Laboratorium programistyczne 4
Iterowany dylemat więźnia

Projekt „Matematyka dla Ciekawych Świata”,
Piotr Morawiecki

2025-01-02

Spis treści

1 O iterowanym dylemacie więźnia

Na wykładzie poznaliśmy Dylemat Więźnia – grę, w której dwóch graczy zostało aresztowanych i muszą niezależnie od siebie podjąć decyzję czy zeznawać przeciwko drugiemu graczowi czy z nim współpracować. Przyjmijmy następującą macierz wypłat:

	B współpracuje	B zdradza
A współpracuje	A dostaje -1 B dostaje -1	A dostaje -12 B dostaje 0
A zdradza	A dostaje 0 B dostaje -12	A dostaje -6 B dostaje -6

Jak zauważyliśmy na wykładzie, jedynym stanem równowagi Nasha w tej grze jest stan, w którym oboje gracze zeznają przeciw sobie. Jest to jedyny stan, w którym żadnemu z graczy nie opłaca się zmieniać swojej strategii.

Na tych zajęciach rozważymy **iterowany dylemat więźnia**, w którym gracze wielokrotnie rozgrywają powyższą grę. Taka gra dopuszcza więcej strategii, gdyż gracze mogą podejmować decyzję na podstawie poprzednich zagrań ich rywali. Przykładem jest strategia „*tit-for-tat*”, w której gracz w pierwszym ruchu współpracuje, a w kolejnych ruchach powtarza poprzedni ruch przeciwnika (czyli zdradza jeśli przeciwnik w poprzednim ruchu zdradził, lub współpracuje jeśli przeciwnik w poprzednim ruchu współpracował).

Celem tych zajęć jest zaimplementowanie tej gry w Pythonie i przetestowanie wybranych strategi metodą Monte Carlo. Gracze będą wielokrotnie dobierani w losowe pary, w których będą rozgrywać grę w iterowany dylemat więźnia. Średnia wartość uzyskanego wyniku pozwoli nam porównać stosowane przez graczy strategie.

1.1 Implementacja strategii

Zaczniemy od napisania funkcji `chooseMove`, która będzie określać ruch wykonany przez danego gracza, w zależności od stosowanej przez niego strategii. Rozważymy trzy strategie:

- `"always_cooperate"` - gracz zawsze będzie współpracować,
- `"always_defect"` - gracz zawsze będzie zdradzać,
- `"tit-for-tat"` - gracz zawsze będzie współpracować w pierwszym ruchu, a następnie będzie kopiował poprzedni ruch drugiego gracza.

Funkcja `chooseMove` przyjmie dwa argumenty:

a) `strategy` - jedną z powyższych strategii w formie napisu,

b) `opponentLastMove` - ostatni ruch przeciwnika (`"cooperate"` w przypadku współpracy, `"defect"` w przypadku zdracy, i `""` podczas pierwszego ruchu.

Funkcja `chooseMove` na wyjściu zwróci wybrany przez gracza ruch — albo `"cooperate"` albo `"defect"`. Funkcję implementujemy następująco:

```
def chooseMove(strategy, opponentLastMove):

  if strategy == "always_cooperate":
    return "cooperate"

  elif strategy == "always_defect":
    return "defect"

  elif strategy == "tit-for-tat":
    # STRATEGIA DO IMPLEMENTACJI

  else:
    print("unknown strategy")
```

Zadanie 1.1.1

Uzupełnij powyższy kod implementacją strategii `"tit-for-tat"`.
Przetestuj ją w następujących przypadkach:

```
print(chooseMove("tit-for-tat", ""))              # pierwszy ruch
print(chooseMove("tit-for-tat", "cooperate"))     # przeciwnik współpracuje
print(chooseMove("tit-for-tat", "defect"))        # przeciwnik zdradził
```

1.2 Implementacja gry

Czas zaimplementować grę w iterowany dylemat więźnia. W tym celu napiszemy funkcję `playGame(strategyA, strategyB, nTurns)`, która dla podanej strategii gracza A oraz gracza B, rozegra podaną liczbę rund. Wynik każdego gracza będzie sumowany, a następnie zwracany przez funkcję.

