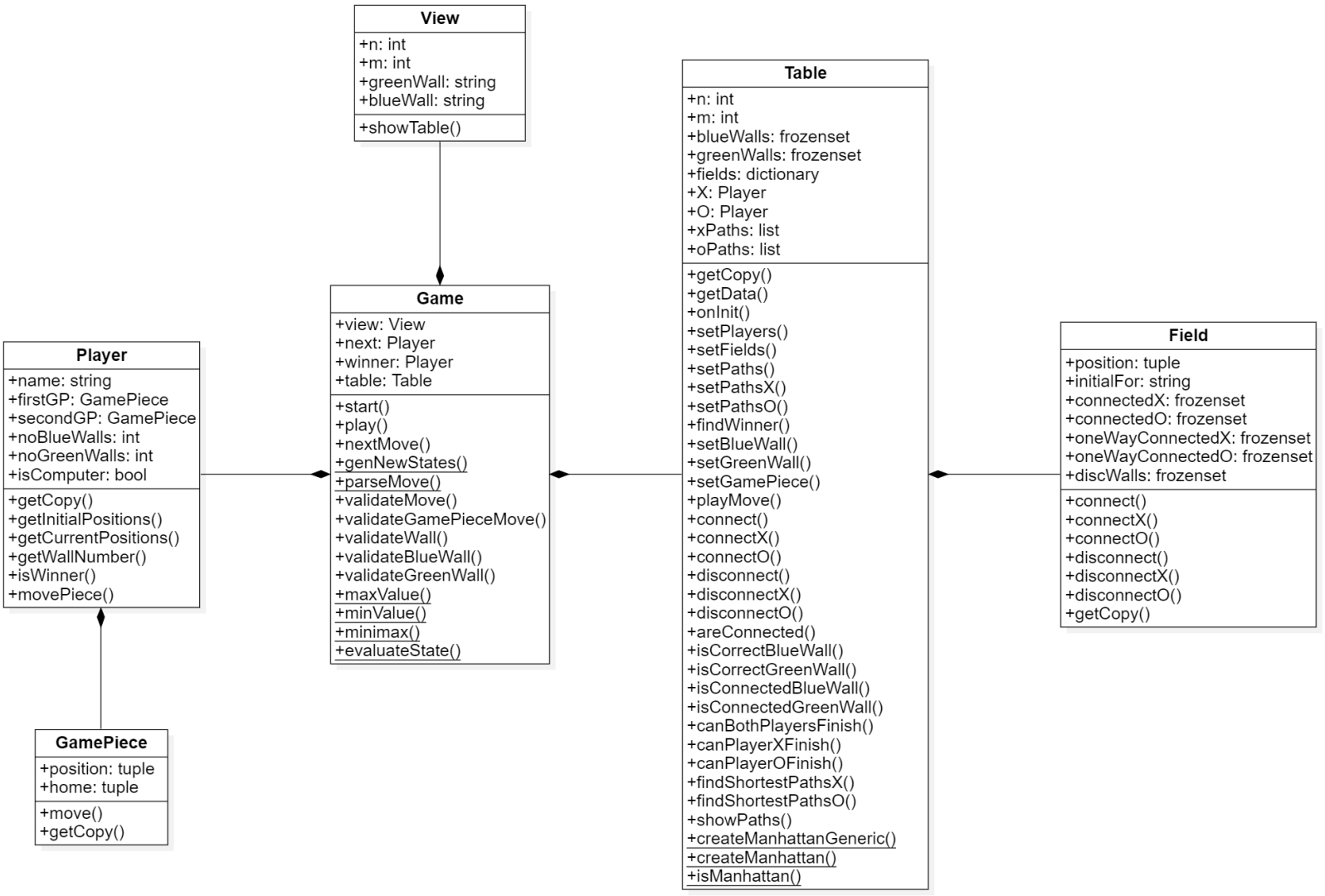
# Извештај пројекта „Blockade”

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

Пре описа треће фазе, у документу наведене су неке од измена које су одрађене пре свега у сврху побољшања (у виду брзине) генерисања нових стања, што је неопходно за израду треће фазе пројекта.

