Programmeringsprojekt del 1

Celine S. Fussing
cefus23@student.sdu.dk

Sandra K. Johansen
sjoha23@student.sdu.dk

Sofie Løfberg
soloe23@student.sdu.dk

24. november 2023

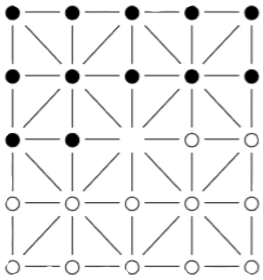
Contents

1 Introduktion

Denne rapport er udarbejdet i november 2023 til et projekt på Syddansk Universitet (Odense) i kurset DM574. Vi har udviklet et program i Python, der implementerer modulerne move.py og board.py med hensigt på at kunne spille spillet "Alquerque" ved brug af en grafisk klient stillet til rådighed af underviser, Luís Cruz-Filipe.

2 Problemstilling og Baggrund

I opgaven implementeres modulerne board.py og move.py. I alt indeholder modulerne 11 funktioner, som gør det muligt at spille Alquerque. Spillet Alquerque er en forfader til spillet dam, hvor to spillere, en sort og en hvid, lagt ud på et 5 x 5 bræt, skal vinde ved at slå alle modstanderens brikker hjem. De sorte brikker ligger øverst og de hvide nederst, mens felt 13 (det midterste) starter med at være tomt.

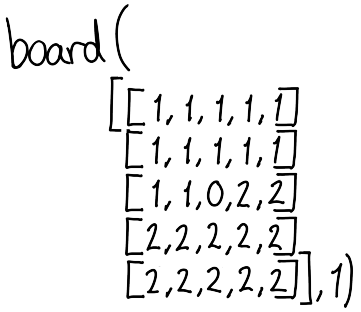


Figur 1 ref. Introduktion til projektet.pdf

For spillets regler refererer vi til projekt-introduktionen.

3 Overvejelser

Som det første i vores projekt undersøgte vi, hvordan vi ville repræsentere vores bræt. Her overvejede vi som det første mulighederne ved at repræsentere brættet som en matrix á 5 lister. Hvor vi samtidig gerne ville holde styr på det træk, vi er nået til. Dette ville blive vist som en matrix efterfulgt at et tal (trækket). Dette bræt fremgår af figur 2.

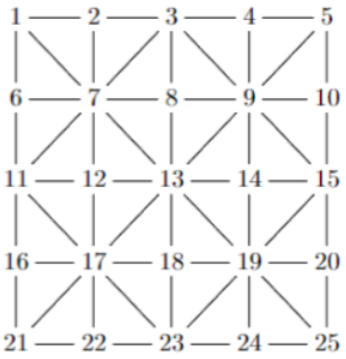


Figur 2

Som det fremgår af spillets regler består brættet af sorte og hvide brikker, som vi valgte at repræsentere som henholdsvis 1-taller og 2-taller. 0 benytter vi til at repræsentere tomme pladser. Det viste sig dog at være udfordrende at tilgå brikkernes indeks. Den umiddelbare løsning var at bruge koordinatsæt, så hver plads havde et tilhørende koordinat-sæt i (y,x) - format. Vi tæller først rækker og derefter kolonner. Eksempelvis: Brikken, på indeks 1 (se figur 3), ville hedde (0,0), mens en brik på indeks 18 ville hedde (3,2).

Dette kunne vi også se store muligheder i, når vi skulle definere lovlige træk senere i spillet. Idéen, at vores y-koordinat kun ændrede sig med 1 - op eller ned - afhængig af, om det er en sort eller hvid brik. Samtidig måtte vores x-koordinat ændre sig med 4-6 - enten op eller ned - baseret på spilleren. I forhold til funktionen is_legal så vi store fordele for implementeringen af brættet som en matrix, som gjorde op for eventuelle udfordringer. Dette ledte til den sidste udfordring, som blev afgørende for vores valg af implementering. Som en del af board.py skulle vi implementere to funktionener, som returnerer en liste med henholdsvist indekserne af alle sorte og hvide brikker. Dette var udfordrende, da vores indekser ville være i koordinat-format, og vi da ville være nødt til at konvertere.

Af denne grund begyndte vi at lede efter andre løsninger for implementeringen af vores bræt. Vi valgte i stedet en liste, hvori 1 og 2 stadig bruges til at repræsentere henholdsvis sort og hvid. Vi valgte, at vores indekser i listen skulle repræsentere pladserne på brættet. Derfor tilføjede vi et 0 på indeks 0. Dette medførte, at vi kunne benytte de nummererede positioner angivet i projektbeskrivelsen, og som også fremgår af figur 3.



