# Sztuczna inteligencja
# Pracownia 2
# Zajęcia 4 i 5

## Obrazki logiczne

**Zadanie 1. (4p)** W zadaniu tym wracamy do obrazków logicznych, tym razem w pełnej wersji. Masz napisać program, który rozwiązuje pełne obrazki logiczne (czyli wczytuje opis i wypisuje odpowiadający mu obrazek).

**Komunikacja ze sprawdzarką** Dane wejściowe przekazywane są w pliku `zad_input.txt`. Format danych jest następujący:

```
<liczba-wierszy> <liczba-kolumn>
<opis-wiersza1>
<opis-wiersza2>
...
<opis-wierszaK>
<opis-kolumny1>
...
<opis-kolumnyM>
```

(opisy wierszy i kolumn są ciągiem oddzielonych spacją liczb)

Twój program ma wygenerować plik `zad_output.txt` zawierający odkodowany obrazek, w którym zera to `.` a jedynki to `#`.

Przykład wejścia:

```
5 5
5
1 1 1
3
2 2
5
2 2
1 3
3 1
1 3
2 2
```

Przykład wyjścia:

```
#####
#.#.#
.###.
##.##
#####
```

**Uwaga**: dane do tego zadania będą wielkości (maksymalnie) 15x15, w rozsądnym czasie (do minuty) powinien sobie z nimi poradzić program implementujący algorytm z listy P1 (z nową funkcją sprawdzającą dopasowanie wiersza/kolumny do opisu). **Uwaga 2**: Na SKOS pojawią się również trudniejsze dane, które nie są obowiązkowe dla tej listy. Jeżeli Twój program będzie umiał je rozwiązać, możesz liczyć na dodatkowe punkty „zaksięgowane" do listy P3.

## Sokoban

**Zadanie 2.** (**4p**)

Będziemy rozważać grę Sokoban[1] w której magazynier ma za zadanie ustawić skrzynki na zadanych pozycjach. Magazynier porusza się w czterach kierunkach (w górę, w dół, w prawo i w lewo). Jeśli przed magazynierem stoi skrzynka, może próbować ją przepchnąć ale nie może jej pociągnąć. Sokoban jest PSpace-zupełny i często nawet małe plansze wymagają heurystycznych metod poszukiwania. W tym zadaniu sprawdzimy jak działa BFS i A$^*$.

Zaimplementuj poszukiwanie BFS w którym rozważamy ruchy magazyniera. Następnie zaproponuj heurystykę i zaimplementuj poszukiwanie A$^*$.

**Komunikacja ze sprawdzarką** Opis planszy przekzaywany jest w pliku `zad_input.txt`. Plik przedstawia mapkę magazynu. Dla planszy o rozmiarze $N \times M$ plik zawiera $N$ wierszy, a każdy wiersz składa się z $M$ znaków opisujących pola planszy: `.` oznacza puste pole, `W` ścianę, `K` magazyniera, `B` skrzynkę, `G` pole docelowe, `*` skrzynkę na polu docelowym i `+` magazyniera stojącego na polu docelowym. Na planszy jet tyle samo skrzynek i pól docelowych oraz dokładnie jeden magazynier.

Znalezione rozwiązanie program ma zapisać w pliku `zad_output.txt` w postaci jednej linii zawierajacej ciąg znaków określających ruchy magazyniera: `U` w górę, `D` w dół, `L` w lewo i `R` w prawo.

Rozwiązanie będzie uznane za poprawne jeśli zwrócona zostanie jedna z najkrótszych sekwencji ruchów poprawnie ustawiających skrzynki.

Przykład wejścia:

```
WWWWWW
W.GWWW
W..WWW
W*K..W
W..B.W
W..WWW
WWWWWW
```

Przykład wyjścia:

```
DLURRRDLULLDDRULURUULDRDDRRULDLUU
```

[1] Np. `https://www.sokobanonline.com/`

Sprawdzarka umożliwia odegranie sekwencji ruchów jeśli uruchomimy ją z opcją `verbose`, np. `python validator.py --verbose zad2 python zad.py`.

**Zadanie 3.** (**5p)** Zaimplementuj poszukiwanie rozwiązań Sokobana używając przestrzeni w której modelujemy jedynie dopuszczalne ruchy skrzynek. Ponieważ gubimy informację o dokładnym położeniu magazyniera, nie jest możliwe znalazienie rozwiązań o najmniejszej liczbie ruchów magazyniera. Zamiast BFS lub A* możemy użyć więc algorytmu poszukiwania „best first search", w którym węzły rozwijane są w kolejności zadawnej przez funkcje heurystyczną.

W tym zadaniu wymagamy jedynie, by program znalazł rozwiązanie w limicie czasu, ale nie nakładamy żadnych warunków na jego długość. Oczywiście rozwiązanie podajemy używając ruchów magazyniera (a nie samych skrzynek).

**Komunikacja ze sprawdzarką** Program ma czytać planszę z pliku `zad_input.txt` i zapisywać wynik do pliku `zad_output.txt`. Formaty plików są identyczne z poprzednim zadaniem, przy czym znalezione rozwiązanie jest uznane za poprawne jeśli program zwróci jakąkolwiek (być może bardzo długą) sekwencję ruchów magazyniera poprawnie układającą skrzynki

## Komandos w labiryncie

**Zadanie 4. (4p)** Rozważamy komandosa, który porusza się w labiryncie (labirynt składa się z kwadratowych pól, tworzących prostokąt). Mamy następujące rodzaje pól:

1. ściany, oznaczane `#`, po których nie można się poruszać,
2. punkty docelowe (oznaczane `G`), do których należy dojść, żeby zdetonować ładunek nad podziemnymi magazynami wroga,
3. punkty startowe (oznaczane `S`), to w nich komandos może się znaleźć w pierwszej turze,
4. punkty startowo-docelowe (oznaczone `B`),
5. punkty pozostałe, oznaczone spacją.

