Laboratorium programistyczne 5
Klasy i Monte Carlo Tree Search
(temat dodatkowy)

Projekt „Matematyka dla Ciekawych Świata”,
Piotr Morawiecki

2025-05-21

Spis treści

1 Klasy w Pythonie

Nauczyliśmy się do tej pory definiować własne funkcje w Pythonie, które dla podanych argumentów wykonują określoną instrukcję.

Innym ważnym narzędziem w programowaniu są tzw. klasy. Pozwalają one definiować własne typy danych (czyli obiekty) wraz z ich atrymutami (zmiennymi) i metodami (funkcjami).

Rozważmy następującą klasę reprezentującą psa:

```
class Dog:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def bark(self):
    print(self.name + " zaszczekał!")
```

W tej klasie są zdefiniowane dwie metody:

- `__init__`, która jest wywoływana przy stworzeniu obiektu danej klasy (jest to tzw. konstruktor), oraz
- `bark`, po której wywołaniu dany pies ”zaszczeka”.

W konstruktorze do psa są przypisane dwa atrybuty: imię (`name`) oraz wiek (`age`). Są one przypisane do danego obiektu i mogę być wykorzystywane w innych metodach, np. metodzie `bark`, która wykorzystuje wcześniej przypisane imię psa (`self.name`).

Tak jak w przypadku funkcji, samo zdefiniowanie klasy jeszcze nic nie robi – musimy ją wykorzystać w dalszym kodzie Pythona, np.:

```
dog1 = Dog("Burek", 3)
dog1.bark()
```

W pierwszej linijce tworzymy psa o imieniu Burek i wieku 3 lat. W drugiej linijce wydajemy do psa komendę o zaszczekanie. Zwróć uwagę na kilka rzeczy:

- `Dog` jest nazwą klasy, a `dog1` jest obiektem (tzw. instancją) danej klasy. W programie możesz stworzyć wiele różnych obiektów tej samej klasy.
- Żeby wywołać metodę danej klasy najpierw piszemy nazwę obiektu, dla którego ta metoda ma zostać wywołana, a następnie po kropce podajemy nazwę metody.
- W definicji metod `__init__` i `bark` pierwszym argumentem jest `self`. Ten argument reprezentuje obiekt danej klasy, dla którego ta metoda została wywołana. Jednak w momencie wywoływania metody nie podajemy tego argumentu w nawiasie.

Zadanie 1.0.1

Dodaj w programie drugiego psa o imieniu Hektor i wieku 2 lat. Następnie "zaszczekaj" Hektorem dwa razy i Burkiem raz.

Zadanie 1.0.2

Do klasy `Dog` dodaj metodę `rename(self, newName)`, która zmieni imię psa na nowe. Sprawdź działanie klasy za pomocą następującego programu.

```
dog3 = Dog("Maks", 6)
dog3.bark()
dog3.rename("Maksio")
dog3.bark()
```

2 Gra w odejmowanie

W tej sekcji stworzymy klasę, która pozwoli zagrać dwóm graczom w prostą grę w odejmowanie. Zasady są następujące. Na stosie umieszcza się określoną liczbę żetonów (np. 20). Następnie gracze na zmianę odejmują ze stosu jedną z dozwolonych liczb żetonów (np. 1, 2, 5 lub 10 żetonów). Wygrywa gracz, który zabierze ostatni żeton.

Stworzymy klasę `Game`, reprezentującą bieżący stan gry. Będzie ona zawierać trzy atrybuty:

- `currentNumber` - bieżąca liczba żetonów na stosie,
- `movesAvailable` - lista zawierająca dozwolone akcje, np. `[1,2,5,10]`,
- `currentPlayer` - number gracza, który teraz wykonuje ruch (`0` w przypadku gracza pierwszego lub `1` dla gracza drugiego).

W klasie zdefiniujemy trzy metody:

- `__init__` - konstruktor pozwalający na ustawienie dowolnej liczby początkowych żetonów i dostępnych akcji,
- `valid_move` - metoda, która zwraca listę dostępnych akcji, pamiętając o tym, że nie można zabrać więcej żetonów niż obecnie znajduje się na stosie,
- `print` - metoda wypisująca bieżący stan gry,
- `play` - która dla bieżącego gracza wykona określoną akcję (`move`), zabierając ze stosu odpowiednią liczbę żetonów. Metoda ta będzie zwracań `True` jeśli zostanie zabrany ostatni żeton, lub `False` w przeciwnym wypadku.

