`sic42` Technical Documentation

Lab42[*]

Version 0.3.0

1 Overview

This document provides an overview of the simulation environment for the Swarm Intelligence Cup (SIC). The code is written in Python 3.10 [1]. To run a tournament, the user should execute ‘main.py‘. This creates an instance of the main class ‘Tournament‘ and calls its ‘run_tournament‘ function. The ‘Tournament‘ class takes an argument the argument ‘config_path‘, which should be a path to a configuration file, whose contents determine the tournament. Running the ‘main.py‘ file saves the results of the tournament (videos, plots and logs of the simulations) into a folder as specified in ‘results_path‘ in the ‘tournament‘ section of the configuration.

A tournament means a face-off of multiple swarms, each with a respective agent behavior module, where the swarms each play games against each other across multiple rounds, and they are ranked based on their ability to maximize the target function relative to the other competing swarms. The competing swarms can be specified by adding ‘.py‘ files into the ‘competitors‘ folder, where each such file should exhibit a ‘main‘ function that determines the behavior of the agents, where the swarm names are inferred from the names of the ‘.py‘ files. The function to maximize can be specified specified in ‘goal_function_name‘ of the ‘tournament‘ seciton of the config.

A game may consist of multiple sets, where each set is a simulation. A simulation means that there is a virtual environment (two-dimensional grid of cells, where each cell can contain one agent or item) that changes over a number of time steps (e.g. via agents performing actions).

2 Configuration

The path specified in ‘config_path‘ should point to a json file that specifies the tournament. This allows a great deal of flexibility via the contents of this configuration file. The configuration file is separated into four different parts, as

[*]`https://lab42.global/arcathon/`

[1]`https://github.com/lab42-global/sic42/tree/main`

described below. The keys are in bold and their types are in brackets.

tournament

- **tournament_seed** (integer): seed determining the randomness used for determining things like the initial order of swarms and seeds for the individual simulations (which determine the initial environment state, i.e. the grid configuration at time step 0 as well as randomness used during the simulation).
- **goal_function_name** (string): An identifier for the goal function that each swarm should aim to maximize which should have a corresponding entry in ‘LOSSFN_MAPPER‘ in ‘sic.py‘ that points to a corresponding swarm-specific function defined over a simulation.
- **n_tournament_rounds** (integer): The number of rounds of games that will be played in the tournament. If this is None, the number of tournament rounds will be set to $\lceil log_2(|swarms|)\rceil$.
- **n_swarms_per_game** (integer): The number of agents that face-off against each other in a single game.
- **n_rounds_per_game** (integer): The number of sets (i.e. simulations) that will be played in each game.
- **aggregation_method** (string): The method by which the scores across the sets of a game for each swarm will be aggregated to define the game score.
- **results_path** (string): The name of the folder into which the results of the tournament (videos, plots, log files, leaderboard, etc.) will be written to.

environment

- **height_bounds** (list of two integers): a lower and upper bound of the allowed number of rows of cells in the grid.
- **width_bounds** (list of two integers): a lower and upper bound of the allowed number of columns of cells in the grid.
- **background_symbol** (integer): which symbol to use to denote empty cells.
- **swarms_meta_config** (dictionary): The specification of the swarms (the same for each swarm, apart from the agent behaviors as specified in ‘competitors‘). The values are as follows:
  - **n_agent_bounds** (list of two integers): a lower and upper bound of the number of agents of that swarm that shall be spawned in the beginning.

