Ćwiczenie 3

Dwuosobowe gry deterministyczne

Adam Nowakowski (300428)

1. Zadanie

Zadaniem była implementacja algorytmu *minimax*, gdzie w przypadku ruchów równie dobrych powinien być wybrany losowo jeden z nich. Poza tym należało napisać grę w *kółko i krzyżyk* umożliwiającą użytkownikowi grę z komputerem i przebadać parametr d głębokości przeszukiwania na tym przykładzie.

2. Wprowadzenie teoretyczne

Gra kółko i krzyżyk należy do deterministycznych gier dwuosobowych o sumie zerowej i pełnej informacji. Wyjaśnijmy sobie powyższe pojęcia:

- **determinizm** w przypadku gier oznacza tyle, że nie występuje w nich element hazardu (losowości). Gracz aby wygrać jest zdany tylko na swoje umiejętności, a nie na szczęście.
- **gra o sumie zerowej** – podczas rozgrywki jesteśmy jednoznacznie określić jej wynik. W tym przypadku jest to wygrana gracza, wygrana przeciwnika, czy remis. Możemy wtedy do każdych z tych zdarzeń przypisać wartość - **wypłatę** np.:
  - wygrana gracza (1)
  - wygrana przeciwnika (-1)
  - remis (0)

  Gracz w takim wypadku będzie dążył, aby wypłata była jak największa, a przeciwnik wręcz odwrotnie.
- **gra z pełną informacją** – każdy z graczy ma pełną wiedzę o tym, jaki jest stan gry. Grając np. w pokera nie mamy takiej wiedzy – nie widzimy, jakie karty mają przeciwnicy.

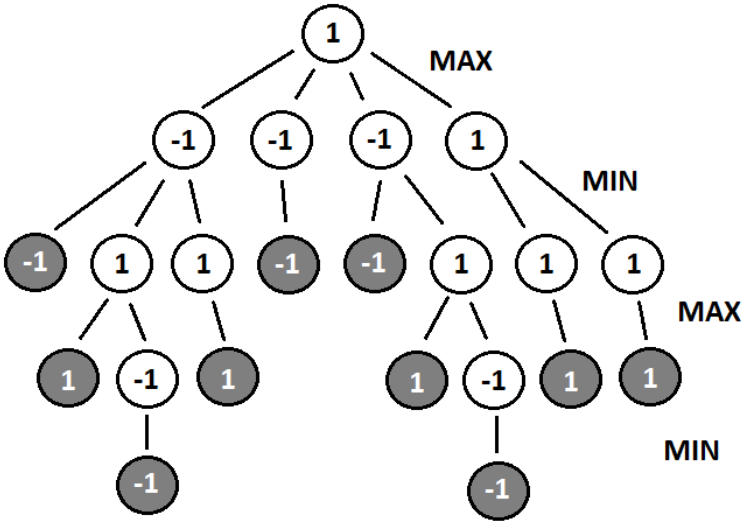
W ogólności takie gry możemy zamodelować zestawem pięciu parametrów $\langle S, P, s_0, T, w \rangle$, gdzie:

- $\boldsymbol{s} \in \boldsymbol{S}$ jest stanem, czyli informacją co się dzieje w grze oraz kto wykonuje ruch. W przypadku naszej gry jest to ustawienie znaków na planszy.
- $\boldsymbol{p} \in \boldsymbol{P}$ jest funkcją następnika, która mając na wejściu dany stan przekształca go na poprawny stan (w ogólności stany) następny. Reprezentuje reguły gry.
- $\boldsymbol{s_0}$ to stan początkowy.
- $\boldsymbol{T} \subseteq \boldsymbol{S}$ jest zbiorem stanów terminalnych – kończących daną grę.

- w – funkcja wypłaty. Przypisuje stanom terminalnym wypłatę.

Do znalezienia optymalnej strategii, podczas gier takich jak kółko i krzyżyk, używa się algorytmu minimax. Załóżmy, że rozpoczynający grę dąży do maksymalizacji swojej wypłaty. Nazwiemy go wtedy graczem **Max**, a przeciwnika **Min**. Startujemy ze stanu początkowego i tworzymy stany następne za pomocą funkcji następnika. Wiedząc, że porusza się wtedy Max, algorytm ze stanów następnych wybiera ten, którego wypłata jest największa i przypisuje stanu początkowemu właśnie tą wypłatę. W przypadku gracza Min jest odwrotnie – bierze najmniejszą wartość. Lecz nasz algorytm nie wie w pierwszej iteracji jaka jest wypłata w następnikach, więc wywołuje się na stanach kolejnych, aż dojdzie do stanów terminalnych. Tam w końcu za pomocą funkcji wypłaty przypisuje do nich wartość.

