**Студент: Јован Николов**

**Индекс: 2016/0040**

**Интелигентни системи 13Е114ИС**

**Домаћи задатак за јануарски испитни рок 2019/20.**

**Опис проблема:**

Реализација игре *Dots and Boxes* коришћењем алгоритама теорије игара. Имплементација три нивоа играча: **почетнички**, **напредни** и **такмичарски**.

Играч **почетник** игра насумично, осим у случају када постоји потез којим може формирати квадрат.

**Напредни** играч користи минимакс алгоритам са алфа-бета одсецањем при чему не разматра потезе који представљају додавање треће линије, осим ако је такав потез једини могућ. У случају да постоји потез којим може формирати квадрат, увек игра прво њега.

**Такмичарски** алфа-бета играч треба на неки начин да оптимизује перформансе претходног играча, било то променом статичке функције процене или неким другим алгоритмом.

**Опис решења:**

Решење је реализовано у програмском језику *Java* коришћењем *JetBrains IntelliJ* развојног окружења. За реализацију корисничког интерфејса коришћења је библиотека *Java Swing*, а сам код за иницијализацију и покретање интерфејса је писан у засебним класама. Класе које се односе на GUI су: *GameFrame, MainMenuContentPane, GameContentPane, GameBoardPanel, ScorePanel* и помоћна класа *SwingUtils*.

Класа *GameFrame* представља главни прозор игре и динамички креира *MainMenuContentPane* и *GameContentPane* у зависности од тренутног стања игре. Ову класу инстанцира класа *Game* која уједно и имплементира главну нит игре над којом се извршава сва логика игре.

Класe *MainMenuContentPane* и *GameContentPane* инстанцирају сав потребан садржај корисничког интерфејса када се налазимо у главном менију и у самој игри респективно. Како би интерфејс био динамичан и скалабилан креиране су класе *GameBoardPanel* и *ScorePanel* које цртају графику игре и то таблу и тренутни број поена респективно.

Главна логика игре је имплементирана кроз више класа од којих су најбитније *Game, State* и класа *Player*.

Као што је претходно напоменуто, класа *Game* иницира одвојену нит за саму логику игре како кориснички интерфејс не би кочио у случају да логика захтева више времена за обраду. Она садржи следеће методе:

**public State getState();** // враћа објекат стања игре

**public Mode getMode();** // враћа тренутни мод у ком је покренута игра (човек против рачунара, човек против човека, рачунар против рачунара корак-по-корак, рачунар против рачунара одједном)

**public boolean isOver();** // враћа **true** уколико је игра завршена, **false** иначе

**public void startThread();** // покреће главну нит, позива се из главног програма

**public void startGame(State state, Mode mode);** // почиње нову игру са задатим почетним стањем и модом игре

**public void playerDone();** // буди главну нит и сигнализира да је играч завршио

**public void nextStep();** // буди главну нит и сигнализира да пређе на следећи корак

**public void endGame();** // буди главну нит и сигнализира да је крај игре

**public void showMainMenu();** // буди главну нит и сигнализира повратак у главни мени

**public void run();** // тело главне нити игре

Класа *State* представља тренутно стање игре и поред саме табле садржи и разне податке као што су тренутни број поена првог и другог играча, стек који садржи све претходне потезе (ради приказивања у корисничком интерфејсу), итд. Неке од битнијих метода су:

**public State getClone();** // враћа копију тренутног објекта (користи се у **getNextBoardState** методи)

**public static boolean tryParseGameStateFromFile(String gameStateFilePath);** // покушава да парсира улазни фајл и поставља статичке променљиве које се даље користе при креирању објекта класе

**public static boolean tryExportGameStateToFile(State state,**

**String gameStateFilePath);** // покушава да експортује стање игре у излазни фајл у истом формату као улазни

**public List<Edge> getAvailableMoves();** // враћа листу могућих неодиграних потеза

**public int getEdgeCount(int x, int y);** // враћа број ивица које су нацртане око квадрата на позицији **x** i **y**

**public int getBoxCount(int edgeCount);** // враћа број квадрата (кутија) које имају по **edgeCount** ивица (0 <= edgeCount <= 4)

**public boolean addsNthEdge(Edge move, int n);** // враћа **true** уколико додавање ивице **move** има за резултат додавање n-те ивице на суседним квадратима

