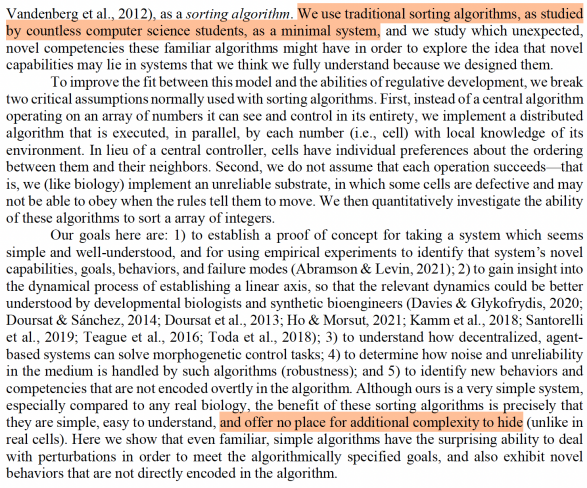
Comments on the Sorting Preprint

Motivation

- This feedback is intended only for you and your team, not as a public critique. I'm a sci-fi writer who mines your work for ideas. I've shamelessly cribbed multi-scale perception, memory-reinterpreting selflets, and the creation of higher-level minds through gap junctions as the core conceits of my first novel. The second novel will be even more shameless. The effort you put into communicating your ideas has helped me more than I can say. So above all, "thank you."
- As per our blog interaction. I wanted to look into this, unsure if I needed to reframe about the limits of basal cognition, or whether there was something suspect in the results. I'm still not sure about the former, but I am convinced there are enough challenges with the study that it isn't persuasive.
- All of my comments are based on the pre-print and the code at Git Hub. I don't have access to the final version of the paper, and I hope the changes were small.
- Do with this as you will, but as a fan of your work, I do hope you take a look and reconsider the conclusions you've reached. As it stands, the paper feels more Wolfram than Levin.

Summary

- The claims of using "traditional sorting algorithms" that offer "no place for additional complexity to hide" are, I believe, wrong.



Vandenberg et al., 2012), as a *sorting algorithm*. We use traditional sorting algorithms, as studied by countless computer science students, as a minimal system, and we study which unexpected, novel competencies these familiar algorithms might have in order to explore the idea that novel capabilities may lie in systems that we think we fully understand because we designed them.

To improve the fit between this model and the abilities of regulative development, we break two critical assumptions normally used with sorting algorithms. First, instead of a central algorithm operating on an array of numbers it can see and control in its entirety, we implement a distributed algorithm that is executed, in parallel, by each number (i.e., cell) with local knowledge of its environment. In lieu of a central controller, cells have individual preferences about the ordering between them and their neighbors. Second, we do not assume that each operation succeeds—that is, we (like biology) implement an unreliable substrate, in which some cells are defective and may not be able to obey when the rules tell them to move. We then quantitatively investigate the ability of these algorithms to sort a array of integers.

Our goals here are: 1) to establish a proof of concept for taking a system which seems simple and well-understood, and for using empirical experiments to identify that system's novel capabilities, goals, behaviors, and failure modes (Abramson & Levin, 2021); 2) to gain insight into the dynamical process of establishing a linear axis, so that the relevant dynamics could be better understood by developmental biologists and synthetic bioengineers (Davies & Glykofrydis, 2020; Doursat & Sánchez, 2014; Doursat et al., 2013; Ho & Morsut, 2021; Kamm et al., 2018; Santorelli et al., 2019; Teague et al., 2016; Toda et al., 2018); 3) to understand how decentralized, agent-based systems can solve morphogenetic control tasks; 4) to determine how noise and unreliability in the medium is handled by such algorithms (robustness); and 5) to identify new behaviors and competencies that are not encoded overtly in the algorithm. Although ours is a very simple system, especially compared to any real biology, the benefit of these sorting algorithms is precisely that they are simple, easy to understand, and offer no place for additional complexity to hide (unlike in real cells). Here we show that even familiar, simple algorithms have the surprising ability to deal with perturbations in order to meet the algorithmically specified goals, and also exhibit novel behaviors that are not directly encoded in the algorithm.

- The algorithms are *not* the well-known algorithms that you would find in Knuth or any textbook.
  - The code takes the spirit of these traditional algorithms and does its best to preserve them in a multi-agent, (mostly) myopic multi-threaded implementation of simultaneous heterogeneous sorts. The net result is a very different animal. They are more different from the originals than they are from each other.
  - In particular, the code does not implement loops in the search algorithms, but only at the thread level, so the "move till you're there" assumptions of Bubble and Insertion sort are violated. This is not "bubble sort with its six lines of code"
  - Further, the standard Selection and Insertion sorts have global state assumptions that are violated by the existence of other instances moving on their own. It works out in the end, but it's different.
  - Further still, the SelectionSortCell code had to be even more creative in mapping to a cell-level implementation with the "ideal position" which leaks a god-level perspective down to the cells. While truer to standard Selection sort, how could a "cell" know its ideal position?
- The asynchronous interaction of dozens of different search processes of different types is *the new algorithm*. This has never been characterized, and appeals to familiarity are misplaced.
- This resulting new algorithm introduces biases into the sorting process that largely (fully?) explain the clustering results. That is the core problem and the heart of what follows.

