POLICY OPTIMISATION

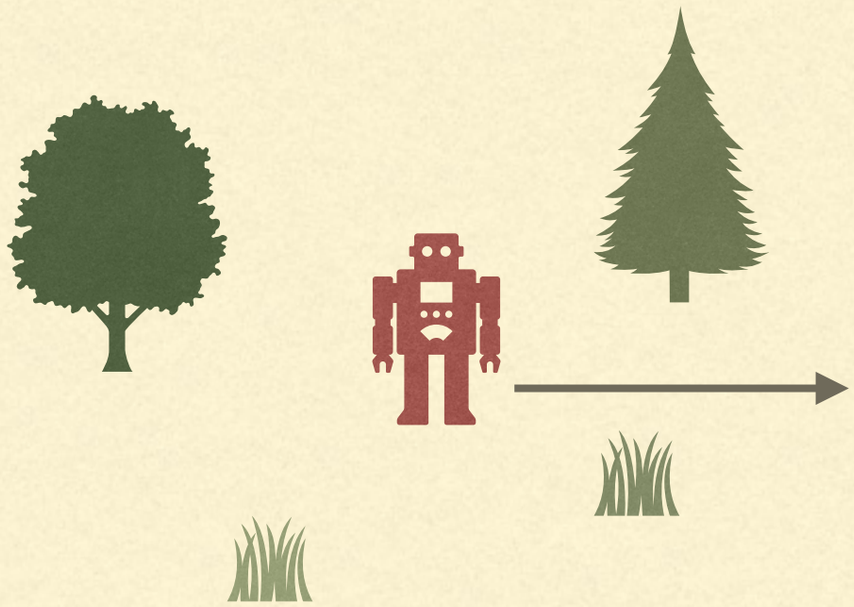
Luca Manzoni

THE SETTING

Suppose that you are an agent in an environment

You perceive the environment

And you have to take an action



How can you learn which action is the best in a given situation?

TERMINOLOGY

- A fixed set A of possible actions
- A fixed set S of states
- From a state $s \in S$, performing the action $a \in A$ we ends up in state $s' \in S$ with a fixed probability $P(s' \mid s, a)$
- For doing action $a \in A$ in a state $s \in S$ we receive a reward $R(s, a)$ either deterministically or with a fixed probability
- The environment is Markovian (i.e., without any memory). Thus, transition probabilities are entirely based on the current state and action

TERMINOLOGY

- Due to these assumptions we **do not** need to store the entire "history" of the actions and states

- We only need a function giving, for the current state $s \in S$, the action to perform

- $\pi\colon S \to A$ is a **policy**. Given the current state $s \in S$, then $\pi(s) \in A$ is the action that we will perform

- We are interested in the optimal policy π^*, which maximises the expected reward during the agent's lifetime

TYPES OF POLICY OPTIMISATION

- Dense policy optimisation
  - Q-learning (not an evolutionary technique)
- Sparse stochastic policy optimisation and Learning Classifier Systems (LCS)
  - Pitt approach
  - Michigan approach

Q-LEARNING

- In Q-learning instead of saving a policy we save a **Q-table**

- $Q(s, a)$ for each $s \in S$ and $a \in A$ gives the **Q-value** of a pair of state and action

- An optimal policy can then be obtained as
  $\pi^*(s) = \text{argmax}_{a \in A} Q^*(s, a)$
  where Q^* is the optimal Q-table

THE Q-TABLE

Q-value

States

Actions

	STRAIGHT ROAD	BOULDER ON THE ROAD
MOVE FORWARD	0.92	0.01
TURN LEFT	0.23	0.47
TURN RIGHT	0.14	0.52

Q-LEARNING

- You can consider the value in $Q^*(s, a)$ as "what is the utility of performing action a in state s and then following an optimal policy?"

