Универзитет у Нишу

Електронски факултет

Лука Белић

Реализација апликације за играње логичке игре "Вук и овце"

Дипломски рад

Студијски програм: Рачунарство и информатика

Ментор:

Проф. др. Владан Вучковић

Кандидат:

Лука Белић 17013

Ниш, октобар 2023. година

Универзитет у Нишу

Електронски факултет

Дипломски рад

Реализација апликације за играње логичке игре "Вук и овце"

Задатак: У теоријском делу проучити основне концепте теорије игара који чине једну логичку игру. У практичном делу рада имплементирати апликацију за играње логичке игре “Вук и овце”.

Ментор:

Проф. др. Владан Вучковић

Кандидат:

Лука Белић 17013

Комисија:

1._____ Датум пријаве: _____

2._____ Датум пријаве: _____

3._____ Датум пријаве: _____

Садржај

1. Увод

Теорија игара је фасцинантно поље математике које се примењује у различитим дисциплинама како би се анализирале интеракције између рационалних доносиоца одлука. Ова грана математике омогућава дубље разумевање динамике одлучивања и стратегијског размишљања у различитим сценаријима. Овај рад истражује и примењује принципе теорије игара кроз конкретан пример - развој апликације под називом "Вук и овце."

Циљ овог рада је развој интелигентног ***AI*** играча способног да донесе оптималне потезе у игри "Вук и овце." Кроз овај процес, показује се како теорија игара може бити примењена у пракси ради анализе стратегија и доношење одлука. Фокус ће бити на теоријским основама теорије игара и њиховој повезаности са игром "Вук и овце," користећи минимакс алгоритам са дубинским ограничењем и алфа-бета одсецањем. Овај рад ће илустровати комплексност стратешког доношења одлука и његову примену у стварним ситуацијама.

У наставку овог рада, прво следи увод у основе теорије игара, истражујући њене кључне концепте и принципе. Након тога, детаљније размотриње игре "Вук и овце" као студијску игру, описујући њену механику и правила. Затим следи представљање минимакс алгоритам са дубинским ограничењем и алфа-бета одсецањем као главне компоненте овог ***AI*** играча. У практичном делу рада, демонстрираће се имплементација ових алгоритама у развоју апликације "Вук и овце." Такође ће се анализирати перформансе ***AI*** играча у сценаријима игре и истражити како се његове одлуке разликују у зависности од различитих стратегија играча.

Овај рад показује како теорија игара може бити моћан алат за анализу и унапређење стратегијског доношења одлука у различитим контекстима. Осим тога, истиче значај примене ових теоријских принципа у стварном свету кроз развој интелигентних ***AI*** играча. Кроз комбинацију теоријских основа и практичне имплементације, надам се да ће овај рад допринети бољем разумевању теорије игара и њених примена.

2. Теорија игара и врсте игара

Теорија игара, грана примењене математике, опремљује нас моћним алатима за анализу ситуација у којима учесници, користећи своје стратегије доносе одлуке које су међусобно повезане. На основу ове повезаности сваки од играча мора добро размислити о стратегијама које користе други играчи како би оформио своју и победио. Сваки играч има план да победи. На путу до победе у зависности од интереса других играча, у току игре мора одабрати оптималну одлуку. Итереси играча могу бити усклађени, супротни или мешани.

Теорија игара се бави анализом друштвених игара, али и поред тога она обухвата далеко више области од само игара намењених за личну забаву. Теорија игара потиче из заједничких напора математичара Нојмана и економисте Оскара Моргенштерна. На почетку теорија игара је развијена са намером да реши електронске проблеме.

У свом утемељеном делу "Теорија игара и економско понашање" (1944), фон Нојман и Моргенштерн тврдили су да математички модели позајмљени из природних наука, који описују неутралне природне појаве, недовољно адресирају сложености економије. Препознали су економију као врсту игре, где играчи антиципирају потезе једни других, захтевајући нови математички приступ, који су назвали теоријом игара. Ова дисциплина је даље развијена 1950-их година од стране америчког математичара Џона Неша, који је поставио математичке основе теорије игара - област математике која истражује интеракције и конфликте између такмичара са различитим интересима.

Теорија игара има практичне примене у разним ситуацијама у којима одлуке учесника заједно обликују крајњи исход. Наглашавањем стратешког аспекта доношења одлука, који је контролисан од стране играча, а не искључиво од стране случаја, теорија игара допуњује и проширује традиционалну теорију вероватноће. Њене примене обухватају предвиђање политичких савеза или корпоративних спајања, оптимизацију цена у конкурентним тржиштима, процену утицаја појединачних бирача или група бирача, избор правосудних порота, идентификацију оптималних локација за производне погоне и разумевање динамике понашања различитих животињских и биљних врста у њиховој борби за преживљавање. Такође, теорија игара је употребљена и за изазивање законитости одређених система гласања.

С обзиром на широк спектар "игара" које се сусрећу, није изненађујуће да ниједна појединачна теорија не може потпуно обухватити све. Бројне теорије су предложене, свака прилагођена различитим сценаријима и свака уводи своје јединствене концепте онога што представља решење.

2.1 Типови игара у теорији игара

У теорији игара, различите врсте игара помажу у анализи различитих врста проблема. Не постоји једна победничка стратегија већ увек зависи од врсте игре која се игра. Различите врсте игара се формирају на основу броја играча укључених у игру, симетрије игре и сарадње међу играчима.

- **Кооперативне и некооперативне игре** – Кооперативна теорија анализира начине на које групе комуницирају када су познате само исплате. Представља игру коалиција играча, а не између појединаца. Поставља се питање на који начин се формирају групе и како распоредити исплате између учесника. Некооперативна теорија игара анализира начине где субјекти са углавном супротним интересима покушавају да остваре своје циљеве. Најпознатија некооперативна игра је шах. Углавном те врсте игара су стртешке игре.
- **Игре нулте суме и не нулте суме** – Игре нулте суме су посебан облик некооперативних игара где постоји директан сукоб играча ка истом циљу. Постоји само један победник, тако да колективна нето корист је увек једнака нули. Пример може бити Шах. Такође и апликација "Вук и овце" спада у игру нулте суме. Игре без нулте суме су оне игре где сви играчи могу изгубити или победити истовремено. Пример могу бити пословна партнерства где обе стране добијају корист.
- **Игре са симултаним потезима и секвенцијалним потезима** – Код игара са симултаним потезима сви играчи стално доносе одлуке у исто време кад и њихов противник. Док код игара са секвенцијалним потезима играч доноси одлуку на основу претходне одлуке другог играча.
- **Игре које се играју једом и игре које се понављају** – Иако је могуће одиграти исту игру једом или више пута узастопно, у зависности од ове поделе коришћење исте стратегије може бити катрастрофално. У неким играма као што је дилема затвореника, стратегија се значајно разликује у односу на број одиграних партија.

Генералне претпоставке играча при доношењу одлука:

1) Сви такмичари су рационални и интелигентни.
2) Број такмичара је коначан.
3) Сваки играч зна правила, последице и детаље игре.
4) Играчи имају за циљ да максимизују добитке и минимизују губитке.
5) Сви играчи имају коначан број акција.
6) Сви играчи се могу одлучити за своју стратегију решавања проблема.

3. Игра “Вук и овце”

3.1 Кратак историјат игре

Иако традиционална игра “Вук и овце” има дуг историјат који датира вековима уназад, њено тачно порекло није познато. Постоји пар претпоставки одакле је настала ова игра. Сматра се да се варијација ове игре играла још у доба Викинга што би значило да је ова игра стара око хиљаду година. Постоји слична варијација ове игре у Скандинавији по називу ***Tafl***. Такође и на блиском истоку постоје игре сличног концепта као што је ***Alquerque***.

У средњем веку ова игра постала је позната, посебно у Француској и Енглеској, али под другим називом и то “Лисица и гуске”. Такође су постојале различите варијације ове игре малим изменама у правилима и величини табле, али у то време најпопуларнија је била верзија са таблом величине ***7x7***. Иако је долазило до стварања знатног броја варијација ове игре у различитим регионима, главни концепт игре је остао очуван – где један играч контролише вука или лисицу, док други играч контролише овце или гуске. Игра је стратешка али једноставна. Циљ вука јесте да стигне до другог краја табле, док овце покушавају да својим кретањем заблокирају пут вуку и коначно га заробе тако да се вук не може више померити то јест нема више ниједан легалан потез.

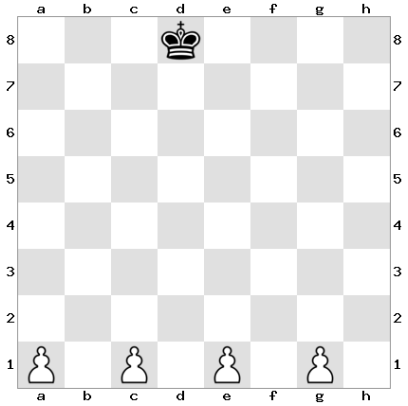
Модерна верзија игре “Вук и овце” је поново оживела у популарности након дужег времена. Са развојем рачунара и вештачке инелигенције ова игра представља праву прилику за тестирање алгоритама због своје статешке природе. Путем интернета игра “Вук и овце” постаје доступна људима широм света, а у комбинацији са мобилним апликацијама, омогућава играчима да уживају у игри такмичећи се једни против других.

Дуга истроија ове игре доказује да је ова игра врло занимљива. Играчи осмишљавају стратегије и вековима се забављају кроз генерације. Врло је привлачна због једноставних правила, али стратешка дубина нуди велики број могућих потеза али и самим тим и могућност да се развије вештина играња ове игре.