Code Changes and Excel Output

- Code Changes
  - I added a jb_snapshots array to StatusProbe to capture the tests and swaps in a way that wouldn't mess with the existing recording that the software at Git Hub captures.
  - Every test for a move in xxxSortCell.py::move() records an entry in jb_snapshots for cases when it wakes up but decides not to call MultiThreadCell::swap() for whatever reason
  - In cases where it proceeds to MultiThreadCell::swap(), I've added a call to record it to jb_snapshot, in parallel to the existing snapshot code. Unless I've made a mistake, this should have no impact on your current record-keeping.

    - The only intentional changes were in multithread_sorting_cell_aggregation_analysis, StatusProbe, MultiThreadCell, and the three xxxSortCells. I initially added some prints that are now mostly commented out, and what remains are the changes to record the tests and swaps in my format.
    - The code is attached per below for your diff'g pleasure
- Excel output
    - The code saves jb_snapshots in csv format, then I open and resave in xlsx format to gain conditional formatting.
    - That layout and formatting is as follows
        1. Col A: the cell/thread that woke up and is thinking about moving
        2. Col B: A boolean of whether the move went ahead or not (was a call made to MultiThreadCell::swap())
        3. Col C-CX: The one hundred cell values
        4. For all columns except B, the format is CellValue-ThreadId-AlgoType. So "4-1-S" in A1 below is a cell with value 4, threadId 1, and is a SelectionSort.
        5. Formatting:
            1. Column A is colored by AlgoType (here Selection is yellow)
            2. Column B is red for TRUE (a swap happened)
            3. Columns C:CX are red if the cell is the one wanting to move (ie it matches column A) and green if it is the one being swapped. If B is False, there is no green cell.

	A	B	C	D	E	F	G	H	I
1	4-1-S	FALSE	4-1-S	6-2-B	8-3-S	1-4-S	8-5-S	9-6-B	2-7-S
2	6-2-B	FALSE	4-1-S	6-2-B	8-3-S	1-4-S	8-5-S	9-6-B	2-7-S
3	8-3-S	FALSE	4-1-S	6-2-B	8-3-S	1-4-S	8-5-S	9-6-B	2-7-S
4	1-4-S	TRUE	1-4-S	6-2-B	8-3-S	4-1-S	8-5-S	9-6-B	2-7-S
5	8-5-S	FALSE	1-4-S	6-2-B	8-3-S	4-1-S	8-5-S	9-6-B	2-7-S
6	9-6-B	TRUE	1-4-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B
7	2-7-S	FALSE	1-4-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B
8	1-8-S	FALSE	1-4-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B
9	5-9-S	FALSE	1-4-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B

Method

- For each pair of AlgoTypes I ran two experiments of a hundred cells with ten repeats of each value from 0 to 9. There are clearly named directories holding your npy files, my csv and xlsx files. The effects were obvious and consistent, so I didn't bother with large runs and statistics.
- I used conditional formatting and formatting in Excel to get an eyeball view of the results. The effects are easy to see.