- **subregion** (string or dictionary): This can either be a string denoting which subregion of the grind to allow for spawning the agents (e.g. "everywhere" for the entire grid, "left_half" for only the left half, "bottom_right_quadrant" for the bottom right quadrant) or a dictionary with the following entries to define a more custom region:
  - **view_function_indicator** (string) can be "circle" or "square" for either a circular or square subregion
  - **radius** (integer) the size of the subregion
  - **center** (integer) the location of the subregion
- **agent_params** (dictionary): configuration of the attributes of the individual agents:
  - **view_distance_bounds** (list of two integers): a lower and upper bound of the view distance of the agents. Each agent gets assigned a separately sampled view distance from this range. The view distance determines how far into its neighborhood an agent can perceive.
  - **view_function_indicator** (string): "circle" if the agent shall have a circular viewfield (where the view distance corresponds to the radius) and "square" if the agent shall have a square viewfield (where the view distance corresponds to half the side-length of the square).
  - **initial_energy_level_bounds** (list of two integers): a lower and upper bound of the initial energy level (at spawn) of the agents. Each agent gets assigned a separately sampled initial energy level from this range.
  - **existence_cost_bounds** (list of two integers): a range for the number of energy level points that shall be deducted from each agent's energy level at the end of each time step. As soon as an agent reaches a non-positive energy level, it dies and correspondingly disappears.
  - **activity_points_per_tick_bounds** (list of two integers): a range for the number of activity points an agent can use at each time step to do actions. Each action costs a certain number of activity points. If an agent wants to do a certain action with an insufficient number of remaining activity points, it can't do so. At the end of each time step, each agents' number of activity points gets refilled. However, unused activity points don't accumulate over time but are lost instead.
  - **actions_config** (dictionary): for each action which the agent should be enabled to perform, an entry in here needs to be made, with the corresponding action name as the key. The entries are as follows:
    - **activity_points_cost_bounds** (list of two integers): range for how many activity points it will cost the agent to execute

this action.

  - **energy_level_cost_bounds**: (list of two integers): range for how many energy points it will cost the agent to execute this action.
  - **distance_metric** (string): which distance metric to use when checking if the distance with/to/at which the action shall be performed is within the allowed distance. The options are "chebyshev" (the maximum of the vertical and horizontal distances), "manhattan" (the sum of the vertical and horizontal distances) and "euclidean" (the square root of the sum of the powers of the vertical and horizontal distances). The default is "chebyshev".
  - **max_distance_bounds** (list of two integers) range for the maximum allowed distance with/to/at which the action can perform the action.
  - **required_energy_level_bounds** (list of two integers) range for the energy level required to be able to perform that action.

* **carry_cost_factor_bounds** (list of two integers): whenever an agent's inventory is filled with an item, the agent looses additional energy points at each time step (on top of its existence cost), namely as many as the product of the carry_cost_factor (as sampled from the specified range) and the weight of the item.
* **max_weight_carriable_bounds** (list of two integers): each agent can carry up to one item, but only if the item's weight is below the max_weight_carriable sampled from thsi range (i.e. only then can the item be picked up).
* **max_memory_size_bounds** (list of two integers): sampling range for the maximum number of bytes that the agent's memory may have. The memory must be a dictinary.
* **attack_strength_bounds** (list of two integers): range for the strength with which an agent attacks another agent. The attacked agent dies if their defense strength is below the attacking agent's attack strength, otherwise, nothing happens.
* **defense_strength_bounds**: see attack strength.
* **max_broadcast_size** (list of two integers): range for the maximum number of characters that a broadcasting message may contain.
* **n_pheromone_symbols_bounds** (list of two integers): range for the number of distinct symbols that an agent may place as pheromones (always the integers 1 to n).
* **initial_pheromone_intensity_bounds** (list of two integers): range for the intensity of the pheromone when it is placed.

        - **pheromone_intensity_decay_bounds** (list of two integers): range for the number of points the intensity of the placed pheromone decreases at each time step.
        - **max_write_size_bounds** (list of two integers): range for the maximum number of characters that the agent may store as information (a string) in an item.
        - **max_allowed_powerups_bounds** (list of two integers): range for the maximum number of powerups an agent may have activated at a time.
        - **max_allowed_upgrades_bounds** (list of two integers): range for the maximum number of upgrades an agent may undergo during a lifetime.

- **item_configs** (dictionary): here, the different item types can be specified by adding corresponding entries, where the keys denote the name of the item type and the values should be dictionaries with the the following parameters:
    - **n_items_bounds** (list of two integers): a lower and upper bound of the number of items to spawn.
    - **subregion** (string): the region on the grid eligible for spawning the items (see subregion of swarm parameters for details).
    - **symbol** (integer): the symbol used to denote items of that type.
    - **weight_bounds** (list of two integers): range for the weight of the item.
    - **energy_bounds** (list of two integers): a lower and upper bound of the energy the item can contain at spawn time.
    - **max_energy_bounds** (list of two integers): range for the maximum amount of energy points the item may have at any time.
    - **energy_regrowth_bounds** (list of two integers): range for the number of energy points the item gains each time step (as long as this does not succeed the maximum energy).
    - **disappears_on_no_energy** (boolean): whether or not the item disappears whenever it reaches zero (due to agents having consumed all its energy).
    - **pickupable** (boolean): whether or not the item can be picked up by an agent and carried around in its inventory and be placed down or not.
    - **edible** (boolean): whether or not agents can consume energy from the item.
    - **infostorage** (boolean): whether or not agents may store information in the item.

