Oscar Teeninga

Uczenie maszynowe

Lab 1

1. Implementacja

Dostępna wraz z sprawozdaniem. Najbardziej problematyczne dla mnie było wyznaczenie _return_value_weight, ponieważ nieintuicyjne było dla mnie, że sumujemy prawdopodobieństwa zamiast wybierać jedno z nich.

```
if update_step >= 0:
    return_value_weight = self._return_value_weight(update_step)
    return_value = self._return_value(update_step)
    state_t = self.states[self._access_index(update_step)]
    action_t = self.actions[self._access_index(update_step)]
    if update_step + self.step_no < self.final_step:
        state_t2 = self.states[self._access_index(update_step + self.step_no)]
        action_t2 = self.actions[self._access_index(update_step + self.step_no)]
        return_value += pow(self.discount_factor, self.step_no) * self.q[state_t2, action_t2]

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

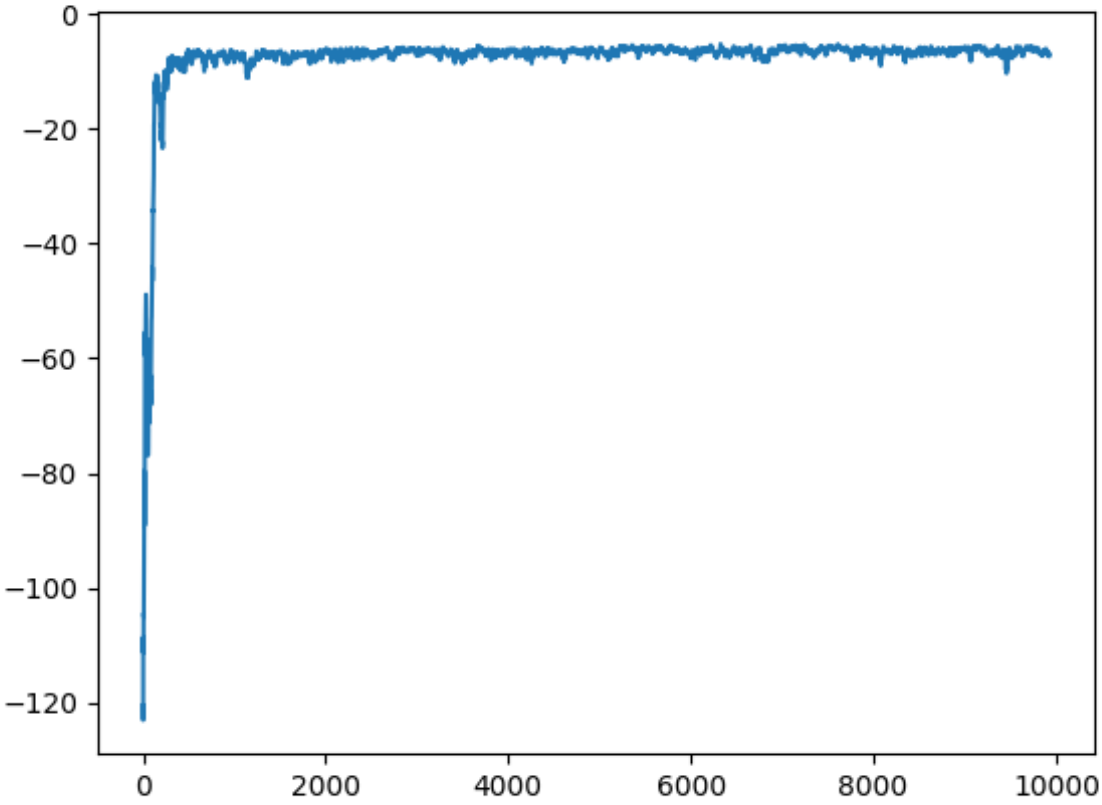
