Walter - Genetic furnishing - subproject report

Mikołaj Aleksandrzak

June 11, 2019

Genetic_fitness() class

init()

init() initializes class. From *Mapper()* it extracts list of free tiles into ***self.****available_positions*, as well as imports from it the whole grid (map) into ***self.****habitat*. ***self.****chromosome_length* is computed from the amount of elements in ***self.****available_positions*. Here initialized are also ***self.****population* as an empty dictionary and ***self.****final_positions*. Then, **create_ancestors()** is called.

create_ancestors()

We initialize empty list *ancestor*. Then, we randomly choose positions (amount regulated by *TABLES_AMOUNT* constant) from ***self.****available_positions* and save them to *random_positions*. This gives us random arrangement of tables on the grid. Then we create binary chromosome, by appending to *ancestor* 1's where on the equivalent position to those from ***self.****available_positions* values from *random_positions* are located and 0's otherwise. After that, we compute the score of the ancestors using the *compute_score()* method.

As we need two ancestors to for reproducing purposes, those operations are calculated twice.

descendants_iteration()

We initialize the empty list *positions*, which will later store decoded table positions from the final arrangement. For amount of turns regulated by constant *REPRODUCTIONS*, we're choosing two individuals with best scores, starting with our ancestors, for reproduction. The are stored in list *parents*. Firstly, we pass to this method our two ancestors. They swap their „genes" (informations about table positioning) up to the predefined threshold. This way we get our first generation of descendants. Class ***mutation()*** gives them a chance to further change their genome. We then choose two best specimens of a whole group for further reproduction, again using class **compute_score()**. This process continues for a fixed amount of turns.

After all the cycles of reproduction are finished, we check for the highest score of all population, and then we're translating it's genome back into *positions*. Finally, we pass this list to ***self.****final_positions*.

compute_score()

Score computing is based on grid from **Mapper()** class and two methods from **Neighborhood()** class, computing different degrees of neighborhood. First, we're creating grid of size equal to game's grid and initialize it with zeros. Then, based on modificators (neighborhood of walls and windows, proximity of other tables, clogging the entrance - respectively constants *NEARBY_WALL*, *WINDOW_VIEW*, *CLOSE_TABLES*, *ENTRANCE_BARRICADE*) we're assigning scores to tiles with tables and it's neighborhood. Based on that, we determine viability of a sequence.

mutation()

Descendants have a small chance (determined by constant *GENETIC_STABILITY*) to mutate some (max amount determined by constant *MAX_MUTATIONS*) of their *genes*. Those are a randomly chosen sample. Mutation changes in selected positions 0's to 1's (and analogously 1's to 0's). This is also a chance to change the amount of tables in the chromosome (as mutations are independent).

get_position()

Returns *final_positions* list, later used by **Mapper()** class.