Kod wygląda następująco:

```
def playGame(strategyA, strategyB, nTurns):
  scoreA = 0              # wynik gracza A
  scoreB = 0              # wynik gracza B
  previousMoveA = ""      # poprzedni ruch gracza A
  previousMoveB = ""      # poprzedni ruch gracza B

  for i in range(nTurns):

    # Gracze wybierają swój ruch
    moveA = chooseMove(strategyA, previousMoveB)
    moveB = chooseMove(strategyB, previousMoveA)

    # Poprzedni ruch jest aktualizowany
    previousMoveA = moveA
    previousMoveB = moveB

    # Wynik każdego z graczy jest aktualizowany
    if moveA == "cooperate" and moveB == "cooperate":
      scoreA += -1
      scoreB += -1

    # POZOSTAŁE PRZYPADKI DO IMPLEMENTACJI

  return scoreA, scoreB
```

Zadanie 1.2.1

1. Uzupełnij powyższy kod pozostałymi możliwymi wynikami gry (gracz A współpracuje, gracz B zdradza itd.).
2. Przetestuj algorytm w następujących przypadkach:

```
print(playGame("always_cooperate", "always_defect", 100))
print(playGame("tit-for-tat", "always_cooperate", 100))
print(playGame("tit-for-tat", "always_defect", 100))
```

1.3 Testy strategii

Teraz napiszemy symulację Monte Carlo turnieju, w którym będzie po czterech graczy stosującą każdą z powyżej zaimplementowanych strategii.

Krok 1. Wprowadźmy listę strategii stosowanych przez poszczególnych graczy:

```
strategies = 4 * ["tit-for-tat"] + 4 * ["always_defect"] + 4 *
 ↪  ["always_cooperate"]
```

Krok 2. Wprowadźmy zmienną z liczbą graczy oraz pustą listę, w której będziemy wpisywać wyniki dla każdego z graczy:

```
n_players = len(strategies)
score = [[] for _ in range(n_players)]
```

Krok 3. W pętli rozegramy 10 000 rund. W każdej rundzie losujemy indeksy dwóch graczy (z przedziału 0-11), którzy będą ze sobą grać. Pamiętaj, że funkcja `random.sample` wymaga zaimportowania pakietu komendą `import random`.

```
for i in range(10000):
  playerA, playerB = random.sample(range(n_players), 2)
```

Krok 4. Rozegrajmy grę dla wylosowanych graczy:

```
  scoreA, scoreB = playGame(strategies[playerA], strategies[playerB], 100)
```

Krok 5. Zapisz wynik każdego z graczy:

```
  score[playerA].append(scoreA)
  score[playerB].append(scoreB)
```

Krok 6. Po zakończeniu pętli wyświetlamy wynik wszystkich graczy:

```
for i in range(n_players):
  print(strategies[i] + ": " + str(sum(score[i]) / len(score[i])))
```

Cały program powinien wyglądać następująco:

```
import random

strategies = 4 * ["tit-for-tat"] + 4 * ["always_defect"] + 4 * ["always_cooperate"]
n_players = len(strategies)
score = [[] for _ in range(n_players)]

for i in range(10000):
  playerA, playerB = random.sample(range(n_players), 2)
  scoreA, scoreB = playGame(strategies[playerA], strategies[playerB], 100)
  score[playerA].append(scoreA)
  score[playerB].append(scoreB)

for i in range(n_players):
  print(strategies[i] + ": " + str(sum(score[i]) / len(score[i])))
```

Zadanie 1.3.1

Przetestuj powyższy program. Gracze stosujący którą strategię osiągnęli odpowiednio najwyższy i najniższy wynik?

W przypadku iterowanego dylematu więźnia strategia `"tit-for-tat"` często okazuje się być zwykle najlepszą strategią.

Zadanie 1.3.2

Wymyśl jeszcze jedną strategię i zaimplementuj ją w powyższym programie. Sprawdź jak radzi sobie ona w porównaniu do pozostałych strategii. Podziel się obserwacjami z prowadzącym i resztą grupy.

2 Turniej gry w dobra publiczne

Na wykładzie poznaliśmy grę dla wielu graczy w dobra publiczne, w których każdy z graczy mógł wpłacić od 0 do 10 złotych do dóbr publicznych. Następnie całkowita kwota wpłacona do dóbr publicznych jest mnożona razy 2 i dzielona po równo pomiędzy wszystkich graczy.

Rozważmy iterowaną wersję w tej gry, w których **pięciu** graczy w kolejnych rundach decyduje o wpłaceniu od 0 do 10 złotych do dóbr publicznych. W każdej rundzie każdy gracz wie, ile pozostali gracze wpłacili do puli.