Figur 3 ref. Introduktion til projektet.pdf

Imidlertid ville vi gerne holde fast i at kunne tælle antal træk, som vi havde tænkt ved implementering af en matrix. Siden det kan bruges i funktionen white_plays. Således kom vores implementering af brættet til at se ud som vist i figur 4.

```
([0,
  1,1,1,1,1,
  1,1,1,1,1,
  1,1,0,2,2,
  2,2,2,2,2,
  2,2,2,2,2],
 1)
```

Figur 4

Vi kunne dog se fremtidige ulemper med at definere lovlige træk med denne implementering. Idet der nu skulle en mere kompliceret funktion til at bestemme gyldige træk samt angrebstræk. Til trods for dette konkluderede vi, at der var større potentiale ved denne implementering fremfor implementering af brættet som en matrix.

4 Teknisk beskrivelse af programmet

Programmet består af to moduler, board.py og move.py. Nedenfor præciseres hvert modul og dens relevante funktioner. Koden, der refereres til, fremgår af afsnittet "Appendix". Vi har lagt stor fokus på at implementere modulerne, så de er udviklet så generelt som muligt.

board.py har til funktion at skabe en repræsentation af et bræt samt holde styr på, hvilket træk der er nået til. I board.py er der funktioner, der definerer lovlige træk, hvis tur det er, samt om spillet er slut. Vi har defineret variabler for læselighedens skyld, som fremgår af appendix.
Som tidligere beskrevet repræsenteres vores bræt som en liste, hvorved indeks på listen passer med pladserne fra 1 til 25 (se figur 3 under overvejelser). Herudover tælles også træk, vi er nået til. I visse funktioner har vi lavet doctest med udgangspunkt i et nyt bræt, dvs. vi bruger funktionen make_board.

Funktionen white_plays tjekker, om det er hvids tur til at foretage et træk. Dette gøres ved en bool. Af reglerne fremgår det, at hvid altid starter spillet. Derfor er første træk hvid. For at tjekke om det er hvids tur, bruger vi antal træk angivet af move i make_board. Herudover er det relevant at gennemgå funktionen is_game_over, da den ikke er særlig effektiv tidsmæssigt. Funktionerne white og black returnerer en liste med indekserne for henholdsvist alle sorte og hvide brikker. Hvis en af disse lister er tomme, vil det derfor betyde, at der ikke er flere af den farve på brættet. Funktionen is_game_over tjekker om disse lister er tomme, og derved om spillet er slut. is_game_over tjekker også om listen, legal_moves returnerer, er tom, da spillet ville være uafgjort og slut. Umiddelbart er dette ikke problematisk - dog fortsætter funktionen selv efter første fundne element. På et bræt af større proportioner med flere elementer, ville det skabe en øget køretid. Da vores alquerque-bræt er småt, er det ikke noget, man bemærker, andet end i selve koden.

I funktion Move starter vi med at tjekke, om det ønskede træk er et gyldigt træk. Vi har to muligheder: Et normalt træk og et angrebstræk. Dette tjekkes ved brug af en anden funktion, is_legal, som forklares senere i dette afsnit. Hvis det er et gyldigt og et normalt træk, tjekker vi, om det er sort eller hvid, der spiller. Dette sikrer, at vores target-plads bliver lavet til den korrekte brik. I vores angrebs-træk tjekker vi, hvilken farve source-pladsen er. Derefter laves både den oprindelige plads, og den plads vi hopper over, tomme. Dette medfører, at den brik vi "flytter" ender det rigtige sted og som den rigtige farve. Modstanderen, som er slået hjem, bliver slettet.
Funktionen is_legal returnerer en boolean, som afgør, hvorvidt et træk er lovligt eller ej. Den består blandt andet af en overordnet if-statement, som starter med at afgøre, om target er en tom plads. Den definerer desuden, at når det er hvids tur, skal der rykkes en hvid brik. Vi havde tidligere ikke tilføjet dette, og det resulterede i en fejl, hvor f.eks. sort kunne rykke en hvid brik (essentielt konvertere den hvide brik til en sort brik). Herefter differentierer vi mellem, hvorvidt det er hvid eller sort, der spiller ved brug af white_plays-funktionen. Vi differentierer også mellem ulige og lige felter. Når hvid laver et almindeligt træk på et ulige felt, kan der rykkes lige opad eller diagonalt opad. Differensen mellem source og target er i de tilfælde 4, 5 eller 6. På et lige felt, kan der kun rykkes lige opad, og derfor skal differensen være lig med 5.
For at undgå de tilfælde hvor brikker springer fra den ene side af brættet til den anden, har vi defineret hjælpefunktionen _row. _row returnerer, hvilken række

