Bomberman Code Structure

Table of Contents

main.py

- First defines the keys (q or ESCAPE) to escape pygame
- Defines the class *Timekeeper*.
    - Manages the time/FPS to update the GUI in pygame
    - This class is used in the function *render*.
- Defines the method *world_controller*
    - Creates a directory for the screenshots if *make_video* is true.
    - Runs the number of rounds defined in *n_rounds*
        - **NB**: For every round, it calls *world.new_round()*
    - Renders the GUI if necessary, using the function *render*.
        - The function render either waits for the next frame, if the CMD argument *skip_frames* is false, otherwise it (may) skip some frames
    - Listens to user input (to show next frame or to play the game)
        - **NB**: There are multiple keys defined in *INPUT_MAP* of *settings.py* to play the game (using *--my-agent user_agent*) or else to show the next frame. For both, one must set *--turn-based*.
    - Performs a step in the world.
    - Creates a video if *make_video* is true.
    - Renders the end screen until next round is initialized or pygame is escaped.

- Defines the method *main*.
    - Parses the arguments.
        - Overall arguments
            - *--turn-based* => Wait for key press for next move/frame
            - *--update-interval* => How often agents take steps, i.e. how often the GUI is updated (ignored without GUI)
            - *--log-dir* => Store logs in ./logs
            - *--save-stats* => Store the game results as .json for evaluation
            - *--make-video* => Make a video from the game
        - Define the mode *play*.
            - *--my-agent* => Play agent of name … against three rule-based agents
            - *--agents* => Explicitly set the agent names for the game (my_agent must be unset)
            - *--scenario* => set game scenario (empty, coin-heaven, loot-crate, classic)
            - *--train* => First … agents should be set to training mode
            - *--continue-without-training* => Continue round even if training agents are dead
            - *--single-process* => *AgentConnector* realised by a separate process (commented out, see *ProcessAgentBackend*)
            - *--seed* => Reset the world's random number generator to a known number for reproducibility
            - *--n-rounds* => How many rounds to play
            - *--save-replay* => Store the game as .pt for a replay
            - *--match-name* => Give the match a name
            - *--silence-errors* => Ignore errors from agents
            - *--skip-frames* => Play several steps per GUI render
            - *--no-gui* => Deactivate the user interface and play as fast as possible.
        - Define the mode *replay*.
            - **NB**: Automatically activates the GUI, sets the number of rounds to 1 and sets the match's name to the file name of the replay.
    - For the *play* mode, it creates BombeRLeWorld
    - For the *replay* mode, it creates *ReplayWorld*
    - Creates an instance of the *GUI* class if necessary.
    - Calls the *world_controller*

settings.py

- Sets the board size (columns & rows)
    - **NB**: A smaller bard size might be useful at the beginning of the training.
- Set game scenarios/modes (empty, coin-heaven, loot-crate, classic)
    - NB: Own game modes can easily be added.
- Sets the max number of agents and steps.
- Sets the GUI properties (height, width, grid size & offset).
- Sets the directory of the assets and the colour of the agents.

- Sets the game rules (bomb timer, explosion time & range)
- Sets the regular step timeout => 0.5 seconds regularly and infinite during training
- **Sets the rewards for killing (5) and collecting a coin (1)**
  - **IMPORTANT: These are the rewards of the total game.**
- Sets the *INPUT_MAP* to play the game or to show the next frame.
- Sets the logging levels.

items.py

- Defines the basic class *Item* with the methods *avatar*, *render* and *get_state*
- Defines the class *Coin* as a child of *Item* having a grid position (*x, y*) and a *collectable* Boolean.
- Defines the class Bomb as a child of *Item* having a grid position (*x, y*), an *owner*, a *timer*, a *power* [= range] and an *active* Boolean
  - It also defines the method *get_blast_coords* to return all game/grid positions of the explosion.
- Defines the class *Explosion* as a child of *Item* having a screen/window position (*screen_coords*), a grid position (*blast_coords*), an *owner*, a *timer* and a *stage*.
  - It also defines the method *is_dangerous* to return *true* if the *stage* is 0.
  - It also defines the method *next_stage* to increase the *stage* and reset the *timer*.