Klasa powinna wyglądać następująco:

```
class Game:
  def __init__(self, initialNumber, movesAvailable):
    self.currentNumber = initialNumber
    self.movesAvailable = movesAvailable
    self.currentPlayer = 0

  def valid_move(self):
    # DO DOKOŃCZENIA

  def print(self):
    print("Obecną liczbą jest " + str(self.currentNumber))
    print("Gracz " + str(self.currentPlayer) + " może odjąć " +
     ↪  str(self.valid_move()))

  def play(self, move):
    if move in self.valid_move():
      # DO DOKOŃCZENIA
    else:
      print("Ruch jest niemożliwy. Wybierz inny.")
      return False
```

Zadanie 2.0.1

Dokończ pisać metodę `valid_move(self)`. Powinna ona zwrócić listę dostępnych ruchów, pamiętając o tym, że nie można ze stosu zabrać więcej żetonów niż się na nim obecnie znajduje.

Zadanie 2.0.2

Dokończ pisać metodę `play(self, move)`. Powinna ona:

- zaktualizować liczbę żetonów,
- w przypadku wygranej wypisać komunikat o zwycięstwie danego gracza i zwrócić `True`,
- w przeciwnym wypadku zmienić aktualnego gracza i zwrócić `False`.

Zadanie 2.0.3

Przetestuj grę, korzystając z następującego skryptu:

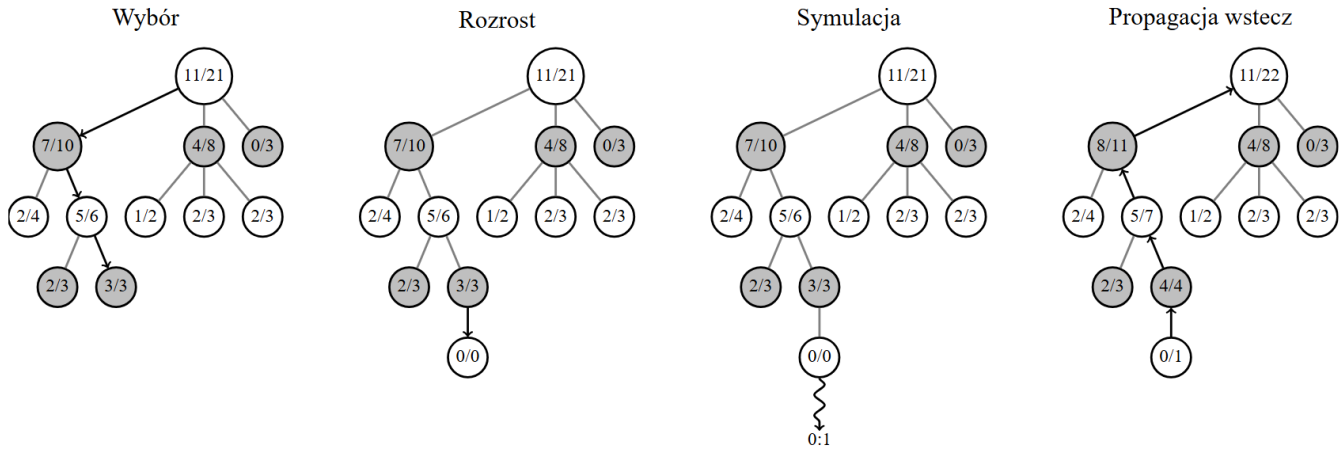
```
def playWithHuman():
  game = Game(20, [1,2,5,10])

  while True:
    game.print()
    print("Choose your move: ", end="")
    move = int(input())
    print()
    if game.play(move):
      print("GRATULACJE! Gracz " + str(game.currentPlayer) + " zwyciężył!")
      break

playWithHuman()
```

Instrukcja `int(input())` pozwala użytkownikowi wpisać ruch, a następnie zamienia go z napisu na liczbę całkowitą.

3 Monte Carlo Tree Search (Connect 4)

Na wykładzie poznaliśmy algorytm Monte Carlo Tree Search (MCTS) pozwalający wybierać strategie w złożonych grach poprzez selektywne eksplorowanie drzewa gry. Algorytm składał się z następujących czterech kroków (https://pl.wikipedia.org/wiki/Monte-Carlo_Tree_Search):

