Alquerque

Eksamensprojekt DM574
Del 3

Vejleder:
Luís Cruz-Filipe

Lavet af:
Søren Rosendahl Christensen
soerc23@student.sdu.dk
project group 20

Resumé

Denne rapport handler om hvordan jeg har implementeret minimax algoritmen til at finde det næste træk for spilleren. Algoritmen skal vælge det bedste træk ud fra alle mulige sekvenser af træk et valgt antal træk forud. Det bedste næste træk har størst sandsynlighed for at vinde fremadrettet.

Abstract

This report is about how I have implemented the minimax algorithm to find the next move for the player. The algorithm should choose the best move based on all possible sequences of moves a chosen number of moves ahead. The best next move has the highest probability of winning in the future.

Forord

Rapporten er lavet på første semester af uddannelsen computer-science på kurset DM574 'Introduktion til programmering' og er i den forbindelse eksamensprojektets tredje og sidste del og vil være baggrund for det mundtlige forsvar af projektet til januar 2024.

Det må noteres at som et gruppeprojekt har dette ikke været den største success.

Indhold

1 Indledning

Projektets formål i denne del er at udvikle et modul for autospilleren der kan benyttes af spilmodulet `alquerquer.py` udvilket i første fase. I det følgende kapitel fastlægges projektets formål og problemformulering, som danner grundlaget for projektet. Rapporten afspejler og dokumenterer det udførte projektarbejde på 1. semester for projektets tredje fase i kurset DM574 "Introduktion til programmering".

1.1 Problemformulering

Formålet med projektet er at undersøge, hvordan en autospiller til spilletmodulet `alquerque.py` kan implementeres ved brug af minimax algoritmen. Løsningen skal benytte teknikker fra pensum til formålet.

1.2 Kravspecifikation

Overordnet så skal modulet minimax indeholde en metode som retunere det bedste træk for næste spiller: `next_move(b: Board, depth: int) -> Move`.

Spiltræet som skal repræsentere spillets mulige tilstande konstureres i to trin.

Trin 1: Minimaxtræet

- Hver node er en tilstand af spillet.
- En dybde der begrænser antal træk man vil analysere.
- Hver node `n` er parent til nodes der er spillets tilstand efter mulige træk fra `n`.

Trin 2: Heuristikken

- En node er enten max- eller minnodes. Maxnodes er tilstande med spillerens tur.
- Alle nodes får tildelt en værdi på baggrund af om det er max- eller minnodes.
- Alle træets blade vurderes på baggrund af heustikken.
- Rodnoden er en maxnode.
- Værdien af en maxnode er værdien af bladet med den højeste værdi.

1.3 Projektafgrænsning

Projektet er ikke udviklet til et slutprodukt. Der vil være plads til forbedringer og optimeringer. Hovedsageligt handler projektet om at gøre brug af de teknikker fra stoffet vi har haft med at gøre. Autospilleren er lavet men ikke begrænset til projektets implementering af Alquerque.

2 Implementering

Implementeringen et minimaxtræet med en heustik baseret på captures og positioner. Det er udviklet til at virke med spilmodulerne `board.py` og `move.py` som repræsenterer brættet og den tilhørende funktionalitet samt datatypen for et træk. I dette kapitel redegører jeg for modulet `minimax.py`.

2.1 Minimaxtræ

Minimaxtræet implementeres ved brug af to definerede datatyper `Node` og `Tree`. Disse danner en dedikeret datastruktur til at håndtere spiltræet. I dette afsnit uddyber jeg valget af funktioner og felter.

2.1.1 Node

Datatypen `Node` repræsentere en tilstand i spillet. Det betyder at den skal indeholde brikkernes positioner og alle spilleres mulige træk. Derudover skal den have en reference til dens *parent node* som er tilstanden der førte dertil. Samtidig skal den også selv være *parent node* til alle mulige tilstande den selv føre til givet spilleres muligheder for at rykke sine brikker. De tilstande den føre til kaldes *child nodes*.

Datatypen er implementeret som en class i python og kan ses i Bilag 2

`Node` har følgende felter:

- `board`
  Brættets der indeholer brikkernes positioner implementeret i `board.py`.
- `parent`
  En reference til dens *parent node*.
- `parent_move`
  Fjendens træk der førte til tilstanden implementeret i `move.py`.
- `moves`
  Liste med træk der må foretages fra tilstanden.
- `child_nodes`
  Liste med referencer til de næste tilstande.
- `maximaxing`
  Boolsk værdi der indeholder information hvis' spiller tilstandens tur tilhører. True for den spillende spillers tur.
- `value`
  Integer der indeholder tilstandes værdi for spilleren. Jo højere, jo bedre.

