Programmeringsprojekt Del 2

Celine S. Fussing
cefus23@student.sdu.dk

Sandra K. Johansen
sjoha23@student.sdu.dk

Sofie Løfberg
soloe23@student.sdu.dk

5. januar 2023

Contents

1 Introduktion

Denne rapport er udarbejdet i december 2023 og januar 2024 til et projekt på Syddansk Universitet (Odense) i kurset DM574. I denne fase har vi udviklet et modul, minimax.py, med hensigt på at kunne udarbejde en autospiller, som selv kan finde det bedste træk på det givne bræt i spillet Alquerque. Vi bruger vores egne moduler fra fase 1 og 2 af projektet.

2 Teknisk Beskrivelse af Programmet

I modulet minimax.py definerer vi 9 funktioner og en dataklasse. Disse beskrives herunder.

2.1 Dataklasse

Vores dataklasse, Node, indeholder felterne: Children, parent, data, value og move. Se figur 1 nedenfor.

```
@dataclass
class Node:
    children: list[Any]
    parent: Any
    data: Board
    value: int
    move: Move
```

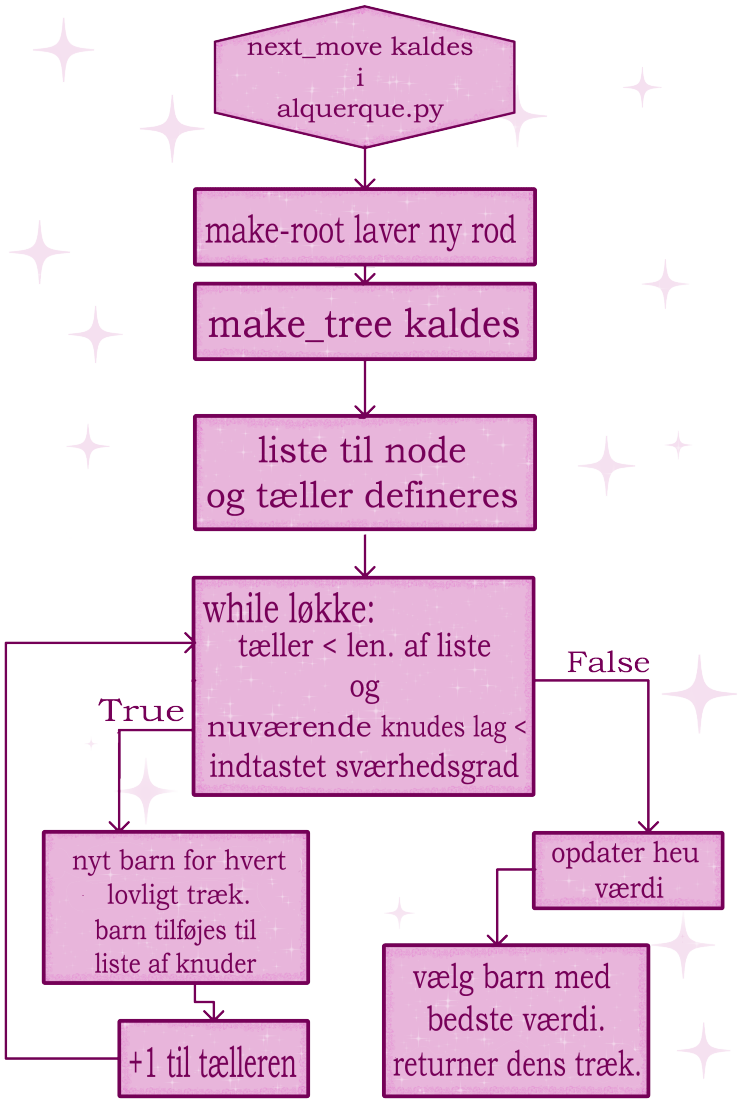
Figur 1

Feltet 'children' er en liste af knuder; referencer til de tilhørende børn. parent viser forældreskabet for den pågældende knude, så man kan overskue, hvor barnet hører til. data er brættet som det ville se ud efter knudes givne træk. Desuden får hver knude i træet en værdi (value), som udtrykker, hvor god brættilstanden er. Dette er den heuristiske værdi, som findes af vores heuristiske funktion, heu. Denne værdi bliver opdateret af funktionen update_value, udfra om det er en min- eller maxknude, efter træet er lavet. move tilføjede vi til dataklassen for at forenkle dele af vores kode. Dette gjorde vi, for at holde styr på, hvilket træk førte til knudens bræt.