  - **powerup** (dictionary): the powerup that the item may contain. If the dictionary is empty, the item won't contain a powerup. The powerup is parameterized as follows:
    * **attribute** (string): the attribute of the agent whose value will be temporarily affected by consuming the powerup.
    * **delta** (list of two integers): range for the temporary difference the value of the attribute during the phase where the powerup is active (e.g. if the attribute is the attack strength, whose value is 4, and the sampled delta is 1, then the agent will temporarily have an attack strength of 5).
    * **duration** (list of two integers): range for the number of time steps for which the powerup will be active (and after which the value of the attribute will be re-set to the value it had before the consumption of the powerup.
    * **spawn_probability** (float): the probability of the item spawning with the powerup (as opposed to without a powerup).
  - **upgrade** (dictionary): the upgrade that the item may contain. If the dictionary is empty, the item won't contain an upgrade. The upgrade is parametrized as follows:
    * **attribute** (string): the attribute of the agent whose value will be changed by consuming the powerup.
    * **delta** (list of two integers): range for the difference of the value of the attribute after consumption of the upgrade.
    * **spawn_probability** (float): the probability of the item spawning with the upgrade.

simulation

- **n_timesteps** (integer): the number of time steps for which to run the simulation.

- **stopping_criterion** (string): a string specifying under which conditions to stop the simulation (currently only "all_agents_dead" supported, meaning that the simulation will be terminated as soon as no agents exist anymore).

- **time_multiplier_in_seconds**: (float) The available total runtime per swarm per set (simulation) in seconds is calculated as the product of time_multiplier_in_seconds, the expected number of agents of a swarm at spawn and the expected number of time steps of the simulation, where the expected number of agents of a swarm at spawn is the middle point of the provided range for n_agents_bounds and the expected number of time steps is simply n_timesteps.

visualization

- **video**
  - **scaling_factor** (integer): the factor by which to upscale the video of the simulation, i.e. how many pixels one grid cell shall be high and wide.
  - **fps** (integer): the number of frames (time steps) per second.
- **save_videos**: (boolean) whether or not to save videos as files.
- **save_plots**: (boolean) whether or not to save plots as files.
- **save_logs**: (boolean) whether or not to save logs files.
- **interactive**: (boolean) whether or not to display the interactive video during the simulation.

Remarks on the configuration file:

- Bounds for integers parameters are inclusive. Hence, to set a parameter that is specified via bounds to a constant, one can simply set the lower bound and upper bound both equal to that hardcoded value. See classes in sic.py for default values of the bounds (oftentimes [0, 0]).

3 Tournament

In order to determine a ranking among the different agent behavior modules (as specified via main functions in separate Python files in the folder "competitors"), a tournament is held. A tournament consists of a number of specifiable rounds n_tournament_rounds (which defaults to $\lceil log_2(|swarms|)\rceil$). In each round of the tournament, each agent plays once (except if the number of swarms is not divisible by the number of swarms per game), and the ranking is updated. The ranking after the last round determines the leaderboard. The competing swarms are ranked by their cumulative scores (cumulative over the tournament rounds). The score a swarm gets for a round is equal to the score it got for the game it played that round, if it did play that round. The score a swarm gets for a game is defined as the aggregated scores it got for the sets of that game, e.g. "mean". The set score (score for a single simulation) is simply the value of the goal function with respect to the swarm, e.g. the average number of agents across all time steps of the simulation.