```
def _return_value_weight(self, update_step):
    return_value_weight = 1.0
    for i in range(update_step + 1, min(update_step + self.step_no, self.final_step)):
        state_t = self.states[self._access_index(i)]
        action_t = self.actions[self._access_index(i)]
        b = self.epsilon_greedy_policy(state_t, available_actions(state_t))[action_t]
        p = self.greedy_policy(state_t, available_actions(state_t))[action_t]
        return_value_weight *= p/b
    return return_value_weight
```

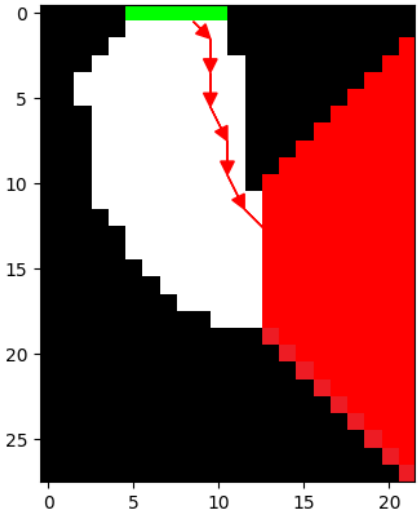
```
def _return_value(self, update_step):
    return_value = 0.0
    for i in range(update_step + 1, min(update_step + self.step_no, self.final_step) + 1):
        return_value += pow(self.discount_factor, i - update_step - 1) * self.rewards[self._access_index(i)]
    return return_value
```

```
