Универзитет Унион

Рачунарски факултет

Лазар Миленковић

Поређење похлепних и бектрек алгоритама за стратегију игре Таблић и њених модификација

10.07.2017

Кандидат:	Лазар Миленковић
Број индекса:	РН 8/13
Наслов:	Поређење похлепних и бектрек алгоритама за стратегију игре Таблић и њених модификација
Ментор:	Др Владимир Миловановић
Чланови комисије:	Др Драган Урошевић

Апстракт

У овом раду пореде се временска, меморијска сложеност, као и ефикасност похлепне стратегије, као и неколико бектрек стратегија за игру Таблић и њених модификација. Таблић је популарна игра картама у којој сваки играч наизменично повлачи једну карту са циљем да на крају игре има највећи збир сакупљених карата. У стандардној верзији, играч зна само карте које му се тренутно налазе у руци као и све карте које су прошле до сада. За ову верзију игре имплементирана је похлепна стратегија по узору на стратегије које користе играчи. Модификована верзија подразумева да играч зна којом стратегијом играју његови противници, као и редослед карата до краја партије. За ову верзију су имплементиране бектрек методе које планирају неколико потеза унапред.

Садржај

1 Увод

Сматра се да су карташке игре откривене у Кини у деветом веку нове ере. До данас су толико распрострањене да представљају средство разоноде широм читавог света. Неке игре попут преферанса, блекџека и покера су толико популарне да постоје професионални турнири са позамашним наградним фондовима. Због такве популарности многи математичари се баве изучавањем карташких игара и њихових особина, покушавајући да нађу победничке стратегије или докажу непостојање истих. З последњих неколико година велики тренд су постали турнири у карташким играма на Интернету, где корисници такође могу зарађивати новац играјући против рачунара или против других играча. Овакав тренд повлачи велики број информатичара који имплементирају ботове за играње тих игара.

Овај рад се бави игром Таблић која је јако популарна на Балкану. У игри обично учествују два играча који користе цео шпил од 52 карте. На почетку се поставе четири карте на сто и сваком играчу се подели по шест карата. Играчи играју наизменично и у сваком потезу могу изабрати скуп карата са стола који у суми дају вредност неке карте из његове руке. Уколико постоји такав скуп, играч ставља све карте из скупа као и одговарајућу карту из руке на своју гомилу. Друга могућност играча је да произвољну карту из руке стави на сто. Када играчима понестане карата, подли им се опет по шест карата. Игра се завршава након четири круга дељења ($4+6\cdot 2\cdot 4 = 52$). Победник је онај играч који има највише штихова [1], а уколико је тај број исти онда је победник онај играч који је сакупио све укупно више карата. Могуће су верзије игре са четири такмичара где су обично два играча у једном тиму, као и верзија са три играча где сваки играч игра за себе. Верзија са три играча подразумева да се у последњем кругу дељења сваки играч добије по четири карте уместо шест. Правила игре су детаљно описана у другом поглављу.

Испитују се особине ботови чије стратегије се заснивају на похлепним и бектрек парадигмама. Посматране су успешност сваког од алгоритама, као и временске перформансе. Успешност је мерена симулирањем више партија где су противници два различита бота. Временске перформансе су најпре теоријски испитиване анализом сложености алгоритма, а након тога је мерено време специфичних имплементација. Неки од ботова који су имплементирани имају предност у виду познавања противникових карата или редоследа свих карата у шпилу. Оваква предност је неопходна да би алгоритми као што је бектрек уопште функционисали.

Мотивација за ову тему је једна од лекција курса *Увод у алгоритме, 6.006* са Масачусетског технолошког института [2]. У лекцији је описа-

[1] штихови су карте $10, A, J, Q, K$

но оптимално играње Блекџека уз познавање свих карата до краја игре. Алгоритам који је разматран користи динамичко програмирање као парадигму.

Друго поглавље бави се историјом таблића и њеним правилима. Описују се све специфичне ситуације које се могу догодити током партије. Разматра се и неколико стратегија које играчи обично користе. Ово поглавље представља потребан увод за разумевање стратегија које су описане у трећој секцији. Треће поглавље описује сваку од коришћених стратегија, мотивацију за њиховим коришћењем, теоријску анализу успешности и сложености, као и имплементацијске детаље. У поглављу са резултатима описује се начин на који су ботови тестирани и анализирају се добијени подаци.

2 О игри Таблић

У овом поглављу најпре је укратко изложена историја карташких игара, као и неке занимљивости о карташким шпиловима и њиховом дизајну. Након тога следи део који се бави неким заједничким особинама и правилима које деле све карташке игре. Ова заједничка правила представљају и основна правила таблића о којима се говори у последњем делу у коме су дата детаљна правила игре, као и све специфичне ситуације које се могу догодити током игре.