Poniżej przykładowe drzewo gry. Stany terminalne zostały zaznaczone kolorem siwym, a wypłaty zostały wpisane w kółka.



Jednak, gdy są bardziej skomplikowane gry, jak szachy, czy warcaby, to przeszukanie całego drzewa gry nie jest możliwe, lub wymaga bardzo dużych zasobów obliczeniowych. Stosuje się wtedy algorytm minimax z głębokością d. Drzewo przeszukuje się wtedy na głębokość d warstw w głąb. Funkcje wypłaty zastępuje się wtedy pewną heurystyką, która mówi nam jaki ruch (który nie doprowadza jeszcze do stanu terminalnego) może być lepszy a który gorszy. Wartości które zwraca nam heurystyka powinny być pewną wartością pośrednią pomiędzy wypłatą Min a wypłatą Max.

3. Implementacja

W katalogu `/implementacja` znajdują się dwa skrypty w kodzie Pythona:

- *`minimax.py`* – zawiera implementację algorytmu minimax.
- *`game.py`* – zawiera klasę, w której zaimplementowana jest gra kółko i krzyżyk oraz przykładowe stworzenie jej instancji.
    - `successor_fun` – generuje z podanego stanu gry stany kolejne według zasad. Stanom terminalnym przyporządkuje wypłatę i sposób zakończenia:
        - wygrał ‘x’
        - wygrał ‘o’
        - remis
    - `heuristics` – fumkcja

Klasa `CirclesAndCrosses` zawiera metody, które opisują grę według zestawu pięciu parametrów wymienionych w punkcie drugim.

- Stan początkowy jest tworzony w `__init__()`. Przypisuje mu się pustą planszę, ustawia się flagę oznaczającą ruch gracza Max na wartość `True`, a flagę terminalności stanu na `False`.
- Metoda `successors_fun()` pełni funkcję generatora następnych stanów gry z jednego podanego. Jeśli któryś z tych stanów okaże się być terminalnym, przypisuje mu od razu odpowiednią wypłatę:
    - wygrywa gracz Max (10)
    - wygrywa gracz Min (-10)
    - remis (0)

  Aby zachować strukturę drzewa, do stanu wejściowego dołączone są stany następne.
- Metoda `heuristics()` zwraca dla podanego stanu pośrednią wartość wypłaty według poniższego algorytmu:
    - dla każdego pola planszy przypisana jest waga wg. macierzy:

$$H = \begin{bmatrix} 3 & 2 & 3 \\ 2 & 4 & 2 \\ 3 & 2 & 3 \end{bmatrix}$$

    - zwracana jest suma wag pól na których postawiony jest znak Maxa (‘o’), pomniejszona o sumę pól na których postawiony znak Mina (‘x’)

Poza tym są jeszcze metody `player_move()` i `opponent_move()`, które modyfikują aktualny stan, na rzecz stanu, do którego chcemy przejść za pomocą ruchu gracza (wpisywanie w konsoli ruchu), czy przeciwnika (algorytmu minimax).

W metodzie `oponent_move()` zaimplementowana jest losowość wybierania ruchów o identycznej wypłacie.

Metoda play() odpowiedzialna jest za rozpoczęcie rozgrywki, wypisywanie na ekran aktualnej planszy gry oraz losowe wybranie gracza, który zaczyna.

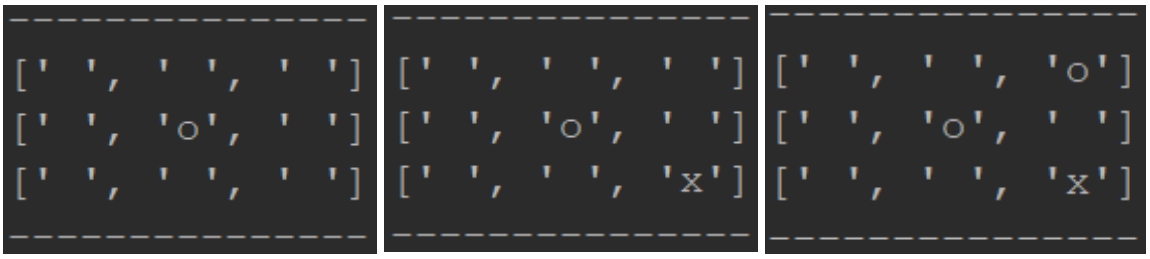
4. Badanie parametru d w algorytmie minimax na przykładzie gry w kółko i krzyżyk.