3.2 Правила игре

3.2.1 Почетна позиција

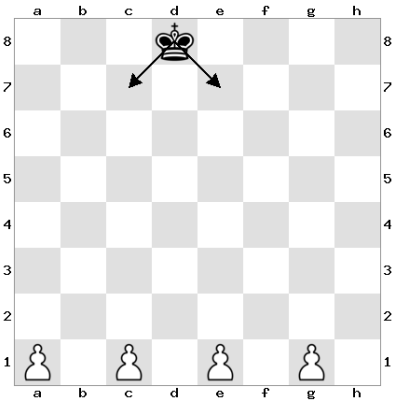
Игра „Вук и овце“ игра се са два играча. Табла на којој се игра ова игра је стандардна шаховска табла величине ***8x8***. Играч Вук поседује једну фигуру која у почетном стању постављена на црно поље у средини првог реда одозго, док играч који контролише овце поседује четири играча који су постављени такође на црна поља али у последњем реду одозго. Када су фигуре правилно постављене на почетне позиције табла изгледа на следећи начин:



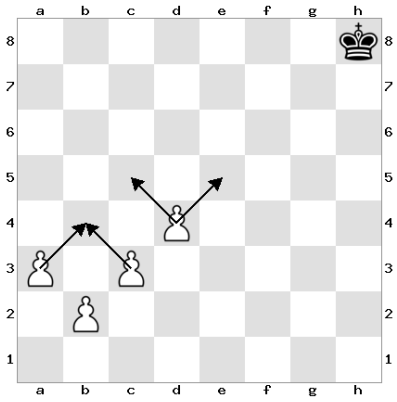
Слика 1 Почетна позиција

3.2.2 Правила кретања

Играч “Вук” се може кретати дијагонално у сва четири правца, али на удаљености од једног поља. Слично томе, играч који контролише “Овце”, када је на реду његов потез, мора померити једну од своје четири фигуре. Његове фигуре се померају само дијагонално унапред такође једно поље. Валидан потез је онај уколико је у он у оквирима табле, и на њему се не налази ни једна друга фигура. Примери легалних потеза:



Слика 2 Легални потези вука

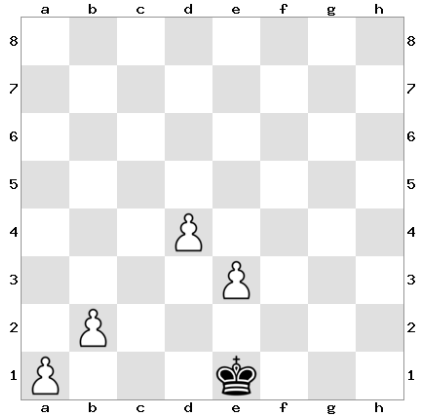


Слика 3 Легални потези овце

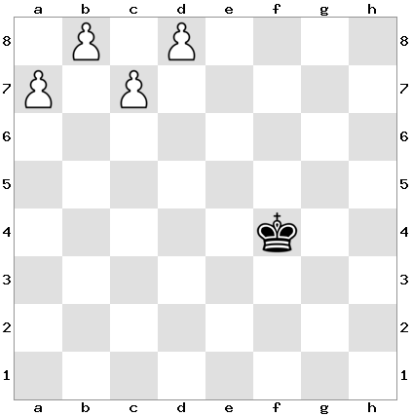
3.2.3 Циљ игре и ток игре

Играч вук има циљ да достигне до било ког поља у реду у ком се налазе овце на почетку игре (први ред одоздо). Циљ играча који контролише овце је да заустави вука стратешким померањем формације својих фигура то јест да зароби вука тако да он не поседује ниједан легалан потез.

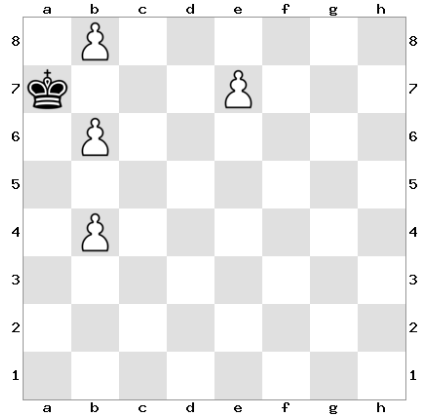
Игру започиње вук, а затим наизменично играчи повлаче потезе. Игра је такође завршена уколико играч овца нема више легалних потеза, и тада је победио вук. Примери крајњих позиција:



Слика 4 Победничка позиција вука



Слика 6 Победничка позиција вука



Слика 5 Победничка позиција овце

4. Структура програма и логичких игара

У овом делу биће описан принцип генерисања потеза и основни елементи структуре игре “Вук и овце”. Основна теоретска подлога за развој програма способних за одигравање вештих потеза кроз доночење одлука извире из основног принципа минимакс алгоритма. [2]

Да би се овај алгоритам применио неопходно је да два играча играју стратешку игру где играч покушава да одабере најоптималнији потез у датом тренутку. Са друге стране, такође и стратегија другог играча је да одабере свој најоптималнији потез. Обе стране поседују тотално супротне интересе, страна која је на потезу покушава одабрати најбољи потез тако што при одабиру узима у обзир и стратегију непријатеља који такође има исти принцип одлучивања. На основу овога формира се минимакс принцип одлучивања који представља основу доношења одлука програма који се баве логичким играма у којима учествују два играча. Применом овог принципа генерише се стабло игре које представља могућа стања и транзиције кроз које пролазе играчи. Стратегија играча се

формира тако што се на основу одређених правила одабира један од путева у стаблу игре. Крајње позиције стабла игре представљају терминални чворови. Сваки од терминалних чворова поседује своју вредност која се израчунава коришћењем евалуационе функције. Користећи овај принцип у сваком од чворова бира се вредност чвора која је најоптималнија за играча који је на потезу па на тај начин најбоља вредност терминалног чвора завршава у почетном чвору који се назива изворни чвор.

Кроз јасан опис овог принципа види се главни проблем који настаје овом методом а то је проблем комбинаторне експлозије. На примеру ове игре где први играч поседује четири могућа потеза, а затим на сваки од тих потеза постоји осам потеза другог играча, врло убрзо долази до генерисања огромног броја могућих потеза. Способност машине да донесе најбољу одлуку директно је повезана са њеном способношћу да уђе у већу дубину игре, па тиме и предвиди више потеза непријатеља, што је чини паметнијом.

Очигледно је да генерисање свих могућих комбинација, и одабир најбоље није ни близу оптималног решења. Како би се постигла већа моћ машине, неопходно је искористити додатне технике које ће превазићи овај проблем. Комбинаторна експлозија се превазилази коришћењем разних алгоритама, техником названом сечење стабла. Сечењем стабла избегава се обрађивање чворова који по процени нису од значаја, па се тиме добија на брзини размишљања машине.

4.1 Основни појмови

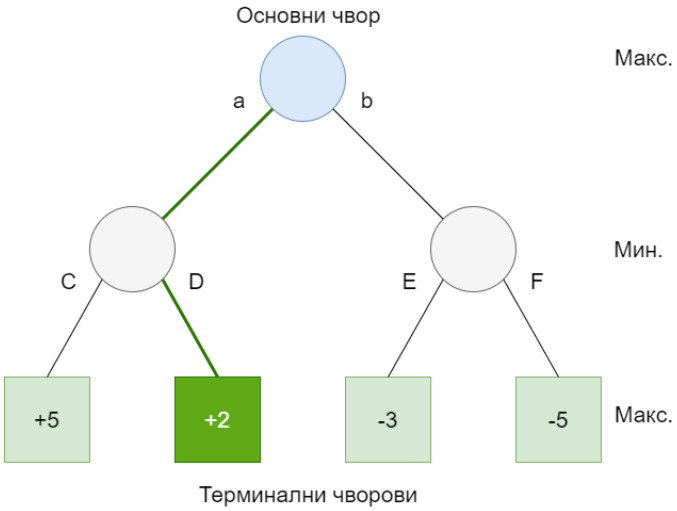
Основне структуре које се могу уочити анализом претходно описане методе су следеће:

- **Стабло игре** – генерисањем осталих позиција на основу почетне позиције генерише се и стабло игре. Постоји више начина генерисања стабла игре, али у реализацији апликације “Вук и овце” коришћена је рекурзија која користи унутрашњи стек за складиштење стања. Стабло игре добило је тај назив јер изгледа као структура података стабло.
- **Терминални чворови** – представљају крајња стања у којима се налази играч у стаблу игре. Свакоме од терминалних чворова додељује се нумеричке вредност добијена евалуационом функцијом и она показује колико је тај потез добар.
- **Алгоритам за тражење** – имплементација овог алгоритма одрађена је рекурзивним позивима тако да формира стабло игре почевши развој од почетног чвора. Такође максимизује евалуиране вредности сваког стања игре.

4.1.1 Стабло игре

Структура стабло игре се формира експанзијом основног чвора, а затим експанзијом свих осталих чворова све до наиласка на терминалне чворове. Стабло игре представља начина на који рачунар размишља и доноси одлуке. План којим ће се машина

водити пролази кроз стабло игре, са обзиром да оно приказује стања која се могу догодити у будућности као и стања којим ће противник одреаговати. Да би се детаљније схватио принцип генерисања стабла, управо ће бити обрађен један једноставан пример. Иако у игри "Вук и овце" вук може да генерише и до четири потеза, а затим овце додатних осам потеза на сваки од тих четири, икоришћен је пример који садржи по два потеза у сваком чвору како би ситуација била једноставнија и прегледнија. Принцип је идентичан са било којим бројем могућих потеза. [1]



Слика 7 Пример минимакс одлучивања

Стања која се односе на играча вук означена су плавим пољима, а белим пољима су означена стања где је играч који игра са овцама. Да би играч одабрао одговарајући потез за дату позицију, мора се прорачунати вредност основног чвора. У овом случају плави играч поседује два наставка стабла, ***a*** и ***b***. На оба наставка се поново генеришу сви могући наставци, али овог пута за белог играча. У овом случају то су два наставка на сваку од позиција (***a***,***C***), (***a***,***D***), (***b***,***E***), (***b***,***F***). У општем случају игра се развија по истом принципу све док се не достигне до терминалних чворова. Јасно је да се на овај начин врло брзо стабло игре знатно развија, и на тај начин долази до комбинаторне експлозије. Овим пример се зауставља на трећем нивоу. Сада је потребно доделити нумеричке вредности терминалним чворовима, а то се постиже коришћењем евалуационе функције. Играч вук покушава да максимизује позицију, а супротно томе играч овца покушава да минимизује позицију. Са обзиром на то, све позитивније вредности чворова представљају све бољи потез играча вук, и обрнуто.

У реализацији ове апликације направљена су три мода евалуације којима се процењују стања игре. Више инфромација о имплементацији тих функција биће у наставку документа.

Одређивање најбољег потеза у датом примеру врши се на следећи начин:

Евалуирају се терминални чворови, а њихове вредности су уписане у чворовима. На сваком од нивоа извршава се минимакс принцип. Вредности треминалних чворова на нивоу три, улазе у ниво два где се врши одређивање најбољег потеза. Са обзиром да је у другом нивоу играч овца, он бира најбољи потез за себе а то је потез који доноси минималну вредност. У датом примеру то су вредности ***min(C,D)*** = 2 у првом чвору и ***min(E,F)*** = -5 у другом чвору. На првом нивуо је играч вук. Он покушава да максимизује вредности, па је његова функција ***max(a,b)*** = 2.

На основу претходног описа може се закључити да апликација приликом доношења одлуке пролази кроз два процеса и то:

- Први процес подразумева ширења стабла које се генерише из основног чвора ка терминалним чворовима. Током овог процеса формирају се међу чворови који представљају могућа стања игре. Терминални чворови се евалуирају и добијају своју вредност.
- Други процес представља генерисање и евалуацију најбољег потеза, то јест израчунавање вредности основног чвора. У овом процесу се полази из супротног правца, из правца терминалних чворова, и користећи претходно описани минимакс принцип израчунава вредност чворова у сваком нивоу респективно.

Појам основна линија представља варијанту која садржи најбоље наставке за играча вук као и за играча овца. Почетак основне линије је увек у основном чвору а крај у неком од терминалних чворова. У овом примеру је то пут (***a***,***D***). Евалуација основне позиција своди се на пропагирање вредности од терминалног чвора до основног чвора у главној варијанти, што је у овом примеру вредност 2. Ова игра је игра нулте суме, што зачи да је увек најбољи потез првог играча најгори потез другог играча и обрнуто. У математици се ово назива и ***Nash equilibrium*** и каже да уколико играч изабере било коју алтернативну опцију а да она није ***Nash equilibrium*** он тиме ништа не може да добије. Сваки потез који је ван основне линије је мање оптималан, то јест никада не доводи до боље позиције.

