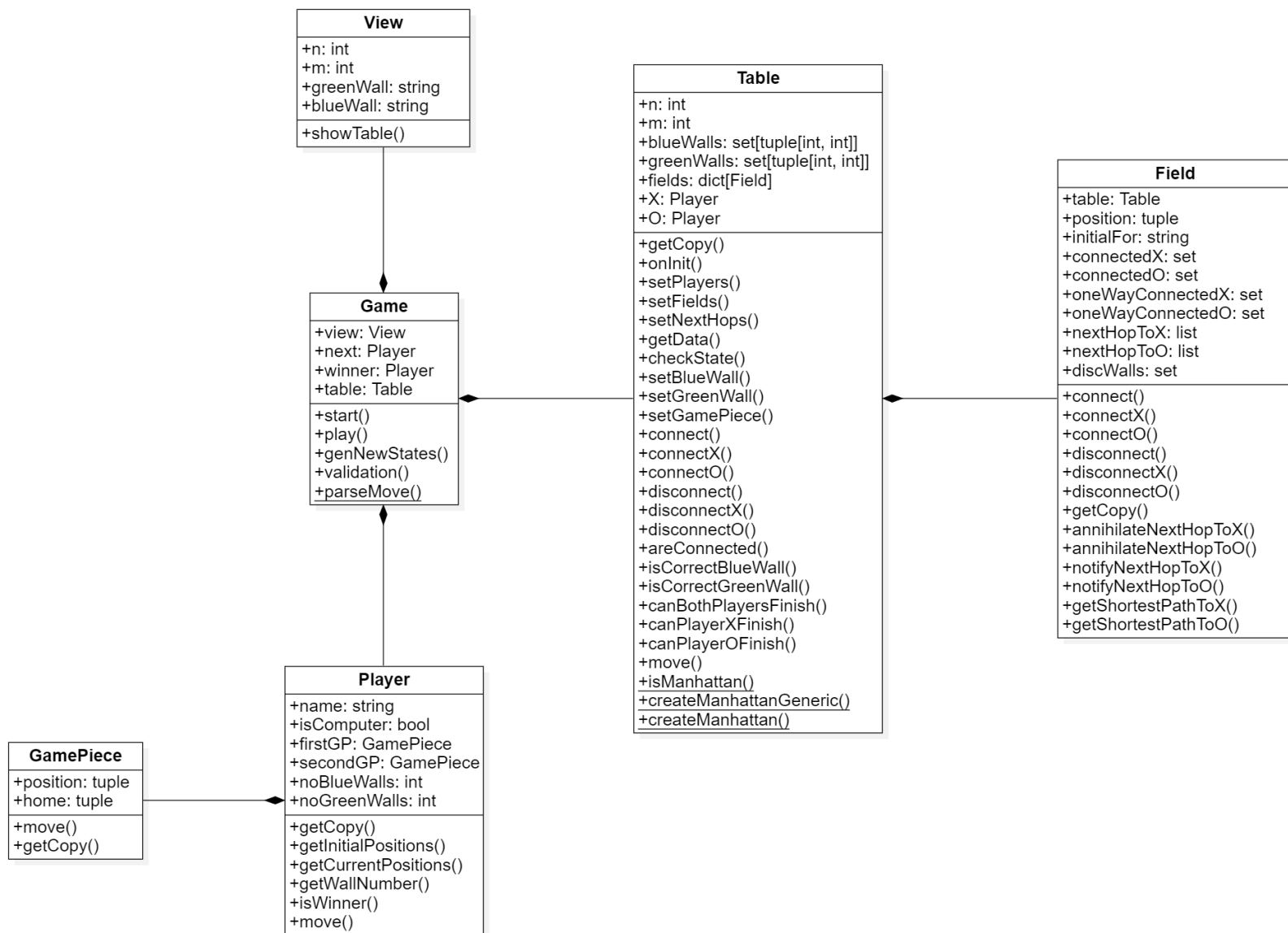




Извештај пројекта „Blockade”

Теодора Коцић 17190
Стефан Алексић 16995

На слици 1 приказан је класни дијаграм у којем су наведене све до сада формиране класе у сврху решавања проблема. С обзиром да су поједине класе претрпеле промене у односу на прву фазу, истаћи ћемо их уколико сматрамо да су битне за нагласити.



