# Raport nSARSA


> Wojciech Ciężobka
> Informatyka - Data Science
> Uczenie Maszynowe
> Laboratorim 3


PS kod źródłowy wraz z kontekstem jest wgrany w UPEL.

## Uzupełnione luki w kodzie

### TODO: Tutaj trzeba zaktualizować tablicę wartościującą akcje Q

```
self.q[state_t, action_t] += \
    self.step_size * \
    return_value_weight * \
    (return_value - self.q[state_t, action_t])
```

### TODO: Tutaj trzeba policzyć zwrot G

```
def _return_value(self, update_step):

        return_value = 0.0
        for i in range(
                update_step + 1,
                min(update_step + self.step_no, self.final_step) + 1
        ):
            discount = self.discount_factor ** (i - update_step - 1)
            reward = self.rewards[self._access_index(i)]
            return_value += discount * reward

        if update_step + self.step_no < self.final_step:
            discount = self.discount_factor ** self.step_no
            estimate = self.q[
                    self.states[self._access_index(update_step + self.step_no)],
                    self.actions[self._access_index(update_step + self.step_no)]
                    ]
            return_value += discount * estimate

        return return_value
```

### TODO: Tutaj trzeba policzyć korektę na różne prawdopodobieństwa ρ (ponieważ uczymy poza-polityką)

```
def _return_value_weight(self, update_step):

    return_value_weight = 1.0
    for i in range(
                update_step + 1,
                min(update_step + self.step_no, self.final_step - 1) + 1
        ):
        state_i = self.states[self._access_index(i)]
        action_i = self.actions[self._access_index(i)]
```

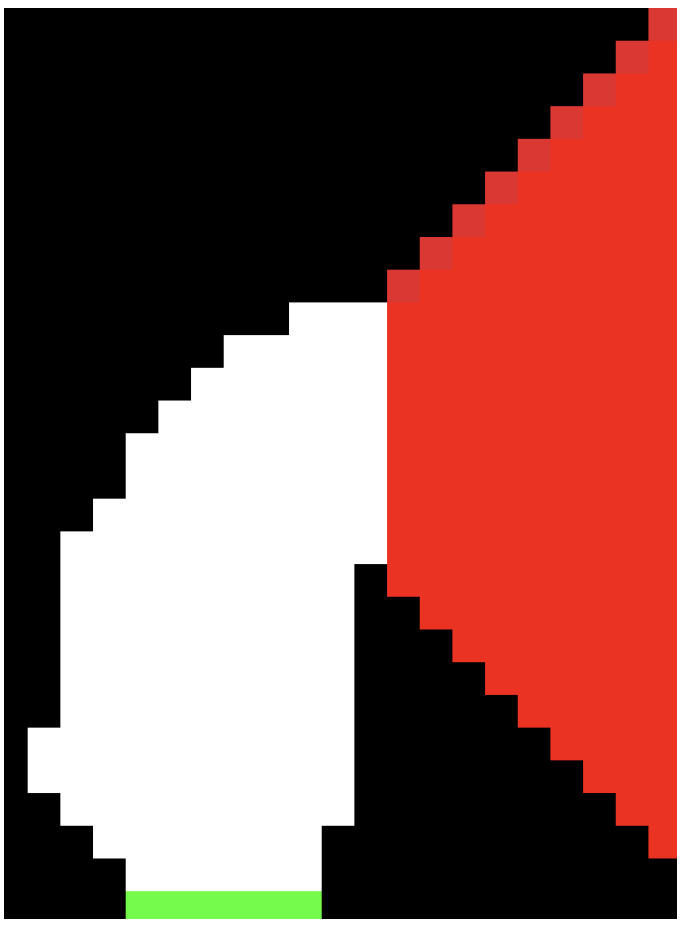
```
        pi = self.greedy_policy(
                        state_i, available_actions(state_i)
                )[action_i]
        behavior = self.epsilon_greedy_policy(
                        state_i, available_actions(state_i)
                )[action_i]
        return_value_weight *= pi / behavior

    return return_value_weight
```