events.py

- Defines all possible events that can happen:
  - MOVED_LEFT, MOVED_RIGHT, MOVED_UP, MOVED_DOWN, WAITED
  - INVALID_ACTION
  - BOMB_DROPPED, BOMB_EXPLODED
  - CRATE_DESTROYED
  - COIN_FOUND [= by destroying crate], COIN_COLLECTED
  - KILLED_OPPONENT, KILLED_SELF
  - GOT_KILLED, OPPONENT_ELINIMATED [= by any agent], SURVIVED_ROUND

agents.py

- Defines the API of the underlying agent code folders in *./agent_code/.*.
- Maps the events (from ./events.py) to the statistics.
- Defines the class *Agent*.
  - This is a wrapper class that is connected to the agent code folder via the backend, which is an object of the class *AgentBackend*. In particular, it calls the callbacks of *callbacks.py* (and the callbacks of *train.py*) in the specified agent code folder by sending and receiving events to/from the backend. The backend has an attribute *runner* of the class *AgentRunner* that runs the callbacks on a (dirty) low level and puts the result back into the attribute *result_queue* of the backend.
    - **NB:** *result_queue* is an instance of the *Queue* class from the *multiprocessing* module. Passing a timeout to the method 'get', which retrieves elements from the queue, blocks the process until the data is available and throws an error if the timeout is exceeded.

            - See: https://docs.python.org/3/library/multiprocessing.html#pipes-and-queues
    - At initialization, it sets the obtained backend, loads the custom *avatar* and *bomb_sprite*, calls the method *setup* and initializes the arguments:
        - *name* => Internal name of the agent
        - *display_name* => Display name of the agent
        - *code_name* => Name of the underlying agent code folder
        - *train* => Whether the agent is training or not
        - *score* => Score of the current round
        - *total_score* => Total score over all rounds
        - *dead* => Whether the agent is dead in the current round
        - *statistics* => Statistics of the current round
        - *lifetime_statistics* => Total statistics over all rounds
        - *trophies* => List of trophies of the current round
        - *events* => List of events that occurred for the last taken action
        - *available_think_time* => Available thinking time for every action
        - *(x, y)* => grid position
        - *bombs_left* => Whether the agent can place the bomb
            - **NB**: The explosion of the bomb must disappear entirely before a new bomb can be placed.
        - *last_game_state* => Current overall state of the game
        - *last_action* => Last taken action of the agent
    - The method *setup* sends and receives the *setup* event to/from the backend (but it does not use the response).
        - If *train* is true, it also sends and receives the *setup_train* event to/from the backend (but it does not use the response).
    - The method *start_round* resets the arguments for the next round.
    - The method base_timeout returns the timeout of the agent from the settings (*TRAIN_TIMEOUT* [= infinity] if the agent is trained, else *TIMEOUT* [= 0.5 seconds])
    - The method *add_event* adds an event (that occurred after the last taken action) to the *events* list
        - If the event is also a statistic, then the method *note_stat* is called
    - The method *note_stat* increments the respective statistic of the current round (*statistics*) and the total statistic over all rounds (*lifetime_statistics*) by 1
    - The method *get_state* returns the state of the agent for the global game state.
    - The method *update_score* increments the score of the current round (*score*) and the total score over all rounds (*total_score*) by *delta*.
    - The method *process_game_events* sends the last & new own state, the last taken action and the resulting events to the underlying agent code folder
        - **NB**: The method is only called if the agent is trained, so that it can learn.
        - **NB2**: The method is called after the agent has acted, so that it can learn from the consequences of its action.
    - The method wait_for_game_event_processing waits until the underlying agent code folder has learned from the last & new own state, the last taken action and the resulting events
        - **NB**: The method is only called if the agent is trained, so that it can learn.

