GIPF

Zadanie składa się z dwóch części:

- Pierwszą jest implementacja silnika uogólnionej wersji gry. Uogólnienie wprowadza listę **L** parametrów opisujących grę. Pozwala to na łatwą zmianę charakterystycznych jej własności, np.: rozmiar planszy, liczba pionów wymaganych dla pewnej kombinacji wyzwalającej pewien efekt w grze, np. bicie/odzyskiwanie pionów, itp.
- Drugą jest implementacja programu rozwiązującego tę grę albo pewne jej stany (np. stany końcowe) dla różnych wartości parametrów listy **L**. Innymi słowy wariant gry opisany niskimi wartościami **L** będzie stanowił dużo mniejszy problem do całkowitego rozwiązanie gry, niż wariant opisany większymi wartościami **L** np. oryginalny. W tym drugim przypadku można próbować rozwiązywać stany gry gdzie "widać", że gra po kilku posunięciach powinna się zakończyć. Podstawowym algorytmem rozwiązującym będzie algorytm min-max (ew. jego wariacja negamax) wraz z przycinaniem alpha–beta. Na dodatkowe punkty można zaimplementować bardziej wyszukane usprawnienia, takie jak: pogłębianie iteracyjne, porządkowanie posunięć, z zastosowaniem Triangular PV-Table lub tablic transpozycji, itp. Alternatywnie można zaimplementować algorytm z rodziny best-first search, np. Proof Number Search w tym przypadku również można zdobyć dodatkowe punkty za implementacje jego bardziej wyrafinowanych wersji, np. PNS^2, Depth-first PN-Search (df-pn), Monte-Carlo PNS itp.

---------- The task consists of two parts:

- The first part is the implementation of the engine for a generalized version of the game. Generalization introduces a list **L** of parameters describing the game. This allows for easy modification of its characteristic properties, such as board size, the number of pieces required for a specific combination triggering a certain effect in the game, like capturing/recovering pieces, etc.
- The second part is the implementation of a program that solves the game or certain states of the game (e.g., end states) for different values of the parameters in the list **L**. In other words, a game variant described by lower values of **L** will pose a much smaller problem for solving the game completely than a variant described by higher values of **L**, such as the original version. In the latter case, one can attempt to solve game states where it is "visible" that the game should end after a few moves. The fundamental solving algorithm will be the minimax algorithm (or its variation, negamax), along with alpha-beta pruning. For extra points, you can implement more sophisticated enhancements, such as iterative deepening, move ordering, Triangular PV-Table or transposition tables, etc. Alternatively, you can implement an algorithm from the best-first search family, such as Proof Number Search. In this case as well, additional points can be earned for implementing its more refined versions, such as PNS^2, Depth-first PN-Search (df-pn), Monte-Carlo PNS, etc.

GIPF to gra deterministyczna, bez elementów losowych oraz ukrytej informacji dla dwóch graczy o sumie zerowej bez remisów. Zasady wersji oficjalnej dostępne są tutaj Zadanie polega na implementacji silnika tej gry dla jej uogólnionej wersji. Uogólnienie polega na tym, że można zmieniać właściwości gry opisane czteroma paramtrami (S, K, GW, GB) będącymi elementami wspomnianej wcześniej listy **L** parametrów opisujących grę:

- S - rozmiar planszy wyrażony liczbą pól planszy wchodzących w skład każdego boku sześciobocznej planszy;
- K - liczba pionów gracza która wyzwala zbieranie pionów, ta wartość nigdy nie powinna być mniejsza od dwóch ponieważ dla jedynki pozycja początkowa na planszy była by zabroniona (wszystkie piony należało by od razu usunąć). Wartość ta nie powinna być również większa od 2*S-1 ponieważ z tylu pól składa się "przekątna" planszy więc nie jest możliwe utworzenie dłuższego ciągu. Idąc dalej tym tropem można zauważyć, że wartość ta powinna być jeszcze mniejsza ponieważ w przypadku kiedy jest równa dokładnie 2*S-1 w grze nie można będzie zbijać pionków przeciwnika. Prowadzi to albo do niekończącej się rozgrywki polegającej na uwalnianiu w nieskończoność swoich pionów do rezerwy albo na zakończeniu rozgrywki poprzez zapchanie planszy, kiedy to żaden z graczy nie może dołożyć już żadnego piona do gry. Opisuje to jednoznacznie zasada z instrukcji: "Nie można wypchnąć piona poza obszar gry, czyli na kropkę po drugiej stronie linii.";
- GW - liczba pionów należących do gracza białego, musi być większa niż trzy które są na starcie umieszczane na planszy gracza. jednak w przypadku kiedy będzie ich mało rozgrywka może się szybko zakończyć z powodu wyczerpania zasobów gracza, wtedy gracz rozpoczynający, czyli biały będzie zawsze na przegranej pozycji (dotyczy to przypadku równej liczby pionów u obydwu graczy);
- GB - liczba pionów należących do gracza czarnego, może się różnić od GW.