The initial ranking (before the first tournament round) is random. The swarms are paired into teams based on their proximity according to the ranking (e.g. if there are four teams and two swarms per game, then at each round, the highest-ranked two swarms will face off and the lowest-ranked two swarms will face off). If the number of swarms is not divisible by the number of swarms per game, then as many swarms as the remainder (e.g. 2 in the case of 8 swarms and 3 swarms per game) are not playing a game each round. The swarms that do not play for any given round is randomly chosen among the swarms with the highest number of games played so far. If a swarm does not play at any given round, instead of the game score, it gets assigned an expected score for that round (that gets added to its overall cumulative score). In case a swarm does not play in the first round, this expected score is defined as the average of the scores that all the swarms which have played have gotten in that first round. In case a swarm does not play in any other but the first round, the expected score is determined as the average of its scores of all previous rounds (regardless of whether they are true or expected scores). The final ranking of the competing swarms (from best to worst) is defined by ordering the swarms (descending) by their final cumulative scores.

During the tournament, some information is printed, such as the current round, game (including names of participating swarms), and set (simulation), and each simulation has a progress bar. The final leaderboard is also printed, at the end of the tournament. More importantly, the outputs of a tournament are stored in a folder as specified in "results_path". The structure is as follows: The folder contains a file called "leaderboard.csv" (with a row for each competing swarm and a column for the cumulative score and one for the rank, as well as a column for the scores of each round) and a file called "tournament_log.json" (which contains both the results of the tournament as well as a summary with the most important information about the tournament, such as the competitors, the goal function, the tournament winner, etc.). The folder further contains one folder for each tournament round, which in turn contains a folder for each game played in that round. Each game-folder contains a "results.csv" file with the scores of the sets (simulations) for that game, and a "times.csv" file with the corresponding runtimes, as well as folders for each set (simulation). Each set folder contains a "visualization.mp4" file, which is a video visualization of the simulation, and two folders "plots" and "logs". The plots folder contains two types of plots: once the graphs for the cumulative action counts time series for each action (in "action_counts") and once graphs for the averages of agent attribute values over time (in "average_agent_attributes"). The logs folder contains two json files for each swarm: one for the actions of the agents of that swarm (a chronological list dictionaries with keys "agent_id", "timestep", "action_name", "action_value", "successful", and "error_message"), and one for the errors (a chronological list of errors, again dictionaries, with keys "timestep", "agent_id" and "error_message", where the errors may range from failures during execution of the agent behavior modules, upon reading of the "agent_output.json" file, etc.).

4 Simulation

4.1 Environment Initialization

At the beginning of a simulation, the environment is initialized, that is, an empty grid of sizes within the specified ranges is constructed and subsequently populated by the swarms and items, as specified in the configuration file. Numbers sampled from ranges (e.g. the height of the grid, or the number of agents of a swarm, etc.) are drawn uniformly. The actual environment grid only contains the entity symbols, the items and agents themselves and the set of empty locations are stored separately (given a location, the environment class can return the corresponding entity at that location, which may be none).

4.2 Simulation Step

For a total of (at most) "n_timesteps", the following things happen in each time step:

1. A random order of all agents of all swarms is sampled. The simulation is asynchronous, meaning, agents do actions one after another (as opposed to synchronously/simultaneously).

2. For each agent, the following things happen:

   (a) If the agent has died in the meantime (e.g. by having been successfully attacked by another agent whose turn was earlier in that time step), nothing happens (the agent will already have died and been removed from the environment correspondingly).

   (b) The agent behavior module receives some local information and shall return a list desired actions for the agent at turn.

   (c) The actions are executed whenever possible, in the same order they are specified in the list of desired actions.

   (d) The per-swarm remaining available runtime gets deducted the execution time, the agent's energy level gets deducted its existence cost, the energy level gets further decreased in case the agent's inventory is filled with an item with non-zero weight, the agent's activity points get re-set, the powerups are handled (i.e. attributes get adjusted to their previous values if the corresponding powerups' durations are over), and the agent dies and with that disappears if it has reached a non-positive energy level.

3. The item energy levels get updated according to their current energy level, regrowth rate and maximum energy.

4. For each item type, items are respawned such that the number of items of that type on the grid is again within the specified range "n_items_bounds" (if possible).

5. The pheromones lose intensity and pheromones without any intensity remaining get removed from the pheromone map.

6. Frame-specific information such as statistics get stored for later visualization.

7. Handling of the stopping criterion (e.g. interrupt the simulation if all agents are dead if specified accordingly).

4.3 Interface

The interface between the agent-behavior module (providing desired actions) and the simulation (providing a local view-field) is specified as follows.

