Selection

May 20, 2018

Contents

1 Selection on a Stream

One of the interesting design decisions that crystallized in engineering this iteration of ROPER has been to treat the population as a cyclical "stream", rather than as a mutable collection.

The stream originates in with the seeder, proceeds through the hatchery, on to the evaluator, and then to the selection and breeding actors, without any need to synchronize a mutable population vector. What makes this feasible is the way that Rust handles the `Send` trait: all that's transferred when a `Creature` is sent across a channel is the deed for ownership. This operation is no slower than indexing into a vector, practically speaking.

The only real speedbump lies with the selection actor. Sticking with tournament selection for the time being, we want to retain some capacity to select the combatants in a tournament *at random.* But randomly selecting from a stream seems to require first collecting the incoming elements into a buffer.

So, let there be a buffer. The selector will wait until `n` creatures have arrived through the channel, and then perform tournament selection on that buffer. Some number `tsize` of those creatures will be chosen for a tournament – perhaps several tournaments, in parallel. It will *take* `tsize` creatures, on a secondary channel, then return `tsize` back, but of those `tsize`, `tsize/2` will be the winners of the tournament, and `tsize/2` will be newborns.

1.1 Spawning the selector

```
 <<bring dependencies into scope>>
  pub fn spawn_selector(
      window_size: usize,
      rng_seed: RngSeed,
  ) -> (Sender<Creature>, Receiver<Creature>, JoinHandle<()>) {
      let (from_selector_tx, from_selector_rx) = channel();
      let (into_selector_tx, into_selector_rx) = channel();

      let window = Arc::new(RefCell::new(Vec::with_capacity(window_size+1)));
      let mut rng_seed = rng_seed.clone();

      let sel_handle = spawn(move || {
          let window = window.clone();
          for creature in into_selector_rx {
              let mut window = window.borrow_mut();
              window.push(creature);
              if window.len() >= window_size {
                  /* then it’s time to select breeders */
                  rng_seed = perform_selection_and_mating(&mut window, rng_seed);
                  /* now send them back. new children will have replaced the dead */
                  for creature in window {
                      from_selector_rx.send(creature)
                  }
              }
          }
      });

      (into_selector_tx, from_selector_rx, sel_handle)
  }
<<perform selection and mating>>
```

1.2 Selection functions

To work with the form of homologous crossover implemented in the `emu::crossover` module, we may wish to use simple mate selection algorithm, which increases the likelihood that mating pairs will have "compatible" crossover masks. But this is a probabilistically delicate operation. We don’t want to create a perverse incentive that will incline the population towards crossover masks that consist entirely of `1` bits (and so which are *maximally compatible* with other masks), simply for the sake of increasing their likelihood of being chosen for tournaments.

On the other hand, this incentive will only turn out to be "perverse" if it overwhelms the selective pressure (which we have theoretically grounded reasons to expect) for sparse crossover masks. It could turn out to be a useful, countervailing pressure that inclines the masks to be as dense as possible, without losing the benefits of sparseness. (The benefit of a sparse crossover mask, of course, is that it reduces the probability of destructive crossover.)

```
fn xover_compat(c1: &Creature, c2: &Creature) -> usize {
    (c1.genome.xbits & c2.genome.xbits).count_ones()
}
```

The static variable `MATE_SELECTION_FACTOR` will be used…

```
fn perform_selection_and_mating(selection_window: &mut Vec<Creature>,
                                  seed: RngSeed) -> RngSeed {
    let mut rng = Isaac64Rng::from_seed(RngSeed);
    /* note: seed creation should probably be its own utility function */
    let mut new_seed: [u8; 32] = [0; 32];
    for i in 0..32 { new_seed[i] = rng.gen::<u8>() }

  assert!(*TSIZE * *MATE_SELECTION_FACTOR <= selection_window.len());
    let indices = rand::seq::sample_indices(&mut rng,
                                            selection_window.len(),
                                            *TSIZE * *MATE_SELECTION_FACTOR);
    let combatants = Vec::new();
    for index in indices {
        combatants.push(&selection_window[index])
    }
    <<
    /* take n times as many combatants as needed, then winnow
     * out those least compatible with first combatant's crossover mask
     */

    new_seed
}

fn tournament(combatants: Vec<&Creature>) -> Vec<Creature> {

}
```