**public boolean makeMove(Edge move);** // одиграва потез (мења стање) **move** и враћа **true** уколико је потез успешно одигран

**public State getNextBoardState(Edge move);** // враћа **копију** тренутног стања измењеног за потез **move** (користи се у минимакс алгоритму)

**public int closeAllAvailableBoxes();** // затвара све квадрате (кутије) на табли док може (постоји потез који додаје четврту ивицу неком квадрату); користи се за имплементацију такмичарског играча (за рачунање хеуристике)

Најбитнији интерфејс који имплементирају све класе везане за играче јесте интерфејс *Solver* који има само једну методу и то:

**Edge getNextMove();** // враћа наредни потез датог играча

Класа *Player* је апстрактна класа која имплементира интерфејс *Solver* и основна је класа за изведене *HumanPlayer* и *AIPlayer*.

Класа *HumanPlayer* садржи једну додатну методу:

**public void setNextMove(Edge nextMove);** // користи је графички кориснички интерфејс како би поставио следећи потез играча када је изабран

Класа *AIPlayer* садржи додатну променљиву maxDepth која означава дубину развијања стабла игре која се поставља приликом покретања игре. Такође на основу String-а који добија из GUI-ја креира свој приватни алгоритам у зависности од тежине која специфицирана у интерфејсу. Ова класа такође чува тренутну хеуристику алгоритма за сваки **разматрани** потез и за то постоје одговарајући getter и setter. Метода која је овде суштински најбитнија је:

**public long computeNextMove();** // позива интерни алгоритам који генерише следећи потез и памти га у променљивој наслеђеној из класе *Player*

Класе које имплементирају саме алгоритме су *RandomSolver, AlphaBetaSolver* и *CompetitiveSolver*. Све имплементирају интерфејс *Solver*.

Класа *RandomSolver* итерира по листи доступних неодиграних потеза и уколико нађе потез који затвара неки квадрат врати га. Уколико не нађе такав потез, врати насумично одабран потез из листе.

Класа *AlphaBetaSolver* садржи помоћне методе ради лакше имплементације алфа-бета минимакс алгоритма. Она садржи методе:

**protected int getHeuristic(State state);** // рачуна статичку хеуристику тренутног стања игре за референтног играча (играч који је креирао инстанцу ове класе)

**protected Node search(State state, List<Edge> moves, int alpha, int beta, int depth, boolean turnByTurn);** // рекурзивна функција која имплементира минимакс алгоритам са алфа-бета одсецањем уз додатни параметар **turnByTurn** који користи такмичарски играч за имплементацију боље хеуристике

**protected Edge getBestMove(List<Edge> moves, boolean turnByTurn);** // позива функцију **search(…)** са одговарајућим параметрима и враћа најбољи потез који је пронађен у стаблу игре

Класа *CompetitiveSolver* наслеђује *AlphaBetaSolver* јер користе исти минимакс алгоритам, а оно што мења јесте формула статичке хеуристике као и ситуације у којима се позива алгоритам претраге. Такође као што је горе поменуто, ова класа позива **getBestMove(…)** са аргументом *turnByTurn = true* чиме модификује алгоритам да ради играч-по-играч уместо потез-по-потез, али само за рачунање хеуристике. Свакако се алгоритам позива сваки потез, али овим се постиже прецизнија хеуристика јер уколико дајемо противнику могућност да затвори квадрате желимо да минимизујемо његову добит. Простим повећавањем дубине сложеност расте експоненцијално, али овом променом хеуристичке функције добијамо сличан резултат са линеарним повећањем сложености.

**Могућа побољшања решења:**

С обзиром на то да је алгоритам имплементиран рекурзивно са копирањем стања за сваки потез, са порастом величине табле и дубине експоненцијално расте временска али и меморијска сложеност. Како би се ово побољшало, алгоритам претраге се може имплементирати итеративно, а уместо копирања стања може се тренутно (једино) стање мењати у месту и враћати (модел do-undo). Тиме би се постигла доста боља меморијска сложеност јер би се копирали само могући потези, а не и цела табла. Самим тим и временска сложеност би опала јер не бисмо сваки пут морали да копирамо таблу (сложеност n\*m). Додатно уместо копирања доступних потеза, они се могу пратити и у самом објекту *State*па би меморијска и временска сложеност додатно могле да опадну.