Слика 1 Класни дијаграм

1. Фаза – Формулација проблема и интерфејс
2. Фаза – Оператори промене стања
3. Фаза – Мин-макс алгоритам

С обзиром на то да уколико један потез подразумева померање једне од фигура $2 * 8$ и постављање зида, плавог или зеленог $(11 - 1) * (14 - 1) * 2$, то на почетку игре резултује укупном броју стања од 4160. Касније овај број опада, међутим и даље је врло висок. Како бисмо успели да реализујемо мин-макс алгоритам, који број стања експоненцијално повећава са порастом дубине тражења, наш тим је одлучио да усвоји предложене измене у правилима:

- Потез подразумева померање једног од пешака, или постављање једне врсте зида, плавог или зеленог. Ово резултује укупном броју стања 276, на почетку игре,
- Играч својим пешаком може блокирати пут противнику, што смањује број испитивања валидних стања,
- Провера да ли постоји пут се врши само уколико дође до надовезивања зидова.

С тим у вези, пре описа треће фазе, навешћемо измене које је претрпела апликација како бисмо ускладили нова правила и што ефикасније решили проблем.

Класа *Game*:

```
@staticmethod
def parseMove(stream):
    ret = {
        'correct': False,
        'message': None,
        'green wall': None,
        'blue wall': None,
        'game piece': None
    }
    m = stream.replace('[', ' ').replace(']', ' ').upper().split(' ')
    if m[0] in ['X', 'O']:
        if m[1] not in ['1', '2']:
            ret['message'] = "Invalid piece ID!"
        else:
            try:
                ret['game piece'] = {
                    'name': m[0],
                    'choice': int(m[1]),
                    'position': tuple([ord(x)-55 if x >= 'A' else ord(x)-48 for x in m[2:4]])
                }
                ret['correct'] = True
            except Exception as e:
                ret['message'] = "Incorrect positional coordinates!"
    elif m[0] == 'B':
        try:
            ret['blue wall'] = { 'position': tuple([ord(x)-55 if x >= 'A' else ord(x)-48 for x in m[1:3]]) }
            ret['correct'] = True
        except Exception as e:
            ret['message'] = "Incorrect positional coordinates!"
    elif m[0] == 'G':
        try:
            ret['green wall'] = { 'position': tuple([ord(x)-55 if x >= 'A' else ord(x)-48 for x in m[1:3]]) }
            ret['correct'] = True
        except Exception as e:
            ret['message'] = "Incorrect positional coordinates!"
    else:
        ret['message'] = "Invalid move choice!"
    return ret
```

Исечак кода 1 – функција *parse()*

Функција је ради читљивости кода промењена да за повратну вредност има *dictionary* са парсираним вредностима, односно индикатором уколико је корисник исправно унео свој потез, поруком која описује проблем при уносу, уколико исти постоји. Уколико се ради о померању неке од фигура, битан нам је индикатор фигуре који је сачињен од њеног типа и броја, као и позиција на коју се поставља. Када је реч о зидовима, једина битна ствар јесте позиција зида. За некоректне вредности, корисник добија поруку о грешки и бива од њега захтевано да поново изврши унос.

```

def validateGamePieceMove(self, move):
    ret = {'valid': False}
    if self.next.name != move['name']:
        ret['message'] = "Not your turn!"
    elif 1 > move['position'][0] > self.table.n:
        ret['message'] = "Row index out of bounds!"
    elif 1 > move['position'][1] > self.table.m:
        ret['message'] = "Column index out of bounds!"
    elif move['position'] in self.next.getCorrectPositions():
        ret['message'] = "You're already on that position!"
    elif not self.table.isConnected(self.next.getCorrectPositions()[move['choice'] - 1],
move['position'], move['name']):
        ret['message'] = "Can't move there!"
    else:
        ret['valid'] = True
        ret['game piece'] = move
    return ret