Zadanie 2.0.1

1. Napisz funkcję `chooseMove(opponentsLastMove, historyOfMoves)`.

   - W argumencie `opponentsLastMove` fukcja otrzymuje listę z kwotami wpłaconymi przez pozostałych czterech graczy w ostatniej rundzie, np. `[2,5,10,0]`. Podczas pierwszej rundy argument `opponentsLastMove` jest pustą listą `[]`.
   - W argumencie `historyOfMoves` fukcja otrzymuje listę, w której na kolejnych pozycjach znajdują się listy ze wszystkimi ruchami wszystkich graczy w kolejnych rundach, np. `[ [5,2,5,10,0], [4,6,3,2,3], [0,0,10,10,6] ]`. Wyniki są posortowane chronologicznie (od pierwszej do ostatniej rundy). Ruch wykonany przez wasz algorytm znajduje się na pierwszej pozycji. Podczas pierwszej rundy argument `historyOfMoves` jest pustą listą `[]`.
   - Funkcja nie musi wykorzstywać obu argumentów, na przykład może bazować tylko na ostatnim ruchu graczy, a nie na całej historii ruchów. Oba argumenty muszą jednak być podane w definicji funkcji.
   - Funkcja `chooseMove` powinna zwrócić wartość z przedziału $[0, 10]$, określającą kwotę, którą gracz przekaże do puli dóbr publicznych. Zwracaną wartością może być ułamek, np. 7.9.

   Przykładowa funkcja może być następująca:

```
def chooseMove(opponentsLastMove, historyOfMoves):
  if len(opponentsLastMove) == 0: # w pierwszej rundzie lista jest pusta
    return 5                      # wówczas wpłacamy 5 do dóbr publicznych
  else:
    return max(opponentsLastMove) # w kolejnych rundach wpłacamy
                                  # najwyższą kwotę wpłaconą
                                  # przez pozostałych graczy
```

2. Przetestuj swoją funkcję w następujących przypadkach:

```
chooseMove([], [])
chooseMove([0,0,0,0], [[5,2,3,5,8]])
chooseMove([10,10,10,10], [[5,0,10,8,7], [10,2,3,8,10]])
chooseMove([0,2,5,10], [[5,2.5,8.5,10,10], [10,0,0,10,5], [10,4,0,10,0]])
```

   W każdym przypadku funkcja musi zwracać wartość z przedziału $[0, 10]$.

3. **Wyślij swój program w wiadomości przez Discord do Piotra Morawieckiego w wiadomości zamieszczając: 1) swoje imię i nazwisko, 2) numer grupy, 3) wymyśloną nazwę bota, oraz 4) zaimplementowaną funkcję (jako link lub załącznik).**

   Każdy uczestnik możne wysłać aż do trzech różnych implementacji funkcji `chooseMove` — wszystkie trzy mogą wziąć udział w turnieju. Za wysłanie co najmniej jednej działającej funkcji, uczestnik otrzyma 2 punkty. Za zajęcie pierwszych miejsc będą przyznawane specjalne nagrody na zakończeniu tegorocznej edycji Matematyki dla Ciekawych Świata. Termin wysyłania programów mija 2 czerwca.

3 Zadania dodatkowe

Zadanie 3.0.1 (współpraca popłaca)

W turnieju iterowanego dylematu więźnia zaimplementowanym na naszych zajęciach, zmień liczbę graczy stosujących poszczególne strategie, tak żeby gracze stosujący strategię `"always_cooperate"` uzyskiwali wyższe wyniki niż gracze stosujący strategię `"always_defect"`.

Zadanie 3.0.2 (agresja i przebaczenie)

1. W turnieju iteracyjnego dylematu więźnia zaimplementuj dwie dodatkowe strategie:
   - `"aggresive_tit-for-tat"` – gracz zawsze będzie **zdradzać** w pierwszym ruchu, a następnie będzie kopiował poprzedni ruch drugiego gracza,
   - `"forgiving_tit-for-tat"` – gracz zawsze będzie **współpracować** w pierwszym ruchu. W następnych ruchach, jeśli przeciwnik w poprzednim ruchu współpracował to gracz też będzie współpracował. Za to jeśli przeciwnik w poprzednim ruchu zdradził, to gracz z prawdopodobieństwem 90% zdradzi, a z prawdopodobieństwem 10% będzie współpracował.
2. Zasymuluj turniej, w którym czterech graczy stosuje każdą z pięciu zaimplementowanych strategii. Która strategia osiąga najwyższe wyniki?