4.1.2 Терминални чворови

Начин евалуације терминалних чворова је једна од главних ставки програма који игра ову игру. У зависности од тога колико је стање добро евалуирано зависиће и начин на који машина одабира потезе. Као што је претходно напоменуто крајњи чворови који се завршавају у стаблу игре називају се терминални чворови. Како би се минимакс процесом одлучивања одабрао прави потез неопходно је евалуирати позицију на прави начин. Функција која евалуира стање назива се евалуациона функција. Што је мање пропуста у евалуационој функцији то ће машина играти оптималније. Стил и тактика којим ће машина играти такође у великој мери зависи од евалуације терминалних чворова.

Основни проблем код евалуатора који је потребно решити је наћи баланс између брзине евалуације и хеуристичког знања које се програмира у функцији за евалуацију. Са обзиром да се углавном доста времена искористи на прорачунавање вредности терминалних чворова, брзина израчунавања је од огромног значаја. Убрзавањем евалуатора, убрзава се рад целог система, па се на тај начин постиже и брже размишљање рачунара, то јест повећање броја нивоа дубине које програм може да прорачуна у неком разумном времену које му је дато за размишљање. Што је већа дубина коју машина може да прорачуна повећава јој се и шанса да је једна од тих позиција крајња, то јест сигурно победничка опција.

4.2 Минимакс псеудокод

Минимакс принцип је главни алгоритам који је коришћен за развој апликације "Вук и овце". На следећем примеру налази се псеудо код овог принципа.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

Функција садржи два улазна параметара. Први представља чвор, то јест стање у којем се тренутно врши минимакс функција. То стање може бити репрезентовано на било који начин који је програмер изабрао. У наредним поглављима биће више речи о начину на који је репрезентовано стање током имплементације игре. Други аргумент је дубина. Овај пример представља минимакс алгоритам ограничен дубином. Последњи параметар је ***bool*** вредност која води рачуна о томе који је играч на потезу.

Као и код било које друге рекурзивне функције на почетку се налазе услови за изалаз из рекурзије. Овај минимакс алгоритам поседује два услова од којих је један да је дубина достигла вредност једнакој нули, што значи да је функција достигла максималну дубину за коју је позвана. Други услов је да је стање терминални чвор то јест да је крајње стање игре. Уколико се установи да је тренутни чвор терминални, врши се евалуација чвора и враћа се као повратна вредност функције. На тај начин долази до претходно

наведеног ефекта да се вредности пропагирају од крајњег чвора све до основног чвора. Уколико није испуњен услов, чвор није терминални, истипује се који је играч на потезу.

Уколико је то играч који покушава да максимизује вредности, он генерише своја легална стања (децу) и рекурзивно се испитује које од његових стања има најбољу вредност за њега. Та вредонст се такође враћа као повратна вредност функције. Уколико је на потезу играч који покушава да минимизује, принцип је идентичан међутим он тражи минималну вредност својих легалних потеза. Иако време израчунавања расте експоненцијално, са обзиром да је функција ограничена дубином, број израчунавања ја коначан и неће доћи до ***stack overflow***-а.

4.3 *Alpha-beta* одсецање

Минимакс алгоритам генерише све могуће комбинације потеза, па на тај начин ствара и комбинаторну експлозију. Након неког времена математичари су увидели начин да убрзају овај процес тако што је могуће одсећи, то јест уопште не разматрати одређене чворове и на тај начин знатно убрзати алгоритам. Овај алгоритам се често користи у логичким играма са два играча као Шах или Икс-Окс. Самим тим представља савршену прилику да се искористи и за имплементацију игре “Вук и овце”. [4][5][6]

Веома је важно напоменути да алфа-бета одсецање одсеца чворове који сигурно неће одсећи бољи потез од постојећих. На тај начин добија се исти ефекат који се добија минимакс алгоритмом, али уз мањи број прорачуна. То допушта овом алгоритму да за исто време прорачуна већи број стања игре па самим тим може ући у знатну већу дубину при израчунавању, што му пружа могућност доношења паметнијих одлука.

4.3.1 Побољшања минимакс алгоритма

Као што је претходно напоменуто предности које пружа алфа-бета алгоритам јесу те да се неке гране стабла могу елиминисати, а да то не утиче на крајње израчунавање основног чвора. Када су чворови у правилном или скоро правилном редоследу може се смањити дубина претраге за пола тако што се изабере први чвор. Ако обележимо просечни фактор гранања као $\boldsymbol{b}$ и дубину као $\boldsymbol{d}$, максимални број чворова је $\boldsymbol{O(b^d)}$ што идентично као и код стандардног минимакс алгоритма. У оптималном случају када је први чвор одмах и најбоља опција, то може поништити остале чворове и на тај начин смањити број чворова са $\boldsymbol{O(b^d)}$ на $\boldsymbol{O(\sqrt{b^d})}$. Да би се ово извело први потез мора да буде најбољи али и сваки следећи потез мора да буде такав да одсеца све остале осим првог. Овакав идеалан случај није толико распрострањен, али када су чворови насумично распоређени процењује се да ће просечан број потеза бити $\boldsymbol{O(b^{3d/4})}$. [7]

У зависности од редоследа генерисаних потеза може доћи до ранијег одсецања или не одсецања, што може драстично убрзати или успорити рад целог система. Веома је важно подесити генератор потеза тако да одабира потезе који ће касније одсећи друге

потезе и на тај начин омогућити машини брже израчунавање. Још једна предност афла-бета алгоритма је да се може модификовати тако да враћа целу главну варијацију стабла игре.

4.3.2 *Alpha-beta* псеудокод

Alpha-beta одсецање је врло једноставно додати на минимакс алгоритам увођењем пар контролних променљивих и услова за испитивање пред одсецање сигурно не бољег стања од претходно посећеног. На следећем примеру налази се псеудо код овог принципа.

