Feature Design Task 2:

Goal: Learn to collect all coins by blowing up crates safely
- maximise coin-collecting speed
- minimise deadly accidents

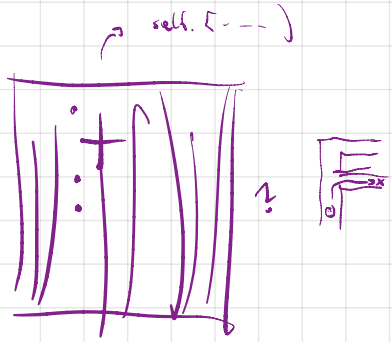
Approach: Develop features that
- make it as easy as possible for the agent to learn good behaviour.
- make it able to also handle complex/edge cases well.

dangerous / safe zones

How to show info about ↗ to agent in time and space?

What is required for good behaviour?
- If a bomb is placed, always move to safety in time → more precisely: if a bomb is in proximity of agent (time/space-wise) move away to safe zone
- Idea: place bombs where the expected number of destroyed crates is highest or such that the crate destruction speed is maximised.
- Idea: Have a high priority on collecting accessible coins

Key tasks:
1. collect coins
2. when no coins reacheable, place bomb where many crates are destroyed.
3. when bomb placed, move to safety (and stay away from explosion while it lasts)



Idea: Learn

Pseudo-algorithm:

```
if not in_safety:                    not_in_safety = within bomb explosion radius
    move to safety
else if no_reachable_coins:          no_reachable_coins = no visible coins or coins out of reach (path blocked by crates)
    place bomb at good_spot
else:                                Idea: good_spot = det. algorithmically
    collect coins as in Task 1       at what spot a bomb placed results in the highest coin collecting speed
```

factor in: time to go to good_spot, time to place bomb and run to safety, time to return to new paths, coin density in crates, multi-bomb strategies?

expected coin speed = coin_density × crates_destroyed / total_time

coin_density = remaining_hidden_coins / remaining_crates

total_time → time_to_reach_good_spot + bomb_cooldown + expected_time_to_reach_coin

↳ maybe approx. is good enough?

Advantages:
- feature architecture with "recommended direction" can be used again, with additional feature „mode" and action „bomb" and „wait" enabled → do these actions require new features?

  3 modes, 6 actions ⇒ Q-matrix $\in \mathbb{R}^{48\times 6}$
- modes maybe individually trainable and probably individually analyze, debug and incentivise
- provided learning of task 1 has become ✓ good and „recommender algorithms" are good, this may actually be very easy to learn.

Disadvantages
- strategy heavily dependant on quality of algorithms
- may prevent agent from employing simultaneous or hybrid strategies *
- (quite like 2019 winner approach)
- learning not really necessary here

*Possible advanced strategies

- Ignore/Deprioritize coins that can be more easily reached by opponents
- place one bomb. Then move in a direction to safety where you (after the first one is gone) can place another bomb. Move to safety exploiting the blast of the first bomb while the second one explodes.
- When going after an accessible coin, place bombs along the way to find more ones
- when a coin is inaccessible behind a few crate, place a bomb specifically to get to that coin
- Further strategies likely exist.
- Hybrid strategies where opponents are present may become even more important.

other ideas:

- Have the features of multiple modes separate from each other
  + potentially allows for learned prioritizing between modes

  ? how to make/incentivize/inform agent when to prioritize one set of features over another?
  - big Q matrix (maybe unnecessary)
  - seems not to increase likelihood of advanced strategies
    ↳ why not?

- Learn when to switch between modes
  + let the low level stuff be handled by algorithms
  + focus „thinking"/learning on deciding between high level strategies

  ? How to do that?

  idea for info categories / naming convention

  low level: info directly from game-state (or almost directly)
  medium level: info from algos such as look_for_targets
  high level: info about strategies and plans

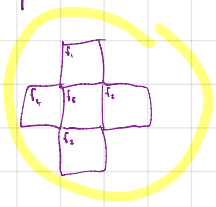
  Idea: Provide medium level infos such that agent knows just the things relevant to high level decisions.

  eg.
  - where never to go: into walls, crates, bombs, agents
  - where to go for coins (direction?, proximity?, what about multiple ones?)
  - where to go to evade danger (direction?, threat level?)
  - where the next best place to place bomb and destroy crate (direction?, gain?)