        - **NB2**: The methods *process_enemy_game_events* and *wait_for_enemy_game_event_processing* are commented out, but this means that it is also possible to learn from the new state of enemies
    - The method *store_game_state* stores the current own state in *last_game_state*
        - **NB**: The method is called just before all agents choose the next action, so that *last_game_state* holds the own state before all these actions.
    - The method *reset_game_events* resets the events
        - **NB**: The method is called just before all agents choose the next action, so that *events* is empty a can be filled by the events of the next action.
    - The method *act* sends the current game state to the underlying agent code folder so that it can choose the next action.
    - The method *wait_for_act* waits until the underlying agent code folder has acted based on the current state.
        - **NB**: The method also increments the thinking time and the number of steps in the statistics. The just taken action is also stored in *last_action*.
    - The method *round_ended* sends the last game state, the last action and the last events to the underlying agent code folder and waits until it has learned from it.
        - **NB**: The method is only called if the agent is trained, so that it can learn.
    - The method *render* draws the agent's avatar to the screen based on the received screen/window position
- Defines the class *AgentRunner*.
    - At initialization, it connects on a (dirty) low level to the callbacks and train API of the underlying agents code folder. It also logs the communication.
    - The method process_event processes a new event by sending it down to the underlying agent code folder. Afterwards, the event name, the duration of the call and the results of the event are inserted in *result_queue*.
- Defines the class *AgentBackend*
    - This class is an interface that acts between the classes *Agent* and *AgentRunner*. On a high level, it creates a backend that connects the agent in the environment to the agent code folder.
    - At initialization, it initializes the arguments:
        - *agent_name* => Internal name of the agent
            - NB: This is the same as the argument *name* of the class *Agent*.
        - *code_name* => Name of the underlying agent code folder
        - *train* => Whether the agent is training or not
        - *result_queue* => Queue that holds the results of the callback events
    - The method *get* calls *get_with_time*
    - The method *get_with_time* extracts the results of a callback event from *result_queue*
- Defines the class *SequentialAgentBackend* as a child of *AgentBackend*
    - This is a backend that is realized in the main thread and is thus easy to debug.
    - At initialization, it initializes the argument *runner*, that holds an instance of the class *AgentRunner*
    - The method *start* stores an instance of the AgentRunner in *runner*.
    - The method *send_event* calls the method *process_event* of *runner*.
- Defines the class *ProcessAgentBackend* as a child of *AgentBackend*
    - This is a backend that is realized by a separate process, which is fast and save.
    - At initialization, it initializes the argument:

        - *wta_queue* => Instance of the *Queue* class from the *multiprocessing* module that holds the callback events to be processed by the underlying agent code folder
            - **NB**: WTA could stand for "Work To be Accomplished".
        - *process* => Instance of the *Process* class from the *multiprocessing* module
            - **NB**: The process is initialized with the standalone method *run_in_agent_runner* as target, that takes the result_queue as atw_queue and initializes the *AgentRunner*.
            - **NB2**: ATW could stand for "Accomplished Work"
    - The method *start* starts the process.
    - The method *send_event* puts the callback event into *wta_queue*

environment.py

- Defines class *Trophy*
    - *coin_trophy* => Trophy for every collected coin
    - *suicide_trophy* => Trophy if the agent is blown up by its own bomb
    - *time_trophy* => Trophy for every time the agents exceed the thinking time
    - **NB**: The method *evaluate_explosions* defines an additional implicit trophy for every agent that is killed
- Defines the class *GenericWorld*
    - At initialization, it calls the method *setup_logging* and initializes the arguments:
        - *name* => Internal name of the agent
        - *args* => All arguments passed down from main.py.
        - *colors* => List of the colours of the agents as defined in settings.py.
        - *round* => Index of current round
        - *round_statistics* => Statistics of every round
        - *running* => Whether a round is currently running (and not finished or in between rounds)
    - The method *setup_logging* creates a logger and stores it in the *logger* argument.
    - The method *new_round* starts a new round.
        - It calls *build_arena*
        - It calls *start_round* for every *active_agent*
        - It initializes/resets the following arguments:
            - *step* => The number of steps for the current round
            - *bombs* => List of the bombs on the field
            - *explosions* => List of explosions on the field
            - *round_id* => The id of the current round (includes name and datetime)
            - *arena* => Matrix representing the arena [1 = wall, 0 = free, 1 = crate]
            - *coins* => List of coins on the field
            - *active_agents* => List of agents on the field
            - *agents* => List of all agents
            - *replay* => Object holding the initial state, all actions of all agents and all permutations [= order in which the agents can act for each step]