def epsilon_greedy_policy(self, state: State, actions: list[Action]) -> dict[Action, float]:
    probabilities = self.experiment_rate * self._random_probabilities(actions) + (1 - self.experiment_rate) * self._greedy_probabilities(state, actions)
    return {action: probability for action, probability in zip(actions, probabilities)}
```

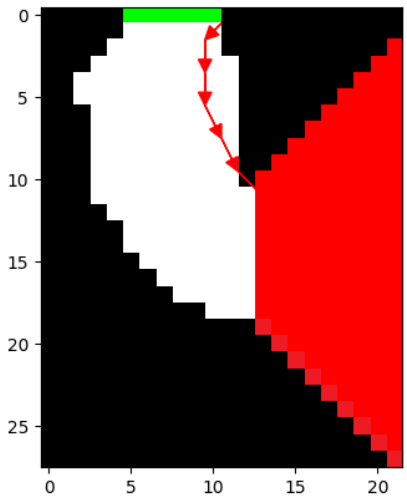
2. Działanie dla zakrętu “b”

Algorytm działa dla bazowych parametrach *step_no* = 5 przy *exp_rate* = 0.05 oraz *step_size* = 0.3. Przy ok. 200 iteracjach mamy znaczą poprawę i optymalną drogę.





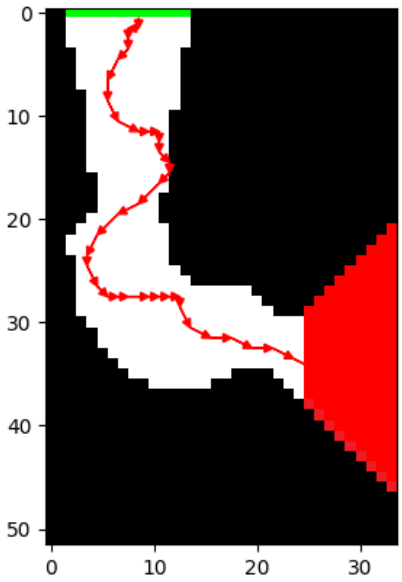
Iteracja 7850



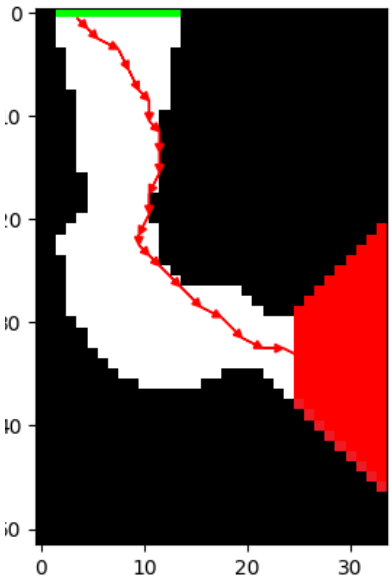
Iteracja 9950

3. Działanie dla zakrętu “c”

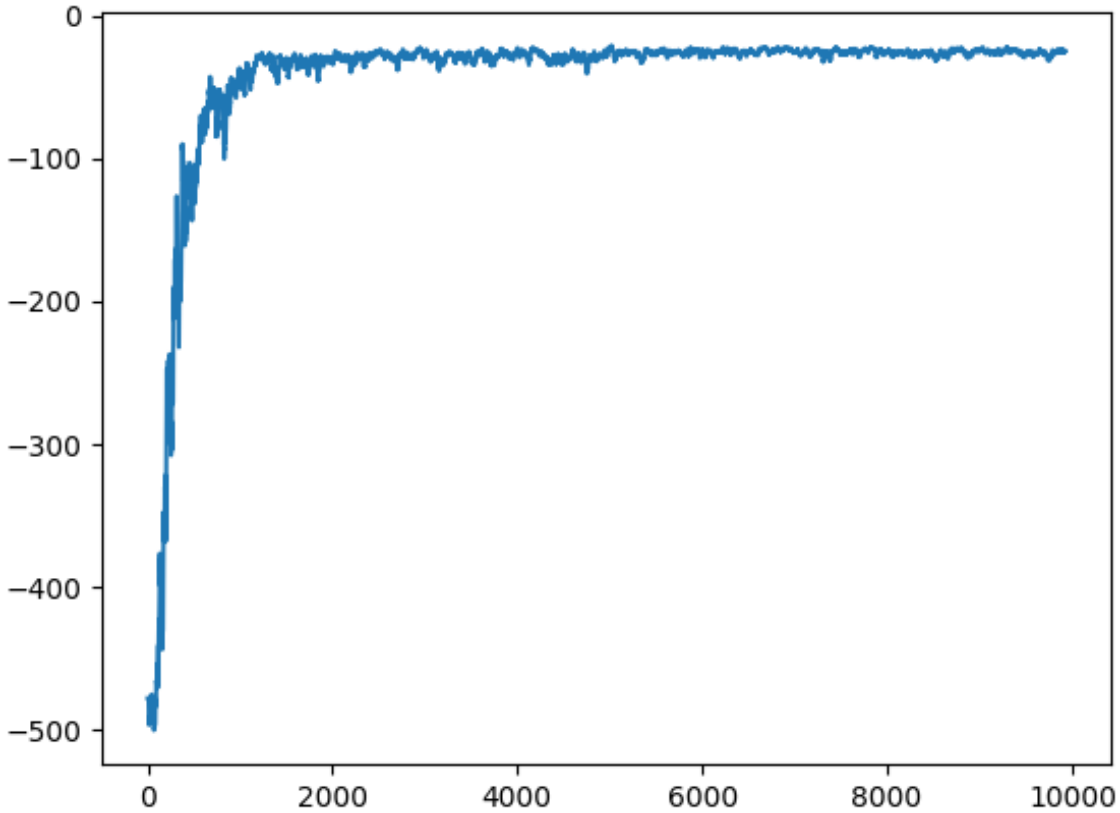
Tutaj badanie trwał nieco dłużej, ok. 8 minut. Pierwszy test przeprowadziłem dla *step_no* = 5, *step_size* = 0.3 i *experimental_rate* = 0.05. Pozwoliło to osiągnąć mało zadowalające wyniki.