- Thus, if the rewards and transition probabilities are known, we can compute $Q^*(s, a)$ as:
$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) \max_{a' \in A} Q^*(s', a')$$
where γ is a discount factor

Q-LEARNING WITH A MODEL

- If transition probabilities and rewards are known we have a model and we can compute Q^* starting from an initial Q-table by iteratively updating it:

- Let Q' be a copy of our current guess Q^*

- Update each entry of our guess for Q^* as
$$Q^*(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) \max_{a' \in A} Q'(s',a')$$

- Repeat until the changes to Q^* are small enough

MODEL-FREE Q-LEARNING

- Q-learning with a model has no local optima
- Unfortunately we usually do not have a model:
  - The rewards for a state are unknown
  - The transition probabilities are unknown
- The agent must discover the probabilities and rewards by trying to perform the actions

MODEL-FREE Q-LEARNING

- We start with $Q(s, a) = 0$ for all pairs
- Suppose that we are in state s and we perform action a, ending up in the state s' and obtaining reward r
- Then we update our Q-table as:
$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a' \in A} Q(s', a'))$$
- This combines old information (old value of $Q(s, a)$) with newly acquired information (the reward r and the state s')

THE PROBLEMS OF Q-LEARNING

- In its traditional form, Q-learning has multiple drawbacks
  - It is purely exploitative and the problem has local optima (more exploitative variants, e.g., with ε-greedy action selection, exist)
  - It does not generalise well — or at all. We need an entry for each pair (s, a), we do not generalise to unseen states
- Those problems are tackled by **sparse** policy optimisation techniques

SPARSE POLICY OPTIMISATION

- The main idea is to aggregate multiple states and provide a specific action for each collection of states

- This can be performed using other algorithm (e.g., neural networks) that learns the q-value (or the policy) also for unseen pairs of state and action (or action to states)

- An explicit way of representing this aggregation is via **rule systems** (sometimes called rulesets)

RULE SYSTEMS

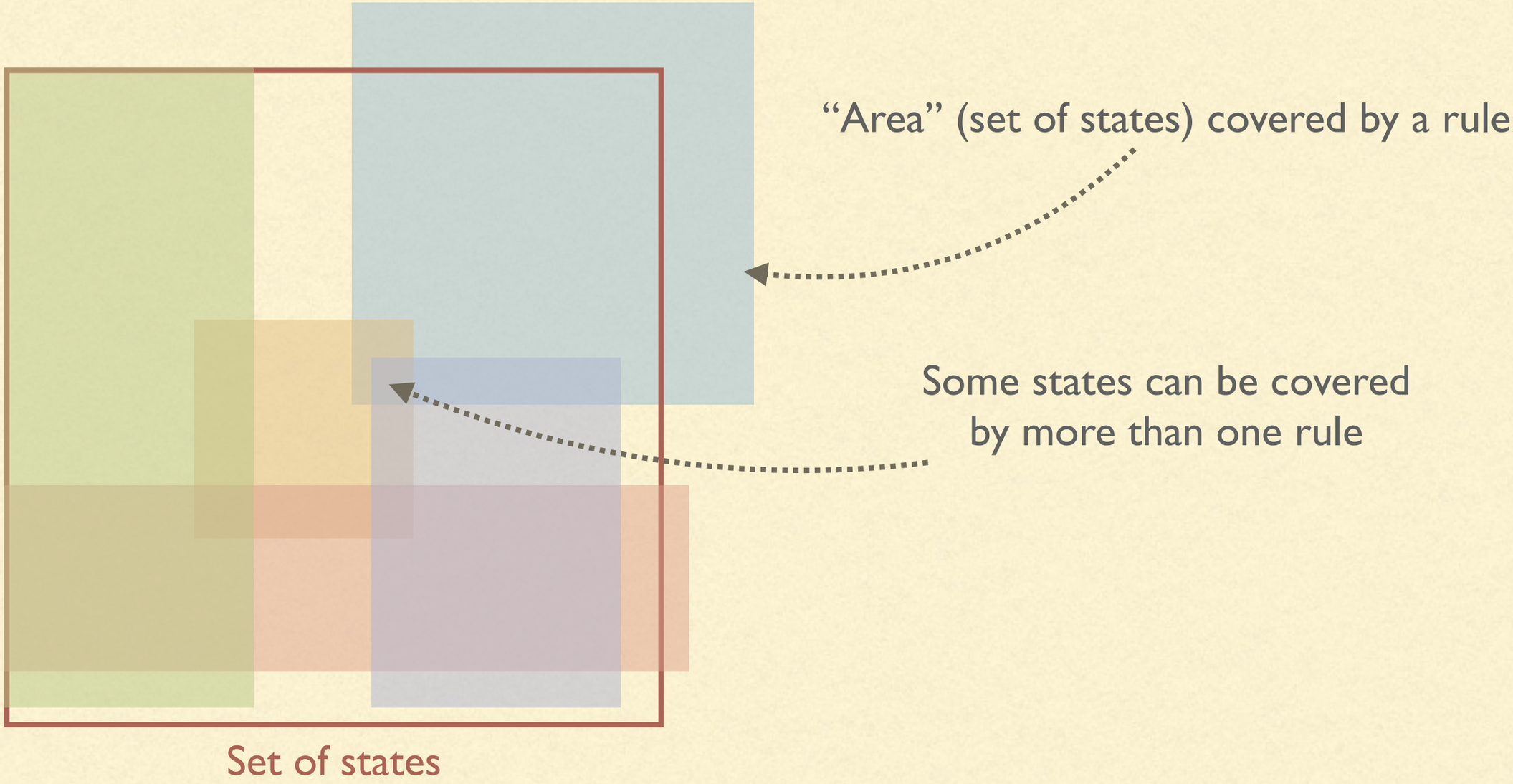
- A rule is a construction of the form:

  if *conditions* **then** *action*

- The first part is the **rule body**, while the second part is the **rule head**.

- A rule system will be a set of rules plus, possibly, an arbitration scheme to select which rule to apply when multiple rules can be applied

RULE SYSTEMS



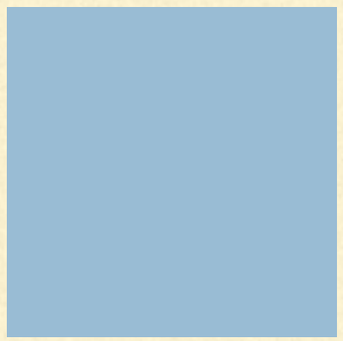
TRIGGERING RULES

- **Exact match**. All the conditions on the rule body must be satisfied
  - Risk of under-specification: a state might not be covered by any rule. Either a default rule or on-the-fly rule creation can be solutions
- **Imprecise match**. We can define a match score, stating how much of the conditions in the rule body are satisfied

RESOLVING CONFLICTS

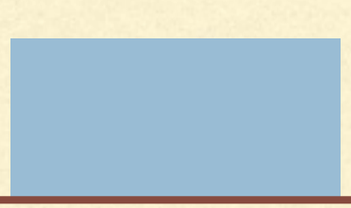
- Since some states might be covered by multiple rules, an arbitration scheme is needed. Usually it takes into account:
  - **Rule utility**. Essentially the Q-value.
  - **Variance of rule utility**. We want a small variance in the reward provided by executing a rule.
  - **Error in rule utility**. The difference between the rule utility and the utility of the rules it leads to.
  - For imprecise matches, the **match score**.