Zadanie 3.0.3 (uczciwszy turniej)

Pewną wadą zaimplementowanego na zajęciach turnieju jest to, że nie wszyscy gracze zostają tyle samo razy wylosowani. W skrajnym wypadku może okazać się, że nie którzy z graczy w ogóle nie zostaną wylosowani.
Zmodyfikuj turniej w taki sposób, żeby każdy z graczy został wylosowany tyle samo razy. Można to osiągnąć na kilka sposobów – wybór implementacji należy do ciebie. Na końcu swojego programu umieść następujący kod do wyświetlania ile razy każdy z graczy został wybrany:

```
for i in range(n_players):
  print("Gracz " + str(i) + " został wybrany " + str(len(score[i])) + " razy.")
```

Dla uproszczenia załóż, że liczba graczy w turnieju jest parzysta.

Zadanie 3.0.4 (symulacja gry w cykora)

Rozważ iterowaną wersję gry w cykora z wykładu. Każdy z dwóch graczy decyduje czy jedzie na czołowe zderzenie z drugim graczem czy ucieka. Przykładowa macierz wypłat jest przedstawiona poniżej.

	Gracz A ucieka	Gracz A jedzie do końca
Gracz B ucieka	A nie dostaje punktów B nie dostaje punktów	A dostaje 1 pkt B traci 1 pkt
Gracz B jedzie do końca	A traci 1 pkt B dostaje 1 pkt	A traci 10 pkt B traci 10 pkt

1. Zaproponuj 5 przykładowych strategii, które mogą stosować graczę w iterowanej grze w cykora.
2. Wykorzystując symulację turnieju podobną do tej napisanej powyżej, przetestuje które strategie dają najlepszy rezultat przy założeniu, że każdą strategię stosuje taka sama liczba graczy.

4 Praca domowa nr 4

Rozwiązania zadań domowych należy przesłać do czasu następnych ćwiczeń. Prowadzący może również zmienić ostateczny termin przesłania pracy domowej, np. na inną godzinę lub dzień.

Za pracę domową można maksymalnie dostać 6 punktów. Można wybierać zarówno łatwiejsze zadania, jak i trudniejsze, za większą liczbę punktów. Można również rozwiązać zadania za większą liczbę punktów, np. za 10. Jeśli wtedy poprawnie będą rozwiązane zadania za 5 punktów, to zostanie przydzielone 5 punktów. Jeśli będą poprawnie rozwiązane zadania za więcej niż 6 punktów, np. 8 czy 10, to taka osoba otrzyma maksymalnie 6 punktów.

Pamiętaj, żeby notebook był czytelny – każde zadanie powinno być umieszczone w osobnym bloku tekstowym oraz poprzedzone numerem (i opcjonalnie opisem) zadania. Komentarze są mile widziane.

Zadanie 4.0.1 — 6 pkt (przyjęcie urodzinowe)

W tym zadaniu rozważymy iteracyjną grę "przyjęcie", w której dwóch graczy musi zdecydować co przynieść na urodziny znajomej, czy tort, czy napoje. Każdy z lenistwa wolałby kupić napoje niż upiec tort, ale zarówno tort jak i prezenty są potrzebne do udanego przyjęcia. Macierz wypłat dla tej gry jest przedstawiona poniżej.

	Gracz A piecze tort	Gracz A kupuje napoje
Gracz B piecze tort	A dostaje 0 pkt B dostaje 0 pkt	A dostaje 2 pkt B dostaje 1 pkt
Gracz B kupuje napoje	A dostaje 1 pkt B dostaje 2 pkt	A dostaje 0 pkt B dostaje 0 pkt

1. Zmodyfikuj funkcję `score()` z zajęć, tak żeby odpowiadał powyższej macierzy wypłat.
2. Zmodyfikuj funkcję `move()` z zajęć, tak żeby:
   - 10 graczy zawsze kupowało napoje,
   - 10 graczy zawsze kupowało tort,
   - 10 graczy z prawdopodobieństwem 2/3 kupuje napoje i z prawdopodobieństwem 1/3 kupuje tort[a],
   - 10 graczy zmieniało swój poprzedni ruch na przeciwny, jeśli w poprzedniej rundzie wybrali tę samą akcję co przeciwnik (np. jeśli w poprzedniej rundzie obaj gracze zagrali tort, to teraz gracz zagra napój); w pierwszym ruchu ci gracze losowo wybierają tort (50%) lub napoje (50%).
3. Zasymuluj turniej i przedstaw wyniki w formie tekstowej. Skomentuj, które strategie osiągnęły najwyższą średnią wyników.

a. ta strategia mieszana spełnia równowagę Nasha