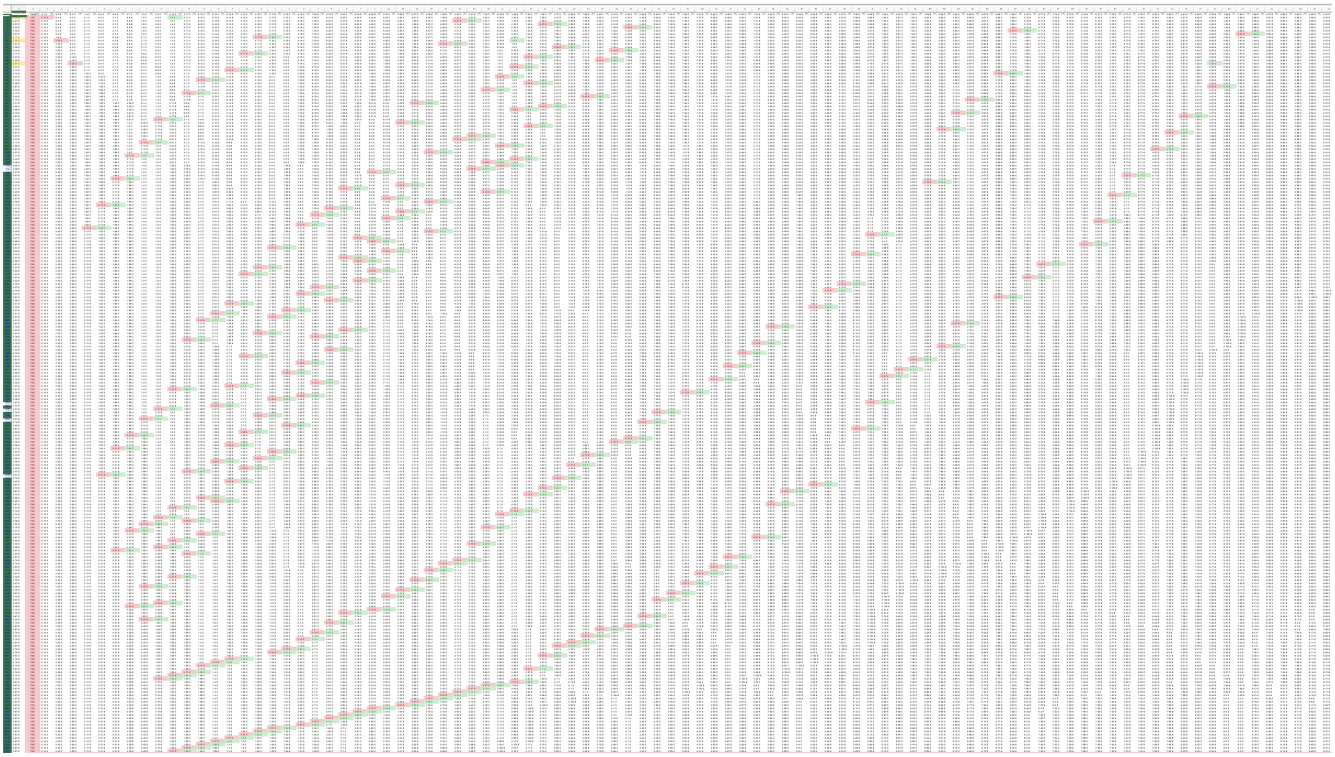
Results

- **Bubble vs Selection**
    1. While the AlgoType is evenly represented in the tests (Col B is True or False), the vast majority of swaps (Col B is True) are for Bubble sort (well over 90%). Selection is just over 100 of the ~1200 swaps in both of the attached files
    2. This is because the Selection cells only have to move once to get where they belong, and Bubble cells have to move dozens of times on average, sometimes in the wrong direction if a neighbor they test against is also misordered.
    3. In other words, Selection cells jump to where they belong, whereas the Bubble cells move one position at a time. It's like a game of chess with only rooks and pawns. The rooks get there first, and the pawns trickle in later. Then, because the swaps only happen to correct a misordering, the pawns will accumulate on top of the initial layer of rooks.
    4. You can see this if you filter column B to look at swaps (B = TRUE) and focus on a single set of values (e.g. Col A starts with 0). You can see the Selection cells (0-x-S) landing first in columns C, D, and E. Then the remaining rows show the trickling down of the Bubble cells (0-x-B) atop them. This is shown in the next two images, where both are filtered to swaps that start with a 0 (ie Col A starts with 0, Col B TRUE))

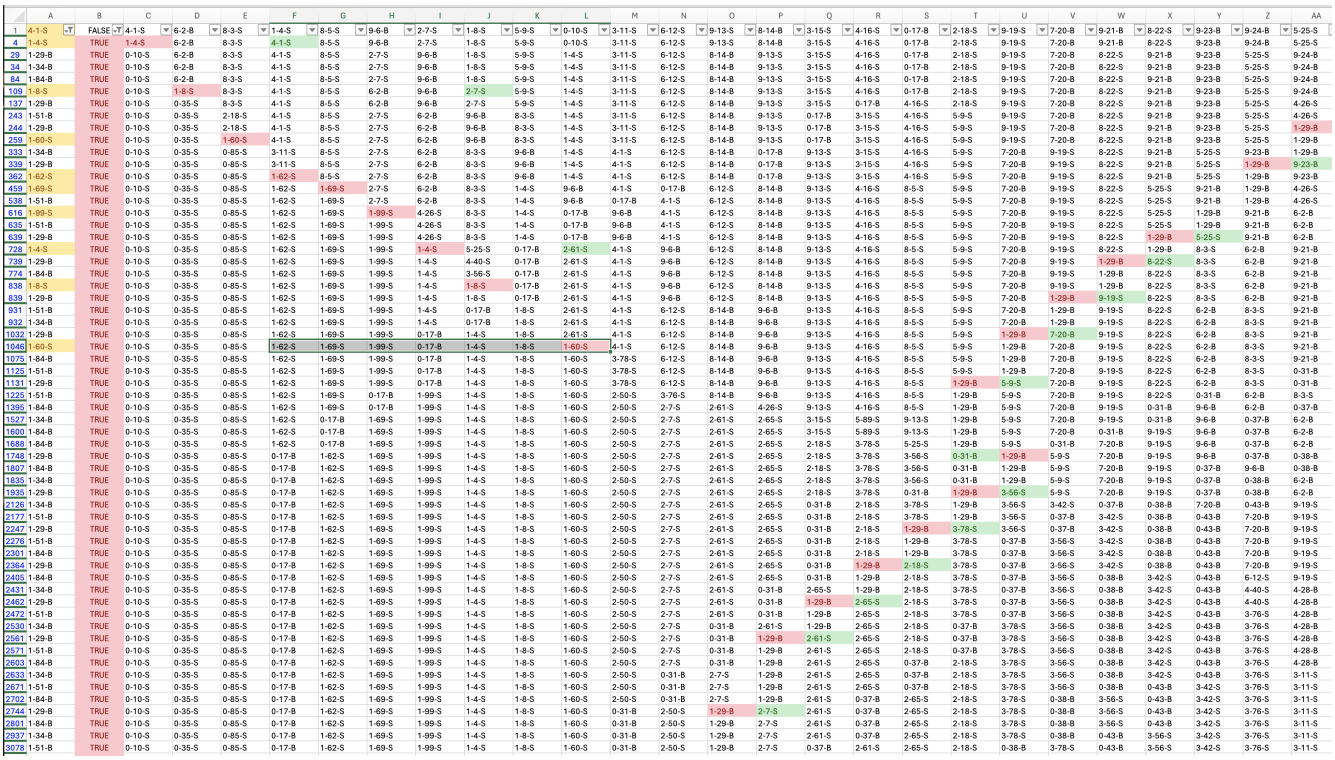
The first three 0-x-S's arrive by row 282, then the first Bubble (0-17-B) trickles in by row 1724