        - **NB**: The method is called at the start of every round by main.py.
    - The method *add_agent* adds a *SequentialAgentBackend* for the agent and creates the agent.
        - **NB:** *ProcessAgentBackend* is commented out.
        - **NB2**: This method is only used by the child class *BombeRLeWorld*
    - The method *tile_is_free* checks if a tile is free (so no crate, wall, bomb or agent)
        - **NB**: A tile with a coin is considered free.
    - The method *perform_agent_action* performs a single action for a specific agent
        - NB: If the tile is not free for the respective action or if the agent performs the action *BOMB* without having any bombs left, the action is *INVALID_ACTION*
    - The method *do_step* performs a step by calling:
        - *poll_and_run_agents*
        - *collect_coins*
        - *update_explosions*
        - *update_bombs*
        - *evaluate_explosions*
        - *send_game_events* [used by the child class *BombeRLeWorld*]
        - **NB**: If *time_to_stop* returns true, it also calls *end_round*
        - **NB2:** Thus, within a step, the agents first move and then the items are updated
    - The method *collect_coins* checks if any agent is on top of a coin and then collects it
    - The method *update_explosions* reduces the timer of every explosion by 1 and removes them if the timer is zero
    - The method *update_bombs* lets those bombs explode that have a timer of zero, for the other bombs it reduces the respective timer by 1
        - **NB**: When a bomb explodes and another bomb is within its radius of explosion, the other bomb does NOT explode
        - **NB2**: The explosion of a bomb goes through crates, i.e. it will not stop at the destroyed crate)
    - The method *evaluate_explosions* checks for every dangerous explosion if any agent is hit by the explosion. If so, it stores the statistics and removes the hit agents.
    - The method *end_round* stops the round (by setting running to *False*), stores the statistics and sends them to the agents.
        - **NB**: This method is used and expanded by the child class *BombeRLeWorld*
    - The method *time_to_stop* returns *True* if no agents are alive OR only one agent is alive and there is nothing to do OR if there were training agents, but no one is left OR if the maximum number of steps is reached (as defined in settings.py)
    - The method *end* calls *end_round* if a round is still running. It also stores the statistics to a file under *./results*.
- Defines the class *BombeRLeWorld* as a child of *GenericWorld*
    - At initialization, it initializes *rng* as a default random generator (for numbers of permutations) and it calls *setup_agents* on the received agents (from main.py)
    - The method *setup_agents* goes through the received agents and calls *add_agent* for every entry (which fills the argument *agents*

    - The method *build_arena* generates the arena by selecting the scenario that was defined in *args*, by randomly adding crates and coins [in crates if possible], by adding the walls and by setting & distributing the start position of the agents
    - The method *get_state_for_agent* returns the state for a specific non-dead agent. This includes:
        - *round* => Index of current round
        - *step* => The number of steps for the current round
        - field => Flattened arena matrix [1 = wall, 0 = free, 1 = crate]
        - *self* => State of the agent
        - *others* => All states of the other agents
        - *bombs* => State of all bombs
        - *coins* => State of all coins
        - *user_input* => Input of the user
        - *explosion_map* => Matrix containing the explosions
    - The method *poll_and_run_agents* passes to every agent is current own game state and tells them to act (if they have thinking time). Then, it goes in a random order [for fairness] through the agents, and does one of the following things:
        - If the agent has no thinking time (because of previous exceedings), it is skipped, and the thinking time is reset to the base timeout.
        - Otherwise, get the chosen action from the agent and perform it.
            - **NB**: If the agent exceeds the allowed thinking time, then the thinking of the next action is appropriately reduced.
    - The method *send_game_events* sends to every agent its new own game state and tells them to learn from the consequences of its action by calling *process_game_events* and *wait_for_game_event_processing*
    - The method *end_round* calls the method *end_round* of its parent class *GenericWorld*, but it also adds for every agent the *SURIVIVED_ROUND* event and it calls the method *round_ended* on every trained agent. Besides, it saves the replay under *./replays* if necessary.
    - The method *end* calls the method *end* of its parent class *GenericWorld*, but it also logs additional messages.
- Defines the class *GUI*.
    - Renders the GUI of the screen and also generates the video with the method *make_video* if necessary.