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        for each child of node
            α := max(α, alphabeta(child, depth - 1, α, β,
not(maximizingPlayer)))
            if β ≤ α
                break                              (* Beta cut-off *)
        return α
    else
        for each child of node
            β := min(β, alphabeta(child, depth - 1, α, β,
not(maximizingPlayer)))
            if β ≤ α
                break                              (* Alpha cut-off *)
        return β
(* Initial call *)
alphabeta(origin, depth, -infinity, +infinity, TRUE)
```

Као што се може уочити са слике , разлика између минимакс алгоритма без алфа-бета одсецања и минимакс алгоритма са алфа-бета одсецањем је у томе што се кроз параметре функције прослеђују два параметра алфа и бета.

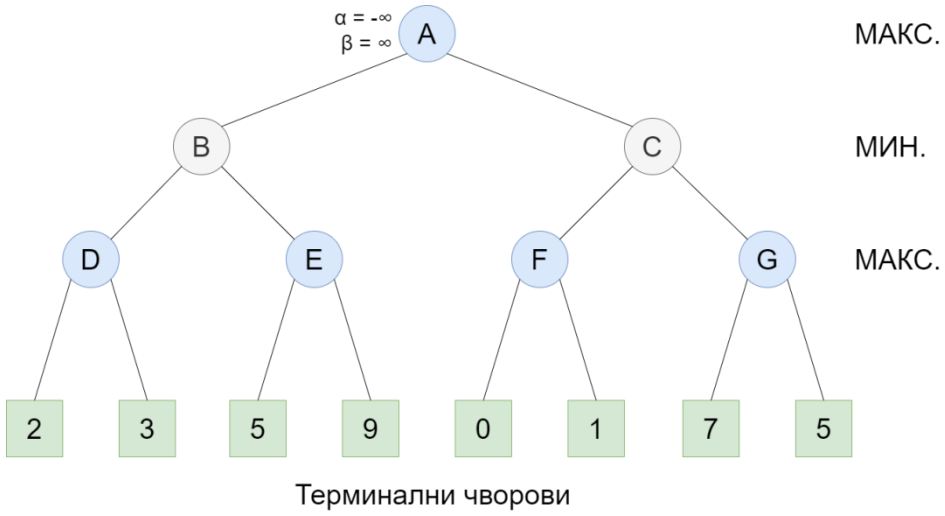
Када је на потезу играч који покушава да максимизује вредности, алфа добија максимум свих рекурзивно добијених вредности своје деце. Уколико је услов ***beta <= alpha***, догађа се бета одсецање и даље се не разматра тај пут у стаблу потеза. У супротном се алфа враћа као повратна вредност функције. Када је на потезу играч који покушава да минимизује вредности поступак је веома сличан али бета добија минималну вредност деце. И у колико услов за одсецање није испуњен, враћа бету као повратну вредност.

Иако на први поглед разлика између минимакс алгоритма са и без алфа-бета одсецања не делује велика, математичарима је било потребно доста година да усаврше

овај алгоритам. Алгоритам се развијао чак око једанаест година и то у периоду од 1975. до 1986.

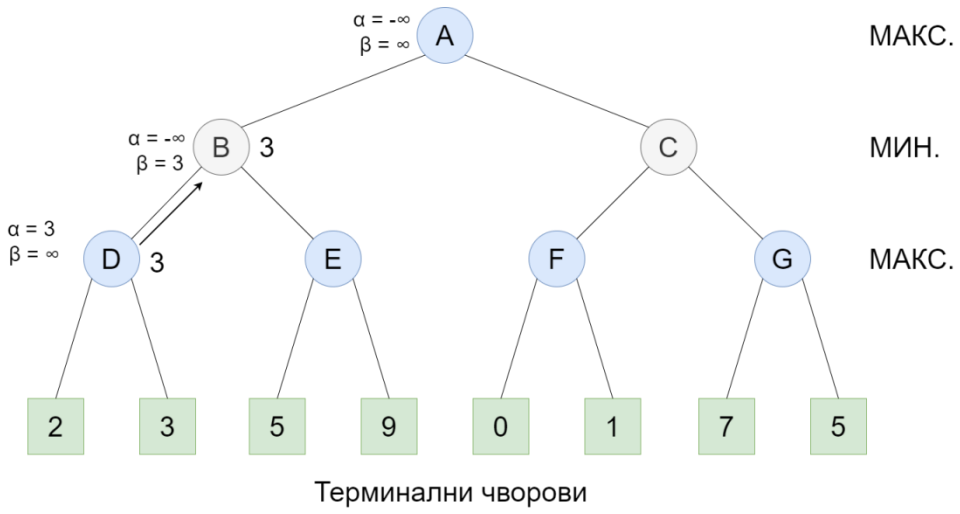
4.3.3 Појашњење алгоритма кроз пример

1. Први на потезу је играч који покушава да максимизује вредности. Он поставља алфа на –бесконачно и бета на +бесконачно. Ове вредности се рекурзивно прослеђују деци, то јест чвор ***A*** прослеђује чвроу ***B***, а чвор ***B*** затим на исти начин прослеђује чвору ***C***.



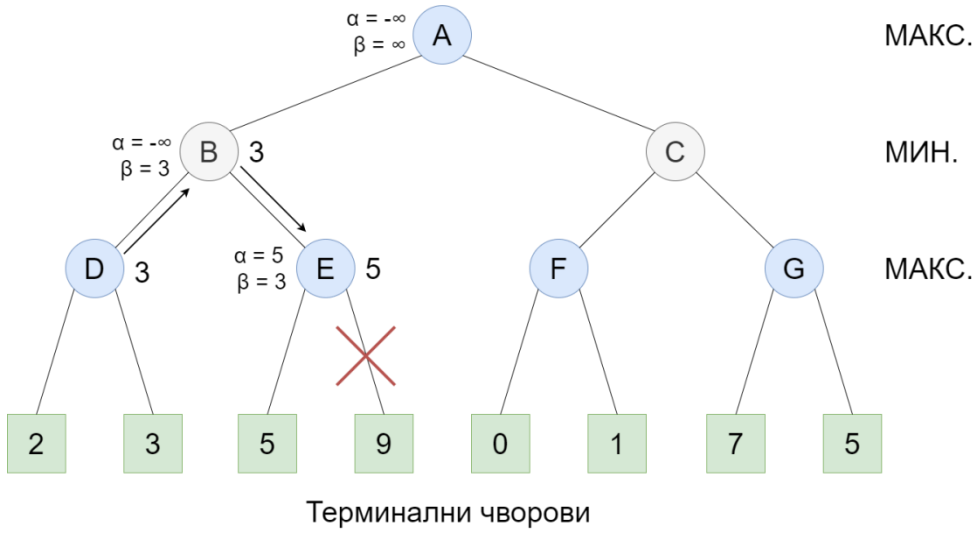
Слика 8 Појашњење алфа-бета алгоритма

2. У чвору ***D*** вредност ће се пропрачунати као максимум његових терминалних чворова. Алфа се пореди са вредностима 2 и 3, алфа постаје ***max(2,3)*** = 3. Вредност чвора Д је такође 3.
3. Функција се враћа на чвор ***B*** где се бета поставља као минимум претходне вредности и 3 тако да тада је алфа -бесконачно, а бета је 3. У следећем кораку вредности алфа и бета се преносе на следеце дете а то је чвор ***E***.



Слика 9 Појашњење алфа-бета алгоритма

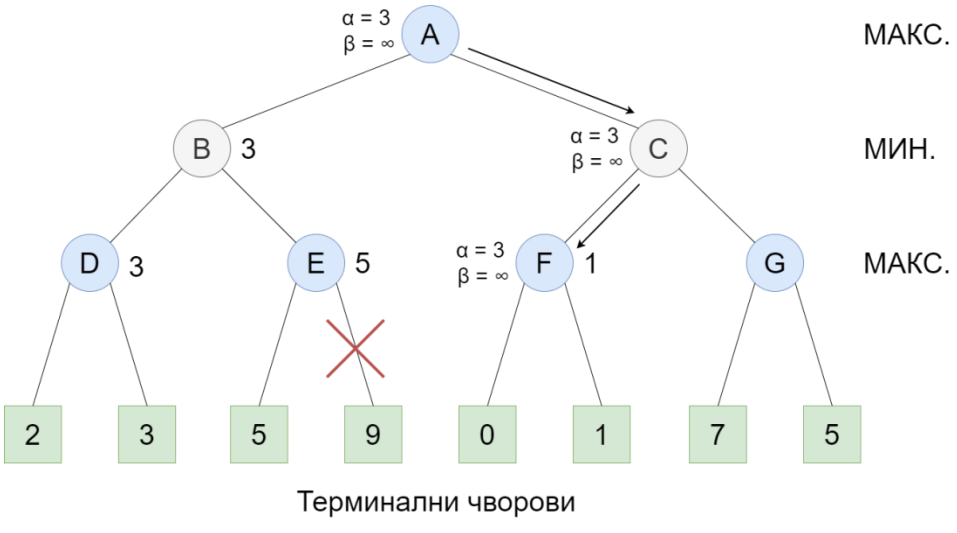
4. Чвор ***E*** је следећи који се обрађује. Код њега се мења вредност промењиве алфа. Тренутна вредност алфе ће се поредити са чвором са вредности 5 алфа је ***max(-5,inf)***. Такође са обзиром да је бета једнака 3 и услов ***alpha*** > ***beta*** је испуњен, наредни наследник чвора ће бити одсечен. Вредност чвора **E** ће бити 5.



Слика 10 Појашњење алфа-бета алгоритма

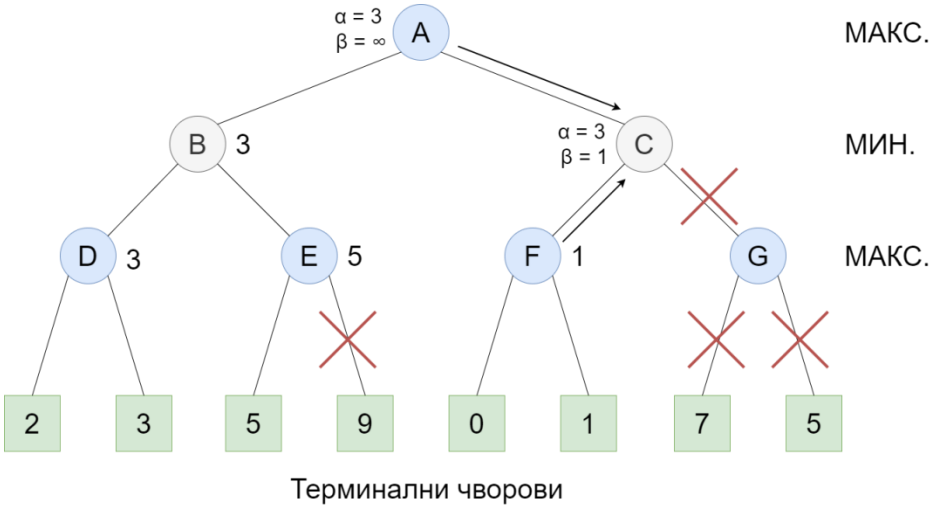
5. Функција се у следећем кораку враћа назад уз стабло из чвора ***B*** у чвор ***A***. Вредност промењиве алфа ће се променити на ***max(-3,inf)*** = 3. Бета остаје +бесконачно и те вредности се преносе даље следећим потомцима из чвора **A** у чвор ***C***, затим из ***C*** у чвор ***F***.

6. У чвору ***F*** вредност промењиве алфа се упоређује са левим дететом које има вредност 0 ***max(3,0)*** је 3, а затим упоређује се са десним дететом које има вредност 1, ***max(3,1)*** је 3. Алфа и даље остаје једнак 3 али вредност чвора постаје 1.



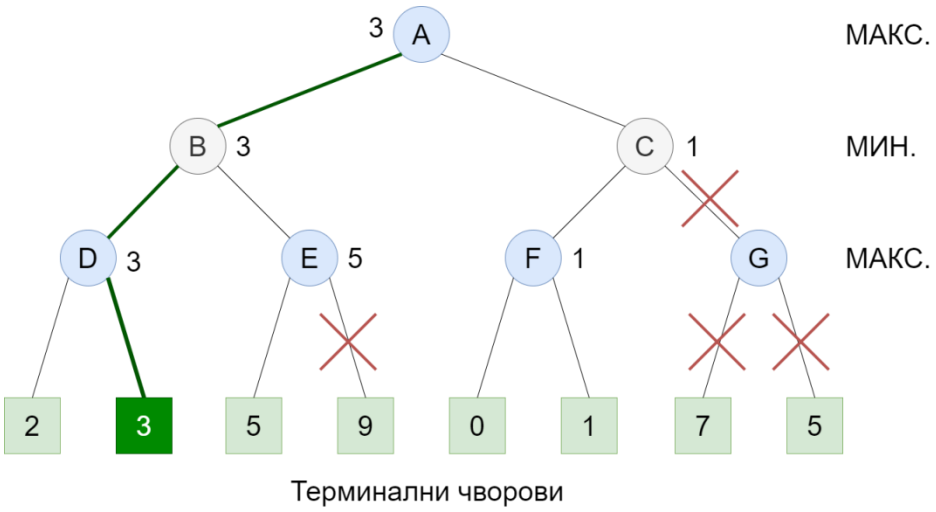
Слика 11 Појашњење алфа-бета алгоритма

7. Чвор ***F*** прослеђује назад вредност 1 чвору ***C*** у којем је тренутно стање алфа је 3 а бета је бесконачно. Мења се бета на ***min(inf,1)*** и сада чвор ***C*** има алфа једнак 3, бета једнак 1. Испуњава се услов ***alpha > beta*** и долази до одсецања чвора ***G***, где алгоритам неће наставити у том правцу.



Слика 12 Појашњење алфа-бета алгоритма

8. ***C*** враћа вредност 1 у чвор ***A*** где се рачуна ***max(3,1)*** = 3. Иако су одређени чворови одсечени из рачунице, то није утицало на крајњи резултат најбољег потеза, што је и карактеристика афла-бета одсецања.



Слика 13 Појашњење алфа-бета алгоритма

5. Имплементација апликације “Вук и овце”

5.1 Опис окружења *Qt*

Целокупна имплементација апликације “Вук и овце” извшена је у окружењу ***Qt***. Алтернативни назив који је често коришћен је ***“cute”*** , је моћан и разнолик оквир за развој апликација. Апликације развијене у овом окружењу могу се експортовати на више платформи. Креиран је од стране компаније ***Qt*** (која је раније била познатија под именом ***Troltech***). Тренутно се одржава од стране компаније ***Qt*** и ***Qt project***. Представља врло популаран избор код програмера приликом израде апликација јер поседује богати графички кориснички интерфејс(ГУИ).

Главне карактеристике ***Qt*-а**:

- **Вишеплатформски развој** – Једна од изузетних особина ***Qt*-а** је способност да врло једноставно пужа опцију креирања апликација на различитим оперативним системима као што су ***Windows, macOS, Linux, Android*** і ***IOS***. На овај начин значајно доприноси програмерима да на лакши начин креирају жељену апликацију, па на тај начин постаје доступна широј публици.
- **Обимна библиотека модула** – Пружа велики опсег модула за различите потребе програмера. Модули садрже све основне структуре података(***Qt core***), напредне графичке компоненте(***Qt widgets***) као и друге бројне модуле.
- **Опен соурце и комерцијална лиценца** - Доступан је под отвореним изворним кодом(***GPL***) и такође пружа комерцијалне лиценце. На тај начин програмери постају слободнији у одабиру модела лиценцирања и модула који су им неопходни за развој њиховог пројекта.
- **Кориснички интерфејс** – Нуди лак и интуитиван приступ програмеру при изради графичког интерфејса. Подржава прилагодљиве теме и садржи разне врсте алата за додавање стилова и анимација, допуштајући програмерима да произведу веома привлачну визуелну апликацију.
- **Интеграција са језиком *C++*** – ***Qt*** поседује дубоку интеграцију са програмским језиком ***C++***, па је прикладан за ***C++*** програмере. Садржи широк асортиман властитих библиотека и функција, те на тај начин пружа могућност побољшавања продуктивности програмера.

Као што је напоменуто, ***Qt*** садржи велики број модула који са собом доносе додатне функционалности. Основни модули које нуди ***Qt*** су:

1) ***Qt core*** – Овај модул садржи основне класе и функције, не графичке класе које се користе у другим модулима. Садржи структуре података, рад са датотекам, обраде догађаја и друге функционалности.

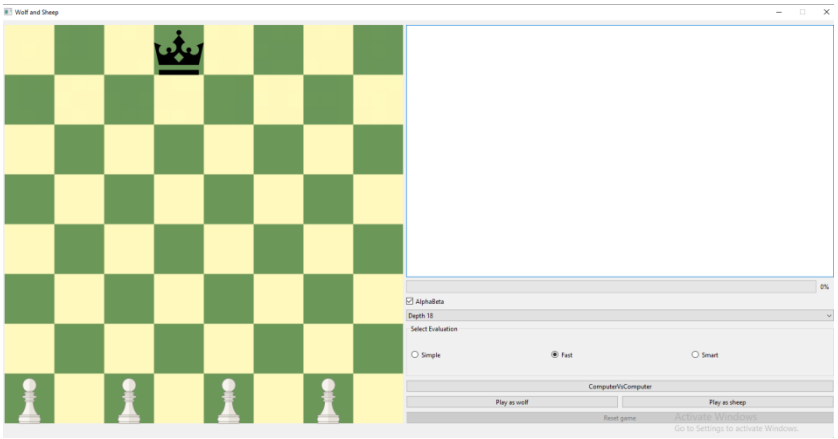
2) ***Qt GUI*** – Овај модул садржи основне класе за графичке корисничке интерфејсе (ГУИ). Укључује и ОпенГЛ.
3) ***Qt Multimedia*** – Садржи класе за аудио, видео, радио и функционалност камере и тиме пружа подршку за репродукцију звука, видеа и интеракцију са камерама.
4) ***Qt Multimedia Widgets*** – Садржи класе базиране на њидгетима за имплементацију мултимедијалних функција.
5) **Qt Нетњорк** – Садржи класе које олакшавају програмирање мрежних апликација, прављање сокета и рад са мрежним сервисима.
6) ***Qt QML*** – Садржи класе за ***QML*** и ***JavaScript*** језике.
7) ***Qt SQL*** – Садржи класе за интеграцију база података користећи ***SQL*** језик.

Наравно поред основних постоје и додатни модули које је врло лако интегрисати у постојећи пројекат. Листа ***Ad On*** модула налази се на званичном сајту ***Qt*-а**. ***Qt installer*** поседује опцију где пружа могућност једноставне селекције којом се скидају и инсталирају нови модули.

5.2 Графички интерфејс апликације

Апликација је релизована у програмском језику ***C++***, што је чини компатибилном са окружењем ***Qt***. Састоји се од две целине, које су логичка која обухвата логику апликације и графичка која је ту да прикаже рад апликације и омогући интеракцију корисника и рачунара.

Qt нуди једноставан начин израде графичког интерфејса. Комбинацијом лејаут елемената за одређивање позиције других елемената унутар њега, слике која представља таблу за игру, дугмади за покретање и заустављање игре као и лабела за приказ информација, добијамо једноставан интерфејс помоћу којих корисник може играти игру. Дизајн апликације изгледа на следећи начин:



Слика 14 Графички интерфејс апликације

Цела логика игре садржана је унутар класе која се назива ***gamelogic***. Свако дугме је повезано са својом функционалности преко слотова. Кликом на дугме за почетак игре диконектују се сви слотови. Уколико је играч први на потезу слот се поново активира. Након сваког потеза играча, такође се сви слотови деактивирају како играч не би могао да направи потезе док рачунар размишља. Након завршеног размишљања рачунара, слот за игру се поново активира како би играч могао да унесе следећи потез. У коду се то једноставно имплементира на следећи начин:

```
void MainWindow::disconnectSlot(){ {Конектовање слотова}
disconnect(clickHandle, &ClickHandle::clicked, this, nullptr);
}

void MainWindow::connectSlot(){ {Дисконектовање слотова}
connect(clickHandle, &ClickHandle::clicked, this, [=](int x, int y){
int i = y / picSize;
int j = x / picSize;
      playGame(i, j);
});
```

5.3 Структуре података

Структуре података представљају основни концепт који се користи за организовање и начин складиштења података у меморији рачунара. У зависности од проблема, избор праве структуре података доводи до тога да се тај проблем може ефикасније решити. Структуре података дефинишу начин на који се чувају подаци и операције које се могу извршити над њима. Основа су за решавање сложенијих проблема у рачунарству и програмирању. Неке од структура података су низови ланчане листе стекови редови приоритетни редови графови и стабла. Сваки корисник има могућност креирања и своје структуре података како би ефикасније решио свој проблем. Структуре података које су коришћене у изради апликације "Вук и овце" су:

- **Низ-матрица** - за репрезентацију стања игре.
- **Унутрашњи стек** - служи да подржи рекурзивне позиве минимакс алгоритма.
- **Додатна структура *Move*** - служи да памти потез.

Са обзиром да се игра "Вук и овце" у реалном свету одиграва на стандардној шаховској табли величине ***8x8***, најочигледнији начин за репрезентовање стања игре је и матрица исте величине. Постоје две врсте играча "Вук и овце", тако да на тај начин постоји 3 стања поља матрице: празно, играч вук, играч овца. За репрезентацију празног поља на табли коришћен је карактер спејс. За репрезентацију играча вук коришћен је карактер 1 и за репрезентацију играча овца коришћени су карактери 2. На основу тога пример једне позиције у игри може се представити на следећи начин:

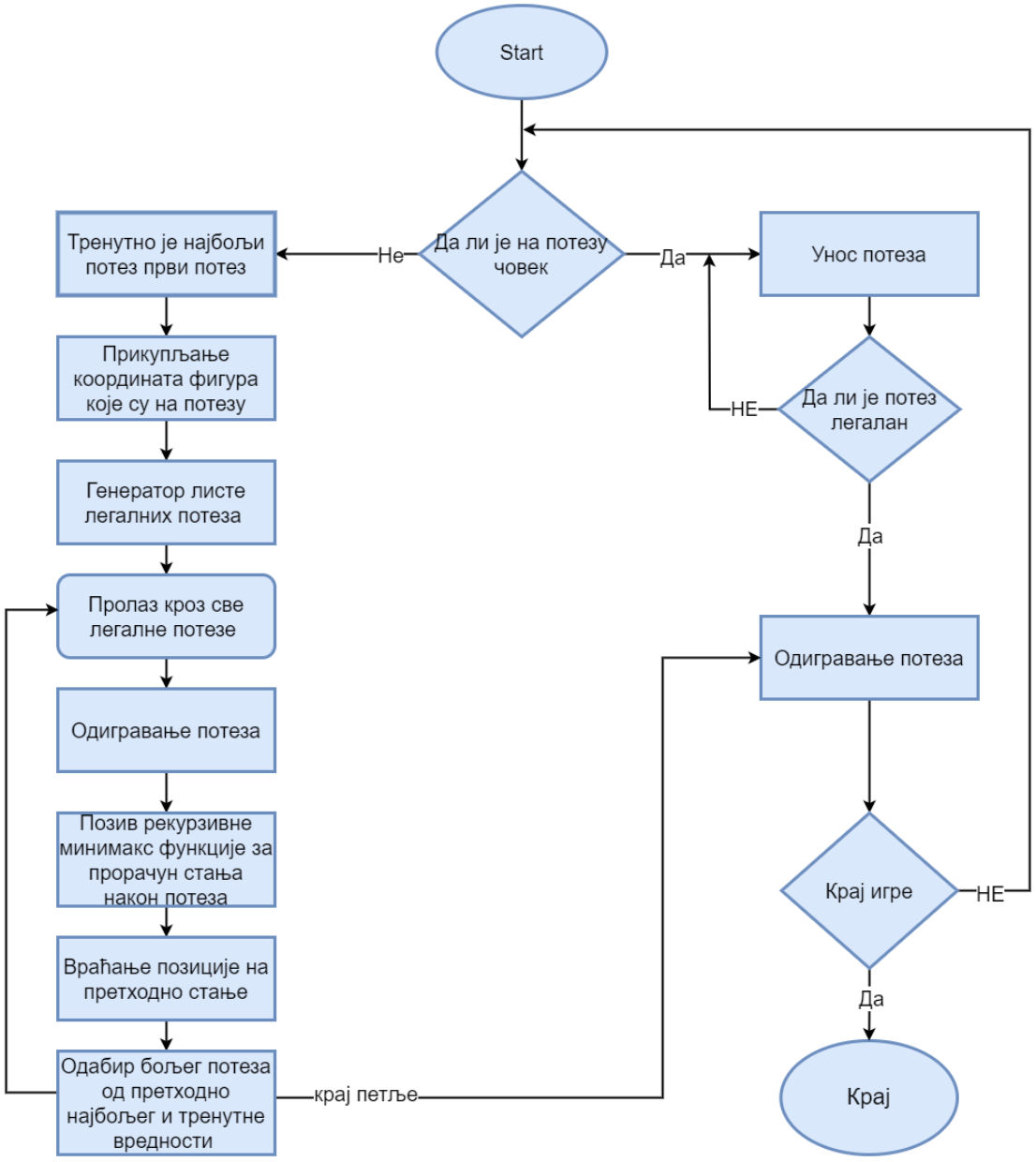
```
{
        { ' ', ' ', ' ', '1', ' ', ' ', ' ', ' '},
        { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
        { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
        { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
        { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
        { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
        { '2', ' ', '2', ' ', '2', ' ', '2', ' '}
};
```

За повратну вредност најбољег потеза као и за одигравање потеза коришћена је структура података ***Move*** која садржи четири променљиве, две почетне и две крајње координате на при померању фигуре. Такође је и коришћена структура ***CELL*** која памти само две координате које представљају и и j позиције поља.