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	4-1-S	FALSE	4-1-S	6-2-B	8-3-S	1-4-S	8-5-S	9-6-B	2-7-S	1-8-S	5-9-S	0-10-S	3-11-S	6-12-S	9-13-S	8-14-B	3-15-S	4-16-S	0-17-B
10	0-10-S	TRUE	0-10-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B	1-8-S	5-9-S	1-4-S	3-11-S	6-12-S	9-13-S	8-14-B	3-15-S	4-16-S	0-17-B
31	0-31-B	TRUE	0-10-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B	1-8-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	4-16-S	0-17-B
37	0-37-B	TRUE	0-10-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B	1-8-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	4-16-S	0-17-B
43	0-43-B	TRUE	0-10-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B	1-8-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	4-16-S	0-17-B
70	0-70-B	TRUE	0-10-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B	1-8-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	4-16-S	0-17-B
87	0-87-B	TRUE	0-10-S	6-2-B	8-3-S	4-1-S	8-5-S	2-7-S	9-6-B	1-8-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	4-16-S	0-17-B
117	0-17-B	TRUE	0-10-S	1-8-S	8-3-S	4-1-S	8-5-S	6-2-B	9-6-B	2-7-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	0-17-B	4-16-S
132	0-35-S	TRUE	0-10-S	0-35-S	8-3-S	4-1-S	8-5-S	6-2-B	9-6-B	2-7-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	0-17-B	4-16-S
134	0-31-B	TRUE	0-10-S	0-35-S	8-3-S	4-1-S	8-5-S	6-2-B	9-6-B	2-7-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	0-17-B	4-16-S
136	0-38-B	TRUE	0-10-S	0-35-S	8-3-S	4-1-S	8-5-S	6-2-B	9-6-B	2-7-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	0-17-B	4-16-S
143	0-43-B	TRUE	0-10-S	0-35-S	8-3-S	4-1-S	8-5-S	6-2-B	9-6-B	2-7-S	5-9-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	3-15-S	0-17-B	4-16-S
219	0-17-B	TRUE	0-10-S	0-35-S	5-9-S	4-1-S	8-5-S	2-7-S	6-2-B	9-6-B	8-3-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	0-17-B	3-15-S	4-16-S
228	0-37-B	TRUE	0-10-S	0-35-S	2-18-S	4-1-S	8-5-S	2-7-S	6-2-B	9-6-B	8-3-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	0-17-B	3-15-S	4-16-S
246	0-43-B	TRUE	0-10-S	0-35-S	2-18-S	4-1-S	8-5-S	2-7-S	6-2-B	9-6-B	8-3-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	0-17-B	3-15-S	4-16-S
282	0-85-S	TRUE	0-10-S	0-35-S	0-85-S	4-1-S	8-5-S	2-7-S	6-2-B	9-6-B	8-3-S	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	0-17-B	3-15-S	4-16-S
309	0-37-B	TRUE	0-10-S	0-35-S	0-85-S	4-1-S	8-5-S	2-7-S	6-2-B	8-3-S	9-6-B	1-4-S	3-11-S	6-12-S	8-14-B	9-13-S	0-17-B	3-15-S	4-16-S
313	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	4-1-S	8-5-S	2-7-S	6-2-B	8-3-S	9-6-B	1-4-S	3-11-S	6-12-S	8-14-B	0-17-B	9-13-S	3-15-S	4-16-S
384	0-70-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	8-5-S	2-7-S	6-2-B	8-3-S	9-6-B	1-4-S	4-1-S	6-12-S	8-14-B	0-17-B	9-13-S	3-15-S	4-16-S
409	0-37-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	4-16-S	2-7-S	6-2-B	8-3-S	1-4-S	9-6-B	4-1-S	6-12-S	0-17-B	8-14-B	9-13-S	3-15-S	8-5-S
413	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	4-16-S	2-7-S	6-2-B	8-3-S	1-4-S	9-6-B	4-1-S	0-17-B	6-12-S	8-14-B	9-13-S	3-15-S	8-5-S
446	0-38-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	3-15-S	2-7-S	6-2-B	8-3-S	1-4-S	9-6-B	4-1-S	0-17-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
487	0-87-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	2-7-S	6-2-B	8-3-S	1-4-S	9-6-B	4-1-S	0-17-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
507	0-37-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	2-7-S	6-2-B	8-3-S	1-4-S	9-6-B	4-1-S	0-17-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
516	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	2-7-S	6-2-B	8-3-S	1-4-S	9-6-B	0-17-B	4-1-S	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
541	0-43-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	2-7-S	6-2-B	8-3-S	1-4-S	9-6-B	0-17-B	4-1-S	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
584	0-70-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	2-7-S	6-2-B	8-3-S	1-4-S	9-6-B	0-17-B	4-1-S	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
628	0-31-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	4-26-S	8-3-S	1-4-S	0-17-B	9-6-B	4-1-S	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
641	0-43-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	4-26-S	8-3-S	1-4-S	0-17-B	9-6-B	4-1-S	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
643	0-38-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	4-26-S	8-3-S	1-4-S	0-17-B	9-6-B	4-1-S	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
682	0-70-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	2-61-S	8-3-S	1-4-S	0-17-B	9-6-B	4-1-S	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
693	0-87-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	2-61-S	8-3-S	1-4-S	0-17-B	9-6-B	4-1-S	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
719	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	2-61-S	8-3-S	0-17-B	1-4-S	4-1-S	9-6-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