Jak w podstawowej wersji każdy gracz zaczyna grę z 3 pionami na planszy, zawsze rozpoczyna biały. Piony umieszcza się na kropkach w rogach, a następnie przesuwa na pierwszy punkt w kierunku środka obszaru gry. Celem gracza jest, zdobycie pionów przeciwnika tak aby nie miał on żadnych pionów w rezerwie. Jednak jak zauważyliśmy wcześniej przy pewnych niefortunnych wariantach gry (niefortunnie dobrane parametry listy **L**) rozgrywka może się nie skończyć albo plansza może zostać zapchana. W pierwszym przypadku wynik rozwiązania gry będzie remisem, a w drugim gracz który nie może wykonać posunięcia będzie przegrywającym. jednak takie warianty gry będziemy traktować jako niewłaściwe starając się nie wykorzystywać ich w testach. Jak łatwo zauważyć, oryginalna gra opisana jest listą **L**=(4, 4, 15, 15) gdzie jedna z ostatnich dwóch wartości może być zwiększona maksymalnie o 3 w celu wprowadzenia asymetrii i tym samym wyrównaniu różnic w poziome gry obydwu graczy. Najmniejsza plansza jaką można sobie wyobrazić to plansza rozmiaru R = 2. Mniejsza plansza nie ma sensu ponieważ dla R = 1 obszar gry składał by się z tylko jednego pola gry, a musi ich być przynajmniej sześć aby pomieścić wszystkie początkowe (2x3) piony graczy.

GIPF is a deterministic game without random elements and hidden information for two players with a zero-sum without draws. The rules of the official version are available here. The task is to implement the engine for this game in its generalized version. Generalization allows for changing the properties of the game described by four parameters (S, K, GW, GB), which are elements of the aforementioned list **L** of game parameters:

- S - the board size expressed as the number of hexagonal board fields on each side of the board;
- K - the number of player pieces that trigger the collection of pieces. This value should never be less than two because a value of one would make the starting position on the board invalid (all pieces would be immediately removed). This value should also not be greater than 2*S-1 because that is the length of the "diagonal" of the board, so a longer sequence cannot be formed. Furthermore, this value should be even smaller because if it is exactly equal to 2*S-1, it would not be possible to capture opponent pieces in the game. This either leads to an endless game where players continuously release their pieces to the reserve or to the end of the game due to a filled board, where neither player can make any more moves. This is unambiguously described by the rule from the instructions: "It is not possible to push a piece beyond the playing area, i.e., beyond the dot on the opposite side of the line.";
- GW - the number of pieces belonging to the white player, which must be greater than the three pieces initially placed on the board for the player. However, if there are too few pieces, the game may quickly end due to resource exhaustion, and in that case, the starting player, i.e., white, will always be in a losing position (this applies when both players have an equal number of pieces);
- GB - the number of pieces belonging to the black player, which can differ from GW.

In the basic version, each player starts the game with 3 pieces on the board, and white always goes first. Pieces are placed on the dots in the corners and then moved to the first point in the direction towards the center of the playing area. The goal of the player is to capture the opponent's pieces so that they have no pieces left in reserve. However, as we mentioned earlier, with certain unfortunate game variants (poorly chosen parameters from the **L** list), the game may not end, or the board may become filled. In the first case, the result of solving the game will be a draw, and in the second case, the player who cannot make a move will be the loser. However, we will consider such game variants as improper and try not to use them in tests. As you can easily notice, the original game is described by the list **L** = (4, 4, 15, 15), where one of the last two values can be increased by a maximum of 3 to introduce asymmetry and balance the differences in the players' gameplay. The smallest imaginable board size is R = 2. A smaller board would not make sense because for R = 1, the playing area would consist of only one board field, and there must be at least six of them to accommodate all the initial (2x3) player pieces.

Silnik gry powinien pozwalać na:

- Wczytanie stanu gry, będzie on podawany w przypadkach testowych w następującym formacie. W pierwszej linii będą podane cztery liczby opisujące grę (zawartość listy **L**). W kolejnej linii pojawi się liczba pionów w rezerwach odpowiednio gracza grającego białymi i czarnymi oraz cyfra określająca aktywnego gracza. W kolejnych liniach znajdą się informacje opisujące stan planszy.
  - Początkowa plansza oryginalnej wersji gry będzie opisana w następujący sposób:

```
4 4 15 15
12 12 W
   W _ _ B
  _ _ _ _ _
 _ _ _ _ _ _
B _ _ _ _ _ W
 _ _ _ _ _ _
  _ _ _ _ _
   W _ _ B
```

  - Mniejsze przykładowe początkowe plansze mogą być opisane w następujący sposób:

    - ```
      2 2 5 5
      2 2 W
       W B
      B _ W
       W B
      ```

    - ```
      3 3 10 9 W
      7 6 W
        W _ B
       _ _ _ _
      B _ _ _ W
       _ _ _ _
        W _ B
      ```

- Zastosowanie posunięcia dla aktualnej planszy opisanego wg schematu: xN - yM gdzie xN to pole na które zostaje położony pion z rezerwy gracza, a pole yM to pole w kierunku którego ten właśnie położony pion zostaje przesunięty. Notacja ta jest zapożyczona z programu Gipf for One (GF1) i wygląda jak na poniższym rysunku:  Zauważmy, że sąsiednie kropki nie będą stosowane w reprezentacji planszy, ponieważ używane są one jedynie podczas pierwszej fazy posunięcia, kiedy gracz kładzie piona z rezerw. Po położeniu piona na kropce gracz musi go przesunąć, jeśli takie przesunięcie spowoduje wypchnięcie piona po przeciwnej stronie na kropkę, to jest ono niedozwolone. W takim przypadku silnik gry, powinien zapamiętać to posunięcie jako błędne i nie powinien już analizować kolejnych posunięć. W przypadku prośby o wydrukowanie stanu planszy będzie drukowany jej stan bezpośrednio poprzedzający ten niedozwolony ruch. W przypadku gdy w tym samym czasie powstanie więcej rzędów składających się z 4 (bądź więcej) pionów tego samego koloru i rzędy te przecinają się, rozkaz musi zawierać informację który rząd ma być usunięty, w innym przypadku będzie to błędne posunięcie i konsekwencje będą identyczne jak w opisanych wcześniej przypadkach.
- Wydrukowanie stanu planszy w postaci identycznej jak przy jej wczytywaniu, tak aby możliwa była ewentualna rozgrywka pomiędzy dwoma programami.
- Wydrukowanie stanu gry:
  - in_progress - gra się nie skończyła

    - white_win - wygrał rozpoczynający
    - black_win - wygrał gracz drugi
    - dead_lock <kolor> - gracz którego tura właśnie trwa a on nie może wykonać posunięcia kolor gracza, dodatkowo drukujemy kolor gracza
    - bad_move <kolor> <xN - yM> - gracz wykonał błędne posuniecie, dodatkowo drukujemy kolor gracza i komendę która wygenerowała błędne posunięcie
- Generowanie wszystkich możliwych posunięć aktywnego gracza.
- Ocenę gry w postaci odpowiedzi na pytanie - czy gra się zakończyła i czy gracz wygrał/zremisował/przegrał? Zauważmy, że jest to równoważne z odpowiedzią na pytanie - czy aktywny gracz może wykonać jakiekolwiek posunięcie. jeśli nie to przegrał a jego przeciwnik wygrał. Remis może nastąpić tylko w przypadku stwierdzenia, że gra nigdy się nie skończy, jednak na potrzeby tego zadania będziemy starali się nie doprowadzać do takich sytuacji.
- W zoptymalizowanej wersji, w przypadku kiedy przynajmniej jedno z posunięć aktywnego gracza prowadzi do jego wygranej, silnik powinien wygenerować tylko jedno z tych posunięć.

Solver, czyli algorytm rozwiązujący grę powinien pozwalać na:

- Rozwiązywanie gry na małej planszy, albo stanów końcowych na bardziej złożonej planszy.
- Odpowiedź na pytanie czy w zadanej liczbie posunięć dla zadanego stanu gry (posunięcie rozumiemy jako ruch pojedynczego pracza, a nie obydwu) aktywny gracz może wygrać.

The game engine should allow for:

- Loading the game state, which will be provided in the following format during test cases. The first line will contain four numbers describing the game (content of the **L** list). The next line will contain the number of pawns in reserves for the player playing with white and black, respectively, and a digit indicating the active player. The subsequent lines will contain information describing the state of the board.
    - The initial board of the original version of the game will be described as follows:

```
4 4 15 15
12 12 W
   W _ _ B
  _ _ _ _ _
 _ _ _ _ _ _
B _ _ _ _ _ W
 _ _ _ _ _ _
  _ _ _ _ _
   W _ _ B
```

    - Smaller sample initial boards can be described as follows:

        - ```
          2 2 5 5
          2 2 W
           W B
          B _ W
           W B
          ```

        - ```
          3 3 10 9 W
          7 6 W
            W _ B
           _ _ _ _
          B _ _ _ W
           _ _ _ _
            W _ B
          ```

- Applying a move to the current board, described according to the schema: xN - yM, where xN is the field where a pawn from the player's reserve is placed, and yM is the field in the direction to which the just placed pawn is moved. This notation is borrowed from the program Gipf for One (GF1) and looks as shown in the diagram below:  Note that neighboring dots will not be used in the board representation because they are only used during the first phase of a move when a player places a pawn from the reserves. After placing a pawn on a dot, the player must move it. If such a move would push a pawn on the opposite side onto a dot, it is not allowed. In such a case, the game engine should remember this move as invalid and should not analyze subsequent moves. When requesting to print the board state, the state preceding this invalid move will be printed. If multiple rows consisting of 4 (or more) pawns of the same color are formed at the same time and these rows intersect, the command must contain information about which row is to be removed. Otherwise, it will be an invalid move, and the consequences will be the same as in the previously described cases.
- Printing the board state in the same format as when loading it, allowing for potential gameplay between two programs.
- Printing the game state:
    - in_progress - the game is not finished
    - white_win - the starting player won
    - black_win - the second player won
    - dead_lock <color> - the turn of the player currently playing, who cannot make a move, additionally printing the player's color
    - bad_move <color> <xN - yM> - the player made an invalid move, additionally printing the player's color and the command that generated the invalid move
- Generating all possible moves for the active player.