```
struct CELL{ {Структура која памти позицију једног поља}
int i;
int j;
};

struct Move{ {Структура која памти позицију једног потеза}
int startI;
int startJ;
int endI;
int endJ;
};
```

5.4 Дијаграм тока података

На следећем дијаграму дат је ток података кроз одигравање потеза унутар апликације.



Слика 15 Дијаграм тока података

Рекурзивни позив минимакс функције функционише на идентичан начин као при одабиру најбољег потеза рачунара, са модификацијом где се је повратна вредност функције вредност стања, а не потез. Након испитивања сваког стања, минимакс функција такође испитује алфа-бета услов за одсецање, и у колико је испуњен, функција се

завршава. Такође се излази из минимакс функције уколико је дубина рекурзивне функције једнака нули.

5.5 Имплементација минимакс алгоритма

У претходним поглављима било је више речи о томе како минимакс принцип ради и пар примера заједно са његовим псеудо кодом. Унутар апликације имплементирана је функција помоћу минимакс алгоритма који садржи дубинско ограничење и алфа-бета одсецање. Такође је и додата мала модификација пре него што се изврши евалуација стања. Имплементација је извршена на следећи начин:

```
int GameLogic::MinimaxValue(char state[][SIZE], int depth, int alpha, int
beta, int isMaximizingPlayer) { {Функција која рекурзивно враћа вредност
позиције}
if (depth == 0 || isGameOver(state) != 0){ {Услов за повратак из функције}
   totalEvaluations++;
   if (isMaximizingPlayer)
         return evaluate(state) + DEPTH - depth;   {Евалуације стања}
   return evaluate(state) - DEPTH + depth;
   }
 if (isMaximizingPlayer){
     int bestScore = INT_MIN, numMoves;

     {Генерисање легалних потеза}
     struct Move* possibleMoves = LegalWolfMoves(state,&numMoves);

     for (int i = 0;i<numMoves;i++){
{Одигравање, рекурзивно израчунавање позиције и враћање потеза за
испитивање следећег}
      makeMoveForWolf(state, possibleMoves[i]);
      bestScore = Max(bestScore, MinimaxValue(state, depth - 1, alpha,
beta, 0));
      revertMoveForWolf(state, possibleMoves[i]);
         if(AlphaBeta) {
             alpha = Max(alpha, bestScore);
                if (beta <= alpha) {  {Алфа-бета одсецање}

                      break;
                 }
         }
 }
 delete[] possibleMoves;
 return bestScore;
}
else{
    int bestScore = INT_MAX, numMoves;

    {Генерисање легалних потеза}
    struct Move*possibleMoves = LegalSheepsMoves(state, &numMoves);

    for (int i = 0; i < numMoves; i++) {
```

```
{Одигравање, рекурзивно израчунавање позиције и враћање потеза за
испитивање следећег}
    makeMoveForSheep(state, possibleMoves[i]);
    bestScore = Min(bestScore, MinimaxValue(state, depth - 1, alpha, beta,
1));
    revertMoveForSheep(state, possibleMoves[i]);
    if (AlphaBeta) {
       beta = Min(beta, bestScore);
       if (beta <= alpha){ {Алфа-бета одсецање}
          break;
        }
    }
 }
 delete[] possibleMoves;
 return bestScore;
}
}
```

Додатна модификација у имплементацији је могућност коришћења и некоришћења алфа-бета одсецања, те на тај начин корисник може бирати да ли жели да машина користи алфа-бета одсецање или обичан минимакс алгоритам. Псеудокод је замењен конкретним функцијама програмског језика ***C++***. Датаљнији опис генератора потеза и евалуационих функција дат је у наставку рада.

5.6 Генератор потеза

Свако стање рекурзивно генерише стања која могу да наступе из њега. То се одвија путем функције за генерисање потеза. Могуће је генерисати све потезе, што је бржа варијанта, али је потребно испитати их касније. Друга варијанта је генерисање легалних потеза, која је једноставнија за имплементацију. Апликација "Вук и овце" поседује имплементацију генератора легалних потеза.

```
struct Move* GameLogic::LegalWolfMoves(char board[][SIZE], int* numMoves){
   {Испитивање легалних потеза прослеђене позиције}
    int i = wolf.i;
    int j = wolf.j;
    *numMoves = 0;
    struct Move* moves = new Move[4];
    {Испитивања легалних потеза}
    if (board[i + 1][j + 1] == ' ') {потез доле-десно}
        moves[(*numMoves)++] = Move{ i, j, i + 1, j + 1 };
    if (board[i + 1][j - 1] == ' ') {Потез доле-лево}
        moves[(*numMoves)++] = Move{ i, j, i + 1, j - 1 };
    if (board[i - 1][j + 1] == ' ') {Потез горе-десно}
        moves[(*numMoves)++] = Move{ i, j, i - 1, j + 1 };
    if (board[i - 1][j - 1] == ' ') {Потез горе-лево}
```

```
        moves[(*numMoves)++] = Move{ i, j, i - 1, j - 1 };
    if (*numMoves < 4) { {Реалокација меморије}
        Move* temp = new Move[*numMoves];
        std::copy(moves, moves + *numMoves, temp);
        delete[] moves;
        moves = temp;
    }
    return moves;
}
```

Генератор легалних потеза за играча вука генерише потезе тако што испитује доле-десно доле-лево горе-десно и горе-лево поље, у том редоследу са циљем да потез на доле у будућности, потенцијално одсече друге потезе обзиром да је циљ вука да достигне до последњег реда, па је потез на горе највероватније дужи пут до победе.

```
struct Move*GameLogic::LegalSheepsMoves(char board[][SIZE],int * numMoves)
{ {Генерисање легалних потеза оваца}
    int z;
    *numMoves = 0;
    struct Move*moves = new Move[8];
    struct CELL*sheep = coordinatesOfSheeps(board);
    for (z = 0; z < 4; z++){ {Испитивање потеза за сваку овцу}
        int i = sheep[z].i;
        int j = sheep[z].j;

        if (board[i - 1][j - 1] == ' ') {Потез горе-лево}
            moves[(*numMoves)++] = Move{ i, j, i - 1, j - 1 };

        if (board[i - 1][j + 1] == ' ') {Потез горе-десно}
            moves[(*numMoves)++] = Move{ i, j, i - 1, j + 1 };
    }
    if (*numMoves < 8){
        Move* temp = new Move[*numMoves]; {Pealokacija memorije}
        std::copy(moves, moves + *numMoves, temp);
        delete[] moves;
        moves = temp;
    }
    delete[] sheep;
    return moves;
}
```

Генератор легалних потеза за играча оваца на први погледа изгледа веома слично, где се прво испитују потези на позицију горе-лево а затим на позицију горе-десно. Међутим кључна ставка генератора потеза за овог играча састоји се у томе да је редослед играча динамички одабран у зависности тренутнеог стања табле.

```
CELL*GameLogic::coordinatesOfSheeps(char board[][SIZE]) { {Редослед
одабирања координата оваца}
    int br = 0, i, j;
    struct CELL*cells = new CELL[4];
    for (i = SIZE - 1; i >= 0; i--) {Почевши од задњег реда}
        if (wolf.j < 4) {Уколико је вук са леве стране табле бирају се
овце са лева на десно}
            for (j = (i + 1) % 2; j < SIZE; j += 2){
                if (board[i][j] == '1'){
                    cells[br++] = CELL{ i, j };
                    if (br == 4)
                        return cells;
                }
            }
        else {Уколико је вук са десне стране табле бирају се овце са десна
на лево}
            for (j = (SIZE - 1) - (i % 2); j >= 0; j -= 2) {
                if (board[i][j] == '1'){
                    cells[br++] = CELL{ i, j };
                    if (br == 4)
                        return cells;
                }
            }
    return cells;
}
```

Као што се може закључити из кода изнад, редослед узимања оваца пред генерисања легалних потеза функционише тако што предност имају овце које су ближе реду у почетној позицији игре, а затим у зависности од позиције вука прво се бирају овце ближе њему. Ово обезбеђује брже пронлажење позиција где су све овце у истом реду, која има велику шансу да одсече друге потезе.

5.7 Евалуациона функција

Евалуациона функција представља функцију која на основу тренутне позиције игре процењује и додељује јој нумеричку вредност. [8] Вредности могу бити у разним опсезима. У реализацији апликације “Вук и овце” евалуационе вредности се крећу од минималне вредности итегера до максималне вредности ингегера (што уједно и представља победничке позиције).

Грануларност евалуационе функције представља минималну вредност за коју се могу разликовати два стања при евалуацији, и у овој апликацији та вредност износи 1. Због тога, веома је битно подесити евалуационе функције тако да прорачунавају стање на прави начин, јер мала грешка у процени позиције може довести до тога да та позиција буде изабрана као гора. У реалној игри евалуациона функција (начин на који играч процењује стање) се може мењати, док у програму једном дефинисана функција се позива

више милиона пута. Променом само пар параметара унутар евалуационе функције може доћи до креирања скроз другог начина игре.

Пре него што имплементација евалуационе функције буде описана, анализирањем претходно приказаног минимакс алгоритма се уочава следећа линија кода:

```
if (isMaximizingPlayer) {Утицај тренутне дубине стања на евалуацију}
       return evaluate(state) + DEPTH - depth;
return evaluate(state) - DEPTH + depth;
```

Она такође представља део евалуационе функције. Променљива ***DEPTH*** памти почетну дубину из које је позван минимакс алгоритам, док променљива ***depth*** памти тренутну дубину до које је стигао минимакс алгоритам. Њихова разлика представља број одиграних потеза до тог тренутка. Евалуацијом стања без увида у то који је потез по реду, може доћи до тога да на пример победничка позиција услед два потеза или победничка позиција услед пет потеза буде евалуирана истом вредношћу. Урачунавањем броја потеза до евалуације избегава се тај проблем, тако да ће машина увек бирати краћи пут. На овај начин такође је могуће одсећи нека стања и на тај начин убрзати програм.

Функција за евалуацију названа је ***evaluate***, и на основу одабране евалуације путем радио дугмади, бира једну од три мода рада.

```
int GameLogic::evaluate(char board[][SIZE]){ {Odabir evaluacione funkcije}
    if (SPEED == 0)
        return evaluateSimple(board);
    else
    if (SPEED == 1)
        return evaluateFast(board);
    else
        return evaluateSmart(board);
}
```

5.7.1 Евалуациона функција *evaluateSimple*

Једна од најбитнијих особина ***AI*** играча је брзина. Већа брзина омогућава програму да уђе у већу дубину и да на тај начин игра паметније. Код неких врло једноставних игара као што је Икс-Окс, обзиром да не постоји велики број могућих комбинација (око 250,000) евалуациона функција уз велику дубину практично може да садржи само крајња стања(победу, губитак и решено) и на тај начин увек одабрати најбољи потез. По сличном принципу реализована је прва евалуациона функција.

```
int GameLogic::evaluateSimple(char board[][SIZE]){ {Најједноставнија
евалуациона функција}
```

```
    if (!canWolfMove(wolf.i, wolf.j, board)){Крајња позиција}
        return INT_MIN;
    else
        if (wolf.i == SIZE - 1)
        return INT_MAX;
    return wolf.i; {Враћа се висина вука}
}
```

Ова евалуациона функција је најједноставнија и садржи провере за крајња стања. Уколико тренутно стање није крајње, враћа се висина вука. Са обзиром да је вук играч који максимизује, минимакс ће преко ове евалуације преферирати поља која су ближа циљу и поља која га доводе до победе, а избегаваће поља која га доводе до губитка. Супротно томе обзиром да овце покушавају да минимизују, трудиће се да вука врате што даље од циља. Иако ова евалуација делује наивно, она допушта програму да смањи своје време израчунавања евалуације, како би ушао у већу дубину и на тај начин предвидео победу.

5.7.2 Евалуациона функција *evaluateFast*

Циљ идеалне евалуационе функције је наћи баланс између времена евалуације и квалитета потеза који су продукт евалуације. Ова евалуација је то и покушала да оствари.