### TODO: tutaj trzeba ustalic prawdopodobieństwa wyboru akcji według polityki ε-zachłannej

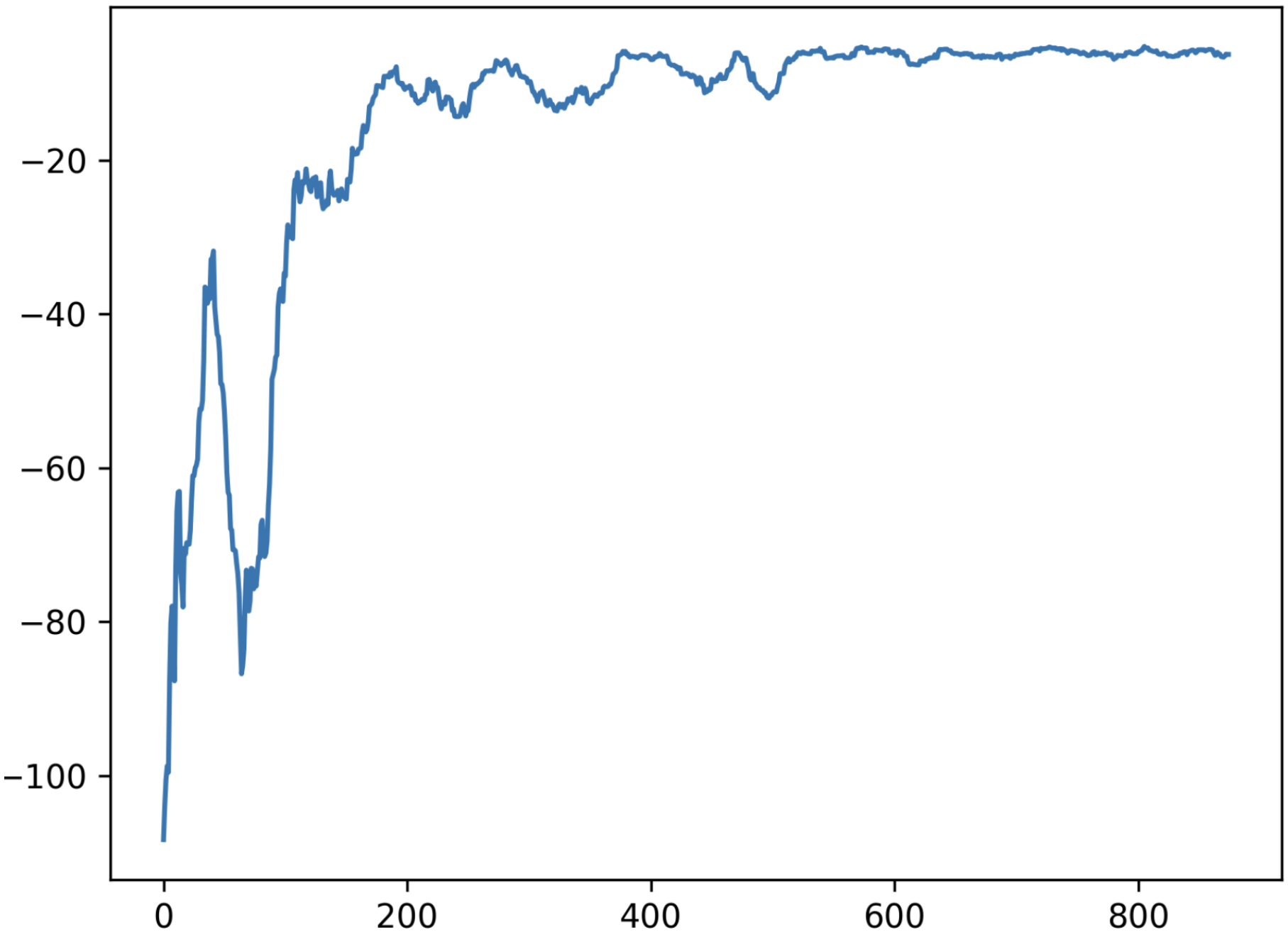
```
def epsilon_greedy_policy(
        self,
        state: State,
        actions: list[Action]
) -> dict[Action, float]:

    values = [self.q[state, action] for action in actions]
    maximal_spots = (values == np.max(values)).astype(float)
    probabilities = maximal_spots * \
                (1 - self.experiment_rate) / (np.sum(maximal_spots))
    probabilities += np.ones_like(values) * self.experiment_rate / len(values)

    return {
                action: probability for action, probability \
                in zip(actions, probabilities)
        }
```