## Класа *Game*:

```
    @staticmethod

    def parseMove(stream):
        ret = {
            'errors': "",
            'wall': {},
            'game piece': {}
        }
        m = stream.replace('[', '').replace(']', '').upper().split(' ')
        if len(m) < 1 or m[0]not in ['X', 'O']:
            ret['errors'] += "Invalid player choice!\n"
        if len(m) < 2 or m[1] not in ['1', '2']:
            ret['errors'] += "Invalid piece identificator!\n"
        if len(m) < 4:
            ret['errors'] += "Missing the new position for the game piece!\n"
        if not ret['errors']:
            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]])
                }
            except Exception as e:
                ret['errors'] += "Incorrect game piece possitional coordinates!\n"
        if len(m) > 4:
            wErr = ""
            if m[4] not in ['B', 'G']:
                wErr += "Invalid wall identificator!\n"
            if len(m) < 7:
                wErr += "Missing the new position for the wall!\n"
            if not wErr:
                try:
                    ret['wall'] = {
                        'type': m[4],
                        'position': tuple([ord(x)-55 if x >= 'A' else ord(x)-48 for x in m[5:7]])
                    }
                except Exception as e:
                    wErr += "Incorrect wall possitional coordinates!\n"
            ret['errors'] += wErr
        return ret
```

*Исечак кода 1 –Функција за „парсирање“ потеза*

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 = {'errors': ""}
        if self.next.name != move['name']:
            ret['errors'] += "Not your turn!\n"
        if 1 > move['position'][0] > self.table.n:
            ret['errors'] += "Row index out of bounds!\n"
        if 1 > move['position'][1] > self.table.m:
            ret['errors'] += "Column index out of bounds!\n"
        if move['position'] in self.next.getCurrectPositions():
            ret['errors'] += "You're already on that position!\n"
        if not self.table.areConnected(
```

```
                self.next.getCurrectPositions()[move['choice'] - 1],
                move['position'],
                move['name']):
            ret['errors'] += "Can't move there!\n"
        if not ret['errors']:
            ret['game piece'] = move
        return ret
```

*Исечак кода 2 – Функција за проверу коректности одиграног потеза у погледу померања фигура*

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

```
    def validateGreenWall(self, wall):

        ret = {'errors': ""}
        if self.next.getWallNumber()[1] < 1:
            ret['errors'] += "No green walls left!\n"
        if 1 > wall['position'][0] > self.table.n - 1:
            ret['errors'] += "Row index out of bounds!\n"
        if 1 > wall['position'][1] > self.table.m - 1:
            ret['errors'] += "Column index out of bounds!\n"
        if not self.table.isCorrectGreenWall(wall['position']):
            ret['errors'] += "Can't put a green wall on the given
position!\n"
        if not ret['errors']:
            if not
self.table.isConnectedGreenWall(wall['position']):
                ret['green wall'] = wall
            else:
                temp = self.table.getCopy()
                temp.setGreenWall(wall)
                if temp.canBothPlayersFinish(True, True):
                    ret['virtual'] = temp
                else:
                    if not temp.canPlayerXFinish():
                        ret['errors'] += "Player X can't
finish!\n"
                    if not temp.canPlayerOFinish():
                        ret['errors'] += "Player O can't
finish!\n"
        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:
            if self.next.isComputer:
                self.table = Game.minimax(self.table, 1, self.next, ([self.table], -50), ([self.table], 50))
                self.nextMove()
```

```
            else:
                parsedMove = Game.parseMove(input(f"{self.next.name} is on the move!\n"))
                if not parsedMove['errors']:
                    move = self.validateMove(parsedMove)
                    if not move['errors']:
                        if move.get('virtual', None):
                            self.table = move['virtual']
                        self.table.playMove(move)
                        self.nextMove()
                    else:
                        print(move['errors'])
                else:
                    print(parsedMove['errors'])
        print(f"{self.winner.name} won! Congrats!")

    def nextMove(self):
        self.next = self.table.X if self.next.name == self.table.O.name else self.table.O
        self.winner = self.table.findWinner()
        self.table.showPaths(True)
        self.view.showTable(*self.table.getData())
```

*Исечак кода 4 – Функција којом се реализује одигравање потеза*

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. Функција је због потребе реализације игре између човека и рачунара измењена тако да уколико је на потезу играч – човек обрада потеза се врши на начин описан у претходном тексту. Уколико је на потезу играч – рачунар потез се одиграва на основу мин-макс алгоритма о којем ће бити више речи касније у тексту.