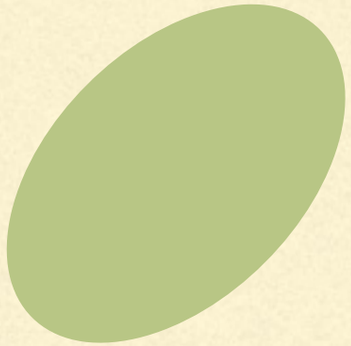
RULES IN REAL-VALUED SPACES



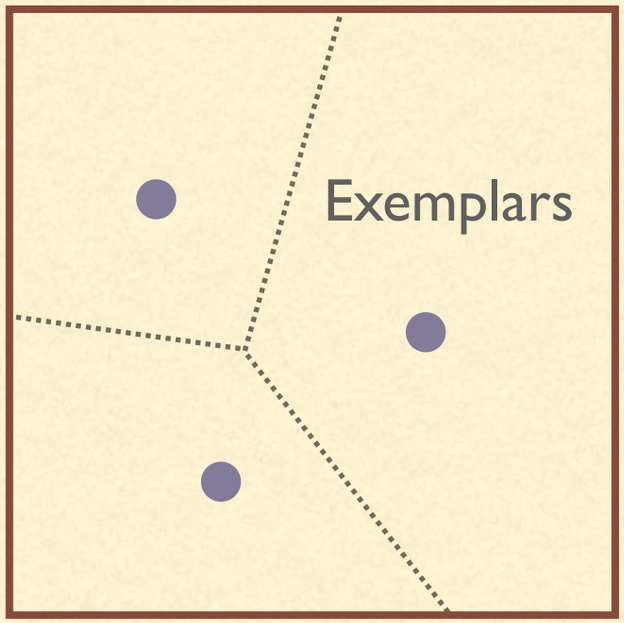
Boxes

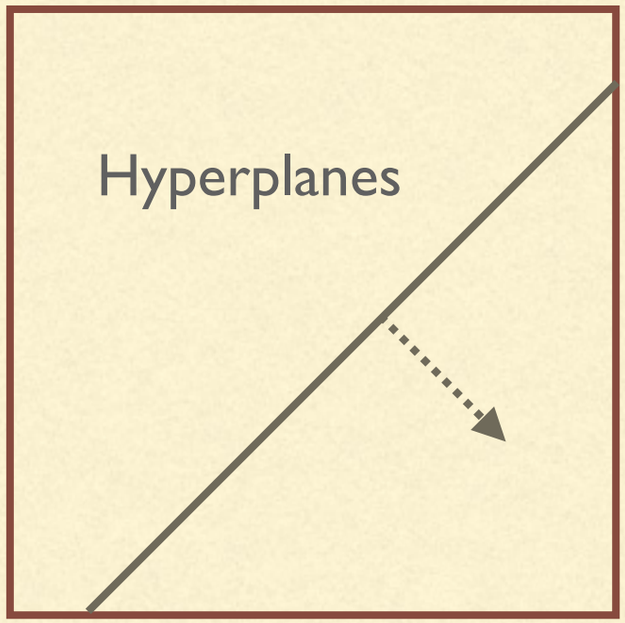
Toroidal Boxes





Hyperspheres or Hyperellipsoids



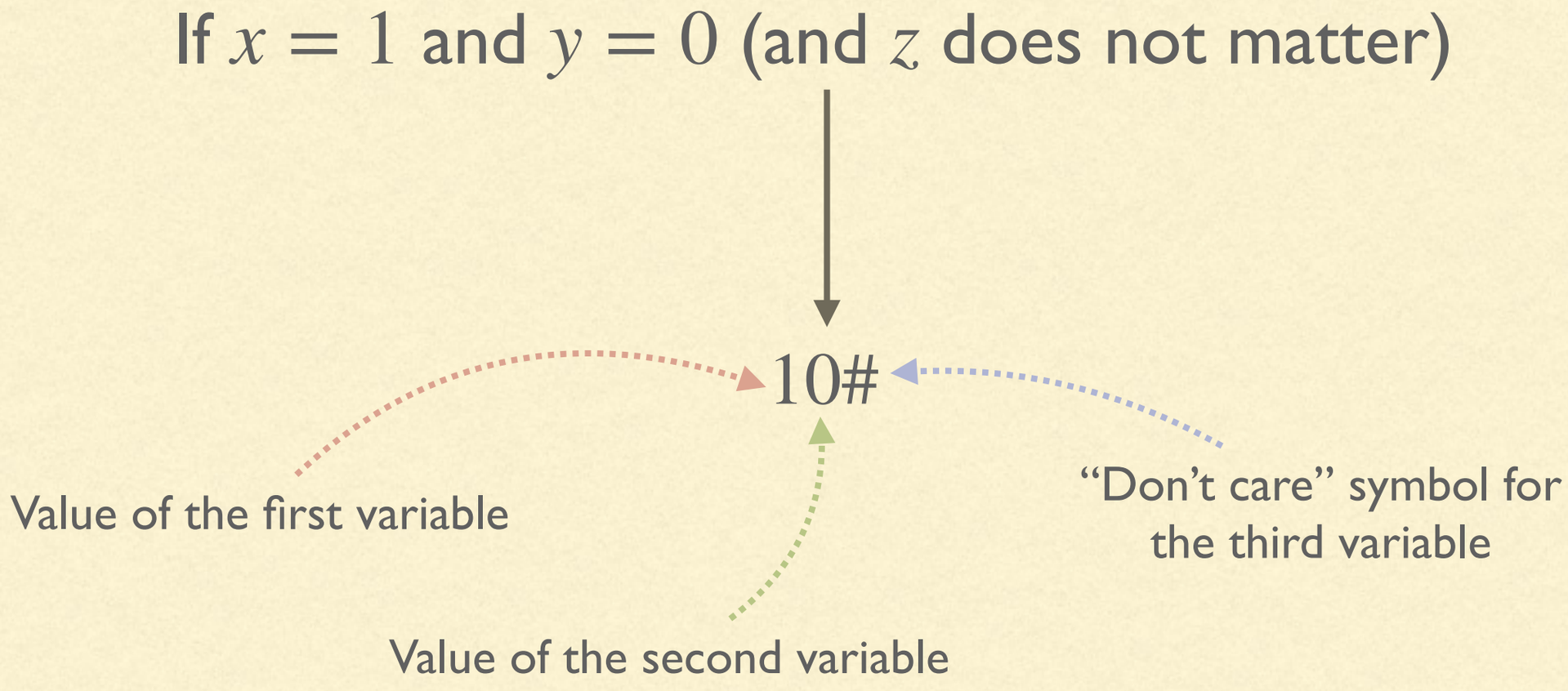


For each approach it is possible to define a match score

The most used method to model rule bodies is via boxes

RULES IN BOOLEAN SPACES

Representation for a three-dimensional Boolean space:

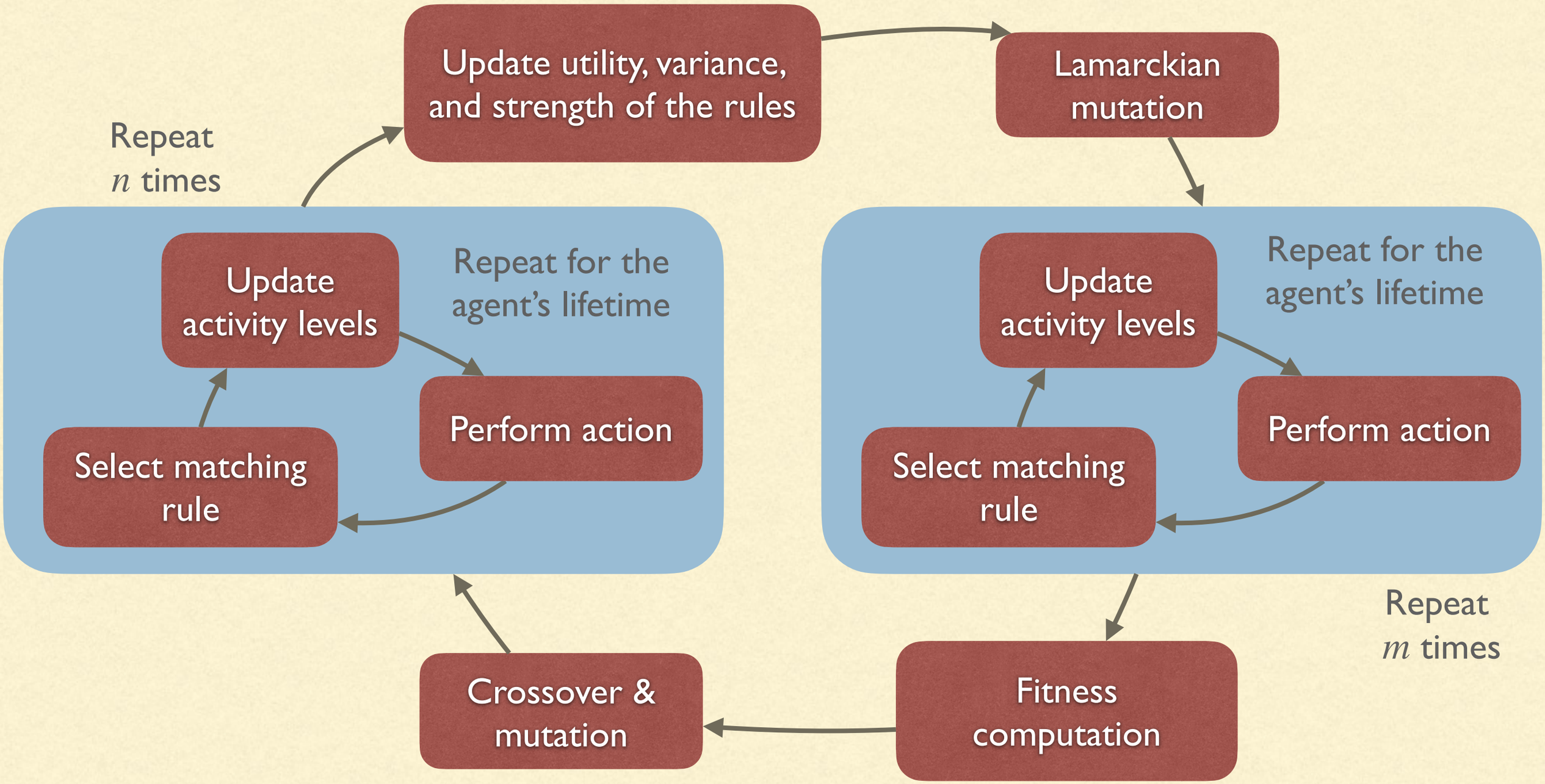


PITT APPROACH

- Each individual is an entire set of rules
- The population is a collection of sets of rule
- We will see a specific implementation in the SAMUEL (Strategy Acquisition Method Using Empirical Learning) system