# Prosty zakręt testowy

## Wizualizacja mapy



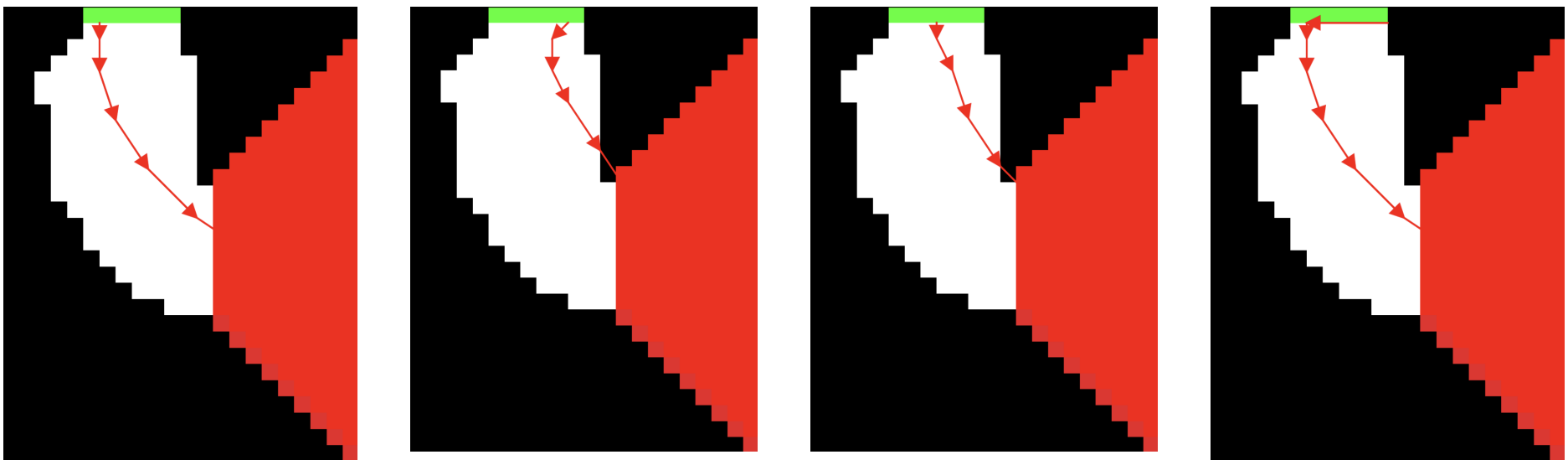
## Czy algorytm się uczy?




Przez pierwsze 200 epok, agent intensywnie się uczy. Po 600 epokach można uznać że agent się osiągnął punkt zbieżności.