- Evaluating the game by answering the question of whether the game has ended and if the player has won/drawn/lost. Note that this is equivalent to answering the question of whether the active player can make any move. If not, they have lost, and their opponent has won. A draw can only occur if it is determined that the game will never end, but for the purposes of this task, we will try to avoid such situations.
- In the optimized version, if at least one of the moves available to the active player leads to their victory, the engine should generate only one of those moves.

The solver, which is the game-solving algorithm, should allow for:

- Solving the game on a small board or end states on a more complex board.
- Answering the question of whether the active player can win within a given number of moves for a given game state (a move is understood as a move of a single pawn, not both).

Wymagania implementacyjne.

Program powinien reagować na następujące komendy:

- LOAD_GAME_BOARD po której podawane są parametry gry oraz stan planszy. Przykładowo, wczytanie oryginalnej planszy będzie wyglądało następująco:

```
LOAD_GAME_BOARD
4 4 15 15
12 12 W
   W _ _ B
  _ _ _ _ _
 _ _ _ _ _ _
B _ _ _ _ _ W
 _ _ _ _ _ _
  _ _ _ _ _
   W _ _ B
```

  Należy sprawdzić czy podany stan planszy jest poprawny, czy suma pionów obydwu graczy na planszy i w rezerwach jest poprawna, czyli identyczna z podaną w liście **L**. Należy sprawdzić czy nie zapomnieliśmy usunąć ciągów pionów, które spełnią kryterium wyzwalające ich zbieranie.
- PRINT_GAME_BOARD drukuje aktualny stan planszy, po wczytaniu gry powinien to być dokładnie ten sam wydruk który został przekazany podczas wczytywania. Oczywiście zakładając, że był on poprawny.
- DO_MOVE <xN - yM> wykonuje przekazany ruch dla aktywnego gracza na aktualnej planszy. Notacja posunięć powinna być zgodna z tą z programu Gipf for One (GF1) W przypadku gdy będzie on poprawny otrzymamy nową planszę a aktywny gracz zmieni się na przeciwnika. Może okazać się, że gra się zakończy jeśli przeciwnik nie będzie miał już pionów w rezerwie. W przypadku gdy ruch będzie niepoprawny należy ustalić odpowiedni stan gry, czyli "bad_move" oraz zapamiętać gracza i ruch które ten stan wywołały. W przypadku kiedy gracz musi zdjąć piony z planszy a istnieje kilka takich możliwości, musi on zadecydować które piony chce zdjąć. Ta informacja musi zostać dodana do komendy w postaci współrzędnych tych pionów. Współrzędne pionów są podawane wg schematu opisanego przykładem początkowej planszy dla standardowej wesji gry:

```
    + + + + +
   + W _ _ B +
  + _ _ _ _ _ +
 + _ _ _ _ _ _ +
+ B _ _ _ _ _ W +
 + _ _ _ _ _ _ +
  + _ _ _ _ _ +
   + W _ _ B +
    + + + + +
```

  Indeksy pól powyższej planszy:

```
    a5 b6 c7 d8 e9
   a4 b5 c6 d7 e8 f8
  a3 b4 c5 d6 e7 f7 g7
 a2 b3 c4 d5 e6 f6 g6 h6
a1 b2 c3 d4 e5 f5 g5 h5 i5
  b1 c2 d3 e4 f4 g4 h4 i4
    c1 d2 e3 f3 g3 h3 i3
      d1 e2 f2 g2 h2 i2
        e1 f1 g1 h1 i1
```

  Indeksy pól mniejszych plansz:

```
  a3 b4 c5
 a2 b3 c4 d4
a1 b2 c3 d3 e3
  b1 c2 d2 e2
    c1 d1 e1

   a4 b5 c6 d7
  a3 b4 c5 d6 e6
 a2 b3 c4 d5 e5 f5
```

```
a1 b2 c3 d4 e4 f4 g4
  b1 c2 d3 e3 f3 g3
    c1 d2 e2 f2 g2
      d1 e1 f1 g1
```

Zatem w przypadku ruchu który doprowadzi do niejednoznacznej sytuacji wymagającej dookreślenie które piony trzeba zdjąć. Po "normalnym" rozkazie podawany jest kolor gracza którego dotyczy wybór a nastepnie ciąg współrzędnych określających zdejmowane piony. Rozkaz będzie wyglądał nastepująco:
DO_MOVE <xN - yM> [w:|b:] x1 xn
W przypadku kiedy tych wyborów będzie więcej zostana one podane w postaci kolejnych ciągów indeksów.

- PRINT_GAME_STATE drukuje aktualny stan gry ("in_progress", "bad_move", itp.).
- GEN_ALL_POS_MOV drukuje listę wszystkich posunięć (bez powtórzeń) prowadzących do unikalnych stanów planszy.
- GEN_ALL_POS_MOV_EXT działa identycznie jak GEN_ALL_POS_MOV z wyjątkiem przypadku kiedy istnieje dowolne posunięcie wygrywające, wtedy drukuje tylko jedno z tych posunięć.
- GEN_ALL_POS_MOV_NUM drukuje liczbę wszystkich posunięć jakie otrzymujemy w wyniku działania komendy GEN_ALL_POS_MOV.
- GEN_ALL_POS_MOV_EXT_NUM działa identycznie jak GEN_ALL_POS_MOV_NUM z wyjątkiem przypadku kiedy istnieje dowolne posunięcie wygrywające, wtedy drukuje 1.
- WINNING_SEQUENCE_EXIST <N> odpowiada na pytanie czy istnieje dla któregokolwiek z graczy wygrywająca sekwencja posunięć mniejsza równa N.
- SOLVE_GAME_STATE odpowiedź na pytanie dla którego z graczy dany planszy gry jest wygrywający. Możliwe są dwie odpowiedzi WHITE_HAS_WINING_STRATEGY albo BLACK_HAS_WINING_STRATEGY.