SAMUEL CYCLE

Notice that there is a phase for "local" improvements before the evolutionary phase



FINDING THE RULE TO APPLY

- Finding the rule to apply is a two-steps process
- For each rule we compute the **match score** (hence an imprecise match is used)
  - A **match set** is selected as the best matching rule (a "truncated selection")
  - The actual rule to apply is selected via a probability proportional to the matching score

FITNESS AND UTILITY

- The fitness is simply defined as the sum of all rewards obtained by the agent
- The **utility** of a rule, instead, is similar to a Q-value, but we do not discount the reward when updating it: $\text{Utility}(R_i) \leftarrow (1-\alpha)\text{Utility}(R_i) + \alpha r$
- We also keep track of the utility variance, which quantify how "consistent" is the rule:
  $\text{UtilityVariance}(R_i) \leftarrow (1-\alpha)\text{UtilityVariance}(R_i) + \alpha(\text{Utility}(R_i) - r)^2$
- Finally, the "quality" of a rule is quantified by the **strength**:
  $\text{Strength}(R_i) \leftarrow \text{Utility}(R_i) + \gamma\text{UtilityVariance}(R_i)$
  where $0 \leq \gamma < 1$

ACTIVITY

- By keeping track of how frequently a rule was active it is possible to remove rules that are almost never active
- At the beginning of the agent's life $\text{Activity}(R_i) = \frac{1}{2}$
- Each time a rule is applied and action a is performed, the activity of all rules having action a in the head is updated as:
  $\text{Activity}(R_i) \leftarrow (1 - \beta)\text{Activity}(R_i) + \beta$
- For all rules without a in their heads the activity levels decay as:
  $\text{Activity}(R_i) \leftarrow \delta\text{Activity}(R_i)$

MUTATION

- SAMUEL has two mutation phases and mutation types:
  - During the assessment of the strength of a rule a **Lamarckian** mutation is performed.
    This kind of mutation is directly exploitative
  - During the normal evolutionary phase standard kinds of mutations are applied.

MUTATION (LAMARCKIAN)

- **Rule Deletion**. If a rule is old and has a low activity value it might be selected for deletion.

- **Rule Specialisation**. If a rule covers a large number of states additional restrictions might be applied. The original rule is also kept in the population.

- **Rule Generalisation**. If a rule is very specific some restrictions might be removed. The original rule is also kept in the population.

MUTATION (LAMARCKIAN)

- **Rule Covering**. Similar to generalisation. If during the evaluation a rule fires frequently but has a partial match, the part that is not matched can be removed. The original rule is also kept in the population.

- **Rule Merging**. If two rules have the same head, are strong enough, and overlap in the set of states that they cover, then they can be merged in a single rule. The original rules are also kept in the population.

MUTATION (CLASSICAL)

- **Standard mutation**. Change some parts of a rule, without keeping the parent in the population.

- **Creep mutation**. Perform small random changes to a rule, trying to mimic a local hill-climbing process.

RECOMBINATION

- **Uniform crossover**. Two individual exchange rules a certain number k of times

- **Clustered crossover**. During fitness evaluation we can keep track of which sequences of rules usually lead to a reward.
  - We identify pairs of rules that appears frequently in the same sequence
  - We perform uniform crossover but keeping the pairs together

SELECTION

- SAMUEL uses a combination of a truncated selection plus a standard selection procedure

- There is a baseline fitness that is kept updated as follow:
  $\text{baseline} \leftarrow (1-\nu)\text{baseline} + \nu(\mu_{\text{fitness}} + \psi\sigma^2_{\text{fitness}})$
  that is, we update the baseline with the current mean and variance (with weight ψ) of the fitness

- Only the individuals that have fitness above the baseline are considered for the standard selection procedure

INITIALISATION

- Random set of rules
- Domain-specific set of rules that might be useful in speeding-up the evolution
- Each individual starts with only rules of the form:
  all states of $S \rightarrow a$
  for each action a. In this case the rules cover the entire set of states and will be specialised by the evolution process

MICHIGAN APPROACH

- Each individual is a single rule
- The solution is given by the entire population
- Similar to single-population a cooperative coevolution approach
- We will see two algorithms:
  - Zeroth Level Classifier System (ZCS)
  - XCS (which apparently does not stand for anything in particular)