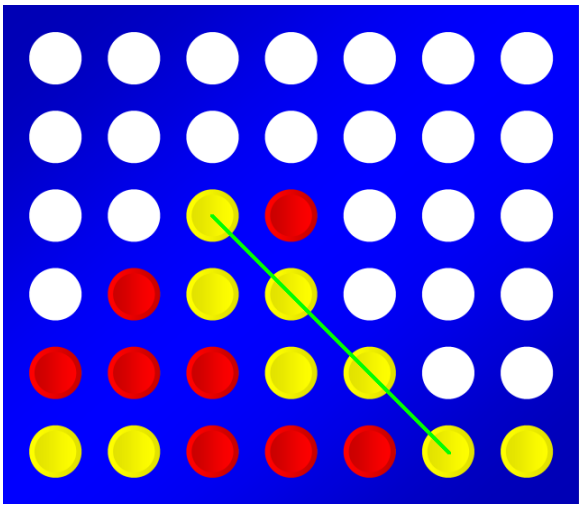
1. **Wybór** (ang. *selection*): Zaczynając od korzenia drzewa gry R wybierz kolejne węzły potomne, aż dotrzesz do liścia drzewa L,
2. **Rozrost** (ang. *expansion*): O ile wybrany liść nie kończy gry, utwórz w nim jeden lub więcej węzłów potomnych i wybierz z nich jeden węzeł C,
3. **Symulacja** (ang. *playout*): rozegraj losową symulację z węzła C,
4. **Propagacja wstecz** (ang *backpropagation*): Na podstawie wyniku rozegranej symulacji uaktualnij informacje w węzłach na ścieżce prowadzącej od C do R.



Rysunek 1: Ilustracja kolejnych kroków algorytmu MCTS (źródło: https://pl.wikipedia.org/wiki/Monte-Carlo_Tree_Search)

Dość prosta implementacja algorytmu MCTS w Pythonie jest dostępna na stronie https://github.com/floriangardin/connect4-mcts/tree/master. Poza implementacją MCTS repozytorium to zawiera

też implementację gry *Connect 4*, którą poznaliście podczas turnieju. W tej grze dwóch graczy na zmianę umieszcza żetony w jednej z siedmiu kolumn. Wygrywa gracz, który umieści w jednej linii 4 swoje żetony (pionowo, poziomo lub na skos).



Rysunek 2: Rozegrana gra w Connect 4 wygrana przez gracza żółtego (źródło: https://commons.wikimedia.org/wiki/File:Connect4_Wins.PNG)

Repozytorium zawiera następujące skrypty

- `main.py` – jest to główny skrypt, którego wykonanie pozwala rozegrać grę z MCTS,
- `connect4/connect4.py` – ten moduł zawiera funkcje analogiczne go tych zaimplementowanych dla *gry w odejmowanie* w Sekcji 2,
- `connect4/mcts.py` – ten moduł zawiera klasę reprezentującą pojedynczy węzeł eksplorowanego drzewa gry,
- `connect4/mcts_ia.py` – ten moduł zawiera funckje umożliwiające trenowanie algorytmu MCTS zgodnie z opisanymi powyżej krokami oraz przeprowadzanie losowej symulacji gry.

Na potrzeby tych zajęć stworzyłem dla Was notebook w Google Colab, do którego przekopiowałem zawartość powyższego repozytorium z pewnymi niewielkimi zmianami: https://colab.research.google.com/drive/1vylKm1nsDxs_mwxOVeUHWAHDsuQ-PwgK?usp=sharing.

Zadanie 3.0.1

Przekopiujcie wszystkie skrypty z powyższego linku do waszego własnego projektu na Google Colab. Sprawdźcie działanie programu, wykonując cały skrypt.

W tej implementacji każdy węzeł drzewa jest reprezentowany przez obiekt klasy `Node` (zaimplementowany w pliku `connect4/mcts.py`). Do każdego węzła są przypisane takie atrybuty jak:

- `parent` – węzeł nadrzędny reprezentujący stan gry w poprzednim kroku
- `move` – zapisuje ruch, który prowadzi do danego stanu,
- `win` – liczba symulacji, które przeszły przez ten węzeł i zakończyły się zwycięstwem danego gracza,
- `games` – liczba wszystkich symulacji, które przeszły przez ten węzeł,
- `children` – lista wszystkich węzłów pochodnych do danego węzła (tzn. lista stanów gry, do których można dojść z danego węzła),
- `state` – zapis stanu planszy, który jest reprezentowany przez ten węzeł,
- `winner` – wartość zwycięzcy jeśli ten węzeł reprezentuje stan kończący grę (-1 jeśli wygrał pierwszy gracz, $+1$ jeśli wygrał drugi gracz i 0 jeśli żaden nie wygrał.

Zadanie 3.0.2

Algorytm MCTS podczas uczenia się iteracyjnie wykonuje funkcję `train_mcts_once`. W grupie przedyskutujcie zawartość tej funkcji. Na czym polega każdy krok algorytmy Monte Carlo Tree Search?

Zadanie 3.0.3

Przedyskutujcie odpowiedzi na poniższe pytania.

