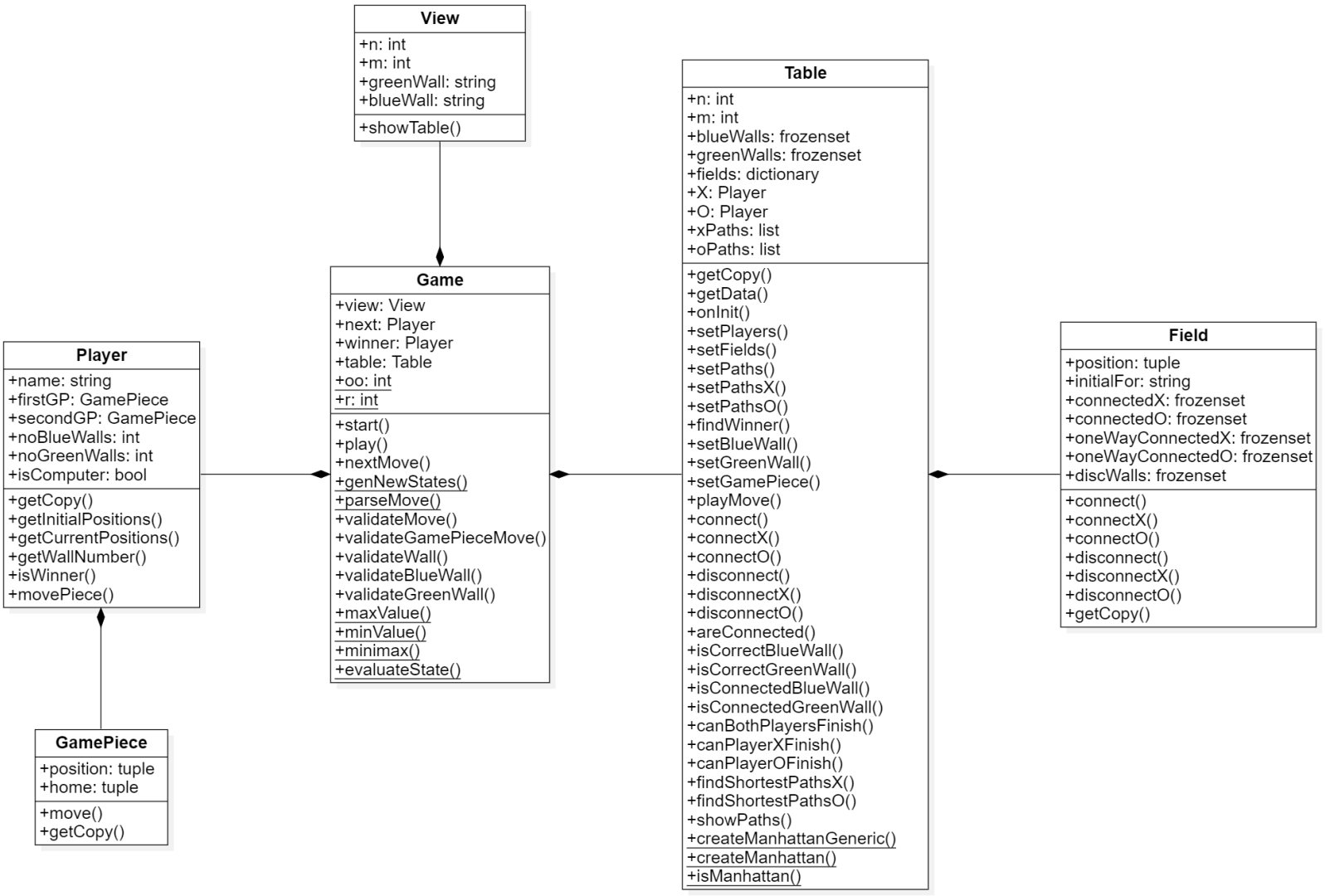
Извештај пројекта „Blockade”

Теодора Коцић 17190
Стефан Алексић 16995

На слици 1 приказан је коначни класни дијаграм у којем су наведене све до сада формиране класе у сврху решавања проблема. Може се приметити да је претрпео скоро незначајне измене у односу на претходну фазу.