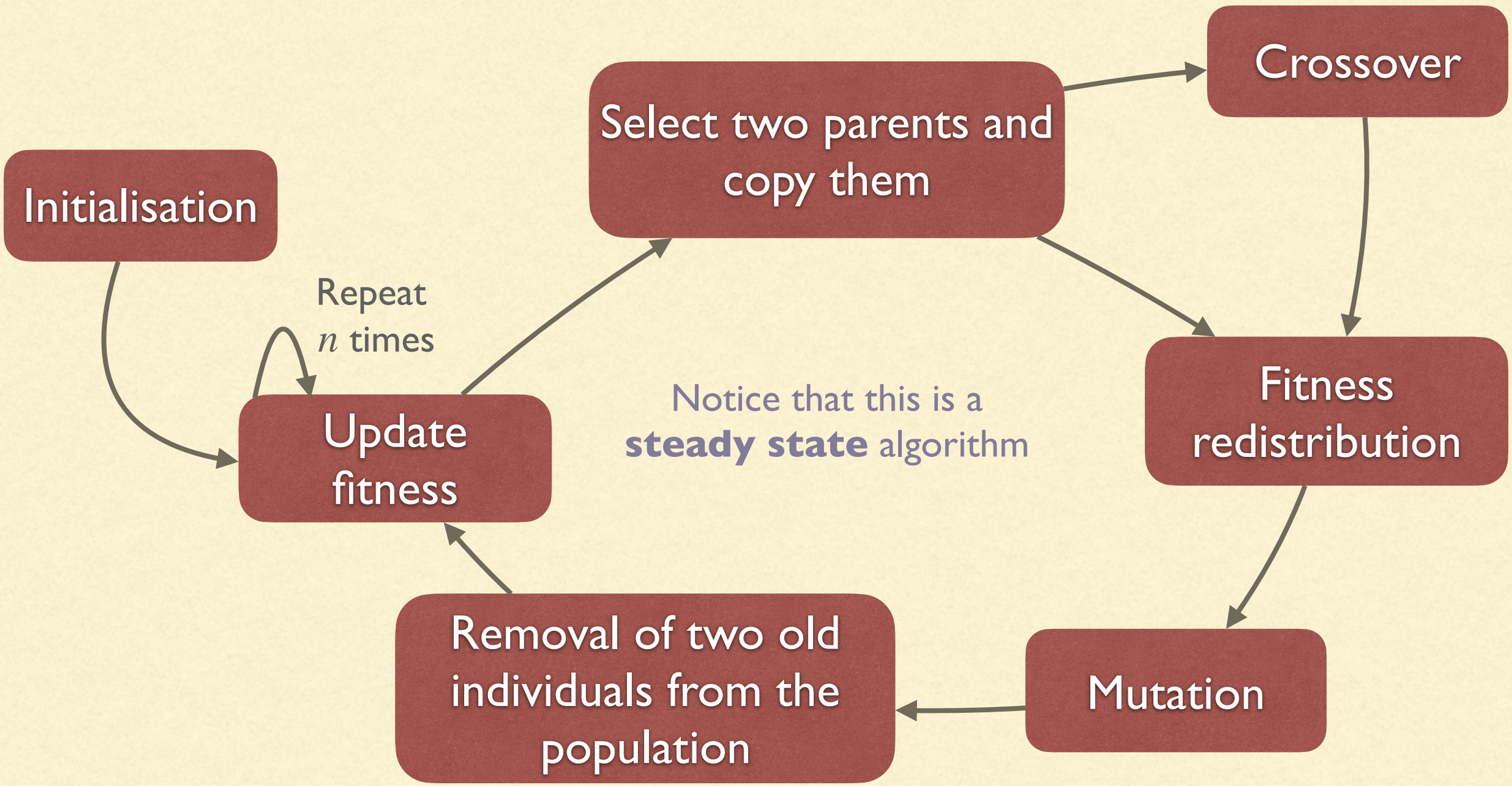
THE ZCS ALGORITHM

- ZCS maintains a collection of **if…then…** rules,
  each one with a fitness corresponding to the *utility* of the rule

- ZCS requires exact match

- If a state s is not covered by any rule a new rule that covers s and possibly other state associated to a random action. The new rule replaces an old rule marked for removal (usually a low-fitness one)