730	0-31-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	5-25-S	0-17-B	2-61-S	4-1-S	9-6-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
741	0-43-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	4-40-S	0-17-B	2-61-S	4-1-S	9-6-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
783	0-70-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	3-56-S	0-17-B	2-61-S	4-1-S	9-6-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
794	0-87-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	3-56-S	0-17-B	2-61-S	4-1-S	9-6-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
805	0-37-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	3-56-S	0-17-B	2-61-S	4-1-S	9-6-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
904	0-37-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	1-8-S	0-17-B	2-61-S	4-1-S	9-6-B	6-12-S	8-14-B	9-13-S	4-16-S	8-5-S
918	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	0-17-B	1-8-S	2-61-S	4-1-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
942	0-38-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	0-17-B	1-8-S	2-61-S	4-1-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
991	0-87-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	0-17-B	1-8-S	2-61-S	4-1-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1002	0-37-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	1-4-S	0-17-B	1-8-S	2-61-S	4-1-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1019	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	0-17-B	1-4-S	1-8-S	2-61-S	4-1-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1040	0-43-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	0-17-B	1-4-S	1-8-S	2-61-S	4-1-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1042	0-38-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	0-17-B	1-4-S	1-8-S	2-61-S	4-1-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1133	0-43-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	0-17-B	1-4-S	1-8-S	1-60-S	3-78-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1134	0-38-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	0-17-B	1-4-S	1-8-S	1-60-S	3-78-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1135	0-31-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	0-17-B	1-4-S	1-8-S	1-60-S	3-78-S	6-12-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1192	0-87-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	1-99-S	0-17-B	1-4-S	1-8-S	1-60-S	2-50-S	3-76-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1214	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	0-17-B	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	3-76-S	8-14-B	9-6-B	9-13-S	4-16-S	8-5-S
1299	0-70-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	0-17-B	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	3-80-S	9-6-B	9-13-S	4-16-S	8-5-S
1302	0-37-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	0-17-B	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	3-80-S	9-6-B	9-13-S	4-16-S	8-5-S
1332	0-31-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	0-17-B	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	3-80-S	8-22-S	9-13-S	4-16-S	8-5-S
1334	0-43-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	0-17-B	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	3-80-S	8-22-S	9-13-S	4-16-S	8-5-S
1384	0-87-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	0-17-B	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	4-26-S	9-13-S	4-16-S	8-5-S
1393	0-37-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	0-17-B	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	4-26-S	9-13-S	4-16-S	8-5-S
1436	0-38-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	1-69-S	0-17-B	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	3-11-S	8-5-S	4-16-S	9-13-S
1525	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	0-17-B	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	3-15-S	5-89-S	9-13-S
1531	0-31-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	0-17-B	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	3-15-S	5-89-S	9-13-S
1537	0-38-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	0-17-B	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	3-15-S	5-89-S	9-13-S
1627	0-31-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	0-17-B	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	2-18-S	4-1-S	9-13-S
1706	0-38-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	0-17-B	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	2-18-S	3-78-S	3-56-S
1715	0-87-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	0-17-B	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	2-18-S	3-78-S	3-56-S
1722	0-31-B	TRUE	0-10-S	0-35-S	0-85-S	1-62-S	0-17-B	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	2-18-S	3-78-S	3-56-S
1724	0-17-B	TRUE	0-10-S	0-35-S	0-85-S	0-17-B	1-62-S	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	2-18-S	3-78-S	3-56-S
1743	0-43-B	TRUE	0-10-S	0-35-S	0-85-S	0-17-B	1-62-S	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	2-50-S	2-7-S	2-61-S	2-65-S	2-18-S	3-78-S	3-56-S