```

Исечак кода 2 – Функција *validateGamePieceMove(...)*

Ово је нова функција креирана у сврху валидације померања пешака. За повратну вредност такође има *dictionary*, а проверава се да ли је прослеђени идентификатор играча идентичан играчу који је на потезу, да ли је позиција пешака пре свега у опсегу табеле, провера да ли корисник случајно не стаје на неку од позиција његових пешака, односно другог пешака и тренутне позиције пешака ког помера, као и то да ли заправо може да се помери на дато поље, на основу тога да ли је тренутно поље на ком се налази пешак „повезано“ (*Manhattan* образац са специјалним случајима када се на путу нађе противников пешак) са пољем на које жели да се помери.

```

def validateBlueWall(self, wall):
    ret = {'valid': False, 'message': ""}
    if self.next.getWallNumber()[0] < 1:
        ret['message'] = "No blue walls left!"
    elif 1 > wall['position'][0] > self.table.n - 1:
        ret['message'] = "Row index out of bounds!"
    elif 1 > wall['position'][1] > self.table.m - 1:
        ret['message'] = "Column index out of bounds!"
    elif not self.table.isCorrectBlueWall(wall['position']):
        ret['message'] = "Can't put a blue wall on the given
position!"
    elif not
self.table.isConnectedBlueWall(wall['position']):
        ret['valid'] = True
        ret['blue wall'] = wall
    else:
        temp = self.table.getCopy()
        temp.setBlueWall(wall)
        if not temp.canPlayerXFinish(True):
            ret['message'] += "Player X can't finish!"
        if not temp.canPlayerOFinish(True):
            ret['message'] += "Player O can't finish!"
        if temp.canBothPlayersFinish():
            ret['valid'] = True
            ret['virtual'] = temp
    return ret

```

```

def validateGreenWall(self, wall):
    ret = {'valid': False, 'message': ""}
    if self.next.getWallNumber()[1] < 1:
        ret['message'] = "No green walls left!"
    elif 1 > wall['position'][0] > self.table.n - 1:
        ret['message'] = "Row index out of bounds!"
    elif 1 > wall['position'][1] > self.table.m - 1:
        ret['message'] = "Column index out of bounds!"
    elif not self.table.isCorrectGreenWall(wall['position']):
        ret['message'] = "Can't put a green wall on the given
position!"
    elif not
self.table.isConnectedGreenWall(wall['position']):
        ret['valid'] = True
        ret['green wall'] = wall
    else:
        temp = self.table.getCopy()
        temp.setGreenWall(wall)
        if not temp.canPlayerXFinish(True):
            ret['message'] += "Player X can't finish!"
        if not temp.canPlayerOFinish(True):
            ret['message'] += "Player O can't finish!"
        if temp.canBothPlayersFinish():
            ret['valid'] = True
            ret['virtual'] = temp
    return ret

```

Исечак кода 3 – Функције за валидацију постављања плавог и зеленог зида *validateGreenWall(...)* и *validateBlueWall(...)*

Обе функције функционишу на идентичан начин. Повратна вредност је и овде типа *dictionary* и садржи индикатор да ли је потез постављања зида валидан, уколико није, придружује се и кључ који ближе описује грешку, а уколико јесте валидан потез, односно корисник има зид одговарајуће врсте који може да постави, позиција зида не излази из опсега табле, такође се функцијама *isCorrectBlueWall(...)* и *isCorrectGreenWall(...)* проверава да ли се зид који корисник жели да постави не преклапа са зидовима који су већ постављени и зид се не надовезује ни на један до сада постављени зид, односно не постоји шанса да се блокира неки пут. Тада је повратна вредност од интереса позиција зида. Алтернатива овога јесте да се десило надовезивање зидова, које резултује проверу блокаде путева у оквиру копије тренутног стања на којој је одигран потез постављања зида, уколико се није десило блокирање пута ниједном од играча, ради ефикасности се враћа то

стање на ком је симулиран потез, у супротном потез није валидан и корисник добија информацију које путеве је блокирао.

```
def play(self):
    while not self.winner:
        parsedMove = Game.parseMove(input(f"{self.next.name} is on the move!\n"))
        if parsedMove['correct']:
            if parsedMove['game piece']:
                move = self.validateGamePieceMove(parsedMove['game piece'])
                if move['valid']:
                    self.table.setGamePiece(move['game piece'])
            elif parsedMove['blue wall']:
                move = self.validateBlueWall(parsedMove['blue wall'] | {'next': self.next.name})
                if move['valid']:
                    if move.get('virtual', None):
                        self.table = move['virtual']
                    else:
                        self.table.setBlueWall(move['blue wall'])
            else:
                move = self.validateGreenWall(parsedMove['green wall'] | {'next': self.next.name})
                if move['valid']:
                    if move.get('virtual', None):
                        self.table = move['virtual']
                    else:
                        self.table.setGreenWall(move['green wall'])
            if move['valid']:
                self.table.showPaths(True)
                self.view.showTable(*self.table.getData())
                self.next = self.table.X if self.next.name == self.table.0.name else self.table.0
                self.table.checkState()
            else:
                print(move['message'])
        else:
            print(parsedMove['message'])
    print(f"{self.winner.name} won! Congrats!")
```