Слика 1 Класни дијаграм

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

У оквиру ове фазе су извршене додатне оптимизације мин-макс алгоритма реализованог у претходној фази како би се ефикасније извршавао.

Класа *Game*:

```
class Game:
    def __init__(self, n=11, m=14, greenWall="\u01c1", blueWall="\u2550", rowSep="\u23AF"):
        self.table = Table(n, m)
        self.view = View(n, m, greenWall, blueWall, rowSep)
        Game.oo = 50
        Game.r = 0
        self.minmaxDepth = 3
        self.next = None
        self.winner = None
```

Исечак кода 1 – Конструктор класе Game

Унутар класе *Game* су додати атрибути класе *oo* и *r*. Атрибут *oo* представља граничну вредност за *alpha-beta* одсецање, док атрибут *r* представља неку врсту псеудо полупречника којим се од одређених позиција противничког пешака врши разматрање постављања плавих и зелених зидова, чиме се смањује број стања које је касније неопходно обрадити.

```
@staticmethod
    def generateNewStates(currentState, player):
        onTheMoveReff = currentState.X if player == "X" else currentState.O
        nextReff = currentState.O if player == "X" else currentState.X
        states = {
            'blue wall 1': frozenset(),
            'blue wall 2': frozenset(),
            'green wall 1': frozenset(),
            'green wall 2': frozenset(),
            'new': list()
        }
        positions = nextReff.getCurrentPositions()
        if onTheMoveReff.noGreenWalls > 0:
            for i in range(positions[0][0] - Game.r, positions[0][0] + Game.r + 1):
                if 0 < i < currentState.n:
                    for j in range(positions[0][1] - Game.r, positions[0][1] + Game.r + 1):
                        if 0 < j < currentState.m and currentState.isCorrectGreenWall((i, j)):
                            temp = currentState.getCopy()
                            temp.setGreenWall({
                                'position': (i, j),
                                'next': onTheMoveReff.name
                            })
                            if not currentState.isConnectedGreenWall((i, j)) or
temp.canBothPlayersFinish(True, True):
                                states['green wall 1'] |= frozenset({temp})
            for i in range(positions[1][0] - Game.r, positions[1][0] + Game.r + 1):
                if 0 < i < currentState.n:
                    for j in range(positions[1][1] - Game.r, positions[1][1] + Game.r + 1):
                        if 0 < j < currentState.m and currentState.isCorrectGreenWall((i, j)):
                            temp = currentState.getCopy()
                            temp.setGreenWall({
                                'position': (i, j),
                                'next': onTheMoveReff.name
                            })
                            if not currentState.isConnectedGreenWall((i, j)) or
temp.canBothPlayersFinish(True, True):
                                states['green wall 2'] |= frozenset({temp})
```

```
        states['green wall'] = states['green wall 1'] | states['green wall 2']
        if onTheMoveReff.noBlueWalls > 0:
            for i in range(positions[0][0] - Game.r, positions[0][0] +  Game.r + 1):
                if 0 < i < currentState.n:
                    for j in range(positions[0][1] - Game.r, positions[0][1] + Game.r):
                        if 0 < j < currentState.m and currentState.isCorrectBlueWall((i, j)):
                            temp = currentState.getCopy()
                            temp.setBlueWall({
                                'position': (i, j),
                                'next': onTheMoveReff.name
                            })
                            if not currentState.isConnectedBlueWall((i, j)) or
temp.canBothPlayersFinish(True, True):
                                states['blue wall 1'] |= frozenset({temp})
            for i in range(positions[1][0] - Game.r, positions[1][0] +  Game.r + 1):
                if 0 < i < currentState.n:
                    for j in range(positions[1][1] -  Game.r, positions[1][1] +  Game.r + 1):
                        if 0 < j < currentState.m and currentState.isCorrectBlueWall((i, j)):
                            temp = currentState.getCopy()
                            temp.setBlueWall({
                                'position': (i, j),
                                'next': onTheMoveReff.name
                            })
                            if not currentState.isConnectedBlueWall((i, j)) or
temp.canBothPlayersFinish(True, True):
                                states['blue wall 2'] |= frozenset({temp})
        states['blue wall'] = states['blue wall 1'] | states['blue wall 2']
        gp = {'name': onTheMoveReff.name}
        if gp['name'] == "X":
            positions = onTheMoveReff.getCurrentPositions()
            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))
                        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 = onTheMoveReff.getCurrentPositions()
            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)
        return states['new']
```

Исечак кода 2 – Функција generateNewStates(...)