Iteracja 1000

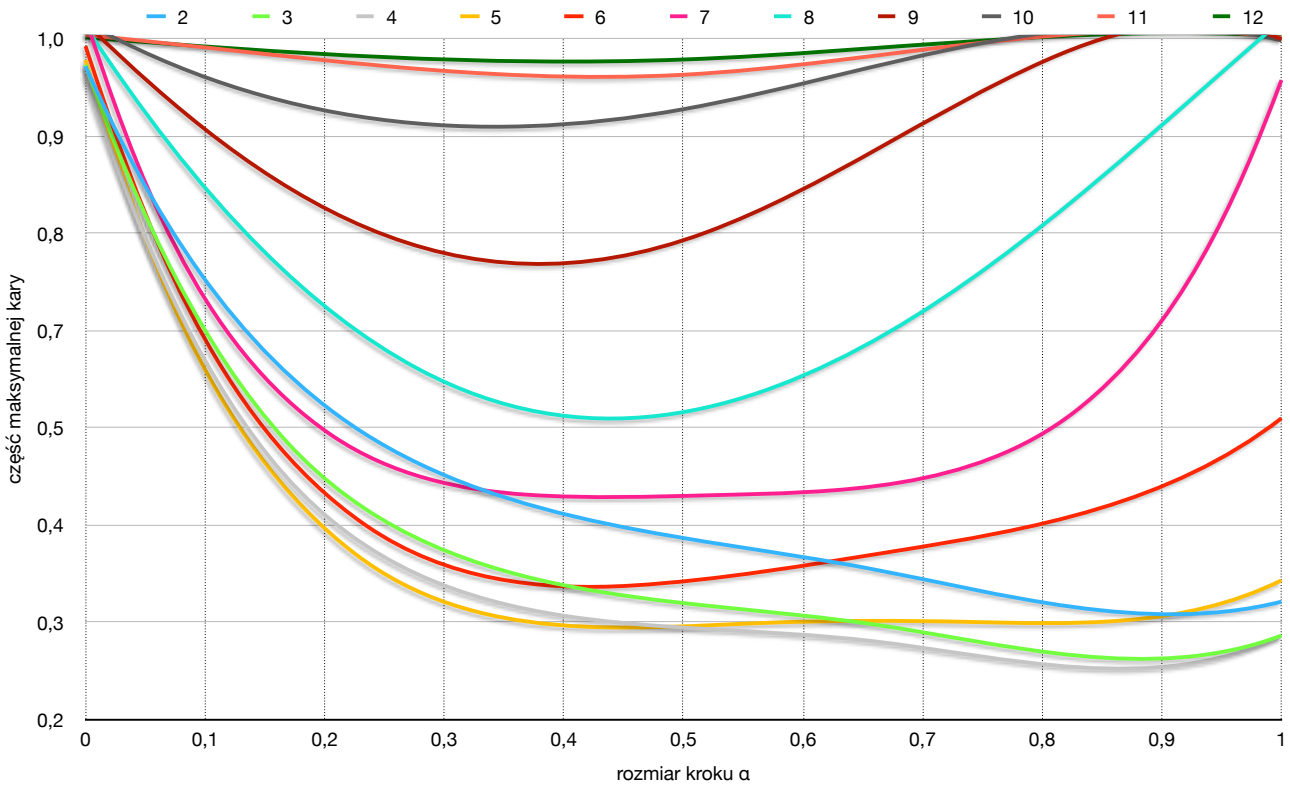


Iteracja 9950



4. Wpływ parametru α i liczby kroków na wyniki dla zakrętu “c”

Pomiary wykonywałem tworząc prosty skrypt zapisujący wyniki do pliku csv. Pomiary wykonywałem na *MAX_LEARNING_STEPS = 1000*. Czas wykonywania obliczeń oscylował w granicach ~3-17 minut. Każdy pomiar jest średnią arytmetyczną kar dla wszystkich 1000 epok. Następnie pomiar dzieliłem przez maksymalną wartość -500. Otrzymane procenty są wartościami na wykresie. Linie są przybliżeniem wielomianem czwartego stopnia.