fallbacks.py

- Avoids that the game crashes if *pygame* or *tqdm* are not available.
    - This allows to handle the errors ourselves.

test.py

- Tests the game by playing 1 round without any GUI.
- Asserts that the game creates the log *game.log*.

replay.py

- Defines the class *ReplayWorld* as a child of *GenericWorld*
    - At initialization, it reads the received replay file and creates the agents as objects of the class *ReplayAgent*.

    - The method *build_arena* reads the arena [1 = wall, 0 = free, 1 = crate] from the replay and places coins & agents. The results are returned.
    - The method *poll_and_run_agents* performs the next recorded step for all agents
    - The method *time_to_stop* returns true if the last step is reached.
- Defines the class *ReplayAgent* as a child of *Agent*.
    - Simulates the recorded agent.

agent_code/fail_agent

- Defines *callbacks.py*.
    - The method *setup* only defines the random seed.
    - The method *act* throws an error.

agent_code/peaceful_agent

- Defines *callbacks.py*.
    - The method *setup* only defines the random seed.
    - The method *act* chooses a random action within RIGHT, LEFT, UP, DOWN
        - **NB**: This agent does not place bombs.

agent_code/random_agent

- Defines *callbacks.py*.
    - The method *setup* only defines the random seed.
    - The method *act* chooses a random action within RIGHT, LEFT, UP, DOWN, BOMB
        - **NB**: This agent does place bombs, but only with a 8% probability

agent_code/user_agent

- Defines *callbacks.py*.
    - The method *setup* is empty.
    - The method *act* returns the user input in the received own game state.

agent_code/coin_collector_agent.py

- Defines *callbacks.py*.
    - The method *look_for_targets* performs BFS on the reachable free tiles until a target [= coins & crates & dead_ends {formed by walls & crates}] is encountered. It then returns the next best tile to move to the closest target.
        - **IMPORTANT**: Actually, coins and dead ends are always prioritized over crates because you can never find a path exactly on the crate since the crate itself is not free.
            - **As a result, the agent only places bombes in dead ends (if reachable)**
            - **NB**: If no target can be directly reached (i.e. no reachable coins or dead ends), then the direction with the shortest grid path to the closest non-reachable targets is chosen.
        - **NB2**: This method is identical to the one for *rule_based_agent*
        - The following arguments are used:
            - *frontier* => Queue holding the next tiles for BFS
            - *parent_dict* => Dictionary for pointing at the parent tiles

- *dist_so_far* => Dictionary holding the distance of the scanned tiles to the start
- *best* => The tile where we want to finally end up in (often several steps away, e.g. the closest reachable target)
- *best_dist* => The distance to the closest target (reachable or not)
    - **NB**: Thus *best_dist* is a hard lower bound and can actually never be reduced during the algorithm, we can only find equal distances.
    - **NB2**: It seems that the algorithm is slightly unintuitive when no target is reachable and the clostest non-reachable target has the same *x* or *y* position as the start point. Because then *best_dist* is the direct x or y distance to the closest non-reachable target, which may be impossible to pursue if there are walls/obstacles in-between. Thus, if there are walls/obstacles in-between, *best_dist* would be impossibly low and since all targets are non-reachable [otherwise the result is clearly the closest reachable target], the result will be start. This seems to be the correct result, because even if there is a path around the walls/obstacles that will lead us closer to the non-reachable target, the path is overall longer so the algorithm just waits at the start until the wall/obstacle disappears, but this is slightly unintuitive.