  Og tilhørende funktioner:

- `make_node(board: Board, parent_move: Move=(0,0), parent: Node=None, maximaxing: bool=True, value: int=None) -> Node`
  Returnere en `Node` med default værdier for rodnoden.
- `evaluate_node(node: Node) -> int`
  Vurderer fordelagtigheden af en tilstand for den spillende spiller baseret på det fjendes sidste træk. Jo lavere, jo bedre. Da vi ønsker at minimere fjendes positive udfald.
- `expand_node(node: Node) -> None`
  Tilføjer alle mulige permutationer af tilstanden til `child_nodes` som `moves` føre til.
- `node_moves(board: Board) -> list[Move]`
  Retunere træk der er captures blandt mulige træk. Hvis de ikke findes retunere den alle mulige træk.
- `_next_move(node: Node) -> Move`
  Retunere trækket der føre tilstanden fra roden mod `node`.

Hjælpefunktionerne:

- `_distance(move: Move) -> int`
  Returnere et træks absolutte afstand.

2.1.2 Tree

Datatypen `Tree` repræsenterer spiltræet som starter ved *roden* og ender i *bladende*. *Bladene* er alle mulige tilstande et bestemt antal træk fra den nuværende tilstand svarende til den valgte dybde.

Datatypen er også implementeret som en class i python og kan ses i Bilag 3. Alle tilstande bruger instances af `Node`.

`Tree` har følgende felter:

- `root`
  Reference til den node der repræsentere spillets nuværende tilstand.
- `leafes`
  Liste med bladene.

Og tilhørende funktioner:

- `make_tree(board: Board, height: int) -> Tree`
  Returnere et `Tree` med dybden `height`.
- `contruct_tree(node: Node, height: int, tree: Tree, acc: int=0) -> None`
  Laver træet og vurdere bladene baseret på heustikken. Bladene tilføjes til `tree.leafes`.
- `find_max(tree) -> Node`
  Retunere bladet med den største værdi.

2.2 Algoritme

Implementationen af minimax algoritmen bruger en kombination af programmeringsteknikker til at opnå dette. Træet konstrueres i funktionen `contruct_tree` rekursivt for for hver *node* og dens *child nodes*. Undervejs akkumeleres værdierne af alle nodes langs stien fra *roden* til *bladet* hvor den tildeles. Her bruges også bruges bivirkninger til at tilføje bladene til træet i `leafes`

`node_moves` sortere i de mulige træk. Det giver færre forgreninger hvilket betyder færre rekursive kald. Derudover tvinger det også autospilleren til at capture hvilket igen simplificere evalueringsfunktionen. Ulempen er at der er tilfælde hvor angreb ikke er den mest optimale taktik.

`expand_node` tilføjer iterativt de næste tilstande til en parent for alle træk tilstanden tillader (*undtagen de frasorterede*). Hvert træk simuleres på en kopi af brættet, hvorefter det nye bræt, trækket og parent noden selv danner de nye tilstande som tilstanden føre til. Max og min nodes bestemmes også her – child nodes er det modsatte af deres parent.

`find_max` bruger reduce og finder bladet med den største værdi. Stien fra bladet til roden følges rekursivt af `_next_move` som returnere trækket fra roden der har en sti mod bladet.

2.3 Køretider

Evalueringsfunktionen `evaluate_node` har en køretid på $\mathcal{O}(1)$. Det er bevist besluttet at benytte bevægede afstande fremfor antallet af hvide og sorte brikker.

`expand_node` har en køretid på $\mathcal{O}(n)$. Vi simulere et træk for alle mulige træk.

`find_max` har en køretid på $\mathcal{O}(n)$ Dette kunne forbedres ved at have en sorteret liste.

Køretiden af `contruct_tree` er polynomisk og afhænger af antallet af mulige træk for de enkelte tilstande. Køretiden stiger voldsomt når en tilstand har mange permutationer.

2.4 Forbedringer

Der er altid plads til forbedringer. Her er en liste over nogle af de tanker jeg har haft om det.