4.3.1 Inputs

The agent-behavior module at each time step for each agent receives a json (called "agent_input.json") containing the following entries:

- **self_view**: A view of the agent's internals i.e. information of the agent at turn (a subset of the agent class attributes, such as the current energy level, the view distance, etc., however not including things like e.g. the absolute position of the agent on the grid).
- **indices**: A list of [x, y] coordinates that specify the view-field of the agent. The coordinates are relative to the agent's current position (set to be at [0, 0]), not the absolute locations that the agent sees on the board.
- **entities**: A corresponding list with the respective entity views for each of the indices (locations) that the agent sees. Hence, the list entries may be none (in case the cell is empty), an item view or an agent view. Note: The items only expose certain information about themselves that becomes visible here, and an agent view consists of only the name of the swarm it belongs to. To clarify: The i-th entry of the entries list denotes the contents of the cell at coordinates as specified in the i-th entry of the indices list.
- **pheromones**: A corresponding list with the pheromones for each location. Note: Unlike in the environment, where for each pheromone intensities and intensity decays are relevant, here, the agent only gets to know whether a certain pheromone (symbol) is present at a location or not. To clarify: The i-th entry of the pheromones list denotes the pheromones present on the cell at coordinates as specified in the i-th entry of the indices list.

4.3.2 Outputs

The agent behavior module should write its output into a file called "agent_output.json". What the agent behavior module shall produce for each time step for each agent (after optionally having read and processed the information in "agent_input.json") is a list of desired actions, where the order of the actions is respected. Each entry in this list of desired actions is a list of two entires, where the first entry corresponds to the identifier string of the action, e.g. "eat", and the second entry corresponds to the respective "action value", e.g. the relative location from which to eat. An agent may perform any of the allowed action as often as it wants and in whatever order it wants (as long as of course the agent is given the ability to perform that action in the first place, and the action itself is valid, i.e. the number of remaining activity points is sufficient, it is within the allowed distance, etc.). If a desired action could not be executed, the agent loses an activity point. The possible actions are described in the next section.

5 Actions

5.1 Definitions and Necessary Conditions

The possible actions that agents may perform are: eat, pickup, putdown, step, memory, broadcast, attack, reproduce, pheromone, inforstorage, powerup, upgrade, itemmerge. Note that in order for a specific action to be executable by an agent, the action needs to be specified in the config file and a number of action- and agent-specific conditions need to hold (e.g. having sufficient remaining activity points, the required energy level, or for any action that is distance-related (i.e. all except the memory action), being within the allowed distance), which, since they apply for most actions, are not listed separately as necessary conditions below.

- **Movement**: The action is called "step" and corresponds to the movement of the agent from one cell to another cell. The action value is the cell the agent wants to move to, relative to its current position. E.g. if it is [1, 2], then the agent wants to move one cell down and two cells to the right. The desired target location needs to be both empty and within reach.
- **Eating**: The action is called "eat" and corresponds to the transmission of energy from an item to an agent. The action value is the location from where to eat, relative to the agent. The agent can only eat if there is indeed an edible item at the specified location, in which case all of the energy points from the item are added to the agent's energy level and the item's energy level is set to zero. The item disappears if it has "disappears_on_no_energy" set to true (but it may regrow its energy otherwise).
- **Picking Up Items**: The action is called "pickup" and corresponds to picking up an item from the grid and taking it into the inventory. The action value is the location of the item to pick up, relative to the agent. The entity at that location needs to be a pickupable item (i.e. an item with the attribute "pickupable" set to true), and its weight needs to be at most the agent's "max_weight_carriable" attribute, in order for the agent to be able to pick up the item. An agent may have at most one item in its inventory at any time. Carrying an item costs the agent as many energy points as the product of the carry_cost_factor and the weight of the item, at each time step.
- **Putting Down Items**: The action is called "putdown" and corresponds to placing down an item from the inventory onto a cell. The action value is the location where the item shall be put down, relative to the agent.

The cell needs to be empty in order for the agent to be able to put down the item.