  How to decide between strategies?
  ↳ maybe learn from longterm experience

$$k_c = s(4,3) = 15$$
$$k_m = s(4,5) \cdot s(1,3) = 210 \text{ at most}$$
k_h

Ausprobieren:



$s^+(5,5)$

f_{1-4}:
- 0: blockiert (nicht passabel oder tödliche Gefahr) → nicht dort hin
- 1: frei aber nicht besonders interessant
- 2: richtung für nächsten coin
- 3: richtung für beste bomb-stelle (good_spot) } ferne entscheidung zwischen diesen alternativen
- 4: sowohl coin als auch bomb → dort hin

f_5:
- 0: auf bombe bzw. tot wenn gelegt → not wait
- 1: uninteressant
- 2: auf coin (nicht good_spot) → wait — kommt sehr selten, unwichtig, da andere policies fast genauso gute Ergebnisse liefern.
- 3: auf good_spot (coin egal) → bomb

Benötigt folgende Algorithmen:

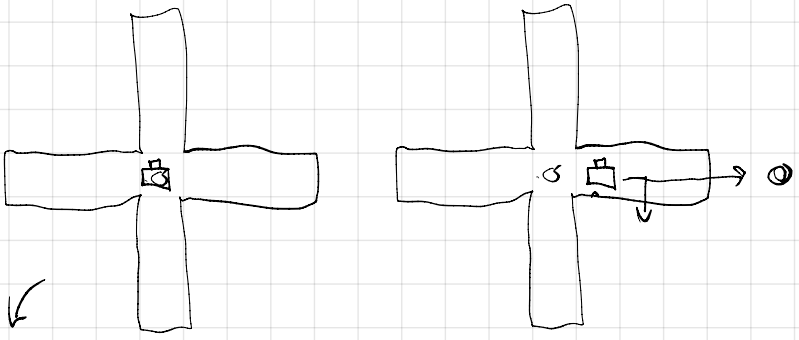
- direction_nearest_coins()
- direction_good_spot()
- direction_lethal_danger()

Outputs: (alle) [self_position, *neighbor_positions]

Herausforderungen:

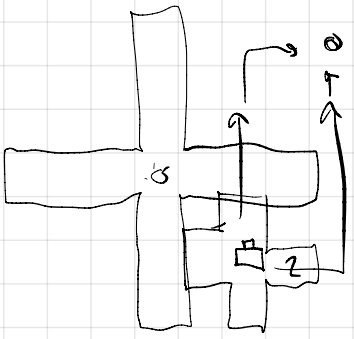
wie priorisieren zwischen nearest_coin und good_spot?

→ kann lernen, aber mit gegebener info nur gemacht was besser ist (bevorzugt aber langfristiges denken, spricht gut nach, bzw. n-step ganz)



$f_5 = 0$

$f_{1-4} = 0$



Basic feature architecture:

state_to_features() Input: game_state Output: [f1, f2, f3, f4, f5]

coins = [(x,y)] ← find_nearest_coins (...) ← would have to be tweaked

spots = [(x,y)] ← find_crate_bombing_spots ()

danger = [(x,y)] ← find_danger () ?

- directions (coins)
- directions (spots)
- directions (danger)

→ make into features

Algorithm design: find_crate_bombing_spots()

Input: „game_state"

Output: [(x, y)] (multiple coordinates possible)

Aim: Solve at what spot a bomb placed results in the highest coin speed.

Task: Calculate the best spot(s) to place the next bomb, such that $\frac{\text{expected ~~collected~~ revealed maybe? coins}}{\text{total time}}$ is maximised

[(x, y)]

After, and if, coins are revealed, the coin and crate algorithms can decide on further action.

Assumptions:
- Only value of bombing crates is coins

Potential issues:
- What about the safety aspect of bombing crates? Isn't there risk of trapping oneself between walls/crates and bombs?
- May be blind to currently non-reachable coins.

Mathematically:

return $\arg\max_{\vec{s}} \left(V_c(\vec{s}, \vec{p}, F) \right)$

$V_c(\vec{s}, \vec{p}, F) = \frac{c(\vec{s}, F)}{t_{tot}(\vec{s}, \vec{p}, F)}$

$c(\vec{s}, F) = k_d(\vec{s}, F) \cdot \rho_c(F)$

$\rho_c = c_h / k(F)$

$t_{tot}(\vec{s}, \vec{p}, F) = t_r(\vec{s}, \vec{p}, F) + t_b$

- $\vec{s} = (x, y)$ — bomb placing spot
- $\vec{p} = (x, y)$ — own current position
- F = game_state['field']
- $V_c(\vec{s})$ — expected coin collection speed
- $c(\vec{s})$ — expected number of coins revealed by bomb
- t_{tot} — expected time until next bomb can be placed

→ good measures?

- $k_d(\vec{s})$ — crates destroyed by bomb at spot $\vec{s}$
- ρ_c — density of remaining hidden coins
- c_h — number of remaining hidden coins
- k — number of remaining crates
- $t_r(\vec{s}, \vec{p})$ — time to reach spot $\vec{s}$ from $\vec{p}$
- t_b — time until next bomb can be placed

Ideas for implementation:

$t_r(\vec{s}, \vec{p}, F)$ could be a map $\in \mathbb{N}^{B \times B}$ (board length = 17 ?) based on own position $\vec{p}$
calculated with a breadth-first search
Non reachable values could be NaN or None

$t_b = 7$ empirical value

$k(F)$ count all 1's in game_state['field']

c_h set up some coin counter that tracks appearing and disappearing coins

$k_d(\vec{s})$ map bomb explosion radius over F
(maybe update only if F changes?)