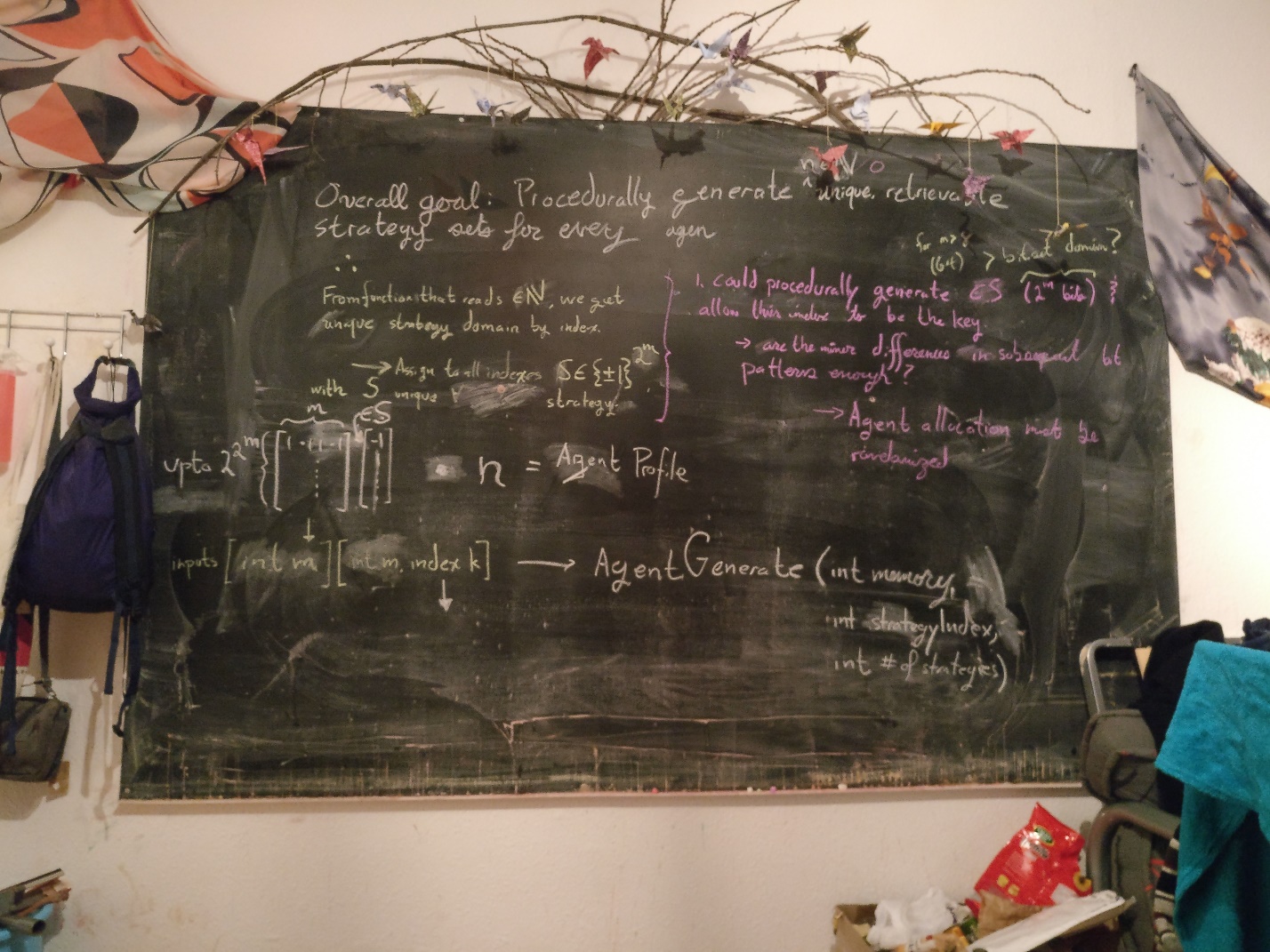
Prelude: All clever inspiration here manifest is brought about through the generous assistance of Gabriel Heinrich, to whom I thusly owe a great debt.