Исечак кода 4 – Функција плаз()

Ова функција је измењена тако да се прилагоди претходно описаним функцијама, односно да испарсира унос, конвертовајући га у потез, изврши валидацију истог, одигра валидан потез, прикаже ново стање, замени играча који је на потезу и провери да ли нема победника, што је услов станка у *while* петљи. Такође, прикаже поруке придружене невалидном потезу.

```
def genNewStates(self):
    states = {
        'blue wall': list(),
        'green wall': list(),
        'game piece': list()
    }
    if self.next.noBlueWalls > 0:
        for i in range(1, self.table.n):
            for j in range(1, self.table.m):
                if self.table.isCorrectBlueWall((i, j)):
                    temp = self.table.getCopy()
                    temp.setBlueWall({
                        'position': (i, j),
                        'next': self.next.name
                    })
                    if not self.table.isConnectedBlueWall((i, j)) or temp.canBothPlayersFinish(True, True):
                        states['blue wall'].append(temp)
    if self.next.noGreenWalls > 0:
        for i in range(1, self.table.n):
            for j in range(1, self.table.m):
                if self.table.isCorrectGreenWall((i, j)):
                    temp = self.table.getCopy()
                    temp.setGreenWall({
                        'position': (i, j),
                        'next': self.next.name
                    })
                    if not self.table.isConnectedGreenWall((i, j)) or temp.canBothPlayersFinish(True, True):
                        states['green wall'].append(temp)
    if self.next.name == "X":
        positions = self.next.getCorrectPositions()
        for choice in range(1, 3):
            for newPos in self.table.fields[positions[choice - 1]].connectedX:
                states['game piece'].append(self.table.getCopy())
```

```

        states['game piece'][-1].setGamePiece({
            'previous position': positions[choice - 1],
            'position': newPos,
            'name': "X",
            'choice': choice
        })
    elif self.next.name == "O":
        positions = self.next.getCorrectPositions()
        for choice in range(1, 3):
            for newPos in self.table.fields[positions[choice - 1]].connected0:
                states['game piece'].append(self.table.getCopy())
                states['game piece'][-1].setGamePiece({
                    'previous position': positions[choice - 1],
                    'position': newPos,
                    'name': "O",
                    'choice': choice
                })
    return states

```

Исечак кода 5 – функција за генерисање нових стања

Функција сада генерише три врсте стања, у зависности од потеза, то су стања померања фигуре, постављања плавог и стање у које се долази постављањем зеленог зида на тренутној табли. Сва стања су независна, а при постављању зида се и овде врши провера да ли се зид надовезује на већ постављане и уколико ово јесте случај, проверавају блокаде пута.

Класа *Table*:

```

def setPaths(self):
    self.setPathsX()
    self.setPathsO()

def setPathsX(self):
    self.xPaths = self.findShortestPathsX(self.X.getCorrectPositions(), self.O.getInitialPositions())

def setPathsO(self):
    self.oPaths = self.findShortestPathsO(self.O.getCorrectPositions(), self.X.getInitialPositions())

```

Исечак кода 6 – Функције за постављање путања

Ове функције позивају друге методе чланице класе *Table*, које проналазе путање од тренутних локација на којима се налазе пешаци одговарајућег играча до почетних поља противничких пешака. Привремено се складиште као атрибут класе *Table* типа лист од по 4 елемента, односно путања, листе позиција поља, кроз која се пролази да би се стигло на тражено поље.

```

def isConnectedBlueWall(self, pos):
    return ((pos[0], pos[1] - 2) in self.blueWalls or
            (pos[0], pos[1] + 2) in self.blueWalls or
            [(pos[0] + x, pos[1] + y)
             for x in range(-1, 2)
             for y in range(-1, 2)
             if (pos[0] + x, pos[1] + y) in self.greenWalls])

def isConnectedGreenWall(self, pos):
    return ((pos[0] - 2, pos[1]) in self.greenWalls or
            (pos[0] + 2, pos[1]) in self.greenWalls or
            [(pos[0] + x, pos[1] + y)
             for x in range(-1, 2)
             for y in range(-1, 2)
             if (pos[0] + x, pos[1] + y) in self.blueWalls])

```

Исечак кода 7 – Функције за проверу надовезивања зидова

У случају плавог зида се испитује да ли се леви или десни сусед, ± 2 за индекс колоне, налазе у скупу већ постављених зидова, односно уколико се позиција плавог зида не пресеца матрицу 3x3 у скупу зелених зидова.