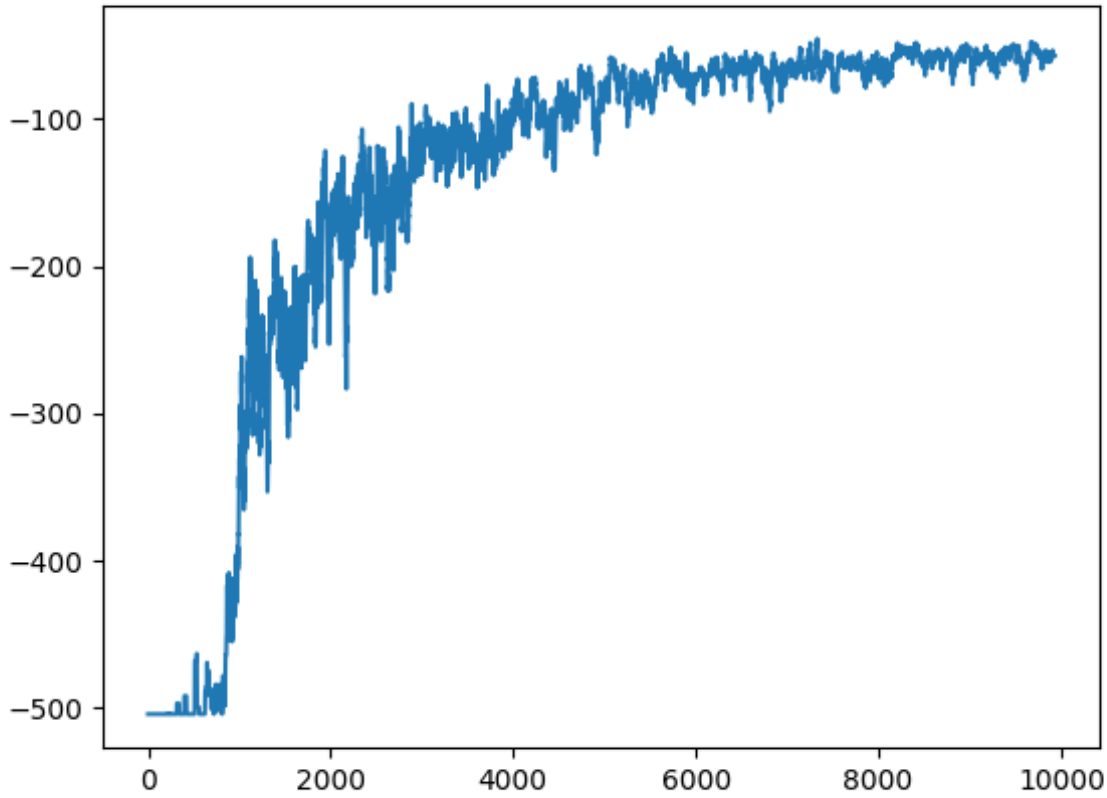


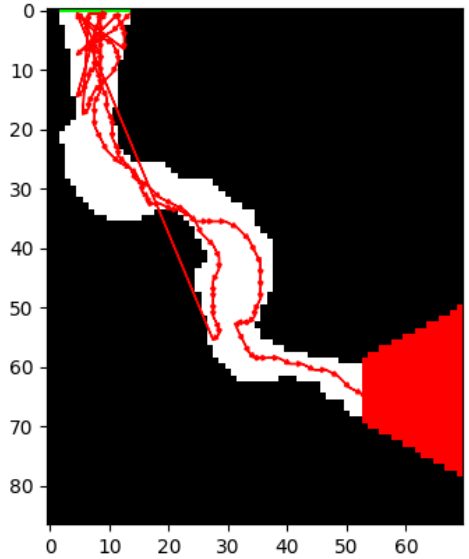
Wykres rozmiaru kroku α do szybkości uczenia dla różnych numerów kroków

Najlepszy efekt osiągnąłem dla *step_no = 4 i α = 0.9*, więc dla takich parametrów będę kontynuował zadanie dla zakrętu d.

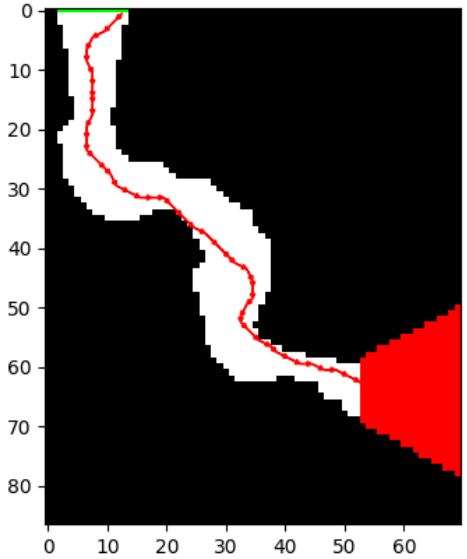
5. Działanie dla zakrętu “d”

Tutaj uczenie trwało najdłużej ~18 minut. Przyjęte parametry to *step_no = 4* i *α = 0.9.* Widać również, że początkowo nie byliśmy w stanie osiągnąć innego wyniku niż minimalnego (takiego w którym nie dochodziliśmy do pozycji końcowej). Przy ok. 1000 iteracji widać znaczną poprawę. Uczenie zakończyło się na ok. 9000.





Iteracja 1000



Iteracja 9950