## Przykładowe przejazdy dla wyuczonego algorytmu

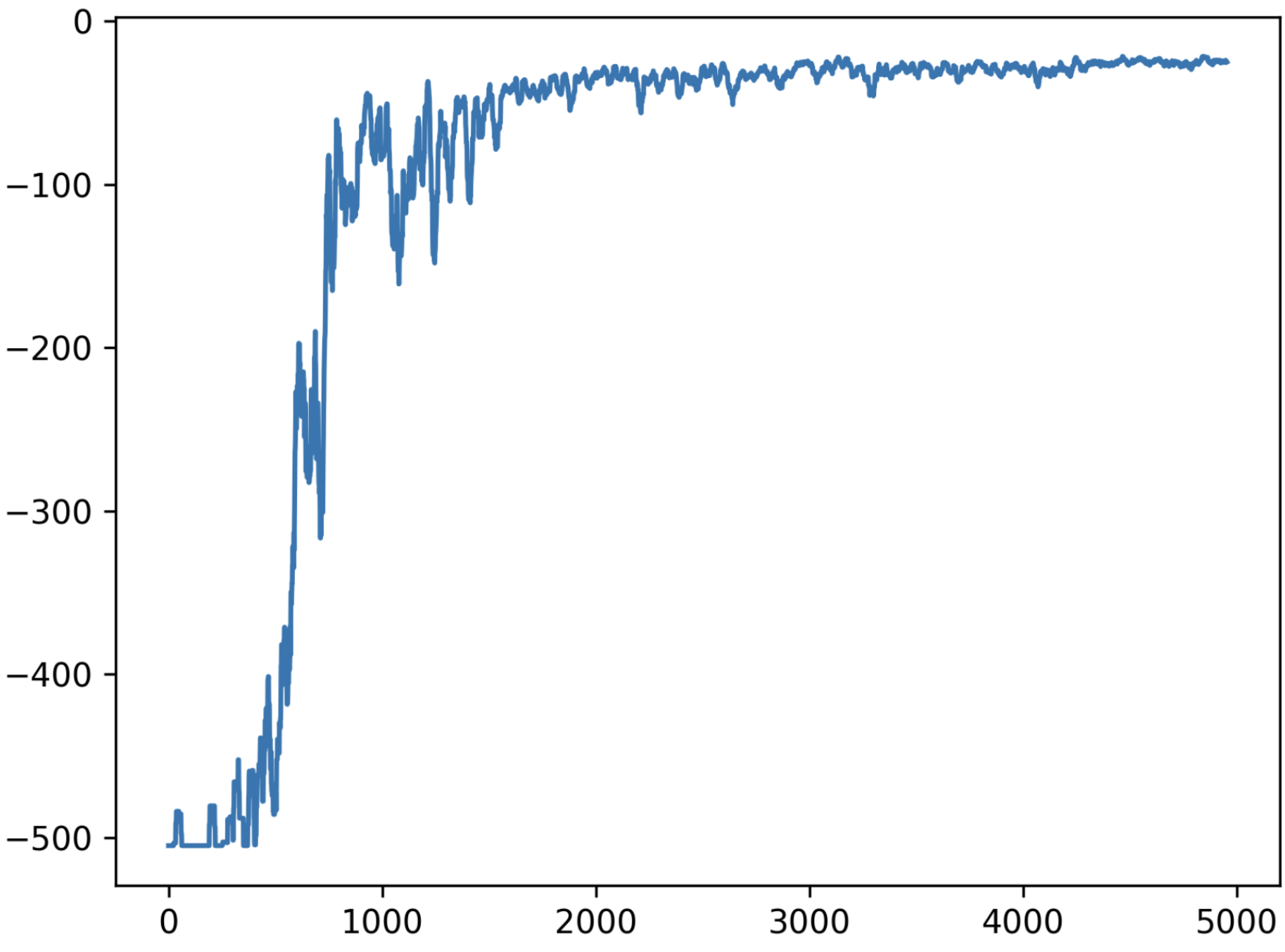
Aby zwizualizować optymalne trasy, należy zapisać stan tabeli wartościującej Q, a następnie uruchomić agenta podejmującego akcje w oparciu o politykę zachłanną.



## Trudniejszy zakręt dla optymalizacji hiperparametrów

### Wizualizacja mapy

**Przebieg uczenia dla domyślnych hiperparametrów**



Na tej mapie, agent potrzebuje więcej czasu na naukę. Zaczyna zbiegać po ok 2000 epizodów.