Идентичан случај је и у питању надовезивања зеленог зида, с тим што се проверава да ли горњи или доњи сусед, ± 2 за индекс врсте, припадају скупу постављених зелених зидова, односно позиција зеленог зида пресеца матрицу 3×3 већ постављених зидова који се налазе у скупу плавих зидова.

```
def canBothPlayersFinish(self, updateX=False, updateO=False):
    return self.canPlayerXFinish(updateX) and self.canPlayerOFinish(updateO)

def canPlayerXFinish(self, update=False):
    if update:
        self.setPathsX()
    return None not in self.xPaths

def canPlayerOFinish(self, update=False):
    if update:
        self.setPathsO()
    return None not in self.oPaths
```

Исечак кода 8 – Функције за проверу блокаде путева играчима.

Ове функције проверавају да ли не постоји неки од 4 пута за одређеног играча, након опционог освежавања путања (поновног тражења путева), уколико је *updateX/updateO* флег сетован. Дакле атрибути класе *Table* *xPaths* и *oPaths* су листе од по 4 елемента, а ти елементи листе одговарајућих путања:

xPaths:

- Тренутна позиција X1 => Почетна позиција O1
- Тренутна позиција X1 => Почетна позиција O2
- Тренутна позиција X2 => Почетна позиција O1
- Тренутна позиција X2 => Почетна позиција O2

oPaths:

- Тренутна позиција O1 => Почетна позиција X1
- Тренутна позиција O1 => Почетна позиција X2
- Тренутна позиција O2 => Почетна позиција X1
- Тренутна позиција O2 => Почетна позиција X2

```
def findShortestPathsX(self, xPos, endPos):
    paths = [False] * 4
    queue = {
        'first game piece': {
            'heads': {
                xPos[0]: [xPos[0]]
            },
            'processing': [[xPos[0]]]
        },
        'second game piece': {
            'heads': {
                xPos[1]: [xPos[1]]
            },
            'processing': [[xPos[1]]]
        },
        'first initial': {
            'tails': {
                endPos[0]: [endPos[0]]
            },
            'processing': [[endPos[0]]]
        },
        'second initial': {
            'tails': {
                endPos[1]: [endPos[1]]
            },
            'processing': [[endPos[1]]]
        }
    }
    while None not in paths and False in paths:
        if queue['first game piece']['processing']:
            current, *queue['first game piece']['processing'] = queue['first game piece']['processing']
            for n in self.fields[current[-1]].connectedX:
                e1 = queue['first initial']['tails'].get(n, None)
                if e1 != None and (not paths[0] or len(paths[0]) > len(current) + len(e1)):
                    paths[0] = current + e1
                e1 = queue['second initial']['tails'].get(n, None)
                if e1 != None and (not paths[1] or len(paths[1]) > len(current) + len(e1)):
                    paths[1] = current + e1
                if not queue['first game piece']['heads'].get(n, None):
                    queue['first game piece']['heads'][n] = current + [n]
                    queue['first game piece']['processing'] += [current + [n]]
            if paths[0] and paths[1]:
                queue['first game piece']['processing'].clear()
        else:
            if not paths[0]:
                paths[0] = None
            if not paths[1]:
```

```

        paths[1] = None
    if queue['second game piece']['processing']:
        current, *queue['second game piece']['processing'] = queue['second game piece']['processing']
        for n in self.fields[current[-1]].connectedX:
            el = queue['first initial']['tails'].get(n, None)
            if el != None and (not paths[2] or len(paths[2]) > len(current) + len(el)):
                paths[2] = current + el
            el = queue['second initial']['tails'].get(n, None)
            if el != None and (not paths[3] or len(paths[3]) > len(current) + len(el)):
                paths[3] = current + el
            if not queue['second game piece']['heads'].get(n, None):
                queue['second game piece']['heads'][n] = current + [n]
                queue['second game piece']['processing'] += [current + [n]]
            if paths[2] and paths[3]:
                queue['second game piece']['processing'].clear()
    else:
        if not paths[2]:
            paths[2] = None
        if not paths[3]:
            paths[3] = None
    if queue['first initial']['processing']:
        current, *queue['first initial']['processing'] = queue['first initial']['processing']
        for n in self.fields[current[0]].connectedX | self.fields[current[0]].oneWayConnectedX:
            if current[0] in self.fields[n].connectedX:
                el = queue['first game piece']['heads'].get(n, None)
                if el != None and (not paths[0] or len(paths[0]) > len(current) + len(el)):
                    paths[0] = el + current
                el = queue['second game piece']['heads'].get(n, None)
                if el != None and (not paths[2] or len(paths[2]) > len(current) + len(el)):
                    paths[2] = el + current
                if not queue['first initial']['tails'].get(n, None):
                    queue['first initial']['tails'][n] = [n] + current
                    queue['first initial']['processing'] += [[n] + current]
            if paths[0] and paths[2]:
                queue['first initial']['processing'].clear()
    else:
        if not paths[0]:
            paths[0] = None
        if not paths[2]:
            paths[2] = None
    if queue['second initial']['processing']:
        current, *queue['second initial']['processing'] = queue['second initial']['processing']
        for n in self.fields[current[0]].connectedX | self.fields[current[0]].oneWayConnectedX:
            if current[0] in self.fields[n].connectedX:
                el = queue['first game piece']['heads'].get(n, None)
                if el != None and (not paths[1] or len(paths[1]) > len(current) + len(el)):
                    paths[1] = el + current
                el = queue['second game piece']['heads'].get(n, None)
                if el != None and (not paths[3] or len(paths[3]) > len(current) + len(el)):
                    paths[3] = el + current
                if not queue['second initial']['tails'].get(n, None):
                    queue['second initial']['tails'][n] = [n] + current
                    queue['second initial']['processing'] += [[n] + current]
            if paths[1] and paths[3]:
                queue['second initial']['processing'].clear()
    else:
        if not paths[1]:
            paths[1] = None
        if not paths[3]:
            paths[3] = None
    return paths

def findShortestPaths0(self, oPos, endPos):
    paths = [False] * 4
    queue = {
        'first game piece': {
            'heads': {
                oPos[0]: [oPos[0]]
            },
            'processing': [[oPos[0]]]
        },
        'second game piece': {
            'heads': {
                oPos[1]: [oPos[1]]
            },
            'processing': [[oPos[1]]]
        },
        'first initial': {
            'tails': {
                endPos[0]: [endPos[0]]
            },
            'processing': [[endPos[0]]]
        },
        'second initial': {
            'tails': {
                endPos[1]: [endPos[1]]
            },
            'processing': [[endPos[1]]]
        }
    }
    while None not in paths and False in paths:
        if queue['first game piece']['processing']:

```



```

        current, *queue['first game piece']['processing'] = queue['first game piece']['processing']
        for n in self.fields[current[-1]].connected0:
            el = queue['first initial']['tails'].get(n, None)
            if el != None and (not paths[0] or len(paths[0]) > len(current) + len(el)):
                paths[0] = current + el
            el = queue['second initial']['tails'].get(n, None)
            if el != None and (not paths[1] or len(paths[1]) > len(current) + len(el)):
                paths[1] = current + el
            if not queue['first game piece']['heads'].get(n, None):
                queue['first game piece']['heads'][n] = current + [n]
                queue['first game piece']['processing'] += [current + [n]]
        if paths[0] and paths[1]:
            queue['first game piece']['processing'].clear()
    else:
        if not paths[0]:
            paths[0] = None
        if not paths[1]:
            paths[1] = None
    if queue['second game piece']['processing']:
        current, *queue['second game piece']['processing'] = queue['second game piece']['processing']
        for n in self.fields[current[-1]].connected0:
            el = queue['first initial']['tails'].get(n, None)
            if el != None and (not paths[2] or len(paths[2]) > len(current) + len(el)):
                paths[2] = current + el
            el = queue['second initial']['tails'].get(n, None)
            if el != None and (not paths[3] or len(paths[3]) > len(current) + len(el)):
                paths[3] = current + el
            if not queue['second game piece']['heads'].get(n, None):
                queue['second game piece']['heads'][n] = current + [n]
                queue['second game piece']['processing'] += [current + [n]]
        if paths[2] and paths[3]:
            queue['second game piece']['processing'].clear()
    else:
        if not paths[2]:
            paths[2] = None
        if not paths[3]:
            paths[3] = None
    if queue['first initial']['processing']:
        current, *queue['first initial']['processing'] = queue['first initial']['processing']
        for n in self.fields[current[0]].connected0 | self.fields[current[0]].oneWayConnected0:
            if current[0] in self.fields[n].connected0:
                el = queue['first game piece']['heads'].get(n, None)
                if el != None and (not paths[0] or len(paths[0]) > len(current) + len(el)):
                    paths[0] = el + current
                el = queue['second game piece']['heads'].get(n, None)
                if el != None and (not paths[2] or len(paths[2]) > len(current) + len(el)):
                    paths[2] = el + current
                if not queue['first initial']['tails'].get(n, None):
                    queue['first initial']['tails'][n] = [n] + current
                    queue['first initial']['processing'] += [[n] + current]
            if paths[0] and paths[2]:
                queue['first initial']['processing'].clear()
    else:
        if not paths[0]:
            paths[0] = None
        if not paths[2]:
            paths[2] = None
    if queue['second initial']['processing']:
        current, *queue['second initial']['processing'] = queue['second initial']['processing']
        for n in self.fields[current[0]].connected0 | self.fields[current[0]].oneWayConnected0:
            if current[0] in self.fields[n].connected0:
                el = queue['first game piece']['heads'].get(n, None)
                if el != None and (not paths[1] or len(paths[1]) > len(current) + len(el)):
                    paths[1] = el + current
                el = queue['second game piece']['heads'].get(n, None)
                if el != None and (not paths[3] or len(paths[3]) > len(current) + len(el)):
                    paths[3] = el + current
                if not queue['second initial']['tails'].get(n, None):
                    queue['second initial']['tails'][n] = [n] + current
                    queue['second initial']['processing'] += [[n] + current]
            if paths[1] and paths[3]:
                queue['second initial']['processing'].clear()
    else:
        if not paths[1]:
            paths[1] = None
        if not paths[3]:
            paths[3] = None
    return paths