- Autospilleren er langsom fordi et nyt helt træ konstruereres ved hvert træk. I stedet kunne roden rykkes og træet udvides.
- Implementere dynamiske programmeringsmetoder.
- Det er ikke nødvendigt med en liste over blade når et nyt træ laves ved hvert træk.
- Autospilleren kan forbederes ved at undgå at sortere trækkene. I stedet kunne man opstille mere sofistikerede regler i evalueringsfunktionen.
- Encapsulating. Felterne på class instances burde have metoder til de specifikke formål.

3 Dokumentation

Alle funktioner har docstrings. Desværre har jeg været lidt presset med tiden. Doctests mangler.

4 Konklussion

I denne rapport har jeg undersøg og redegjort for hvordan autospiller modulet `minimax.py` kan implementeres. Implementationen lever ikke helt op til forventningerne. F.eks. tildeles alle nodes ikke værdier, men værdierne akkumeleres istedet under konstruktionen af træet. Det er en design fejl. Der er ikke taget højde for at at træet skal genbruges.

Alle mulige træk bliver heller ikke taget i betragning. Det er et dårligt trade off.

Og så måske nogle flere diagrammer i rapporten.

Når det er sagt, så har det været en utrolig spændende opgave som jeg har nydt at arbejde med hen over jul og nytår. Jeg er nogenlunde tilfreds med navngivningen og læsbarheden af programmet.

Således er det lykkedes at implementere `minimax.py`. Jeg håber at der er små forbedringer at se (*især i rapporten*). Det har været en fornøjelse. Tak for opmærksomheden.

5 Bilag

```python
def next_move(b: Board, n: int = 3) -> Move:
    """Returns the next move for the autoplayer."""
    tree = make_tree(b, n)
    return _next_move(find_max(tree))
```

Listing 1: next_move

```python
@dataclass
class Node:
    board: Board
    parent: None          # Node
    parent_move: Move
    moves: list[Move]
    child_nodes: list     # list[Node]
    maximizing: bool
    value: int

def make_node(board: Board,
              parent: Node=None,
              parent_move: Move=(0, 0),
              maximizing: bool=True,
              value: int=None) -> Node:
    """Returns a node representing the state of the game."""
    return Node(board, parent, parent_move, node_moves(board), [], maximizing, value
    )

def node_moves(board: Board) -> list[Move]:
    """Returns the legal capturing moves if any otherwise returns the legal moves.
    """
    moves = legal_moves(board)
    captures = [m for m in moves if 6 < abs(m[0] - m[1]) or 3 > abs(m[0] - m[1])]
    return captures or moves

def expand_node(node: Node) -> None:
    """Adds possible state permutations to a state."""
    for m in node.moves:
        new_board = copy(node.board)
        move(m, new_board)
        node.child_nodes.append(make_node(new_board, node, m, not node.maximizing))

def evaluate_node(node: Node) -> int:
    """Evaluates the how positive a state is based on the previous move (the enemy
    move)."""
    if node.child_nodes == []:
        # Evt win value
        return 0
    elif node.maximizing:
        # Fjendens captures er skidt for spilleren
        return 0 -_distance(node.parent_move)
    else:
        return  _distance(node.parent_move)
def _distance(move: Move) -> int:
    """Returns the absolute distance of a move."""
    return  abs(move[0] - move[1])
```

Listing 2: Node

```python
@dataclass
class Tree:
    root: Node
    leafes: list[Node]

def make_tree(board: Board,height: int) -> Tree:
    """Initializes a GameTree with a given height to a given board."""
    tree = Tree(make_node(board), [])
    construct_tree(tree.root, tree, height)
    return tree

def construct_tree(node: Node, tree: Tree, height: int, acc: int=-1)  -> None:
    """Builds the heuristic tree of a given height"""
    if height == 0 or node.moves == []:
        node.value = acc+evaluate_node(node)
        tree.leafes.append(node)
    else:
        expand_node(node)
        # Expand subnodes rekursivt
        for i in range(len(node.child_nodes)):
            construct_tree(node.child_nodes[i], tree, height-1, acc+evaluate_node(
    node.child_nodes[i]))

def find_max(tree) -> Node:
    """Returns the leaf with the maximum accumulated value"""
    return reduce(lambda x,y: x if x.value > y.value else y, tree.leafes)

def _next_move(leaf: Node) -> Move:
    if leaf.parent.parent_move == (0, 0):
        return leaf.parent_move
    else:
        return _next_move(leaf.parent)
```

Listing 3: Tree