2.2 make_tree

make_tree bruger roden samt en brugervalgt sværhedsgrad til at bygge et beslutningstræ med samme højde som inputtet. Vi definerer en liste (nodes) til knuder, som fra starten indeholder roden, og en variabel i som bruges som tæller. Vi benytter en while-loop, hvis guard er følgende: tælleren skal være mindre end længden af listen med knuder, og laget af knuden vi arbejder med, skal være

mindre end sværhedsgraden. Derefter bruges et for-loop til at tilføje en ny knude for hvert lovligt træk, som kan laves på brættet af knuden, der ligger på plads i i vores liste. Alle nye knuder tilføjes til listen nodes. Vi tilføjer så 1 til tælleren i. Dette kører indtil vores træ når samme højde, som sværhedsgrad-input fra brugeren, eller at vi har arbejdet med alle knuder på listen.



Figur 2

2.3 update_value

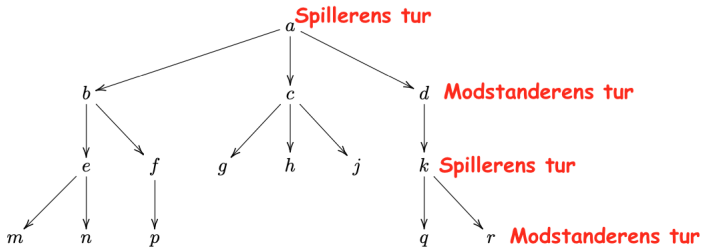
update_value kaldes til sidst i make_tree funktionen. Efter vores træ er bygget, skal knudernes værdier opdateres efter hvorvidt de er min eller max knuder. Funktionen tager roden som input. Hvis roden har børn, kalder vi funktionen rekursivt for børnene, så vi ender nederst i træet. Derefter kan vi tjekke om det er en max eller min knude, og opdatere værdien ved at finde barnet med den henholdsvist laveste eller højeste værdi.

2.4 make_root

make_root laver en knude, vi bruger som roden. Den er, ligesom alle eventuelle knuder i træet, en Node defineret af vores dataklasse. Det er dog vigtigt at bemærke, at det er copy fra del 1 af projektet, som bruges som data for brættet. Dette gøres for at undgå at ændre i vores egentlige bræt, mens vi stadig bruger et opdaterede bræt.

2.5 _max

Hjælpefunktionen _max bruger modulus 2 på laget af knuden for at determinere, om en given knude er en max- eller minknude. Det første lag (roden) er en maxknude, og derefter hver anden knude. Se figur 3 nedenfor.



Figur 3

2.6 layer

Layer returnerer laget for en given knude. current_node starter med at være den knude, man indtaster. Hvis den node ikke er roden, går vi ét lag op, og current_node bliver lavet til forældreknuden. Sådan tælles der op, indtil roden nås.

2.7 add_child

add_child laver et nyt barn, som tilføjes til forældre knudes liste af børn. Derudover tilføjes en reference til forældren hos barnet. Til sidst returneres barnet.

2.8 _move_copy

Denne hjælpefunktion er relativ simpel, men yderst vigtig for programmets funktionsevne. _move_copy skaber blot en kopi at det nuværende bræt, udfører barnets træk på kopien og returnerer det nu "nye" bræt. Herefter kan det nye bræt vurderes af vores heuristiske funktion, og knuden kan få sin værdi.

2.9 heu

Vores heuristiske funktion kaldes "heu". Heuristikken er lavet således, at vi tæller, hvor mange af modstanderens brikker, der er tilbage. Det vil sige i den bedste tilstand, er der 0 af modstanderens brikker tilbage (vi har vundet), så vil output af heu være 12. I værste tilfælde har modstanderen alle sine brikker, og output er 0. Se eventuelt figur 9 under test, hvor den heuristiske værdi kan ses for startbrættet.

2.10 next_move

next_move finder det bedste træk for det pågældende træ - og dermed det pågældende bræt. Dette gøres ved, at den først fastsætter rodens første barn til at være det optimale træk, kaldet best. Herefter gennemgår den alle rodens børn og opdaterer best, hvis der findes et barn med en bedre heuristik. Efter barnet med den bedste heuristik er fundet, returneres dets træk.

3 Test af Programmet

I dette afsnit belyses test af vores modul. Ligesom i de forrige dele af projektet har vi ønsket at teste funktionerne enkeltvist så vidt muligt. Nedenfor eksemplificeres test af de funktioner, som vi har testet separat. Alle testene indeholder variablen b, som er make_board fra del 1 af projektet.