## Studium Parametrów

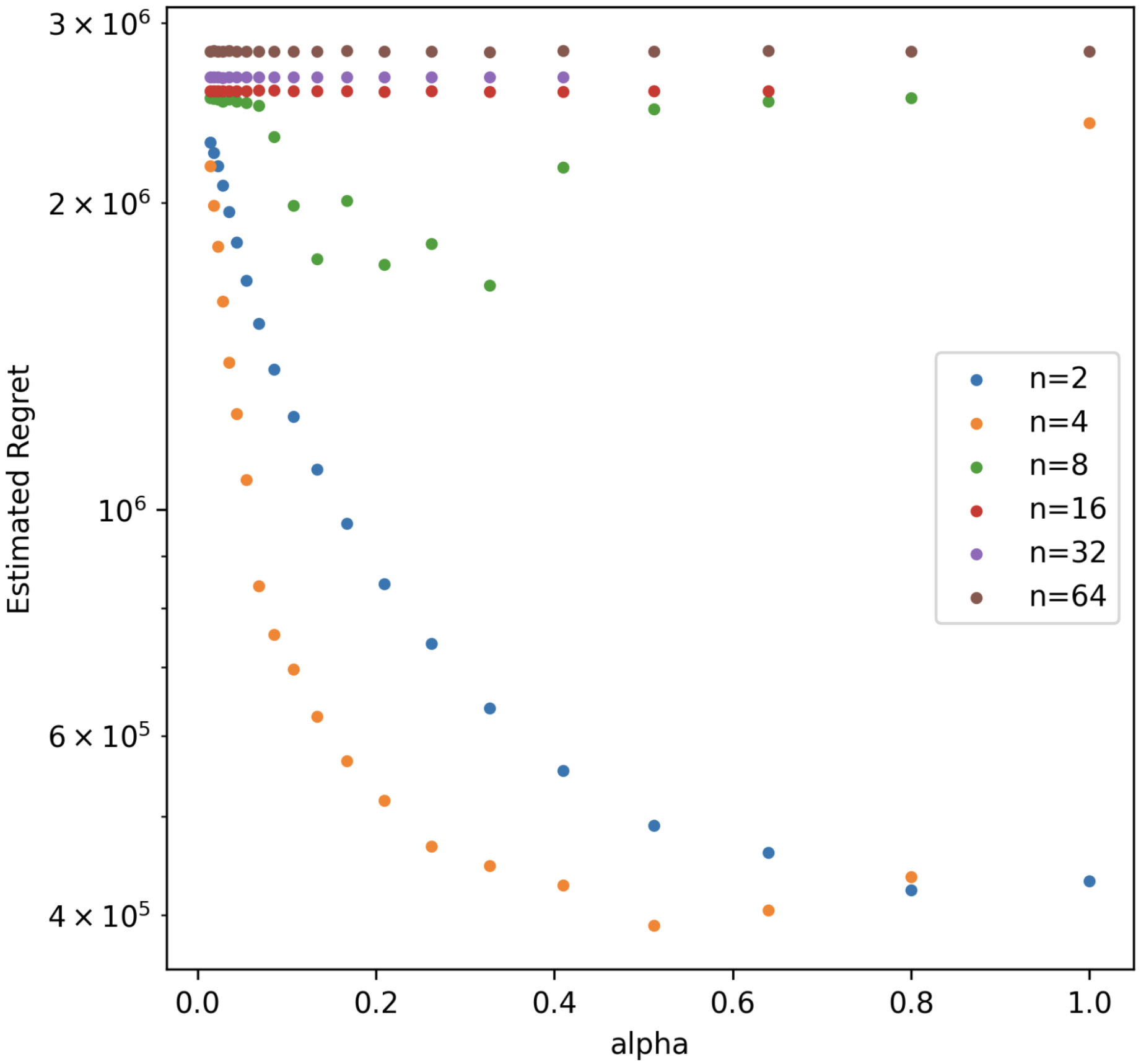
Przeszukuję przestrzeń parametrów $\alpha$ oraz liczby kroków $n$ aby znaleźć najlepiej działającego agenta. Uznałem, że miarą jakości agenta będzie całkowity *żal* (ang. regret). Ponieważ nie znam idealnej nagrody, to uznałem że dobrym szacunkiem optymalnej nagrody jest wartość $0$, kiedy agent po pierwszym kroku ląduje na mecie. W takim przypadku, estymowany żal, to po prostu wartość kary.

### Środowisko testowe HPC

Eksperymenty przeprowadziłem na klastrze obliczeniowym Ares, uruchamiając równolegle symulacje za pomocą poniższego skryptu

```
#!/bin/bash -l

TIMESTAMP=$(date +%y%m%d_%H%M%S)
FOLDER_SPEC="nsarsa"
GITREPO="nSARSA"

# Create directory structure to store logs
mkdir -p $SCRATCH/Logs/$FOLDER_SPEC/
mkdir -p $SCRATCH/Logs/$FOLDER_SPEC/$TIMESTAMP/


ALPHA_SPAN=20
N_STEP_SPAN=9

for (( n = 1; n < N_STEP_SPAN; n++ )); do
    for (( a = 0; a < ALPHA_SPAN; a++ )); do

        N_STEP=$((2**n))
        ALPHA=$(echo "scale=5; 1.25^(-$a)" | bc)

        LOG_OUTPUT_DIR=$SCRATCH/Logs/$FOLDER_SPEC/$TIMESTAMP/n$N_STEP-a$ALPHA-output.out
        LOG_ERROR_DIR=$SCRATCH/Logs/$FOLDER_SPEC/$TIMESTAMP/n$N_STEP-a$ALPHA-error.err
        sbatch \
                    --job-name="nsarsa"
                    --account="plgsano4-cpu"
                    --partition="plgrid"
                    --output="$LOG_OUTPUT_DIR"
                    --error="$LOG_ERROR_DIR"
                    --time="4:00:00"
                    --nodes="1"
                    --ntasks-per-node="1"
                    --cpus-per-task="1"
                    --mem-per-cpu="1GB"
                    $SCRATCH/Files/GitRepos/$GITREPO/hpc-runner.sh $N_STEP $ALPHA
    done
done
```