```
int GameLogic::evaluateFast(char board[][SIZE]){ {Брзи евалуатор}
    if (!canWolfMove(wolf.i, wolf.j, board)) {Крајња позиција}
        return INT_MIN;
    else
        if (wolf.i == SIZE - 1)
        return INT_MAX;
    struct CELL*sheep = coordinatesOfSheeps(board);
{Уколико су све овце у истом реду, а вук изнад њих, позиција се сматра
сигурно победничком у будућности}
    if (sheep[0].i == sheep[1].i && sheep [1].i == sheep[2].i &&
        sheep[2].i == sheep[3].i && wolf.i < sheep[0].i){
        int ret = INT_MIN / (sheep[0].i + 1);
        delete[] sheep;
        return ret;
    }
{Уколико је вук у истој или мањој висино од најниже овце, позиција се
сматра сигурно победничком у будућности}
    if (wolf.i >= sheep[0].i){
        delete[]sheep;
        return INT_MAX / 2;
    }
    int left = 0, i;
{Уколико су овце по две у свакој половини табле}
    for (i = 0; i < 4; i++)
        if (sheep[i].j < 4)
```

```
            left++;
    if (left == 2){
        delete[]sheep;
        return wolf.i * 2 + INT_MIN / 4;
    }
    int Sum = 0;
{Сума растојања између вука и оваца}
    for (i = 0; i < 4; i++)
        Sum += abs(sheep[i].i - wolf.i);

    delete[] sheep;
    return wolf.i * 2 - Sum;
}
```

Као што се одмах може уочити функција ***evaluateFast*** је доста комплекснија од функције ***evaluateSimple***. Ова функција покушава да пронађе довољно добру позицију и да на тај начин што пре заврши евалуациона функција. Аспекти које узима при прорачунавању евалуације су и то по редоследу:

1) **Крајња позиција вука или крајња позиција овце –** Уколико није наставља се евалуација.
2) **Све овце су у истом реду, а вук је изнад њих** - Та позиција је практично увек победничка, где год да се налазио вук уколико је изнад њих. Ова позиција се сматра полупобедничком и како би се уштедело време излази из функције. Наравно како би се разликовала од исте те формације али ближе првом реду, дели се са висином реда. На овај начин овце преферирају сигуран пут до победе, а још бржем доласку до ове позиције доприноси начин генерисања легалних потеза који такође охрабрује овце да се крећу формацијом линије. Уколико ни овај услов није испуњен прелази се на следећи корак.
3) **Висина реда у коме се вук налази је иста или нижа од висине реда у којем се налази последња овца –** Обзиром да се овце увек крећу унапред, када вук достигне висину последње овце, то значи да не немогуће заробити вука. Тада је победа вука загарантована.
4) **Да ли су овце једнако распоређене по редовима –** Уколико су овце распоређене две на левој и две на десној половини табле евалуација се завршава.
5) **Уколико ни један од услова није испуњен** – Вук покушава да проближавањем овцама смањи број њихових могућих потеза и на тај начин их натера да изаберу потез који може довести до њиховог губитка.

5.7.3 Евалуациона функција *evaluateSmart*

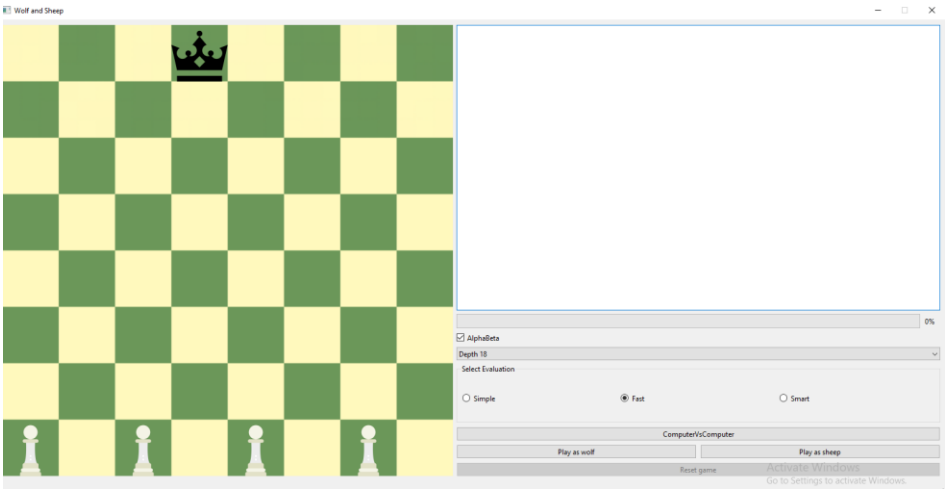
Ова евалуациона функција покушава да жртвује време које је потребно за израчунавање позиције у нади да ће позиција бити боље евалуирана, и на тај начин одсећи друге лошије потезе а у исто време и одигравати паметније потезе.

```
int GameLogic::evaluateSmart(char board[][SIZE]){ {Паметни евалуатор}
    if (!canWolfMove(wolf.i, wolf.j, board)) {Крајња позиција}
        return INT_MIN;
    else
        if (wolf.i == SIZE - 1)
        return INT_MAX;
    struct CELL*sheep = coordinatesOfSheeps(board);

{Уколико је вук у истој или мањој висино од најниже овце, позиција се
сматра сигурно победничком у будућности}
    if (wolf.i >= sheep[0].i){
        delete[] sheep;
        return INT_MAX / (7 - wolf.i);
    }
    int eval = 0;

{Уколико су све овце у истом реду, а вук изнад њих, позиција се сматра
сигурно победничком у будућности}
    if (sheep[0].i == sheep[1].i && sheep[1].i == sheep[2].i &&
        sheep[2].i == sheep[3].i && wolf.i < sheep[0].i)
        eval += INT_MIN / (sheep[0].i + 5);

{Уколико су овце по две у свакој половини табле}
    int left = 0, i;
    for (i = 0; i < 4; i++)
        if (sheep[i].j < 4)
            left++;
    if (left == 2)
        eval += INT_MIN / 4;

{Сума растојања између вука и оваца}
    int volvesSum = 0;
    for (i = 0; i < 4; i++)
        eval += abs(sheep[i].i - wolf.i) + abs(sheep[i].j - wolf.j);

{Број легалних потеза оваца и вукова}
    int WolvesMoves;
    struct Move*possibleWolfMoves = LegalWolfMoves(board, &WolvesMoves);
    int SheepMoves;
    struct Move*possibleSheepMoves = LegalSheepsMoves(board, &SheepMoves);

    delete possibleWolfMoves;
    delete possibleSheepMoves;

    eval += 10 * WolvesMoves;
    eval -= SheepMoves;

    delete[] sheep;
    return wolf.i - eval; {Утиче и висина вука}
}
```

Функција ***evaluateSmart*** користи исте услове као и функција ***evaluateFast***. Разлика је у томе што се не зауставља након испуњених услова, те прорачунава позицију на тачнији начин. Такође узима у обзир број валидних потеза које имају оба играча и покушава да их повећа. Иако су ове две функције доста сличне међу собом, играч игра другом стратегијом јер су и стања процењена на други начин.

6. Коришћење апликације

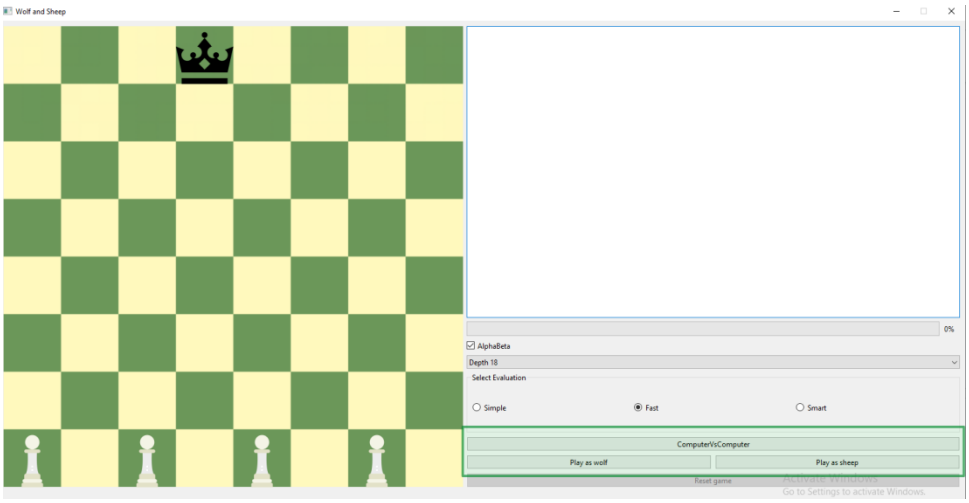
Приликом покретања апликације корисник се сусреће са графичким интерфејсом који изгледа као на слици 16.



Слика 16 Почетни приказ апликације

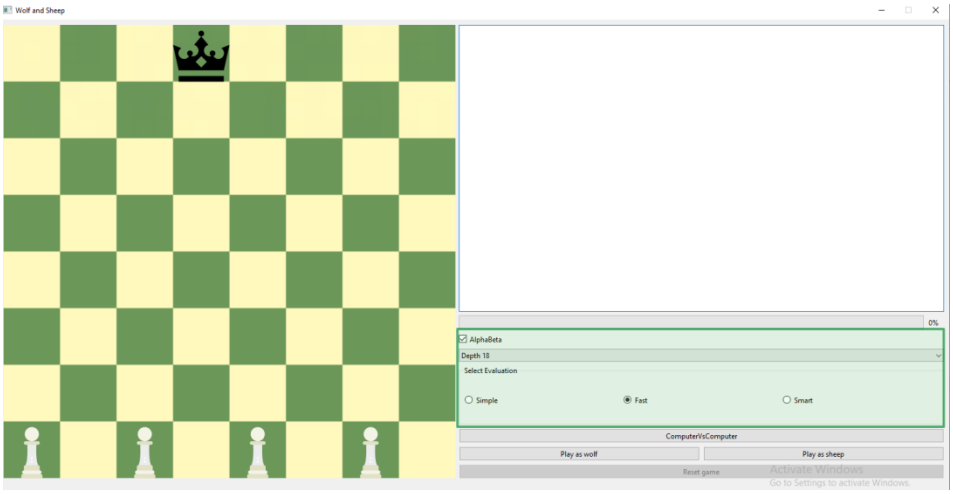
Пре покретања игре корисник има могућност да одабере дубину којом ће размишљати машина, да ли ће користити алфа-бета одсецање или не, и једну од три мода евалуације потеза. Након одабраних опција, корисник може одабрати једну од три опције:

- Мод играња игре као играч вук
- Мод играња игре као играч овце
- Мод рачунар против рачунара



Слика 17 Модови игре

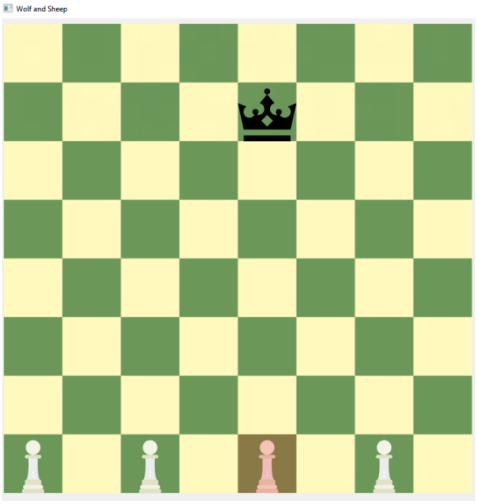
Корисник поседује опцију одабира коришћења или не коришћења алфа-бета одсецања,као и један од модова евалуације. Могуће дубине играња се крећу у распону од 1 до 18, Слика 18.



Слика 18 Опције игре

Приликом одабира мода играња као вук, обзиром да је вук први на потезу корисник бира слободно поље које одговара правилима кретања вука. Уколико дође до одабира поља које није легално, кориснику ће искочити обавештење. Након ког треба одабрати легално поље. Уколико је поље успешно одабрано, сви слотови се деактивирају док рачунар не одигра свој потез. Приказује се потез рачунара и играч је поново на потезу.

Приликом одабира мода играња као овце, прва на потезу је машина, те су слотови за играча искључени. Након одиграног потеза машине, играч селектује једну од своје четири фигуре. Фигура која је селектована постаје означена на екрану, Слика 19, и апликација чека док корисник не одабере где жели да помери одабрану фигуру. Тада је на реду вук, и принцип се понавља.



Слика 19 Одабрана фигура

Моду рачунара против рачунара је осмишљен како играч може посматрати и анализирати начин рада рачунара, и на тај начин стећи увид у његове стратегије. Током овог мода корисник може само посматрати партију. Временско ограничење одигравања потеза рачунара је постављења на минимално једну секунду како би одигравање потеза био прегледније.

7. Тестирање система

7.1 Мерење протеклог времена

Јачина машине у корелацији је са брзином израчунавања стања. Уколико две машине исто стање евалуирају истом евалуационом функцијом за другачији временски интервал, може се рећи да је она која има већу брзину јача. Тако да као главна метрика процене користиће се време потребно за израчунавања тест примера.

Std библиотека садржи додатне модуле унутар ње као што је ***crono*** а затим ***steady_clock***. За рад на апликацији "Вук и овце" једино су потребне функције за почетак мерења времена и крај мерења времена. Позив функција се врши на следећи начин:

```
auto start = std::chrono::steady_clock::now();{Мерење времана}
//poziv funkcije
auto end = std::chrono::steady_clock::now();
std::chrono::duration<double> duration = end - start;
```

На овај начин интервал који је протекао између почетка и краја мерења времена је садржан у променљивој ***duration.count()*** као вредност у покретном зарезу и то у секундама.

7.2 Тестирање минимакс алгоритма са и без алфа-бета одсецања

За најлакши приказ побољшања којег доноси алфа-бета одсецање, искористиће се табела која садржи дубину на којој се испитује игра, број евалуираних стања без минимакс алгоритма и број стања након алфа-бета одсецања.

Дубина	Бр. евалуација са алфа - бета одсецањем	Бр. евалуација са алфа-бета одсецањем
2	28	10
3	172	46
4	516	54
5	3132	272
6	12474	308
7	75455	1601
8	228865	3413
9	1382730	16829
10	5126360	9795
11	31504600	64504

Табела 1 Број евалуација пре и након алфа-бета одсецања ***evaluateSimple***

Ова табела приказује број потеза пре одсецања и након одсецања приликом примене евалуационе функције ***evaluateSimple***. Занимљиво је да чак и повећањем дубине догодило се то да је број евалуација са алфа бета одсецањем опао. Значајно убрзање у времену израчунавања у дубини 10, које се свело са 0.54521 секунди на 0,0097756 секунди.

Дубина	Бр. евалуација са алфа бета одсецањем	Бр. евалуација са алфа бета одсецањем
2	28	10
3	172	47
4	516	59
5	3 132	300
6	12474	363
7	75455	1706
8	228865	3157
9	1382730	17219
10	5126 360	10503
11	31504600	65723

Табела 2 Број евалуација пре и након алфа-бета одсецања ***evaluateFast***

Дубина	Бр. евалуација са алфа бета одсецањем	Бр. евалуација са алфа бета одсецањем
2	28	16
3	172	148

4	516	115
5	3132	651
6	12474	323
7	75455	1817
8	228865	9192
9	1382730	47005
10	5126360	12851
11	31504600	135350

Табела 3 Број евалуација пре и након алфа-бета одсецања ***evaluateSmart***

На дубини од 11 број евалуација је смањен на читавих 50 пута! Време за израчунавање овог броја евалуација без алфа бета одсецања је око једне секунде. Свакој следећој дубини ово време би се повећало за 7 или 4 пута у зависности од тога који је играч на потезу. Овим једноставим примером закључује се да једноставна техника одсецања као што је алфа бета доприноси значајном убрзању програма. А на овај начин омогућава програму да чак уђе у дубину 18.

7.3 Тестирање рада евалуационих функција

У овом одељку вршиће се упоређивање времена потребног евалуационим функцијама да одиграју потез на истим дубинама. Те ће се рачунати однос њихових брзина.

	evaluateSimple		***evaluateFast***		***evaluateSmart***	
Дубина	Број евалуација	Време у секундама	Број евалуација	Време у секундама	Број евалуација	Време у секундама
11	64504	0.0170735	65723	0.0430945	135350	0.148381
12	119500	0.0625326	110907	0.056967	239833	0.218428
13	207805	0.0781877	216105	0.112166	757678	0.712332
14	236052	0.12282	241399	0.147009	375965	0.404753
15	1.54352e+06	0.314853	1.43539e+06	0.564912	3.03974e+06	2.91087
16	1.23482e+06	0.507997	1.31712e+06	0.70745	4.31611e+06	4.02458
17	5.18759e+06	1.0538	5.22554e+06	2.05646	1.85257e+07	17.156
18	6.78015e+06	2.56493	6.41409e+06	3.71816	9.25777e+06	9.46862

Табела 4 Упоређивање брзина евалуационих функција

Као по претпоставци, најједноставнији евалуатор је и најбржи, док је најкомплекснији најспоријi. Занимљивост је да у дубини 18 ***evaluateFast*** је смањио број евалуација испод евалуација једноставног евалуатора али због своје веће комплексности и даље је спорији. Брзина израчунавања је од великог значаја, на крају крајева најзначајнији део је одабир правог потеза, тако да иако је најједноставнији евалуатор најбржи, није нужно да ће направити најбољи потез.

Експреиментално је утврђено да рачунар са најједноставнијим евалуатором (***evaluateSimple***), генератором потеза споменутим у секцији имплементације и дубином од 18, када играта као играч овце никада неће моћи да изгуби.

7.4 Тестирање рада апликације на различитим уређајима

Тестирање се врши са алфа-бета одсецањем, играч на потезу је вук.

	Рачунар 1			Рачунар 2			Рачунар 3			Рачунар 4		
Дубина	Ф1	Ф2	Ф3	Ф1	Ф2	Ф3	Ф1	Ф2	Ф3	Ф1	Ф2	Ф3
14	0.117	0.149	0.38	0.048	0.09	0.14	0.238	0.366	1.05	0.10	0.1	0.3
15	0.321	0.523	2.85	0.112	0.21	1.03	0.755	1.313	7.58	0.29	0.5	3.0
16	0.546	0.774	3.98	0.190	0.26	1.38	1.371	1.860	11.5	0.54	0.7	3.9
17	1.047	1.992	17.6	0.363	0.73	6.35	2.713	5.094	45.2	1.03	2.0	17
18	2.709	3.516	9.43	0.909	1.37	3.37	6.920	9.274	24.6	2.80	3.8	9.6

Табела 5 Упоређивање брзине рада апликације над више уређаја

Легенда:

- Ф1 – број секунди за израчунавање функције ***evaluateSimple***
- Ф2 – број секунди за израчунавање функције ***evaluateFast***
- Ф3 – број секунди за израчунавање функције ***evaluateSmar***

Спецификације рачунара:

- Рачунар 1 – ***Inter Core i5 3320M 2 Core @2.6GHz, 12GB 800MHz DDR3 RAM,Intel HD Graphic 4000***
- Рачунар 2 – ***Intel Core i7-12700KF 8 Core @5GHz 32GB 1800MHz DDR4 RAM, Nvidia Gforce RTX 3070 8GB GDDR6***
- Рачунар 3 – ***Intel Celeron CPU N3060 2 Core @1.6GHz, 4GB 1600MHz DDR3 RAM, Intel HD Graphics***
- Рачунар 4 – ***Intel Core i5-6300U 2 Core @2.5GHz, 8GB 2133MHz DDR4 RAM,Intel R(HD) Graphic 520***

7.5 Потенцијална побољшања апликације у будућности

Апликација је имплементирана на једноставан начин. Иако је то случај, у способности је да игра на врло интелигентан и брз начин. Додатним модификовањем могуће је још више побољшати њен рад. Неки од примера који могу донети побољшање апликацији су:

- **Коришћење друге репрезентације табле** преко на пример графова. Граф би био повезан и усмерен, тако да прати могуће легалне потезе играча.
- **Додавањем транспозиционих табела** како би програм могао да предвиди боље потезе.
- **Коришћење додатних херуистика** као што је ***history*** херуистика.
- **Додавање база отварања** и **база завршнице**.
- **Имплементацијом минимакс алгоритма који није ограничен дубином** већ временом.
- **Додавања могућности где машина размишља док чека на свој потез** и на тај начин пуни своје транспозиционе табеле.

8. Закључак

Овај пројекат је био путовање кроз фасцинантан свет теорије игара и практичне примене овог знања у области развоја софтвера. Кроз развој наше ***Qt*** апликације са вештачком интелигенцијом, не само да смо стекли дубље разумевање појединих концепта теорије игара, већ смо такође видели како се они могу превести у решења која функционишу у стварном свету.

Док размишљамо о изазовима и успесима које смо доживели током овог пројекта, очигледно је да теорија игара служи као кључна основа за дизајнирање стратегијских и узбудљивих игара. Научили смо како да моделирамо различите сценарије доношења одлука, развијемо вештачке противнике способне да доносе интелигентне одлуке и створимо кориснички интерфејс који доприноси привлачном искуству играча.

Овај пројекат истиче лепоту повезивања апстрактних концепата с конкретним, функционалним апликацијама. Надамо се да ће наш рад инспирисати друге да истраже узбудљиву тачку сусрета теорије игара и развоја софтвера и да стварају игре које изазивају, забављају и просветљују.

9. Литература

[1] Vladan Vuckovic "Prilog teoriji i praksi naprednih šahovskih algoritama", doktorska disertacija, Elektronski fakultet u Nišu. oktobar 2006.
[2] Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd изд.), Upper Saddle River, New Jersey: Prentice Hall, стр. 163—171, ISBN 0-13-790395-2
[3] Osborne, Martin J., and Ariel Rubinstein. A Course in Game Theory. Cambridge, MA: MIT, 1994. Print.
[4] Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd изд.). Upper Saddle River, New Jersey: Pearson Education, Inc. стр. 167. ISBN 978-0-13-604259-4.
[5] Heineman, George T., Gary Pollice, and Stanley Selkow (2008). „Chapter 7: Path Finding in AI”. Algorithms in a Nutshell. Oreilly Media. стр. 217—223. ISBN 978-0-596-51624-6.
[6] Judea Pearl, Heuristics, Addison-Wesley, 1984
[7] McCarthy, John (27. 2. 2006). „Human Level AI Is Harder Than It Seemed in 1955”. Приступљено 20. 12. 2006.
[8] Shannon, Claude (1950), Programming a Computer for Playing Chess (PDF), Ser. 7, vol. 41, Philosophical Magazine, retrieved 12 December 2021
[9] Ross, Don, "Game Theory", The Stanford Encyclopedia of Philosophy (Fall 2023 Edition), Edward N. Zalta & Uri Nodelman (eds.), https://plato.stanford.edu/archives/fall2023/entries/game-theory

10. Списак слика