Implementation Requirements.

The program should respond to the following commands:

- LOAD_GAME_BOARD, followed by the game parameters and the state of the board. For example, loading the original board would look like this:

```
LOAD_GAME_BOARD
4 4 15 15
12 12 W
   W _ _ B
  _ _ _ _ _
 _ _ _ _ _ _
B _ _ _ _ _ W
 _ _ _ _ _ _
  _ _ _ _ _
   W _ _ B
```

  It should be checked whether the provided board state is valid, whether the sum of pawns for both players on the board and in reserves is correct, i.e., identical to the one provided in list **L**. It should also be checked whether any sequences of pawns triggering their collection have been removed.
- PRINT_GAME_BOARD prints the current state of the board. After loading the game, it should be exactly the same as the printout provided during loading, assuming it was valid.
- DO_MOVE <xN - yM> performs the given move for the active player on the current board. The move notation should be consistent with the notation used in the Gipf for One (GF1) program. If the move is valid, a new board will be generated, and the active player will switch to the opponent. The game may end if the opponent no longer has any reserve pieces. If the move is invalid, the appropriate game state should be established as "bad_move," and the player and move that caused this state should be remembered. In cases where a player needs to remove pieces from the board and there are multiple possibilities, he must decide which pieces he want to remove. This information must be added to the command in the form of coordinates for those pieces. The coordinates of the pieces are provided according to the scheme described in the example of the initial board for the standard version of the game:

```
    + + + + +
   + W _ _ B +
  + _ _ _ _ _ +
 + _ _ _ _ _ _ +
+ B _ _ _ _ _ W +
 + _ _ _ _ _ _ +
  + _ _ _ _ _ +
   + W _ _ B +
    + + + + +
```

  Indexes of the fields for the above board:

```
    a5 b6 c7 d8 e9
   a4 b5 c6 d7 e8 f8
  a3 b4 c5 d6 e7 f7 g7
 a2 b3 c4 d5 e6 f6 g6 h6
a1 b2 c3 d4 e5 f5 g5 h5 i5
  b1 c2 d3 e4 f4 g4 h4 i4
    c1 d2 e3 f3 g3 h3 i3
      d1 e2 f2 g2 h2 i2
        e1 f1 g1 h1 i1
```

Indexes of the fields for the smaller boars:

```
  a3 b4 c5
 a2 b3 c4 d4
a1 b2 c3 d3 e3
  b1 c2 d2 e2
    c1 d1 e1

   a4 b5 c6 d7
  a3 b4 c5 d6 e6
 a2 b3 c4 d5 e5 f5
a1 b2 c3 d4 e4 f4 g4
  b1 c2 d3 e3 f3 g3
    c1 d2 e2 f2 g2
      d1 e1 f1 g1
```

Therefore, in the case of a move that leads to an ambiguous situation requiring clarification of which pieces should be removed, the command includes the color of the player for whom the choice applies, followed by a sequence of coordinates specifying the removed pieces. The command will look as follows:
DO_MOVE <xN - yM> [w:|b:]x1 xn ...
If there are multiple choices, they will be provided as subsequent sequences of indexes.

- PRINT_GAME_STATE prints the current game state ("in_progress," "bad_move," etc.).
- GEN_ALL_POS_MOV prints list of all possible moves (without repetitions) leading to unique board states.
- GEN_ALL_POS_MOV_EXT works the same as GEN_ALL_POS_MOV, except when there is any winning move available, it only prints one of those moves.
- GEN_ALL_POS_MOV_NUM prints the number of all moves obtained as a result of the GEN_ALL_POS_MOV command.
- GEN_ALL_POS_MOV_EXT_NUM works the same as GEN_ALL_POS_MOV_NUM, except when there is any winning move available, it prints 1.
- WINNING_SEQUENCE_EXIST <N> answers whether there exists a winning sequence of moves, of length less than or equal to N, for any of the players.
- SOLVE_GAME_STATE answers the question of which player has a winning strategy for the given game state. The possible answers are WHITE_HAS_WINNING_STRATEGY or BLACK_HAS_WINNING_STRATEGY.