### Wykres oczekiwanego “żalu” od parametrów $\alpha$ oraz $n$




Wykres nie jest tak ładny jak dla zabawkowego problemu z polecenia, jednak kształty krzywych oraz ich wzajemne położenie zgadzają się z teorią. Okazuje się, że agent najlepiej sobie radzi z parametrem $\alpha = 0.512$ oraz $n = 4$. Takie parametry ustawiłem od tej pory jako domyślne i przeszedłem do testów na najtrudniejszej mapie.

# Najtrudniejszy przypadek dla walidacji

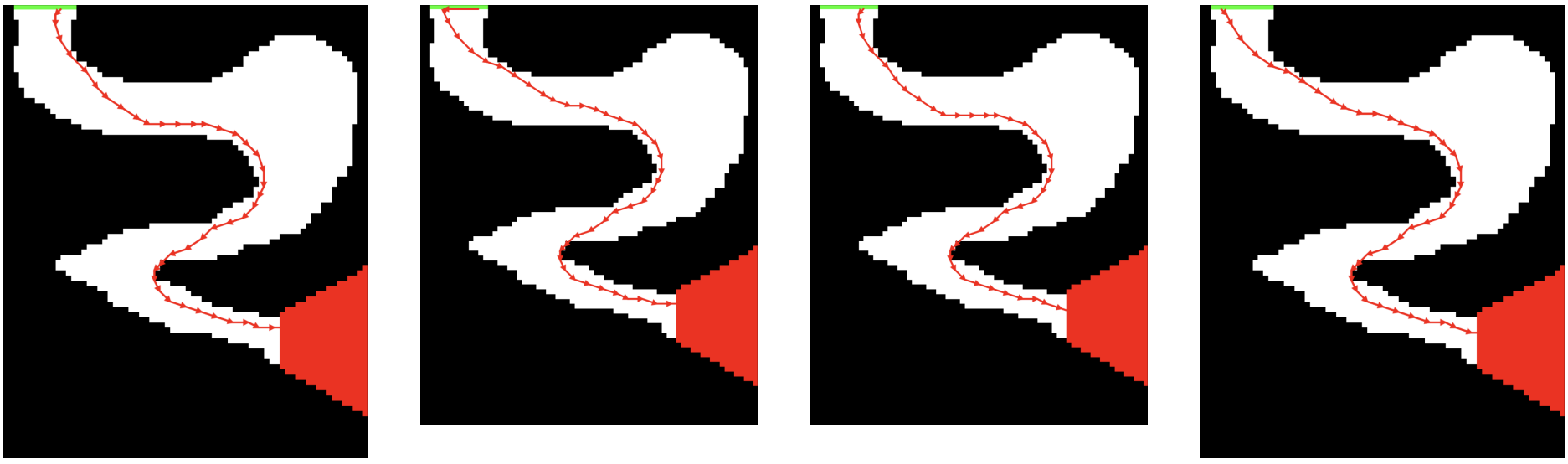
## Wizualizacja mapy

**Przebieg uczenia**




Agent z dostrojonymi parametrami na mapie **C**, radzi sobie bardzo dobrze na trudniejszej mapie **D**. Co prawda trening zajmuje więcej czasu (ok. 11 000 epok), ale zbieżność jest bardzo ładna i cały proces uczenia jest stabilny.

## Przykładowe przejazdy dla wyuczonego algorytmu



Agent nauczył się bardzo efektywnie pokonywać zakręt. Niezależnie od pozycji startowej dociera do mety oraz ścina zakręty aby jak najbardziej skrócić czas przejazdu.