```

Исечак кода 9 – Алгоритам проналажења путева

Ово је највећа измена у односу на претходно предат програм. *Distance vector* алгоритам искоришћен за проналажење путева у претходној фази је замењен *branching* алгоритмом. Алгоритам је модификован тако да се за један пролаз по ширини анализирају сви следбеници првог и другог почетног поља, као и сви претходници првог и другог крајњег поља. На овај начин се постиже да се радијално из 4 позиције шире потези све док не дође до преклапања одређених елемената у оквиру путања. Предност овога јесте та да се веома брзо пронађе ограђено поље, односно пешак, јер се његова број суседа које је неопходно обрадити

кроз мали број корака испразни, а уколико није пронађен пут и ред за обрађивање суседа испражњен, то аутоматски означава непостојање путева до тог поља.

Дакле повратна вредност јесте листа од 4 елемента, где сваки представља одређени пут. Ови путеви су сви на почетку постављени на *False*, а *while* петља врти све док је један од њих *False*, или се јави било који „немогућ“ пут. На почетку је такође иницијализован *dictionary* који складишти још 4 *dictionary*-ја за одговарајуће позиције, а сваки од њих садржи по *dictionary hash*-иран по последњем елементу у одговарајућим путањама, а њима упарена вредност су заправо те парцијалне путање које су до тада пронађене. У оквиру *processing* кључа се складишти ред путања које се обрађују. Путања је пронађена када се глава, односно последњи елемент у путањи из неког почетног поља јави као претходник репу, односно последњем, технички првом, елементу у оквиру путање која полази из крајњег поља, или супротно, када се реп крајњег поља јави међу главама почетних поља. Када се деси један од ова 2 пресека, идентификује се путања, уколико та путања има најмањи број скокова у односу на све остале путање које могу да се идентификују за тренутног родитеља, онда је то заправо најкраћа путања. Уколико су се идентификовале обе путање за одређено почетно, односно крајње поље, ред који обрађује његове следбенике по ширини бива затворен, у противном се редови пуне све док има суседа за обраду, а путање попуњавају. За случај петљи се поново позивамо на *dictionary* који складишти све до тада идентификоване главе, тј. репове одговарајућих путања и на овај начин се досеже до истих на што бржи начин, што обрада по ширини подразумева.

Остале класе су претрпеле само уклањање атрибута, метода и позива функција које смо користили за *Distance vector* алгоритам, који је имао превише позива, те смо се зато одлучили за итеративни приступ уз помоћ алгоритма гранања.

Даље следи опис имплементације мин-макс алгоритма.