- The method *setup* only defines the random seed.
- The method *act* does the following things:
    - First, the information is extracted from the current own game state.
    - Then, the valid possible next actions amongst LEFT, RIGHT, UP, DOWN, BOMB are determined and inserted in *valid_actions*
        - **NB**: bomb_map is a matrix with the same shape as arena where the bombs but also their radius is marked [5 = no bomb (radius), 1-4 = bomb (radius with timer), 0 = bomb (radius) that will explode in this step]
        - **NB2**: The rules BOMB_TIMER and BOMB_POWER are not used here, instead the values are hardcoded.
    - The list *action_ideas* contain a basic, shuffled set of possible next actions, but better actions are appended later. The latest added action is finally chosen.
        - NB: Actions can appear multiple times in the list.
    - Possible targets [= coins & crates & dead_ends {formed by walls & crates}] are defined
    - With the use of the method *look_for_targets*, the best (free) direction to come closer to the nearest target is found.
        - **IMPORTANT**: Actually, coins and dead ends are always prioritized over crates because you can never find a path exactly on the crate since the crate itself is not free.
            - **As a result, the agent only places bombes in dead ends (if reachable)**

        - **NB**: If no target can be directly reached (i.e. no reachable coins or dead ends), then the direction with the shortest grid path to the closest non-reachable targets is chosen.
    - The action BOMB is added to *action_ideas* if the agent is in a dead end or if he is already in front of a crate.
    - An action amongst UP, DOWN, LEFT, RIGHT is added to *action_ideas* if the agent must run away from a bomb.
    - The latest added action in *action_ideas* is taken, but only if it is also valid, else the second latest added action is taken and so on …

agent_code/rule_based_agent

- Defines *callbacks.py*.
    - The method *look_for_targets* performs BFS on the reachable free tiles until a target [= coins & crates & dead_ends {formed by walls & crates}] is encountered
        - **IMPORTANT**: Actually, coins and dead ends are always prioritized over crates because you can never find a path exactly on the crate since the crate itself is not free.
            - **As a result, the agent only places bombes in dead ends (if reachable)**
            - **NB**: If no target can be directly reached (i.e. no reachable coins or dead ends), then the direction with the shortest grid path to the closest non-reachable targets is chosen.
        - **NB2**: This method is identical to the one for *coin_collector_agent*
        - The following arguments are used:
            - *frontier* => Queue holding the next tiles for BFS
            - *parent_dict* => Dictionary for pointing at the parent tiles
            - *dist_so_far* => Dictionary holding the distance of the scanned tiles to the start
            - *best* => The tile where we want to finally end up in (often several steps away, e.g. the closest reachable target)
            - *best_dist* => The distance to the closest target (reachable or not)
                - **NB**: Thus *best_dist* is a hard lower bound and can actually never be reduced during the algorithm, we can only find equal distances.
                - **NB2**: It seems that the algorithm is slightly unintuitive when no target is reachable and the clostest non-reachable target has the same *x* or *y* position as the start point. Because then *best_dist* is the direct x or y distance to the closest non-reachable target, which may be impossible to pursue if there are walls/obstacles in-between. Thus, if there are walls/obstacles in-between, *best_dist* would be impossibly low and since all targets are non-reachable [otherwise the result is clearly the closest reachable target], the result will be start. This seems to be the correct result, because even if there is a path around the walls/obstacles that will lead us closer to the non-reachable target, the path is overall longer so the

algorithm just waits at the start until the wall/obstacle disappears, but this is slightly unintuitive.