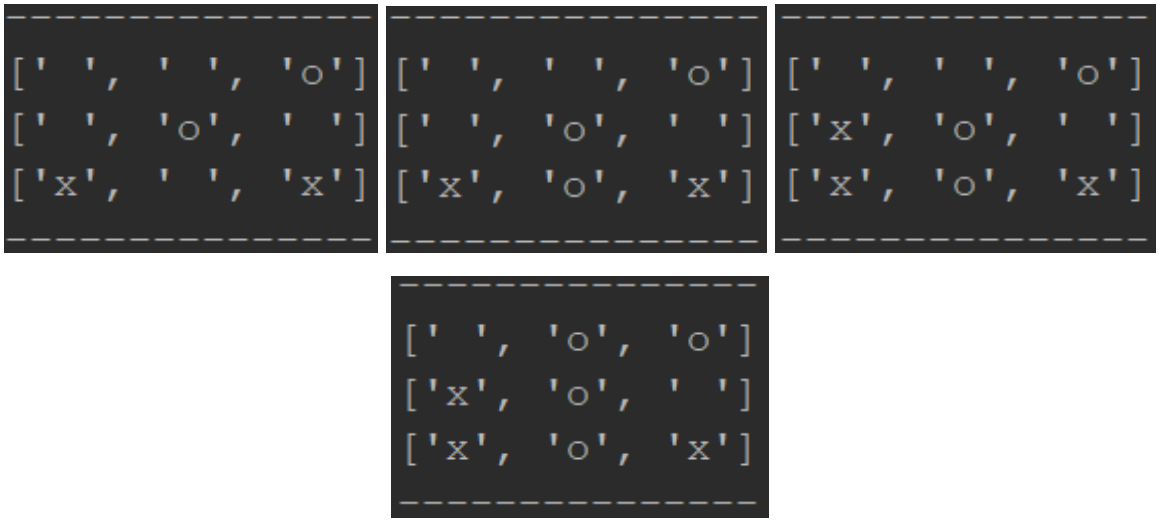
Parametr d określa nam, na jaką głębokość przeszukiwane jest nasze drzewo gry. Im większy, tym lepiej działa nasz algorytm – przewiduje więcej ruchów w przód. Niestety spada wtedy szybkość obliczenia takiego ruchu. Poniżej znajduje się przebieg kilku rozgrywek:

- Dla wartości $d = 1$:
  - zaczął algorytm

Jak widzimy algorytm w tym przypadku nie był zbytnio „mądry”. Na piątym screenshocie widzimy, że popełnił błąd. Zamiast postawić znak na polu 1-2 i zablokować przeciwnika, zajął pole 3-3. Wynika to z tego, że algorytm patrzy tylko na następny ruch, czyli kieruje się wskazaną przez nas heurystyką. Wybrał je, gdyż miało większą wagę, niż to które powinien wybrać. Trudno mu się dziwić, bo nie miał informacji o tym, że jego przeciwnik może w następnym ruchu wygrać.

- Dla wartości $d = 2$
  - zaczął algorytm





Jak widzimy, w tym przykładzie algorytm już nie popełnił tego samego błędu, co wcześniej. Zablokował mój ruch i wykorzystując mój błąd wygrał. Udało mi się jednak na tym poziomie znaleźć słaby punkt algorytmu:

- zaczął gracz

Algorytm popełnia błąd na screenshocie czwartym. Jest to spowodowane tym, że i w tym przypadku nie mógł przewidzieć mojej wygranej. Dzieje się ona dopiero na czwartym poziomie w głąb w momencie, gdy minimax przewidywał ruchy. Aby wyeliminować ten błąd wartość parametru d powinna wynosić co najmniej 4. Co się sprawdza, gdyż i przy $d = 3$ algorytm wpada w tą samą pułapkę. Stany związane z kolejnym ruchem na rogach mają takie same wypłaty, więc jeśli wylosowane zostanie to nieszczęsne pole, algorytm może przegrać.

Ustawiając parametr d algorytmu minimax należy znaleźć pewien kompromis pomiędzy jakością naszej strategii a czasem wykonywania się algorytmu. Moglibyśmy ustawić $d = 9$, gdyż algorytm przeszukuje wtedy całe drzewo gry. Niestety znacznie to wydłuża czas obliczeń. W przypadku gry kółko i krzyżyk optymalną wartością jest $d = 4$. Zapewniamy wtedy, że nasz algorytm na pewno nie przegra.

Ciekawe wyniki daje nam pełne przeszukiwanie drzewa gry. Do tej pory gdy zaczynał algorytm, to stawiał swój znak na środku. Ten z pełną informacją stawia go w losowym miejscu. To samo się dzieje, gdy gracz postawi swój znak na polu o wadze 2. Algorytm losowo wybiera wtedy pole z całej planszy.