1. Jakie znaczenie ma funkcja `get_uct(self)` klasy `Node` podczas procesu uczenia się algorytmu MCTS?
2. Funkcja ta poza uwzględnieniem tego ile gier przechodzących przez dany węzeł zakończyło się zwycięstwem, zawiera człon również `np.sqrt(2*np.log(self.parent.games)/self.games)`. Wyjaśnij dlaczego.
3. Jakie jest znaczenie liczby 2 w tym wzorze? Jak na zachowanie algorytm MCTS wpłynęłaby zmiana tej stałej na przykład na 20?
4. Sprawdź swoją odpowiedź na poprzednie pytanie doświadczalnie rozgrywając grę z MCTS dla zmodyfikowanej funkcji `get_uct`.

Zadanie 3.0.4

W kroku trzecim algorytmu MCTS algorytm z repozytorium wykonuje symulację losową za pomocą funkcji `random_play_improved(grid)`, ale także repozytorium zawiera funkcję `random_play(grid)`.

1. Przeanalizujcie i wspólnie przedyskutujcie różnice między tymi dwoma funkcjami.
2. Sprawdź czy zmiana `random_play_improved(grid)` na `random_play(grid)` wpłynęłaby na skuteczność algorytmu MCTS.

Zadanie 3.0.5

W grupie przedyskutujcie jak jeszcze można by zwiększyć efektywność implementacji algorytmu MCTS z powyższego repozytorium.

4 Zadania dodatkowe

Zadanie 4.0.1

1. Zmodyfikuj program z sekcji 3, tak żeby algorytm MCTS grał sam ze sobą. W tym zadaniu nie ma potrzeby tworzyć dla każdego gracza osobnego drzewa gry.
2. Zbadaj za pomocą metody Monte Carlo (wielokrotnie rozgrywając grę algorytmu MCTS z samym sobą), kiedy algorytm wygrywa częściej – kiedy zaczyna grę czy kiedy gra jako drugi gracz?

   Ważne: Niestety ze względu na czas uczenia się algorytmu MCTS, wykonanie reprezentatywnej próbki symulacji zajmuje sporo czasu. W celu przyspieszenie procesu możesz spróbować skrócić czas trenowania algorytmu.

Zadanie 4.0.2 (dla ambitnych)

Wykorzystując klasę `Game` z Sekcji 2 oraz implementację algorytm Monte Carlo Tree Search omawiane w sekcji 3, napisz program, w którym gracz będzie mógł zmierzyć się z algorytmem MCTS w grze w żetony.

Uwaga: Jest to trudne zadanie, gdyż gry są inaczej reprezentowane. W Connect4 plansza jest reprezentowana przez tablicę 6×7, a w naszej grze przez dwie liczby – liczbę żetonów na stosie oraz numer bieżącego gracza. Musisz odpowiednio dostosować reprezentację gry w algorytmie MCTS.

5 Praca domowa

Zadanie 5.0.1 (2 punkty)

Dodaj do klasy `Game` z Sekcji 2 funkcję `EasyIA`, która wykona akcję, korzystając z następującego algorytmu.

- Jeśli istnieje ruch wygrywający, to wykona ten ruch.
- W przeciwnym razie wybierze losowy ruch.

Napisz funkcję `PlayWithEasyAI()`, w której gracz będzie mógł zagrać z botem korzystającym z tej strategii.

Zadanie 5.0.2 (2 punkty)

Dodaj do klasy `Game` z Sekcji 2 funkcję `MediumIA`, która wykona akcję, korzystając z następującego algorytmu.

- Jeśli istnieje ruch wygrywający, to wykona ten ruch.
- W przeciwnym razie sprawdź które ruchy pozwolą przeciwnikowi wygrać w następnej turze. Wykonaj losowy z pozostałych ruchów.
- Jeśli niezależnie od ruchu bieżącego gracza przeciwnik zawsze będzie mógł w kolejnym ruchu wygrać, to wykonaj losowy ruch.

Napisz funkcję `PlayWithMediumAI()`, w której gracz będzie mógł zagrać z botem korzystającym z tej strategii.

Zadanie 5.0.3 (2 punkty)

Napisz funkcję `BattleOfAI()`, w której bot wykonujący funkcję `MediumIA` będzie grał przeciwko botowy, który wykonuję funkcję `EasyIA`. Funkcja powinna losowo wybierać bota rozpoczynającego. Wykonując funkcję `BattleOfAI()` 1000 razy, zbadaj za pomocą metody Monte Carlo, jakie jest prawdopodobieństwo wygranej dla każdego bota.