```
def genNewStates(currentState, player):
        print(f"Generating new states for player {player}. Please be patient...")
        currTime = monotonic()
        playerReff = currentState.X if player == "X" else currentState.O
        states = {
            'blue wall': list(),
            'green wall': list(),
            'new': list()
        }
        if playerReff.noBlueWalls > 0:
            for i in range(1, currentState.n):
                for j in range(1, currentState.m):
                    if currentState.isCorrectBlueWall((i, j)):
                        temp = currentState.getCopy()
                        temp.setBlueWall({
                            'position': (i, j),
                            'next': playerReff.name
                        })
                        if not currentState.isConnectedBlueWall((i, j)) or temp.canBothPlayersFinish(True, True):
                            states['blue wall'].append(temp)
        if playerReff.noGreenWalls > 0:
            for i in range(1, currentState.n):
                for j in range(1, currentState.m):
                    if currentState.isCorrectGreenWall((i, j)):
                        temp = currentState.getCopy()
                        temp.setGreenWall({
                            'position': (i, j),
                            'next': playerReff.name
                        })
                        if not currentState.isConnectedGreenWall((i, j)) or temp.canBothPlayersFinish(True, True):
                            states['green wall'].append(temp)
        gp = {'name': playerReff.name}
        if gp['name'] == "X":
            positions = playerReff.getCurrectPositions()
            for choice in range(1, 3):
                gp['choice'] = choice
                if not states['blue wall'] and not states['green wall']:
                    for newPos in currentState.fields[positions[choice - 1]].connectedX:
                        gp['position'] = newPos
                        temp = currentState.getCopy()
                        temp.setGamePiece(dict(gp))
                        if temp.canBothPlayersFinish(True, True):
                            states['new'].append(temp)
```

```
            else:
                for newPos in currentState.fields[positions[choice - 1]].connectedX:
                    gp['position'] = newPos
                    for bws in states['blue wall']:
                        temp = bws.getCopy()
                        temp.setGamePiece(dict(gp))
                        states['new'].append(temp)
                    for gws in states['green wall']:
                        temp = gws.getCopy()
                        temp.setGamePiece(dict(gp))
                        states['new'].append(temp)
    elif gp['name'] == "O":
        positions = playerReff.getCurrectPositions()
        for choice in range(1, 3):
            gp['choice'] = choice
            if not states['blue wall'] and not states['green wall']:
                for newPos in currentState.fields[positions[choice - 1]].connectedO:
                    gp['position'] = newPos
                    temp = currentState.getCopy()
                    temp.setGamePiece(dict(gp))
                    states['new'].append(temp)
            else:
                for newPos in currentState.fields[positions[choice - 1]].connectedO:
                    gp['position'] = newPos
                    for bws in states['blue wall']:
                        temp = bws.getCopy()
                        temp.setGamePiece(dict(gp))
                        states['new'].append(temp)
                    for gws in states['green wall']:
                        temp = gws.getCopy()
                        temp.setGamePiece(dict(gp))
                        states['new'].append(temp)
    print(f"Generated {len(states['new'])} states in {monotonic() - currTime}s!")
    return states['new']
```

*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.getCurrectPositions(), self.O.getInitialPositions())

    def setPathsO(self):
        self.oPaths = self.findShortestPathsO(self.O.getCurrectPositions(), 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 атрибути 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 3x3 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']
             el = queue['first initial']['tails'].get(current[-1], None)
```

```
                if el != None and (not paths[0] or len(paths[0]) > len(current) + len(el) - 1):
                    paths[0] = current + el[1:]
                el = queue['second initial']['tails'].get(current[-1], None)
                if el != None and (not paths[1] or len(paths[1]) > len(current) + len(el) - 1):
                    paths[1] = current + el[1:]
                for n in self.fields[current[-1]].connectedX:
                    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']
                el = queue['first initial']['tails'].get(current[-1], None)
                if el != None and (not paths[2] or len(paths[2]) > len(current) + len(el) - 1):
                    paths[2] = current + el[1:]
                el = queue['second initial']['tails'].get(current[-1], None)
                if el != None and (not paths[3] or len(paths[3]) > len(current) + len(el) - 1):
                    paths[3] = current + el[1:]
                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']
                el = queue['first game piece']['heads'].get(current[0], None)
                if el != None and (not paths[0] or len(paths[0]) > len(current) + len(el) - 1):
                    paths[0] = el + current[1:]
                el = queue['second game piece']['heads'].get(current[0], None)
                if el != None and (not paths[2] or len(paths[2]) > len(current) + len(el) - 1):
                    paths[2] = el + current[1:]
                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']
                el = queue['first game piece']['heads'].get(current[0], None)
                if el != None and (not paths[1] or len(paths[1]) > len(current) + len(el) - 1):
                    paths[1] = el + current[1:]
                el = queue['second game piece']['heads'].get(current[0], None)
                if el != None and (not paths[3] or len(paths[3]) > len(current) + len(el) - 1):
                    paths[3] = el + current[1:]
                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 findShortestPathsO(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]].connectedO:
                    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]].connectedO:
                    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]].connectedO | self.fields[current[0]].oneWayConnectedO:
                    if current[0] in self.fields[n].connectedO:
                        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]].connectedO | self.fields[current[0]].oneWayConnectedO:
                if current[0] in self.fields[n].connectedO:
                    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 се извршава 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.

## *Класа Game:*