Zoomed-out view of showing the remaining 0-x-B's trickling in (the preceding image occupies the top left corner of this one)



You can see a similar effect for the "1" cells, where the -S cells land first, then slowly get pushed right as a block when the 0-x-B cells trickle in (those swap rows not shown), and simultaneously the 1-x-B cells start trickling down on top of them



Once fully sorted, we can take the last row and make an array out of it. You see the effect of Selection cell clustering is very strong for the lower numbers, and a little muddled for higher values, where "swappee's" may have had a lucky shot and been kicked back by a prior swap into the range they belong in

0-10-S	0-35-S	0-85-S	0-17-B	0-31-B	0-37-B	0-38-B	0-43-B	0-70-B	0-87-B
1-62-S	1-69-S	1-99-S	1-4-S	1-8-S	1-60-S	1-29-B	1-34-B	1-51-B	1-84-B
2-50-S	2-7-S	2-61-S	2-65-S	2-18-S	2-41-B	2-52-B	2-74-B	2-86-B	2-100-B
3-78-S	3-56-S	3-42-S	3-76-S	3-11-S	3-15-S	3-80-S	3-58-B	3-59-B	3-72-B
4-26-S	4-1-S	4-40-S	4-16-S	4-93-S	4-55-B	4-57-B	4-63-B	4-67-B	4-28-B
5-30-S	5-64-S	5-89-S	5-9-S	5-25-S	5-49-B	5-81-B	5-83-B	5-90-B	5-91-B
6-12-S	6-97-S	6-68-S	6-53-S	6-75-S	6-66-B	6-73-B	6-77-B	6-44-B	6-2-B
7-46-S	7-32-S	7-39-B	7-96-S	7-36-S	7-45-B	7-47-B	7-20-B	7-94-B	7-98-B
8-33-B	8-22-S	8-54-B	8-5-S	8-3-S	8-79-S	8-48-S	8-14-B	8-71-B	8-82-B
9-27-S	9-6-B	9-13-S	9-88-S	9-23-B	9-92-S	9-21-B	9-24-B	9-19-S	9-95-B

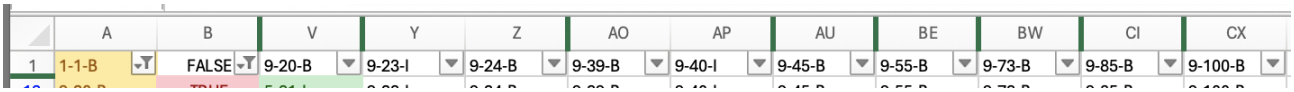
Similar thing for the second run of Bubble vs Selection.

0-29-S	0-68-S	0-85-S	0-22-B	0-30-B	0-48-B	0-52-B	0-60-B	0-74-B	0-87-B
1-53-S	1-59-S	1-63-S	1-44-S	1-4-S	1-49-S	1-18-B	1-19-B	1-47-B	1-65-B
2-16-S	2-35-S	2-69-S	2-97-S	2-13-S	2-94-S	2-36-B	2-55-B	2-93-B	2-96-B
3-20-S	3-8-S	3-15-S	3-92-S	3-43-B	3-6-B	3-83-B	3-84-B	3-24-B	3-88-B
4-75-S	4-82-S	4-9-B	4-41-S	4-5-B	4-28-B	4-46-B	4-23-B	4-67-B	4-91-B
5-27-B	5-34-B	5-66-S	5-11-S	5-70-S	5-100-S	5-45-B	5-58-B	5-21-B	5-79-B
6-61-S	6-64-S	6-51-S	6-32-B	6-57-B	6-39-S	6-81-S	6-2-B	6-86-B	6-90-B
7-62-S	7-71-S	7-1-S	7-17-S	7-26-S	7-99-S	7-37-B	7-76-B	7-77-B	7-31-B
8-14-S	8-3-S	8-54-S	8-40-S	8-12-S	8-50-S	8-72-S	8-73-S	8-89-B	8-95-B
9-33-B	9-10-S	9-7-S	9-25-S	9-38-S	9-42-B	9-56-S	9-78-B	9-80-B	9-98-B

5. *Note also that thread order is strongly preserved until the sort breaks it (see the middle digits of Column A in the first 100 rows). I don't believe this is a big problem, but it might interact with the other biases and is easily fixed by a random initial sleep assignment. On the other hand, it's a very useful property for analyzing the problem in Bubble-Insertion (next)*

- **Bubble vs Insertion**
  1. Insertion moves are roughly 1/3 of the total: ~800/2000
  2. This variant of Insertion sort as coded is very far from a true Insertion sort. An Insertion sort that can only move one cell at a time is basically a uni-directional bubble sort. It can't slide all the way down to its target due to the implementation of the meta-algorithm.
  3. Bubble sort being able to move either direction would explain the approximate 2:1 ratio of moves.
  4. Because neither Bubble nor Insertion can jump over another cell of the same value, the original order of cells of a given value will be preserved.

5. This is easily seen because the threadId's are assigned just prior to the sort, so the ordering of threadId's for cells with a given value will not change from the first row until the end (roughly 60-70 thousand rows later). Here's the sequence in Row 1. Note the monotonically increasing ThreadId's (middle value, starting in Col V: 20, 23, 24 ...).