- **Attacking**: The action is called "attack" and corresponds to attacking another agent. The action value is the location of the agent to be attacked. The entity at the target location needs to be an agent. If the attack strength of the attacker is greater than the defense strength of the victim agent, the victim agent's energy level gets deducted the difference between the attack strength and the defense strength, otherwise nothing happens.

- **Reproduction**: The action is called "reproduce" and corresponds to the agent producing offspring. The action value is the location of the cell where the offspring shall be born onto. Reproduction (as most other actions) can be parametrized such that there is a minimum energy level required to be enabled to perform it (corresponding to "unlocking" an action). Reproduction works very simply, namely by placing a precise copy of the parent agent at the desired target location, with the following exceptions: The agent id will be different, the energy level will be halved, all powerups get deactivated, the inventory as well as the memory will be empty. If the reproducer has made upgrades, the offspring will also have the upgraded attributes. Note that the reproducer also gets its energy level halved.

- **Memory Updating**: The action is called "memory" and corresponds to an agent updating the contents of its memory. The action value is simply the memory itself (which must be a dictionary), and the size of the memory in bytes should be at most max_memory_size. If the agent does not perform any memory update, its memory remains (and will again be passed to the agent behavior module as part of the "self-view" in the next time step).

- **Information Storage**: The action is called "infostorage" and corresponds to the agent storing a piece of information (string of characters) into an item. The action value is a list of two entries, where the first entry is the location of the cell where the item that the agent wants to write to is located, relative to the agent, and the second entry is the string that shall be stored. The string that the agent wants to write must be at most "max_write_size" long and the entity at the specified location needs to be an item that has the ability to store information. In case the item already had some information stored, this information will simply get overwritten.

- **Information Broadcasting**: The action is called "broadcast" and corresponds to the agent sending a number of messages to other agents of the same swarm within its viewfield. The action value is a list of lists (where each sub-list has two entries), where the first entry of each sub-list corresponds to the relative location of an agent to which to send the message, and the second entry of each sub-list corresponds to the message that the agent wants to send to that agent. Hence, within a single broadcast action, an agent may broadcast to multiple agents, and it need not be a single message that is being sent. However, each agent to which shall be broadcasted increases the activity points cost, i.e. the cost of a broadcast action is the product of the broadcast activity points cost and the number of recipients. Each message length needs to be within the specified "max_broadcast_size". Each recipient receives the message into its inbox, which is contained in the self view of that recipient agent. However, the inbox of each agent gets emptied at the end of its turn.

- **Pheromone Placement**: The action is called "pheromone" and corresponds to the agent placing a pheromone (symbol) onto any grid cell within reach. The action value is a list of two entries, where the second entry corresponds to the symbol that shall be placed and the first corresponds to the location where it shall be placed, relative to the agent. Each agent can only write up to "n_pheromone_symbols" distinct symbols, which need to be integers in the range $[1, n]$. Hence, the meaning of the symbols is entirely left to the agent behavior module, i.e. pheromones only allow a set of distinct symbols to be perceived, not arbitrary messages (also, pheromones are visible to all swarms). Pheromones can also be placed on cells that are occupied by an entity (item or agent). The pheromone intensity is initially "initial_pheromone_intensity" and decreases by "pheromone_intensity_decay" after each time step.

- **Powerup**: The action is called "powerup" and corresponds to consuming a powerup from an item. The action value is the location of the item from which to consume the powerup, relative to the agent. Of course, at the specified location there needs to be an item that also contains a powerup. Upon consumption of the powerup, it gets immediately activated, and temporarily affects the respective attribute of the agent by the specified delta for the specified duration. However, powerups can only be consumed if the number of active powerups is below "max_allowed_powerups" and the powerup does not affect any of the attributes of the agent for which already another powerup is active. After consumption, the powerup disappears from the item.

- **Upgrade**: The action is called "upgrade" and corresponds to consuming an upgrade from an item. The action value is the location of the item from which to consume the upgrade, relative to the agent. The location needs to correspond to a cell with an item that features an upgrade. Upgrades - as opposed to powerups - permanently alter the corresponding attribute of the agent. The "max_allowed_upgrades" specifies how many upgrades an agent may undergo at most during its lifetime.

