Sztuczna inteligencja
Pracownia 1
Zajęcia 1 i 2

Można używać dowolnego języka programowania. Modelowe rozwiązania pisane były w Pythonie. Terminy podane były na forum aktualności. Jest to wersja poprawiona, w której uściślone zostały pewne drobiazgi i format zadań został dostosowany do sprawdzaczki.

Zadanie 1. (4p) Będziemy rozważać końcówki szachowe, w których jedna strona (czarne) dysponuje tylko królem, a druga królem i wieżą. Twoim zadaniem będzie podanie dla danej sytuacji na planszy minimalnej liczby ruchów dzielących tę sytuację od mata dla białych. Przy czym zakładamy, że czarny król współpracuje z białymi (w tradycji szachowej takie zadania nazywa się matem pomocniczym lub kooperacyjnym).

Przypominamy, że wieża porusza się o dowolną liczbę pól w wierszu lub kolumnie, król porusza się o jedno pole (w pionie, poziomie lub po skosie). Szach to 'zagrożenie biciem' króla, mat to zagrożenie biciem, w którym strona matowana nie może tego zagrożenia usunąć. Dodatkowo król nie może wchodzić na pole szachowane (uwaga dla szachistów: zakładamy, że białe zrobiły roszadę).

Sytuacja na planszy podawana jest w pliku tekstowym (`zad1_input.txt`), w którym po kolei podawany jest kolor gracza, który ma ruch, pozycja białego króla, pozycja białej wieży i pozycja czarnego króla. Przykładowe dwie sytuacje (i odpowiadające im dwa wiersze danych).

```
black c4 c8 h3
white a1 b2 c3
```

Twój program powinien działać w dwóch trybach:

1) wsadowym, w którym czyta kolejne sytuacje z pliku z danymi i wypisuje do pliku (`zad1_output.txt`)
2) w trybie 'debug', w którym jest możliwość obejrzenia ruchów prowadzących do mata (kolejne stany gry możesz „narysować" za pomocą instrukcji `print`, nie jest konieczne prawdziwe rysowanie). Możesz również skorzystać z kodu wizualizującego pozycję na planszy.

Wskazówka (rot13.com): mnqmvnłn ghgnw cebfgr cemrfmhxvjnavr jfmrem (OSF).

Zadanie 2. (3p) W tym zadaniu zajmiemy się rekonstrukcją tekstu. Twoim zadaniem jest napisanie funkcji, która dla zbioru słów S (znajdującego się w pliku `polish_words.txt` i tekstu niezawierającego spacji (t) podaje podział tego tekstu na słowa (a dokładniej zwraca ten tekst z dodanymi spacjami). Podział wyznaczony jest zgodnie z następującymi zasadami:

1. Po usunięciu spacji powinniśmy otrzymać oryginalny tekst,
2. każde słowo (czyli ciąg znaków otoczony spacjami) powinno należeć do zbioru S,
3. spośród możliwych podziałów należy wybrać ten, który maksymalizuje sumę kwadratów długości słów (oznacza to, że preferujemy długie słowa).

Dla tekstu: `tamatematykapustkinieznosi` wynikiem działania tej funkcji powinno być `"ta matematyka pustki nie z`
a nie np. `"tama tematy kapustki nie z nosi"`.

Program powinien działać tak, że z pliku `zad2_input.txt` wczytuje kolejne wiersze (kodowane UTF-8) i wypisuje optymalne podziały na wyrazy dla tych wierszy do pliku `zad2_output.txt`

Program powinien działać szybko, to znaczy przetworzyć X zdań w mniej więcej Y sekund (TODO)

Zadanie 3. (3p) Przeczytaj o układach pokerowych na przykład na stronie https://pl.wikipedia.org/wiki/Poker. Będziemy rozważać wariant pokera, w którym:

1. Nie ma żadnego dobierania, gracze losują 5 kart i porównują siłę swoich układów.
2. Jeden z graczy, Figurant, losuje 5 kart z talii zawierającej tylko asy, króle, damy i walety.
3. Drugi z graczy, Blotkarz, losuje 5 kart z talii zawierającej tylko blotki (dwójki, trójki, ..., dziesiątki)

Napisz program, który umożliwia szacowanie szansy zwycięstwa Blotkarza w starciu z Figurantem. Zwróć uwagę, że nie musisz implementować pełnych zasad porównywania układów pokerowych (dlaczego?).

Sprawdź za pomocą tego programu (wykonując kilka eksperymentów), jak zmienia się prawdopodobieństwo sukcesu, jeżeli pozwolimy Blotkarzowi wyrzucić pewną liczbę wybranych kart przed losowaniem (inaczej mówiąc, pozwalamy Blotkarzowi na skomponowanie własnej talii, oczywiście złożonej z blotek).

Czy potrafisz skomponować zwycięską talię dla Blotkarza (mającą możliwie dużo kart)?

Wskazówka: anwcebfgfmlz fcbfborz ebmjvąmnavn grtb mnqnavn wrfg ancvfnavr cebtenzh, xgóel jvrybxebgavr ybfhwr hxłnql xneg, npmxbyjvrx fą grż vaar, b xgóelpu cbjvrzl an ćjvpmravnpu.

Zadanie 4. (2p) Przeczytaj, czym jest łamigłówka nazwana *obrazki logiczne*. (na przykład `https://www.wydawnictwologi.pl/obrazki-logiczne`). Będziemy rozważać jej uproszczony wariant, w którym w każdym wierszu lub kolumnie znajduje się co najwyżej 1 blok czarnych kwadracików (zatem do opisu wiersza czy kolumny wystarcza jedna liczba, 0 oznacza nieistnienie żadnego bloku).

Napisz funkję `opt_dist`, która bierze jako argument listę liczb 0/1 i liczbę naturalną D (długość bloku), a następnie zwraca minimalną liczbę operacji zamiany bitu, która da jeden ciąg jedynek o długości D otoczony zerami **(lub przylegający do krańców listy)**. Przykładowo, funkcja `opt_dist` dla argumentów

```
0010001000 i 5 powinna zwrócić 3
0010001000 i 4 powinna zwrócić 4
0010001000 i 3 powinna zwrócić 3
0010001000 i 2 powinna zwrócić 2
0010001000 i 1 powinna zwrócić 1
0010001000 i 0 powinna zwrócić 2
```

Zadanie 5. (5p) Napisz program rozwiązujący **uproszczone** obrazki logiczne, realizujący następujący algorytm (inspirowany algorytmem WalkSat, o którym będziemy jeszcze mówić).

- Zaczynamy od pewnego przypisania koloru wszystkim polom obrazka (np. losowo, albo same zera).
- W trakcie działania algorytm będzie on zmieniał kolor wybranego pola obrazka (jednego), aż do momentu, w którym obrazek będzie gotowy
- Wybór pola realizujemy w następujący sposób:
  - Losujemy wiersz (lub kolumnę), która jest niezgodna ze specyfikacją (tzn. czarne piksele nie tworzą jednego bloku o wymaganej długości)
  - Wybieramy piksel o współrzędnych (i,j), którego zmiana w największym stopniu poprawi sumaryczny poziom dopasowania w wierszu i oraz w kolumnie j.
- Wszystkie wybory z niewielkim prawdopodobieństwem robimy nieoptymalnie (psujemy dobry wiersz, naprawiamy tylko wiersz, zamiast wiersza i kolumny, itd).
- Jeżeli nie otrzymaliśmy przez dłuższy czas rozwiązania, to losujemy jescze raz ustawienia początkowe.