	A	B	V	Y	Z	AO	AP	AU	BE	BW	CI	CX
1	1-1-B	FALSE	9-20-B	9-23-I	9-24-B	9-39-B	9-40-I	9-45-B	9-55-B	9-73-B	9-85-B	9-100-B

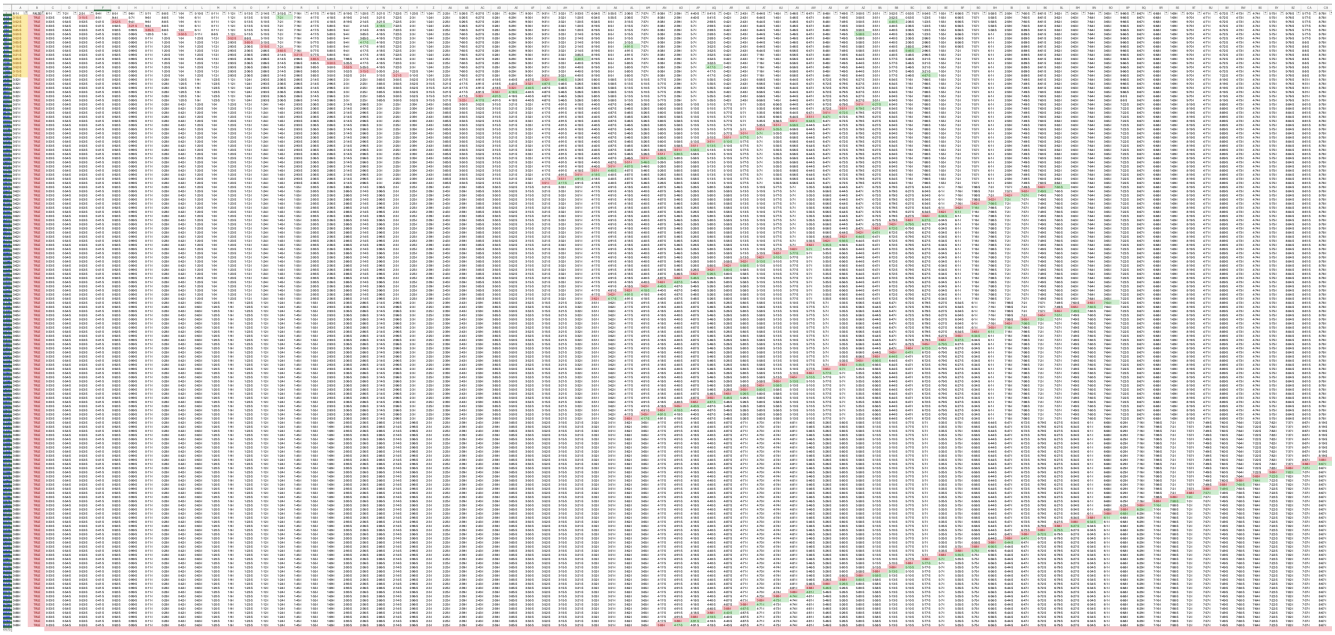
And here's the sorted sequence from the final row presented as an array. Not only do the 9's preserve the initial thread order, but threadID increases monotonically for every row—covering the values from 0 to 9

0-5-B	0-10-I	0-16-I	0-35-I	0-44-I	0-60-I	0-61-I	0-76-B	0-77-B	0-90-B
1-1-B	1-9-B	1-18-B	1-36-I	1-37-I	1-52-B	1-54-I	1-62-I	1-88-B	1-98-I
2-26-I	2-27-B	2-41-I	2-49-I	2-50-B	2-65-B	2-80-B	2-81-I	2-87-I	2-93-I
3-6-B	3-15-I	3-25-I	3-28-B	3-30-B	3-38-I	3-57-I	3-63-I	3-82-I	3-97-B
4-2-B	4-14-I	4-19-I	4-22-B	4-31-B	4-34-I	4-70-B	4-74-I	4-79-B	4-95-B
5-3-B	5-13-I	5-17-B	5-21-I	5-29-I	5-43-B	5-46-I	5-47-B	5-53-B	5-58-I
6-7-B	6-8-B	6-12-I	6-48-I	6-64-B	6-66-B	6-67-B	6-71-I	6-83-B	6-99-I
7-4-I	7-51-I	7-59-B	7-68-B	7-72-I	7-89-I	7-91-B	7-92-I	7-94-I	7-96-B
8-11-I	8-32-B	8-33-I	8-42-B	8-56-I	8-69-B	8-75-B	8-78-I	8-84-I	8-86-I
9-20-B	9-23-I	9-24-B	9-39-B	9-40-I	9-45-B	9-55-B	9-73-B	9-85-B	9-100-B

- **Selection vs Insertion**

1. Now we can make predictions. This one is like Bubble-Selection with Selection as the rooks and Insertion taking the role of pawns, except that the pawns can only move in one direction. This means the whole sort takes much longer because the pawns can't move until all misordered cells to the left get cleared out.
2. At a glance, looking at the swaps for 0-valued cells (ie Col A starts with 0 and Col B is TRUE) we see a similar Rooks and Pawns pattern. All of the Selection sorts jump to the first slots for 0-valued cells (Selection sort moves have the yellow color in the leftmost column), and then the Insertion sort cells trickle in. Shown here for the zeros (0-x-x)