The idea is to procedurally generate all the necessary strategies from indexes rather than maintain them in any kind of system memory, to reduce the program’s memory overhead and improve overall speed and scalability, which becomes especially relevant when looking into adaptive systems with dynamically generated and rated strategy regimes. Given that retrieving memory from ram takes at least two orders of magnitude greater time than any ALU operation, we can readily deduce that even if a two random number generator calls, bit manipulation and tens of ALU operations were required to regenerate any necessary reference value, it should still be considerably faster than any memory retrieval. Ergo, we shall try as best we can to ensure as little memory retrieval is required.

Our overall strategy regime looks as follows: (turn to graphic when pen starts working again)



Thus we’ll need to create a set of all possible binary combinations of a given memory length string, and then create strategies by assigning to every combination a binary number as well. All strategies must be unique, though they may overlap upon random assignment to the agents.

Ergo, we crafted bitset functions that read integers corresponding binary representation and convert it to the necessary -1/+1 pattern. (shifts 0 to -1) By enumerating through , we create all the binary patterns for a given memory length that a given strategy has to map to. These values can then be recalled by index by the reverse function (also made via bitset operations) Our initial part of the initialization therefore only takes the memory length as an argument, and generates all the possible combinations of that as the domain of the strategy. Our next step is producing the strategy mappings through combinations (though really just the first (Number of agents)\*(number of strategies per agent) number of them, though it’s best to have an active mapping to them all (we’ll only be generating those we need anyway) in order to ensure effective randomness in the agent strategy distribution). This may be done in one of several ways:

1. Again, via bitset operations, we can simply read out the enumerated binary digits through and convert them to 1 and -1 as prior. However, as the longest length of bits readable may be 64, this constrains the possible memory length to be less than 8.
2. Random assignment of strategies for m>8, as strategies are unlikely to over lap (there are over ) possible strategies for m=8.
3. A random number generator might seed anther random number generator that then assigns unique sequences of -1 and 1 of the required length , a procedural mechanism that allows for indexing all the necessary elements via their initial rng seed. As we don’t care if the agents have the same strategies, this is likely the way to go, even for m<8.

This brings the question of how these strategies are to be rated, however. While it would be possible to keep an array of all the strategies in use for a given memory length, and then grade them at the end of every round, this requires maintaining a different ledger for every strategy given a different memory length. (unless we choose to allow them to freely go between memory length, but that is contrary to the original experiment). Alternatively, we could have the agents keep track of their own strategies themselves. Though this would require only a tiny bit more memory and computation (as according to their strategy overlap with other players) it would prevent the logistical difficulty of dynamically creating rated strategy pools of all strategy regimes.

With agent memory:

* Strategies are issued randomly to agents.
* Agents use the strategies, and the result vector is updated.
* As all agent strategies are know from an index as a rng seed, we regenerate the random numbers and the corresponding strategies, and see how they would have performed. If they would have preformed and update their scores accordingly. Their score would be stored in an array whose indexes are the rng seed, and they still need to be identified somehow with their memory length. (name would suffice, but can you procedurally generate vectors/array pointers of different name?
* Ergo we have a function that given a memory length and agent population creates strategies and assigns them to agents (i.e. the first two indices might represent the first agent’s two strategies, and the values of that array their corresponding score.), and returns the vector of strategies initialized (to 0) ratings and the input memory length. This allows you to pass both these to another function that would then use this information to store uninitialized array in the “right” place in an array of arrays (strategies of differing memory lengths).

No matter the implementation, unless specified there will be regimes of relative computational inefficiency; consider using the individual agent memory pool methodology in the high agent number and low memory regime, then there is clearly wasted computational resources as they repeatedly evaluate the same strategies. However, we could create a check so that like strategies and consolidated, and the agents look to their consolidation point instead of their original assignment. In this regime, it might even be easier to simply create a pool of all the strategies and keep track of all of them in an array, and randomly assign them. (i.e. for alpha< 1). It should be noted that I used alpha < 1, but this assumes that ram memory retrieval is comparable to having to do the computation over, which is decidedly not the case, so it’s likely that that ratio will have to be off set by this factor; a factor of 100-200 or so, this likely actually only for regimes where: .

With Memory Pools:

* Strategies are all formed, and an array is used to keep track of their scores
* Their scores are all updated at the end of each round, and the corresponding best is called by the agents.