Kryterium oceny

Zadanie nie będzie w pełni automatycznie testowane, a student będzie musiał zaprezentować które funkcjonalności udało się zaimplementować. Proszę zatem na dołączenie w postaci pliku tekstowego wszystkich komend na które program powinien reagować. Przykładowo mile widziane będą zestawy komend polegające na wczytaniu pewnego stanu gry, wykonaniu pewnej liczby posunięć, wydrukowaniu planszy oraz stanu gry, a następnie sprawdzeniu czy jest on zgodny z oczekiwaniami. powyższe scenariusze powinny dotyczyć serii posunięć prowadzących do wszystkich możliwych stanów gry (in_progress, itp.). Podczas prezentacji funkcjonalności silnika gry generującej listę wszystkich możliwych posunięć, dodatkowo należy zaprezentować, usprawnienie polegające na wygenerowaniu tylko jednego posunięcia wygrywającego (oczywiście jeśli takowe istnieje). Wagi oceny prezentują się następująco:

- Poprawne wczytanie planszy. (10%) Test 0
- Poprawne wydrukowanie planszy. (10%) Test 1
- Poprawne wykonanie posunięcia nie zmieniającego stanu gry, chodzi o "zwykłe posunięcia" w stanie gry "in_progress". Test 2-5 (40%)
- Prezentacja posunięć prowadzących do wszystkich możliwych stanów gry stanu gry, czyli powodujące przejście ze stanu gry "in_progress" w "white_win", black_win, itp. (10%)
- Wygenerowanie wszystkich możliwych posunięć dla wczytanego stanu gry. (10%) Test 6
- Detekcja posunięcia wygrywającego podczas generowania wszystkich możliwych posunięć, widoczna poprzez wydrukowanie pojedynczego posunięcia wygrywającego. (10%) Test 7
- Prezentacja poprawnego działania Solver'a odpowiadającego czy dany gracz może wygrać/przegrać w zadanej, liczbie posunięć. Student powinien wygenerować takie stany gry, na których do zakończenia gry pozostało N posunięć gdzie $N \geq 1$ i $N \leq 2$. (10%)
- To samo co wyżej ale dla $N > 2$ i większych. Wiadomo, że dla mniejszych plansz powinno to być to łatwiejsze. (10%)
- Prezentacja prostych, końcowych stanów gry, które udało się rozwiązać. (10%) Test 9
- Prezentacja bardziej ambitnych stanów gry, które udało się rozwiązać wraz z sekwencją posunięć wygrywających w postaci drzewa dowodzącego gdzie na jedno optymalne posunięcie gracza rozpoczynającego przypadają wszystkie możliwe posunięcia przeciwnika. (10%)
  Prezentacja wyników działania Solvera wykorzystującego bardziej zaawansowanie algorytmy, porównanie ze zwykłym mini-max'em + aplha-beta. Prezentacja faktu, że udało się rozwiązać bardziej złożone plansze albo zysku czasowego/pamięciowego . (10%-30% bonusowe)

Teoretycznie można uzyskać 160% czyli 56 pkt.

Dodatkowa literatura:

- GAME-TREE SEARCH USING PROOF NUMBERS: THE FIRST TWENTY YEARS

Evaluation Criteria

The task will not be fully automated and the student will need to present which functionalities have been implemented. Please include in a text file all the commands to which the program should respond. For example, sets of commands involving loading a certain game state, making a certain number of moves, printing the board and game state, and then checking if it matches the expectations. These scenarios should cover a series of moves leading to all possible game states (in_progress, etc.). During the presentation of the game engine functionality that generates a list of all possible moves, an additional improvement should be demonstrated, which involves generating only one winning move (if such exists). The grading weights for the presentation are as follows:

- Correct loading of the board. (10%) Test 0
- Correct printing of the board. (10%) Test 1
- Correct execution of moves that do not change the game state, referring to "regular moves" in the "in_progress" state. Test 2-5 (40%)
- Presentation of moves leading to all possible game states, causing a transition from the "in_progress" state to "white_win," "black_win," etc. (10%)
- Generation of all possible moves for the loaded game state. (10%) Test 6
- Detection of a winning move during the generation of all possible moves, indicated by printing a single winning move. (10%) Test 7
- Demonstration of the Solver's correct functionality in determining whether a player can win/lose in a given number of moves. The student should generate game states where there are N moves left until the end of the game, where $N \geq 1$ and $N \leq 2$. (10%)
- The same as above but for $N > 2$ and larger. It is known that it should be easier for smaller boards. (10%)
- Presentation of simple final game states that were successfully solved. (10%) Test 9
- Presentation of more challenging game states that were successfully solved, along with the sequence of winning moves in the form of a proof tree, where all possible opponent moves are attributed to one optimal move by the starting player. (10%)
  Presentation of the results of the Solver using more advanced algorithms, comparison with ordinary mini-max + alpha-beta. Presentation of the fact that more complex boards were solved or gained time/memory efficiency. (10%-30% bonus)

In theory, it is possible to achieve 160%, which corresponds to 56 points.

Additional Literature:

- GAME-TREE SEARCH USING PROOF NUMBERS: THE FIRST TWENTY YEARS

Testy

Tutaj

Test 0 - Wejście

Testowanie poprawności wczytywanej planszy. W pierwszej linii pojawi się instrukcja LOAD_GAME_BOARD a następnie plansza którą należy wczytać i sprawdzić jej poprawność pod kątem, liczby pionów oraz jej rozmiarów (liczby pól we wczytywanych liniach wejścia).