Komandos może się poruszać w 4 kierunkach (UDLR, oznaczane jak w Sokobanie). Ruch w kierunku ściany nie wywołuje zmiany stanu. Komandos zostaje zrzucony w nocy i nie wie, gdzie dokładnie zrzut miał miejsce (w którym punkcie startowym), zna natomiast mapę labiryntu. Interesuje nas znalezienie sekwencji ruchów, która **na pewno** doprowadzi do któregoś stanu końcowego (czyli nasz żołnierz wykonuje swój plan, będący ciągiem ruchów, po których wykonaniu może odpalać bombę, bo niezależnie, gdzie znajdował się na początku wędrówki, pod koniec będzie w jednym z punktów docelowych. Taki plan będziemy nazywać *zwycięskim*.

Dane wejściowe przekazane w pliku `zad_input.txt` to opis labiryntu:

```
######################
# G   G              #
#    #       #S     #
#  S #       #      #
######### ############
#    G#           G  #
##   ##    ###########
#    #       S      #
# ## ###########     #
#    #            S  #
#S         ####      #
######################
```

Jak chcesz, to możesz założyć, że labirynt zawsze otoczony jest ścianami.

Program ma zapisać do pliku `zad_output.txt` jedną linię zawierającą rozwiązanie, przykładowo dla powyższego labiryntu `LLUULLULLLLURRULUUUULLLLLLLLRRRRR`. Po podaniu flagi `--verbose` sprawdzaczka pokazuje planszę po każdym kroku rozwiązania.

**Początek właściwego zadania 4:** Napisz program, który rozwiązuje zadanie z komandosem, czyli dla każdego przypadku testowego wypisuje zwycięski plan. W tym zadaniu plan nie musi być optymalny, wymagane jest jedynie, by był krótszy niż 150 ruchów (dla każdego przypadku). Rozwiązanie powinno mieć zaimplementowane dwie fazy:

1. wykonywanie losowych/zachłannych ruchów zmniejszających niepewność,
2. wykonanie przeszukiwania BFS (nie wolno korzystać z $A^*$).

Twoim zadaniem jest zatem między innymi sprawdzić, jaką niepewność[2] jest w stanie zaakceptować BFS i zaproponować schemat działań dla części pierwszej, który niepewność jest w stanie zredukować do wymaganego poziomu.

Ocena zależy liniowo od liczby przypadków testowych, które program obsłuży w limicie czasu.

**Zadanie 5. (4p)** Rozwiązujemy to samo zadanie, co powyżej, ale wykorzystując $A^*$. Dodatkowo wymagamy, by tym razem zwycięski plan był optymalny (nie dłuższy od żadnego innego zwycięskiego planu). Testy dla tego zadania będą tak skonstruowane, że nie będzie konieczny etap 1, zmniejszający niepewność.

Ocena zależy liniowo od liczby przypadków testowych, które program obsłuży w limicie czasu.

**Zadanie 6. (2p)** Zaproponuj sensowną modyfikację heurystyki z poprzedniego zadania, by stała się ona niedopuszczalna. Heurystyka powinna mieć parametr (nazwijmy go: stopień niedopuszczalności), sprawdź jaki zysk czasowy udaje Ci

[2]Liczba możliwych położeń komandosa. W oczywisty sposób im większa ta liczba, tym większa przestrzeń stanów.

się osiągnąć dla różnych stopni niedopuszczalności, oraz jaki koszt za to płacisz w sumarycznej długości znalezionych planów.

Zadanie testujemy na tych samych testach co poprzednie. Wskazówka: istnieje rozwiązanie, które wymaga ekstremalnie mało dodatkowego kodu.

**Zadanie 7. (1p)** To zadanie jest połączeniem trzech poprzednich zadań. Powinieneś zaproponować jakąś kombinację metod z tych zadań, która:

a) Jest w stanie rozwiązać testy do zadania 4 w limicie czasu,

b) Osiąga lepsze wyniki (w sensie sumy długości planów) niż te z zadania z BFS-em.

Dodatkowe punkty za to zadanie zostaną przyznane podczas pierwszego tygodnia pracowni P3 i będą zależały od tego, jak dobrze (w sensie krótkości planów) radzi sobie Twój program na tle programów koleżanek i kolegów.

## Sprawdzaczka

Dla zadań 1, 2, 3, 4 i 5 przygotowaliśmy sprawdzaczkę. Zadanie 6 sprawdzamy testami dla zadania 5. Zadanie 7 sprawdzamy testami do zadań 4 i 5. Przykłady użycia sprawdzaczki:

```
1. uruchomienie wszystkich testów dla danego zadania:
python validator.py zad1 python rozwiazanie.py

2. uruchomienie wybranych testów
python validator.py --cases 1,3-5 zad1 a.out

3. urochomienie na innych testach
python validator.py --testset large_tests.yaml zad1 python rozwiazanie.py

4. Wypisanie przykadowego wejścia/wyjścia:
python validator.py --show_example zad1

5. Wypisywanie plansz dla Sokobana i Komandosa
python validator.py --verbose zad1 python rozwiazanie.py
```

Prosimy o przygotowanie rozwiązań w formacie komatybilnym ze sprawdzaczką.