У оквиру функције *generateNewStates(...)* је извршена и сама имплементација r, атрибута класе *Game*. Идеја иза овога је била да при сваком потезу ми, као играч, за циљ који имамо ограђивање пешака противнику, наравно не у потпуности, па нема смисла испитивати поља која су на превеликом растојању од истог, барем не за дубину мин-макс алгоритма коју ми желимо да остваримо. С тим у вези је постављање зидова сведено на квадрате $2 * rx2 * r$, са центрима у пољима где се налазе противнички пешаци. Овим се невероватно смањује број генерисаних стања, који са порастом дубине мин-макс алгоритма експоненцијално расту, па је за огроман број стања физички немогуће извршити рачуницу за прихватљив временски период. Из овог разлога је овај параметар постављен иницијално на 0, односно постављање зидова ће се одвијати само поред противничких пешака.

```
@staticmethod
    def maxValue(states, depth, player, alpha, beta):
        if depth == 0:
            return (states, Game.evaluateState(states[-1], player))
        else:
            for branchState in Game.generateNewStates(states[-1], player):
                alpha = max(alpha, Game.minValue(states + [branchState], depth - 1, player, alpha, beta),
key=lambda x: x[1])
                if alpha[1] >= beta[1]:
                    #print("Beta odsecanje")
                    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.generateNewStates(states[-1], player):
                beta = min(beta, Game.maxValue(states + [branchState], depth - 1, player, alpha, beta),
key=lambda x: x[1])
                if beta[1] <= alpha[1]:
                    #print("Alpha odsecanje")
                    return alpha
        return beta

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

Исечак кода 3 – Функције min(...), max(...) і minmax(...)

У оквиру ове три функције је исправљен пропуст који смо имали у претходној фази, где смо ручно мењали за ког играча тражимо евалуацију стања, и с обзиром да смо ишли само до дубине 1, нисмо могли да уочимо тренутно отклоњену грешку.

```
@staticmethod
    def evaluateState(state, player):
        if state.findWinner():
            return Game.oo - 1
        else:
            if state.canBothPlayersFinish(True, True):
                xMinPath = min(map(lambda x: len(x), state.xPaths))
                oMinPath = min(map(lambda x: len(x), state.oPaths))
                if player == "X":
                    return (oMinPath + 1 / xMinPath)
                else:
                    return (xMinPath + 1 / oMinPath)
            else:
                return -Game.oo + 1
```

Исечак кода 4 – Функција за процену стања (Хеуристика)

На крају је дата и сама хеуристика нашег пројекта у виду функције *evaluateState(…)*, односно процене ваљаности стања које се обрађује. Функција се своди на налажење најкраћег победничког пута за сваког од играча и мапирања пута на његову дужину. Сама евалуација зависи од тога да ли је крај игре, ово одсецање има веома високу вредност, или уколико је генерисани потез случајно невалидан, што има врло ниску вредност. Док уколико ниједно од наведених није случај, онда се разматрају дужине путања оба играча. Уколико се мин-макс алгоритам позива у корист играча X, онда се од дужине минималне путање до краја играча O одузима реципрочна вредност дужине минималног пута играча X. То јест, стање које ће имати највећу вредност за играча X, је или победничко, или оно које има велику дужину пута играча O, што говори да смо зид поставили тако да се препречио на путу до циља, а притом, у збиру са реципрочном вредношћу дужине нашег најкраћег пута, добијамо и одговор да ли је наш пут најкраћи могућ. Односно, фаворизоваће се стања у којима O има дуге путеве, а X што краће. Уколико се максимизује играч O, слична је логика, само окренуте вредности, фаворизују се стања у којима X има дуге путеве, а O кратке.