et felt ligger på. Ved et almindeligt træk definerer vi, at differensen mellem rækken source ligger på, og target ligger på, skal være 1 eller -1. Så undgår vi, at man kan rykke ud af brættet diagonalt, fordi da ville man rykke 2 rækker. Ved et angrebstræk definerer vi, at differensen mellem de to rækker skal være enten -2, 0, 2. Dette forhindrer, at man rykker til siden og over på den anden side af brættet, da differensen i det tilfælde ville være 1 eller -1. Det forhindrer også, at man kan rykke diagonalt ud af brættet, da differensen så ville være 3 eller -3. For at et angrebstræk er lovligt, skal brikken mellem source og target hverken være tom eller samme brik som source. Desuden skal differensen mellem source og target være en af følgende tal; (-12, -10- 8, -2, 2, 8, 10, 12). Dette tillader at rykke til siden, opad, nedad og diagonalt. Der differentieres selvfølgelig igen mellem lige og ulige tal.

Funktionen is_legal har den ulempe, at den ikke er specielt letlæselig. Den har nogle lange if-statements, som benytter en del and-operatorer for at samle flere betingelser. Dette gør den lidt uoverskuelig at læse. Denne ulempe kunne mindskes ved brug af hjælpefunktioner. Vi kunne for eksempel have inkluderet en funktion, der definerer et angrebstræk, så nogle af betingelserne i vores if-statements kunne være ekskluderet. På grund af tidspres har vi ikke nået dette, men vi stræber efter at rette det i fremtiden.
I move.py defineres en ny datatype, Move, som indeholder src og trg. move.py holder styr på vores "brikker" ved brug af indeks på listen fra board.py. Herunder er der tre funktioner; source, target og make_move, der interagerer med board.py for at ændre hvilke tal, der står på hvilke indeks.

4.1 Test og overvejelser

Vi har valgt at teste modulerne og funktionerne hver for sig så vidt muligt. Denne tilgang har vi også brugt generelt, når vi har ændret i dele af funktionerne. Udfordringen har været at teste modulerne i sammenhæng med de andre. Med dette menes, at det har været svært at eksemplificere test, hvor modulerne fungerer i sammenhæng med hinanden uden bare at køre hele programmet. Alligevel er der flere funktioner som tydeligt virker når hele programmet køres. Eksempelvis virker white_plays, da det skiftevis bliver hvids og sorts tur. Det samme gør sig gældende for is_game_over, hvor vi både har testet, om den registrerer, hvis det er uafgjort eller en sejr. Vi ræsonnerer derfor også, at make_move, source og target fra move.py virker. Fordi det ellers ville give problemer når programmet køres. Nedenfor vil vi vise udsnit af de test, vi har fortaget os på funktioner, hvor test har været relevant:
Af figur 5 fremgår test af make_board(), som er brættet, man starter med.

```
make_board()
Board(board=[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], move=1)
```

Figur 5

Nedenfor ses figur 6, som viser de hvide og sorte brikkers indekser, før spillet begynder.

```
white(b)
[14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
    black(b)
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Figur 6

Af figur 7 ses en kopi at brættet samt det træk, den er nået til - i dette tilfælde er det en kopi af startbrættet.

```
copy(b)
Board(board=[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], move=1)
```

Figur 7

Det er også relevant at se figur 8, som viser de gyldige træk, hvid kan lave for at begynde spillet. Her fremgår det også, at de eneste gyldige træk svarer til spillets regler.

```
legal_moves(b)
[Move(src=17, trg=13), Move(src=18, trg=13), Move(src=19, trg=13)]
```

Figur 8

5 Konklusion

Vi kan nu slutte af med at konkludere følgende; vi har formået at indfri vores tre centrale ønsker: Skabe et program som følger spillets regler, lave velovervejede tests og få vores program til at køre gnidningsfrit med klientmodulet alquerqueGUI.py. Først og fremmest sikrede vi os, at spillet følger de givne regler i funktionen is_legal, som skulle kunne differentiere mellem almindelige og angrebstræk. Dette formåede vi. Herudover var det vigtigt at lave kontrollerede test på mindre dele af programmet så vidt muligt, og overveje hvorvidt programmet som helhed fungerer ved at kigge på koden og spillet. Vi er lykkedes med at finde løsninger til de givne funktioner, som fungerer med vores valgte implementering: en liste med 3 konstanter. Sidste ønske var, at programmet skulle køre gnidningsfrit med alquerqueGUI.py, hvilket vi er lykkedes med. Vi kan spille spillet, man kan ikke lave ugyldige træk, og spillet kan slutte.

6 Appendix

```
from move import *
from dataclasses import dataclass
import math
@dataclass
```