And here for the threes (3-x-x)



The final ordering, colored by AlgoType, is here, showing Selection's advantage in clustering to the left

0-33-S	0-54-S	0-53-S	0-41-S	0-92-S	0-99-S	0-11-I	0-28-I	0-42-I	0-63-I
1-20-S	1-9-I	1-23-S	1-12-I	1-24-I	1-45-I	1-55-I	1-69-I	1-90-I	1-97-I
2-93-S	2-36-S	2-86-S	2-14-S	2-96-S	2-3-I	2-25-I	2-39-I	2-43-I	2-59-I
3-85-S	3-50-S	3-52-S	3-15-S	3-21-S	3-32-I	3-51-I	3-62-I	3-65-I	3-88-I
4-17-S	4-91-S	4-18-S	4-40-S	4-87-S	4-71-I	4-73-I	4-74-I	4-81-I	4-95-I
5-46-S	5-26-S	5-80-S	5-58-S	5-13-S	5-10-S	5-77-S	5-7-I	5-35-S	5-75-I
6-56-S	6-44-S	6-47-I	6-72-S	6-79-S	6-27-S	6-34-S	6-1-I	6-68-I	6-29-I
7-16-I	7-98-S	7-2-I	7-57-I	7-49-S	7-60-S	7-64-I	7-22-S	7-82-I	7-37-I
8-67-I	8-19-S	8-89-S	8-84-S	8-61-S	8-78-I	8-8-S	8-38-I	8-5-I	8-48-I
9-83-I	9-4-I	9-6-I	9-66-S	9-70-I	9-94-S	9-76-S	9-30-I	9-31-I	9-100-S

Conclusion

- Key Findings
  - The algorithms as coded are simply not the well-known and well-studied Bubble, Selection, and Insertion sorts. They are clever adaptations to a cell view, where the control loop has been passed up to a mix of Python threading and a super-class. The combination of multithreaded agents of different types makes this a very different algorithm from the ones it is imitating.
  - Insertion sort, in particular, is modified to the point that it is a poor man's Bubble sort.
  - Because only Selection can jump over misordered cells and arrive where it belongs in one step, it preferentially grabs the initial slots in the sequence for any given value. Bubble and Insertion (as implemented) precipitate down and fill in the rest of the slots over time.
  - Because a swap only happens to solve a misordering, late-arriving Bubble and Insertion cells will never disrupt the layer of Selection sort cells that got there first, making "grouping" an inevitable consequence of the control loop, not an emergent behavior. (Note sometimes there are intrusions from Bubble or Insertion cells that started near the right place)
- Thoughts
  - To the likely pushback that "one can always dig down post-hoc and claim it's all physics, but this paper has discovered emergent goals in something that we thought we understood," my answer would be twofold.
    - No, there is nothing familiar here. This is a new algorithm, with a multi-processing governor overseeing novel daughter algorithms of different capabilities. The paper then describes the computational behavior of this new algorithm. It might be interesting (e.g. like Wolfram) but it's not a new insight into something we thought we knew.
    - Further, the new algorithm does not manifest emergent goals or proto-cognition. It's just cells falling downhill in a created landscape behind a thin veil of stochasticity. The grouping effect is demonstrably a direct consequence of the differing movement capabilities, and I'm confident with a little digging, the delayed gratification would prove to be the same.
  - By analogy, if I made an argument saying, "I just made a Galton board, you know them, just marbles and pegs, there's nowhere for intelligence to hide." But then I fed it marbles of different weights from a feeder that preferred heavy ones. From a purely statistical analysis of the distribution, you'd find the heavy ones at the bottom, and perhaps be tempted to explain it in terms of emergent behavior. I think this would be a mistake.

    - I admire your rigorous empiricism, but this case leads to an interesting question. If you assume you know the process, and only look at the results, can you be misled as badly as if you'd considered the process at the risk of letting biases creep back in? I'd say yes, but it's a slippery slope. Scylla and Charybdis.
    - That's it. I love your work.

Other Business

- Next Steps
    - I didn't look into Delayed Gratification claim, as it strikes me as quite possible this email won't be at all interesting to you. If you're interested, I can.
    - With a little effort you might be able to undo the biases of the meta-algorithm and cause your observed Clustering effects to disappear. But I guess that would torpedo the findings.
- Small Things
    - I found small code issues with the versions in the git repo
        - multithread_cell_sorting_analysis() doesn't run without two changes (probably just a version issue?)
            - define swapping_count = [0]
            - import time
            - (Non-critical) it doesn't save an npy file
        - multithread_sorting_cell_aggregation_analysis left me with questions (it runs fine though)
            - Why the duplicate randomization
            - Why is the "CellGroup" constructor inside the for loop?
            - What are CellGroups for anyway? They don't affect the flow of control as far as I could tell.
- Attachments
    - Full repo with changed files as described in "Code Changes"
    - Results in csv/jb/{sort-type}