ZCS ALGORITHM CYCLE



FITNESS UPDATE

- Let s be the current state
- Let M be the set of rules whose body matches the state s
- Select one rule to apply (usually with fitness-proportional selection)
- Extract the action a from the rule head
- Find the subset A of M of all rules having a as the head. This is the **action set**

FITNESS UPDATE

- Perform the action a obtaining reward r
- Compute the next match set M' and action set A'
- Update the fitness of each rule A_i rules in A as:

$$\text{Fitness}(A_i) \leftarrow (1-\alpha)\text{Fitness}(A_i) + \alpha\frac{1}{|A|}(r + \gamma \sum_{A'_j \in A'} \text{Fitness}(A'_j))$$

- This is basically the same idea as model-free Q-learning

FITNESS UPDATE

- In addition to updating the fitness of the rules in A, ZCS also punishes the rules in $B = M - A$ (i.e., the ones that match but with different actions)

- The fitness of each $B_i \in B$ is updated as
$\text{Fitness}(B_i) \leftarrow \beta\ \text{Fitness}(B_i)$
with $\beta \in [0,1]$

ZCS FITNESS REDISTRIBUTION

- When two parent rules P_1 and P_2 are selected we copy them as $C_1$and C_2 (the two parents are not removed from the population)

- If crossover is **not** performed the fitness is redistributed between parents and children as

$$\text{Fitness}(C_i) \leftarrow \frac{1}{2}\text{Fitness}(P_i) \text{ and } \text{Fitness}(P_i) \leftarrow \frac{1}{2}\text{Fitness}(P_i)$$

- If crossover is performed, the fitness is redistributed to the children by using the average of the fitness of the parents:

$$\text{Fitness}(C_i) \leftarrow \frac{1}{4}(\text{Fitness}(P_1) + \text{Fitness}(P_2))$$

and $$\text{Fitness}(P_i) \leftarrow \frac{1}{2}\text{Fitness}(P_i)$$

THE XCS ALGORITHM

- XCS is an improvement over ZCS with four main differences:
  - How the action is selected
  - How the fitness is updated
  - How the fitness is redistributed
  - How selection is performed in the evolutionary part

XCS: MEASURES

- Differently from ZCS, XCS uses four different quality measures for each rule:
  - **Rule utility**. This is separated from the fitness
  - **Utility error**. Measures the difference between the utility of the rule and of the rules activated by this one
  - **Accuracy**. Derived from the error. Lower error means higher accuracy
  - **Fitness**. Instead of representing utility is an "historical accuracy"

XCS: ACTION SELECTION

- XCS tries to determine the best possible action by using all rules in the match set M
- For each action a that appears in the head of a rule in M we compute $R_a \subseteq M$ as the set of all rules in M with head a
- We compute a score for the action a as $\frac{\sum_{r \in R} \text{Utility}(r) \times \text{Fitness}(r)}{\sum_{r \in R} \text{Fitness}(r)}$
- We use the score to select the best action (or another random action with probability ε)

XCS: UTILITY UPDATE

- We have to update utility, utility error, and fitness
- Utility is updated in a Q-learning style as

$$\text{Utility}(A_i) \leftarrow (1-\alpha)\text{Utility}(A_i) + \alpha(r + \gamma b)$$

where:

  - r is the reward, $0 \leq \gamma \leq 1$, and b is the utility of the best action at the next iteration

- The utility error is updated as:

$$\text{UtilityError}(A_i) \leftarrow (1-\alpha)\text{UtilityError}(A_i) + \alpha\,|\,b - \text{Utility}(A_i)\,|$$

XCS: FITNESS UPDATE

- To update the fitness we need the **accuracy** of a rule:
  - If the error is below a threshold ε then the accuracy is 1
  - Otherwise the accuracy a_i if rule A_i is defined as
    $$\delta\left(\frac{\varepsilon}{\text{UtilityError}(A_i)}\right)^{\beta}$$
    for two parameters δ and β

- The fitness is then updated using the accuracy:
  $$\text{Fitness}(A_i) = (1-\alpha)\text{Fitness}(A_i) + \alpha\frac{a_i}{\sum_{A_j \in A} a_j}$$

XCS: FITNESS REDISTRIBUTION AND SELECTION

- Differently form ZCS, the fitness redistribution does not change the fitness of the parents, but cuts down (by half) the fitness of the children
- The same happens for utility and utility error
- Selection of the individuals to be crossed over and mutated is not performed in the entire population but only in the **action set**