3.1 make_tree

Nedenfor er koden for make_tree indsat, som er den del, vi har kørt test på. Det fremgår af denne kode, at vi har tilføjet et print-statement. Dette skyldes, at make_tree returnerer 'None', og vi gerne ville have et output, som vi kunne vurdere.

```
def make_tree(r: Node, depth: int) -> None:
    """Make a tree with depth chosen by player."""
    nodes = [r]
    i = 0
    while i < len(nodes) and layer(nodes[i]) < depth:
        print(nodes[i].move)
        for m in legal_moves(nodes[i].data):
```

```
            new = add_child(nodes[i], m)
            nodes.append(new)
        i = i + 1
    update_value(r)
```

Figur 4 viser outputtet af overstående, når sværhedsgraden er 2. I den første linje af outputtet printes 'None', da spillet først er begyndt, og man er ved roden af træet, som ikke indeholder et træk. De næste tre linjer er mulige første træk, man kan lave. Dette er i overensstemmelse med, at den tænker to træk frem og derved laver en dybde på 2 i træet.

```
make_tree(make_root(b), 2)
None
Move(src=17, trg=13)
Move(src=18, trg=13)
Move(src=19, trg=13)
```

Figur 4

Af figur 5 fremgår den sidste selvstændige test af make_tree. Vi har lavet en print-statement i slutningen af koden efter den kalder update_value(r). make_tree vil printe den opdaterede værdi af roden. Her tester vi for, at den heuristiske værdi bliver opdateret. Dette gøres ved at se, at output ikke er 0.

```
make_tree(make_root(b), 4)
1
```

Figur 5

3.2 make_root

Af figur 6 fremgår test af make_root, som laver roden i træet. Her ses det, at de rigtige felter fra dataklassen indgår.

```
>>> make_root(b)
Node(children=[], parent=None, data=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), value=0, move=None)
```

Figur 6

3.3 layer

I figur 7 fremgår test af layer, hvor roden er indsat. Figuren viser, at roden er lag 0.

```
>>> layer(make_root(b))
0
```

Figur 7

3.4 _max

Nedenfor fremgår test af funktion _max, som viser, at roden er en maxknude. Se figur 8.

```
>>> _max(make_root(b))
True
```

Figur 8

3.5 heu

Heu er også blevet testet separat (se figur 9). Dette gøres den ved at indsætte b. Det ses, at ved starten af spillet er heu 0. Dette viser også, at det er den dårligste tilstand at være i, da modstanderen har alle sine brikker.

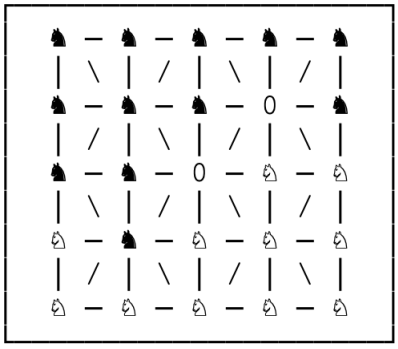
```
>>> heu(b)
0
```

Figur 9

3.6 Spillefase

I sidste testfase kører vi spillet. Her testes de funktioner, som returnerer 'None' og/eller kun fungerer i sammenhæng med andre. Dette fremgår af figur 10 nedenfor.

```
The magical machine moves knight 9 to field 17
```



Figur 10

Ovenfor ses det at autospilleren får sin tur og kan udføre et træk. I dette tilfælde udfører den sorte autospiller et træk fra plads 9 til 17. Vi har desuden testet i alle kombinationer af, hvem der spiller; autospiller versus menneske og autospiller versus autospiller. I begge tilfælde kan autospilleren udføre et træk, hvis træks effektivitet også bliver styret af den sværhedsgrad (dybde), man har valgt. En udfordring, vi opdagede ved denne testfase, er køretiden. Det opleves, at når man spiller med en sværhedsgrad på 7, vil der desto længere inde i spillet,

man når, være en lille ventetid. Spillet fungerer, men man kan komme til at vente på, at autospilleren gennemgår alle mulige træk og bestemmer trækket med den bedste heuristik. På baggrund af vores test kan vi udlede, at vores modul virker.

4 Overvejelser

4.1 Heuristik

Som en del af vores overvejelser, vil vi gerne indrage den valgte heuristik. Vores heuristik er meget simpel. Den giver udelukkende "point" for, at der bliver færre af modstanderens brikker. På den ene side fungerer vores heuristik, og autospilleren vil derfor gå efter en brættilstand med så få af modstanderens brikker som muligt. På den anden side kan heuristikken også være for enkel. Det kan ske, at autospilleren vælger et træk der medfører 2 færre af modstanderens brikker, men samtidigt medfører, at man selv mister 3 brikker. Her kan det diskuteres, om det overhovedet er et godt træk. Det vil vi ikke mene, da man selv hurtigere bliver taget ud af spillet. Dette tager vores heuristik ikke højde for.