```
        @staticmethod
        def evaluateState(state, player):
            winner = state.findWinner()
            if winner:
                if winner.name == player:
                    return -49
                else:
                    return 49
            else:
                if state.canBothPlayersFinish(True, True):
                    xMin = min(map(lambda x: len(x), state.xPaths))
                    oMin = min(map(lambda x: len(x), state.oPaths))
                    if player == "X":
                        return xMin - oMin
                    else:
                        return oMin - xMin
                else:
                    return -49
```

*Исечак кода 10 – Функција процене стања*

Функција процене стања најпре врши проверу да ли постоји победник за прослеђено стање. Ако је победник онај играч који је тренутно на потезу враћа неку веома малу вредност (alpha = -49), односно велику вредност (beta = 49) уколико је победник играч који није тренутно на потезу. Уколико нема победника прво се врши испитивање да ли ни једна од фигура није блокирана. Уколико овај услов није задовољен враћа се поново неки веома мали број, међутим ово није валидно стање и не би требало да се дође до овога. Али у супротном, ако поменути услов јесте задовољен тражи се минимум дужина најкраћих путева за оба играча па је онда повратна вредност функције разлика одређених одговарајућих минимума играча и то у зависности од тога који је играч на потезу.

```
@staticmethod
    def maxValue(states, depth, player, alpha, beta):
        if depth == 0:
            return (states, Game.evaluateState(states[-1], player))
        else:
            for branchState in Game.genNewStates(states[-1], player):
                alpha = max(alpha, Game.minValue(states + [branchState], depth - 1, "X" if player ==
                            "O" else "O", alpha, beta), key=lambda x: x[1])
                if alpha[1] >= beta[1]:
                    return beta
        return alpha

    @staticmethod
    def minValue(states, depth, player, alpha, beta):
        if depth == 0:
            return (states, Game.evaluateState(states[-1], player))
        else:
            for branchState in Game.genNewStates(states[-1], player):
                beta = min(beta, Game.maxValue(states + [branchState], depth - 1, "X" if player ==
                           "O" else "O", alpha, beta), key=lambda x: x[1])
                if beta[1] <= alpha[1]:
                    return alpha
        return beta
```

*Исечак кода 11 – Функције које враћају најбоље вредности за играче (Макс и Мин редом у овом контексту)*

Прва и друга функција (*maxValue(...)* и *minValue(...)*) имају доста сличности и ради се о рекурзивним функцијама. Из рекурзије се излази када се дође до максималне дубине и тада се позива, претхдно објашњена, функција процене стања. За све дубине које су мање од максималне пролази се кроз сва стања која се генеришу на основу тренутне ситуације, при чему се за сва ова стања најпре одреди вредност alpha (при чему је почетна вредност -49 што је дефинисано у функцији процене стања) као максималну вредност од тренутне вредности коју узима alpha и вредности која је најбоља за противника, која се добија позивом функције *minValue(...)* где су аргументи слични као у позиву функције *maxValue(...)* где је листа стања проширена и тренутно обрађиваним стањем, а дубина смањена за један (због рекурзије). Ако ова ново изгенерисана вредност alpha-е није мања од вредности коју узима beta (иницијална вредност је 49 што је

дефинисано у функцији процене стања), функција *maxValue(...)* као повратну вредност има beta = tuple (листа стања, вредност коју узима beta). Ако претходно описивани услов никада не буде задовољен фунција враћа alpha = tuple (листа стања, вредност која је изгенерисана за alpha).

Функција *minValue(...)* веома је слична претходној функцији, једино се разликује што се за сва стања у рекурзији одређује вредност beta као минимум тренутне вредности коју узима beta и вредности која је најбоља за противника, која се добија позивом функције *maxValue(...)* где су аргументи слични као у позиву функције *minValue(...)* где је листа стања проширена и тренутно обрађиваним стањем, а дубина смањена за један (због рекурзије). Ако ова ново изгенерисана вредност beta-е није већа од вредности коју узима алпха (иницијална вредност је -49 што је дефинисано у функцији процене стања), функција *minValue(...)* као повратну вредност има alpha = tuple (листа стања, вредност коју узима alpha). Ако претходно описивани услов никада не буде задовољен фунција враћа beta = tuple (листа стања, вредност која је изгенерисана за beta).

```
@staticmethod
    def minimax(state, depth, player, alpha, beta):
        return Game.maxValue([state], depth, player.name, alpha, beta)[0][1]
```

*Исечак кода 12 – Функција која користу мин-макс алгоритам са alpha-beta одсецањем*

Функцију *minimax(...)* позивамо приликом одигравања потеза за играча који је рачунар. Коришћењем претходно дефинисаних функција ова функција дефинише потез који је најбоље одиграти за игрча који је рачунар.