```
class Board:
    board: list[int]
    move: int

def make_board() -> Board:
    """Return a new board."""
    return Board([0,
                  1,1,1,1,1,
                  1,1,1,1,1,
                  1,1,0,2,2,
                  2,2,2,2,2,
                  2,2,2,2,2],
                 1)

def white_plays(b: Board) -> bool:
    """Check if it is white's turn to play.
    >>> white_plays(make_board)
    True
    """
    return b.move % 2 == 1

def white(b: Board) -> list[int]:
    """Return a list containing the indices of every white piece.
    >>> white(make_board)
    [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
    """
    white_indices = []
    for i in range(len(b.board)):
        if b.board[i] == white_piece:
            white_indices.append(i)
    return white_indices

def black(b: Board) -> list[int]:
    """Return a list containing the indices of every black piece.
    >>> black(make_board)
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    """
    black_indices = []
    for i in range(len(b.board)):
        if b.board[i] == black_piece:
            black_indices.append(i)
    return black_indices

def is_game_over(b: Board) -> bool:
    """Determine whether the game is over.
    >>> is_game_over(make_board)
```

```
    False
    """
    res = False
    if black(b) == []:
        res = True
    if white(b) == []:
        res = True
    if legal_moves(b) == []:
            res = True
    return res

def copy(b: Board) -> Board:
    """Return a copy of the given board.
    >>> copy(make_board)
    Board(board=[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], move=1)
    """
    copy = [item for item in b.board]
    return Board(copy, b.move)

def move(m: Move, b: Board) -> None:
    """Update the board by simulating the given move."""
    if is_legal(m, b):
        b.board[source(m)] = empty_space
        if white_plays(b):
            b.board[target(m)] = white_piece
        else:
            b.board[target(m)] = black_piece
        if not(4 <= abs(source(m) - target(m)) <= 6):
        #Checks if it is not a nonattack move.
            b.board[(source(m) + target(m))//2] = empty_space
            b.board[source(m)] = empty_space
    b.move = b.move + 1

def is_legal(m: Move, b: Board) -> bool:
    """Check if the given move is a legal move on the given
    board.
    """
    legal = False
    if ((1 <= target(m) <= 25) and (1 <= source(m) <= 25)
        and b.board[target(m)] == empty_space
        and ((white_plays(b) and b.board[source(m)] == white_piece)
             or (not white_plays(b) and b.board[source(m)] == black_piece))):
        if white_plays(b):
            if (source(m) % 2 == 1 and _row(source(m)) - _row(target(m)) == 1
            #Only allows pieces on odd numbered squares to move diagonally.
```

```
                    and source(m) - target(m) in (6, 5, 4)):
                        legal = True
                if (source(m) % 2 == 0 and _row(source(m)) - _row(target(m)) == 1
                    and source(m) - target(m) == 5):
                        legal = True
            else:
                if (source(m) % 2 == 1 and _row(source(m)) - _row(target(m)) == -1
                    and source(m) - target(m) in (-6, -5, -4)):
                    legal = True
                if (source(m) % 2 == 0 and _row(source(m)) - _row(target(m)) == -1
                    and source(m) - target(m) == -5):
                        legal = True
            if (_row(source(m)) - _row(target(m)) in (-2, 0, 2)
                and b.board[(target(m)+source(m))//2] != b.board[source(m)]
                and b.board[(target(m)+source(m))//2] != empty_space):
                #Checks whether the requirements for an attack move is fulfilled.
                if (source(m) % 2 == 1
                    and source(m) - target(m) in (-12, -10, - 8, -2, 2, 8, 10, 12)):
                    legal = True
                if (source(m) % 2 == 0
                    and source(m) - target(m) in (-10, -2, 2, 10)):
                    legal = True
    return legal

def _row(n: int) -> int:
    """Returns the row a piece is located on.
    >>> row(5)
    1
    """
    return math.ceil(n/5)


def legal_moves(b: Board) -> list[Move]:
    """Return a list containing all the legal moves
    from the given board."""
    moves = []
    for i in range (1, 26):
        for j in range (1, 26):
            current_move = make_move(i, j)
            if is_legal(current_move, b):
                moves.append(current_move)
    return moves

white_piece = 2
black_piece = 1
empty_space = 0
```

```
b = make_board()
```

```
from dataclasses import dataclass
@dataclass
class Move:
    src: int
    trg: int

def make_move(src: int, trg: int) -> Move:
    """Makes the new move between the given squares."""
    return Move(src, trg)

def source(m: Move) -> int:
    """Returns the 'from'-square in a move"""
    return m.src

def target(m: Move) -> int:
    """Returns the 'to'-square in a move"""
    return m.trg
```