Test 0 - Wyjście

Jeden z poniższych stringów:

- BOARD_STATE_OK - plansza poprawna
- WRONG_WHITE_PAWNS_NUMBER - zła liczba białych pionów
- WRONG_BLACK_PAWNS_NUMBER - zła liczba czarnych pionów
- WRONG_BOARD_ROW_LENGTH - zła długość wiersza planszy

Test 1 - Wejście

Drukowanie planszy. W pierwszej linii pojawią się dodatkowa instrukcja: PRINT_GAME_BOARD drukująca zawartość planszy.

Test 1 - Wyjście

W odpowiedzi na PRINT_GAME_BOARD drukowana jest zawartość planszy o identycznym układzie jak ten z instrukcji LOAD_GAME_BOARD. W przypadku drukowania pustej planszy, co ma miejsce po próbie wczytania błędnej planszy pojawia się komunikat: "EMPTY_BOARD".

Test 2 - Wejście

Testowanie poprawności wykonywanych posunięć. W pierwszej linii pojawią się dodatkowe instrukcje:

- DO_MOVE [w:|b:] y1 yn - wykonuje posunięcie z pola x1, które musi być polem startowym w kierunku pola x2 które musi z nim sąsiadować.

Test 2 - Wyjście

Jeden z poniższych stringów:

- MOVE_COMMITTED - ruch poprawny, zatwierdzono zmiany na planszy i zmieniono aktualnego gracza.
- BAD_MOVE_<x1>_IS_WRONG_INDEX - zły indeks, pokazuje na pole spoza planszy. W przypadku kiedy podano więcej niż jeden zły indeks, drukowany jest tylko pierwszy napotkany.
- UNKNOWN_MOVE_DIRECTION - nie można określić kierunku ruchu.
- BAD_MOVE_<x1>_IS_WRONG_STARTING_FIELD - wybrano złe, pole startowe (powinno to być jedno z położonych na obrzeżach planszy)
- BAD_MOVE_<x2>_IS_WRONG_DESTINATION_FIELD - wybrano złe, pole docelowe (powinno to być pole wolne albo zawierające pion któregoś z graczy)
- BAD_MOVE_ROW_IS_FULL - nie można wykonać ruchu bo wiersz jest zapełniony.

Test 3 - Wejście

Testowanie poprawności wczytywanej planszy. W pierwszej linii pojawi się instrukcja LOAD_GAME_BOARD a następnie plansza którą należy wczytać i sprawdzić jej poprawność pod kątem istnienia wierszy zawierających ciąg długości przynajmniej K.

Test 3 - Wyjście

Jeden z poniższych stringów:

- BOARD_STATE_OK - plansza poprawna
- ERROR_FOUND_<N>_ROW_OF_LENGTH_K - błędna plansza znaleziono N wierszy zawierających ciąg długości przynajmniej K.

Test 4 - Wejście

Rozszerzenie testu 2 pozwala na testowanie poprawności wykonywanych posunięć i drukowania planszy w przypadku konieczności zbierania pionów. Piony zbierane są z całego ciągu w którym jeden kolor tworzy nieprzerwany podciąg długości K albo większej. Piony tego koloru wracają do rezerw gracza pozostałe piony nie wracają do gry.

Test 5 - Wejście

Rozszerzenie testu 5 pozwala na testowanie poprawności wykonywanych posunięć i drukowania planszy w przypadku konieczności zbierania pionów w niejednoznacznych sytuacjach w których należy doprecyzować który rząd ma zostać zdjęty.

- DO_MOVE <x1-x2> [w:|b:] y1 yn - w wyniku wykonania posunięcia należy usunąć ciąg białych albo czarnych pionów (z zależności czy podano w: czy b:) o skrajnych pionach na polach y1 i yn.

Uwaga w testach zarówno kolor gracza jak i skrajne piony mogą być podane błędnie w takim przypadku należy wydrukować adekwatną informację:

- WRONG_COLOR_OF_CHOSEN_ROW - podano błędny kolor.
- WRONG_INDEX_OF_CHOSEN_ROW - podano błędny index.

Test 6 - Wejście

Testowanie funkcjonalności generującej wszystkie możliwe ruchy gracza prowadzące do unikalnych stanów planszy. Na wejściu najpierw nastąpi wczytanie planszy komendą LOAD_GAME_BOARD. W kolejnym kroku nastąpi komenda GEN_ALL_POS_MOV_NUM. W tej komendzie nie są testowane posunięcia prowadzące do niejednoznacznego zbierania pionów z planszy.

Test 6 - Wyjście

Komenda <N>_UNIQUE_MOVES - zwracająca informację o liczbie N unikalnych stanów planszy.

Test 6a - Wejście

Test 6 z dodanymi komendami GEN_ALL_POS_MOV do komend GEN_ALL_POS_MOV_NUM.

Test 6b - Wyjście

Wydrukowane wszystkie możliwe stany gry, ten test nie jest umieszczany na STOS'ie aby nie wymuszać kolejności generowania stanów gry. Programy studentów powinny zachowywać się identycznie co do kolejności drukowanych stanów.

Testy 7 i 7a

Rozszerzenie testu 6 o funkcjonalność komendy GEN_ALL_POS_MOV_NUM generującej posunięcia prowadzące do niejednoznacznego zbierania pionów z planszy. Test 7a podobnie jak 6a nie jest wykonywany na STOS'ie a ma za

zadanie dostarczyć szczegółowych informacji o wygenerowanych ruchach w przypadku problemów z testem 6.
Testowanie metody IS_GAME_OVER.