- The method *setup* only defines the random seed and the following arguments:
  - *bomb_history* => List of length 5 containing the past bomb positions
  - *coordinate_history* => List of length 20 containing the past coordinates
  - *ignore_others_timer* => Timer that during which, if positive, enemies will be ignored (i.e. they will not be considered as targets during BFS)
  - *current_round* => Index of current round
- The method *reset_self* resets all arguments (except *current_round*)
- The method *act* does the following things:
  - First, the information is extracted from the current own game state.
  - If the agent has been 3 times at the same location during the last 20 steps, he will ignore enemies during the next 5 steps (*ignore_others_timer* will be set to 5)
  - Then, the valid possible next actions amongst LEFT, RIGHT, UP, DOWN, BOMB are determined and inserted in *valid_actions*
    - **NB**: bomb_map is a matrix with the same shape as arena where the bombs but also their radius is marked [5 = no bomb (radius), 1-4 = bomb (radius with timer), 0 = bomb (radius) that will explode in this step]
    - **NB2**: The rules BOMB_TIMER and BOMB_POWER are not used here, instead the values are hardcoded.
    - **NB3**: BOMB is only a valid action if it has not been placed at the same location during the last 5 steps.
  - The list *action_ideas* contain a basic, shuffled set of possible next actions, but better actions are appended later. The latest added action is finally chosen.
    - NB: Actions can appear multiple times in the list.
  - Possible targets [= enemies & coins & crates & dead_ends {formed by walls & crates}] are defined
    - **NB**: Enemies are only possible targets if the enemies are not ignored OR if there are no coins and no crates left.
  - With the use of the method *look_for_targets*, the best (free) direction to come closer to the nearest target is found.
    - **IMPORTANT**: Actually, coins and dead ends are always prioritized over crates because you can never find a path exactly on the crate since the crate itself is not free.
      - **As a result, the agent only places bombes in dead ends (if reachable)**
      - **NB**: If no target can be directly reached (i.e. no reachable coins or dead ends), then the direction with the shortest grid path to the closest non-reachable targets is chosen.
    - **NB2**: Enemies are only considered as walls/obstacles if they are currently ignored.
  - The action BOMB is added to *action_ideas* if the agent is in a dead end OR if he is already in front of a crate/enemy.

- **NB**: Neighbouring enemies are always bombed, even if they are currently ignored by the timer.
- An action amongst UP, DOWN, LEFT, RIGHT is added to *action_ideas* if the agent must run away from a bomb.
- The latest added action in *action_ideas* is taken, but only if it is also valid, else the second latest added action is taken and so on …

agent_code/tpl_agent

- This is a template structure for the own agent.
- Defines *callbacks.py*.
  - Defines the possible actions in ACTIONS.
  - The method *setup* initializes the argument model that consists of a dummy list of normalized weights for the different possible actions OR of a previously stored model.
  - The method *act* is called by act which is triggered by *poll_and_run_agents* at the very start of a step. The agent should here take an action in the non-training and the training mode.
    - **NB**: When not in training mode, the maximum execution time for this method is 0.5 seconds.
    - **NB2**: When in training mode, a good balance between exploration and exploitation must be found.
  - The method *state_to_features* is a suggestion to convert the received *game_state* to the feature vector and return it.
- Defines *train.py*.
  - Defines the named tuple Transition that holds the old state, the action, the next state and the received reward.
    - **NB**: The constant *TRANSITION_HISTORY_SIZE* defines how many past transitions are stored.
    - **NB2**: The constant *RECORD_ENEMY_TRANSITIONS* defines the probability with which the transitions of enemies are kept, but this is not implemented yet.
  - The method *setup_training* initializes the argument *transitions*. The purpose of this method is to initialize all arguments for the training. Note that the method is called after the *setup* method in *callback.py*.
  - The method *game_events_occured* is called by *process_game_events* which is triggered by *send_game_events* at the very end of a step. The received argument *events* will contain a list of all game events relevant to the agent that occurred during this step [also referenced to as “previous” step]. Based on the events, the reward is computed with *reward_from_events* and a *Transition* is appended to *transitions*
    - **NB**: An idea is to append own events such like *PLACEHOLDER_EVENT* based on the old & new game states before calling *reward_from_events*
  - The method end_of_round is called by *round_ended* which is triggered by *end_round* at the very end of every game or when the agent died. The received argument *events* will contain a list of all game events relevant to the agent that occurred during the final step. Based on the events, the reward is computed with *reward_from_events* and a *Transition* is appended to *transitions*

        - **NB**: An idea is to append own events such like *PLACEHOLDER_EVENT* based on the old & new game states before calling *reward_from_events*
        - **NB2:** In this method, the argument *model* is also stored in a file.
    - The method *reward_from_events* is a suggestion to compute the reward based on the received argument *events*.