- **Item Merging**: The action is called "itemmerge" and corresponds to merging two items. The action value corresponds to the location of another agent, whose inventory item shall be merged with the item in the action-performing agent's inventory, relative to the agent position. The location needs to correspond to a location containing an agent of the same swarm with an inventory that contains an item, and the agent at turn also needs to have an item in its inventory. The merged item gets placed into the inventory of the action-performing agent, and the other agent's inventory becomes empty, correspondingly. Item merging happens as follows: If the combination of two items has never been merged before during that simulation, a new item is being created for which each attribute value is randomly sampled with some bias towards better attribute values. If the combination of two items has been merged before during that simulation, the randomness is removed, i.e. the item is the same as the first time of a merger of equivalent two items (apart from its id, of course). For each numeric item attribute, the value is sampled from the range specified by the minimum of the respective attributes of the two items to be merged and the sum of them (hence the attribute value of the merged item may be larger, but never smaller). For boolean attributes, the more favourable value (e.g. true for "pickupable") has a higher probability of being sampled. Powerups (and upgrades) also get "merged". If neither of the two items contains a powerup (upgrade) , the merged item also does not contain a powerup (upgrade). If at least one item contains a powerup (upgrade), then one of the available powerups (upgrades) is chosen and its "delta" increased by up to 100%, with a bias for higher values, and its "attribute" is changed towards the more favourable direction (e.g. for the existence cost, lower values are more favourable, for the attack strength, higher values are more favourable, etc.), in the following manner: If the favourable direction is towards lower values, then the value may reduce by up to 50%, otherwise the value may increase by up to 100%. The "favourable direction" is defined as the one that would allow the agent to either perform more actions (e.g. via a higher number of activity points), or better-informed actions (e.g. via a larger view distance), etc..

5.2 Examples

The following are some examples of action values for each of the featured agent actions. Note that the coordinates are such that the first entry x of a coordinate [x, y] means down (or up, if negative) and the second entry y means right (or left, if negative). The conditions that are required for the attempted actions to succeed are not mentioned again here, they can be found in the previous section where the actions are explained in more detail. Assume the agent is located at location [x, y] (at the time of attempting to perform the respective action). The following 9 actions are very straight-forward in terms of their action values, as in all cases the action value is a location relative to the agent's current position:

- eat: [1, 2]: eat from the item at [x+1, y+2].
- move: [-3, 4]: move to [x-3, y+4].

- pickup: [2, 2]: pick up the item at [x+2, y+2].
- putdown: [-1, 2]: put down the item at [x-1, y+2].
- attack: [5, 6]: attack agent at [x+5, y+6].
- reproduce: [3, 3]: place offspring at [x+3, y+3].
- powerup: [2, 1]: consume powerup from item at [x+3, y+3].
- upgrade: [4, 1]: consume upgrade from item at [x+3, y+3].
- itemmerge: [1, 1]: merge inventory item with item of agent at [x+3, y+3].

The action values for the following actions are a bit more involved:

- infostorage: [[1, -1], ”hi”]: write messgage ”hi” into item at [x+1, y-1].
- pheromone: [[2, -3], 1]: place pheromone 1 onto cell [x+2, y-3].
- memory: {”i”: 1, ”j”: 0}: update memory to {”i”: 1, ”j”: 0}.
- broadcast: [[[1, 0], ”above you”], [[-1, 0], ”below you”]]: broadcast message ”above you” to agent at [x+1, y] and broadcast message ”below you” to agent at [x-1, y].

The contents of ”agent_output.json”, as constructed for each agent at each turn, should be a list of desired actions. The list may be empty, in which the agent does not attempt to do any actions, or contain only a single desired action (e.g. [[[”eat”, [1, 1]]]), but may contain a sequence of desired actions that are attempted to be executed one after the other. Here are some examples of action sequences:

- an agent picking up the item directly above and placing it directly below: [[”pickup”, [-1, 0]], [”putdown”, [1, 0]]]
- an agent picking up an item directly left of it, moving two to down, and then placing it left of itself again, thus effectively moving the item two down: [[”pickup”, [0, -1]], [”step”, [2, 0]], [”putdown”, [0, -1]]]
- an agent merging its item with the item of the agent right of it and then updating its memory, e.g. as to indicate that the item in it’s inventory is a merged item: [[”itemmerge”, [0, 1]], [”memory”, {”merged”: 1}]]