4.2 Køretid

I afsnittet "Test" erfarede vi, at der er en ulempe i køretiden af vores kode. Vi oplevede, at desto højere sværhedsgrad, desto længere køretid vil der være. Dette betyder, at når man spiller mod autospilleren med en høj sværhedsgrad skal man vente på, at den gennemgår mulige træk. Det samme gør sig gældende, når autospilleren spiller mod sig selv, hvor der kan gå nogle minutter før spillet afsluttes. Dette betyder en køretid, som ikke er optimal. Det påvirker naturligvis spillekvaliteten, dog fungerer vores program ellers fint.

5 Konklusion

På baggrund af rapporten kan vi konkludere, at vi har indfriet vores fundamentale ønske: Lavet en heuristik, som vurderer et træk, og tillader at vælge det bedste. Dens primære opgave var at vurdere baseret på en simpel tankegang. Denne tankegang gik på, at få af modstanderens brikker er godt. På den anden side er der også ulemper ved en simpel heuristik. Vi har desuden formået at lave minimax.py, som kan lave et træ med en dybde svarende til sværhedsgraden. På baggrund af dette træ med de tilhørende heuristiske værdier, kan programmet vælge det bedste træk. Herudover er det lykkedes os at lave test på alle vores funktioner, som viser spillet virker. Køretiden af vores kode er den eneste ulempe som påvirker spillekvaliteten. Afslutningsvist kan vi konkludere, at minimax.py fungerer sammen med vores implementeringer fra del 1 og 2 af projektet.

6 Appendix

```
from board import *
from move import *
from dataclasses import dataclass
from typing import Any

@dataclass
class Node:
    children: list[Any]
    parent: Any
    data: Board
    value: int #Den heuristiske værdi
    move: Move

def make_tree(r: Node, depth: int) -> None:
    """Make a tree with depth chosen by player."""
    nodes = [r]
    i = 0
    while i < len(nodes) and layer(nodes[i]) < depth:
        for m in legal_moves(nodes[i].data):
            new = add_child(nodes[i], m)
            nodes.append(new)
        i = i + 1
    update_value(r)


def update_value(t: Node) -> None:
    """Update the value of the nodes according to whether they
    are min or max nodes.
    """
    if t.children != []:
        for child in t.children:
            update_value(child)
        if _max(t):
            highest = 0
            for c in t.children:
                if c.value > highest:
                    highest = c.value
            t.value = highest
        else:
            lowest = 12
            for c in t.children:
                if c.value < lowest:
                    lowest = c.value
            t.value = lowest
```

```
def make_root(b: Board) -> Node:
    """Make the root of the tree.
    >>> make_root(b)
    Node(children=[], parent=None,
    data=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), value=0, move=None)
    """
    n = Node([], None, copy(b), 0, None)
    return n

def _max(n: int) -> bool:
    """Determine whether a node is a max node.
    _max(make_root(b))
    True
    """
    return layer(n) % 2 == 0

def layer(n: Node) -> int:
    """Return the current layer for a node.
    >>> layer(make_root(b))
    0
    """
    current_layer = 0
    current_node = n
    while current_node.parent != None:
        current_node = current_node.parent
        current_layer = current_layer + 1
    return current_layer

def add_child(p: Node, m: Move) -> Node:
    """Add a child to the given parent. The order is irrelevant."""
    new_board = _move_copy(m, p.data)
    new_child = Node([], p, new_board, heu(new_board), m)
    p.children.append(new_child)
    return new_child

def _move_copy(m: Move, b: Board) -> Board:
    """."""
    c = copy(b)
    move(m, c)
    return copy(c)

def heu(b: Board) -> int:
    """Determine the value of a node. This is the heuristic function.
```

```
    The best value is 12 and the worst 0.
    >>> heu(b)
    0
    """
    counter = 0
    if white_plays:
        for opponent in black(b):
            counter = counter + 1
    else:
        for opponent in white(b):
            counter = counter + 1
    return 12 - counter


def next_move(b: Board, depth: int) -> Move:
    """Find the optimal move for the autoplayer."""
    root = make_root(b)
    make_tree(root, depth)
    best = root.children[0]
    for c in root.children:
        if c.value > best.value:
            best = c
    return best.move
```