Testy 8 i 8a

Testowanie metody GEN_ALL_POS_MOV_EXT_NUM.

Testy 9 i 9a

Testowanie metody SOLVE_GAME_STATE.

Tests

Here

Test 0 - Input

Testing the correctness of the loaded board. The first line will contain the instruction LOAD_GAME_BOARD, followed by the board that needs to be loaded and checked for correctness, including the number of pawns and its dimensions (the number of fields in the input lines).

Test 0 - Output

One of the following strings:

- BOARD_STATE_OK - board is correct
- WRONG_WHITE_PAWNS_NUMBER - incorrect number of white pawns
- WRONG_BLACK_PAWNS_NUMBER - incorrect number of black pawns
- WRONG_BOARD_ROW_LENGTH - incorrect board row length

Test 1 - Input

Printing the board. The first line will contain an additional instruction: PRINT_GAME_BOARD, which prints the content of the board.

Test 1 - Output

In response to PRINT_GAME_BOARD, the content of the board will be printed with the same layout as the one in the LOAD_GAME_BOARD instruction. In case of printing an empty board, which occurs after attempting to load an incorrect board, the message "EMPTY_BOARD" will appear.

Test 2 - Input

Testing the correctness of the performed moves and the printing of the game board. The first line will contain additional instructions:

- DO_MOVE - performs a move from field x1, which must be a starting field, to field x2, which must be its neighboring field.
- PRINT_GAME_BOARD - prints the content of the game board

Test 2 - Output

One of the following strings:

- MOVE_COMMITTED - move is valid, changes on the board are confirmed, and the current player is changed.
- BAD_MOVE_<x1>_IS_WRONG_INDEX - wrong index, pointing to a field outside the board. If multiple wrong indexes are provided, only the first one encountered is printed.
- UNKNOWN_MOVE_DIRECTION - unable to determine the direction of the move.
- BAD_MOVE_<x1>_IS_WRONG_STARTING_FIELD - incorrect starting field selected (should be one of the fields located at the edges of the board).
- BAD_MOVE_<x2>_IS_WRONG_DESTINATION_FIELD - incorrect destination field selected (should be either an empty field or a field containing a pawn of one of the players).
- BAD_MOVE_ROW_IS_FULL - unable to make a move because the row is full.
- In response to PRINT_GAME_BOARD, the content of the board is printed with the same layout as the one in the LOAD_GAME_BOARD instruction.

Test 3 - Input

Testing the correctness of the loaded game board. The first line will contain the LOAD_GAME_BOARD instruction, followed by the board that needs to be loaded and checked for correctness in terms of the existence of rows containing a sequence of length at least K.

Test 3 - Output

One of the following strings:

- BOARD_STATE_OK - board is correct
- ERROR_FOUND_<N>_ROW_OF_LENGTH_K - incorrect board, N rows containing a sequence of length at least K were found.

Test 4 - Input

The extension of test 2 allows testing the correctness of moves and printing the board in case it is necessary to collect pieces. The pieces are collected from a sequence in which one color forms a continuous subsequence of length K or greater. The pieces of that color return to the player's reserve, while the remaining pieces do not return to the game.

Test 5 - Input

The extension of test 5 allows testing the correctness of moves and printing the board in case it is necessary to collect pieces in ambiguous situations where it is necessary to specify which row should be removed.

- DO_MOVE <x1-x2> [w:|b:] y1 yn - as a result of executing the move , a sequence of white or black pieces (depending on whether w: or b: is provided) with the outermost pieces on fields y1 and yn should be removed.

Note that in the tests, both the player's color and the outermost pieces may be provided incorrectly. In such cases, an appropriate information should be printed:

- WRONG_COLOR_OF_CHOSEN_ROW - incorrect color provided.
- WRONG_INDEX_OF_CHOSEN_ROW - incorrect index provided.

Test 6 - Input

Testing the functionality that generates all possible moves leading to unique board states for the player. The input will begin with loading the game board using the command LOAD_GAME_BOARD. The next step is to execute the command GEN_ALL_POS_MOV_NUM. This command does not test moves that result in ambiguous collection of pieces from the board.

Test 6 - Output

The command <N>_UNIQUE_MOVES returns information about the number N of unique board states.

Test 6a - Input

Test 6 with the added commands GEN_ALL_POS_MOV within the command GEN_ALL_POS_MOV_NUM.

Test 6b - Output

Printed all possible game states. This test is not included in the STOS to avoid enforcing the order of generating game states. Student programs should behave identically in terms of the order of printed states.

Tests 7 and 7a

Extending test 6 with the functionality of the GEN_ALL_POS_MOV_NUM command, which generates moves leading to ambiguous pawn captures on the board. Test 7a, similar to 6a, is not performed on the STOS and is intended to provide detailed information about the generated moves in case of issues with test 6. Testing the IS_GAME_OVER method.

Tests 8 and 8a

Testing the GEN_ALL_POS_MOV_EXT_NUM method.

Tests 9 and 9a

Testing the SOLVE_GAME_STATE method.