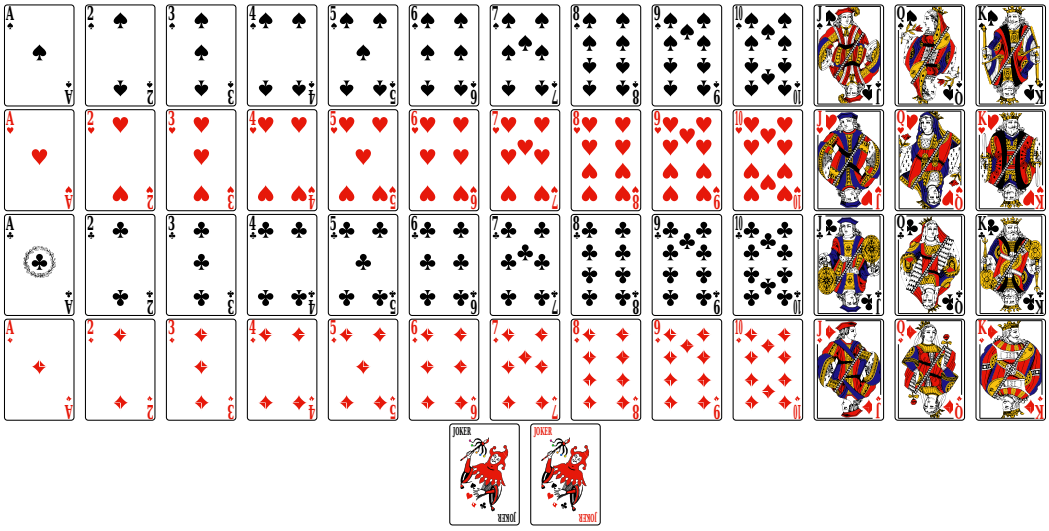
2.1 Историја карташких игара

Сматра се да су прве карташке игре настале у 9. веку нове ере у Кини за време владавине династије Танг. Ова претпоставка везана је за текст који описује ћерку тадашњег владара како 868. године игра „игру листова“ са члановима породице свог мужа. Постоји књига о овој игри за коју се претпоставља да ју је написала управо владарева кћи. Карташке игре повезују се открићем технике штампања на папир која је између осталог коришћена и за штампање шпилова карата.

Играње карата се проширило из Кине у Персију и Индију, где су шпилови имали различите боје односно знаке, налик савременим шпиловима. Сваки знак је имао 12 карата поређаних тако да карте са највећим бројевима осликавају краљеве и везире. До 11. века су се карташке игре раширile по целој Азији а дошле су и у Египат. Археолози су нашли египатске карте који потичу из 12. века. Њихови шпилови су имали 52 карте подељене у четири знака. Сваки знак је имао 10 обичних карата и 3 карте највеће вредности које осликавају владаре по хијерархији. Овакав изглед шпила доста личи на модеран шпил који је данас у употреби широм света. Оваква верзија шпила најпре је стигла у јужне делове Европе средином 14. века. Једина промена јесте у знацима који су се користили. У стом веку карташке игре су се прошириле и у Француску, Каталонију и Швајцарску. Ускоро се у Немачкој појављују професионални произвођачи карташких комплета који би штампали шпилове. Те штампане шпилове би касније ручно осликавали уметници.

Први симболи који се данас користе, листови и срца, уведени у Немачкој где су касније уведени и детелина и пик. Највећа карта која је означавала краља појављује се 1377. године у Немачким и Швајцарским шпиловима, где се први пут спорадично појављује и краљица на тринаестој карти. Још један битан моменат у развоју јесте и осликавање броја на ивице карте због лакшег држања у једној руци. Први овакав шпил датира из 1693, док се учестанија употреба усталила тек након 18. века. Могућност ротирања, односно симетрично осликавање датира из 1745. године. Ова карактеристика карата убрзо бива забрањена од Француске владе која је у то време контролисала дизајн. У другим европским земљама се овај дизајн усталио све до модерних шпилова. Заобљене ивице на картама настале су јер се оштре ивице брзо оштете што пружа могућност играчима да отркију о којој карти је реч на основу оштећења. Текстуре на полеђини карте настају из сличних разлога: карте са текстурама прикривају могућа ожтећења полеђине. Први шпилови који су садржали џокере појављују се у Америци где је у то време постојала игра која их користи. Постоји референца из 1875 која помиње употребу џокера као карте која мења другу карту, што је у данашњим играма обично случај [1].

Шпилови који се данас користе састоје се од 52 карте и џокера. Карте су подељене у четири знака: лист (пик), детелина (треф), ромб (каро) и срце (херц). Сваки од ових знакова има по 13 карата $A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K$, где A представља аса, J представља жандара, Q краљицу, а K краља.



Слика 1: *Модеран шпил карата састоји се од 52 карте у четири различита знака и џокера.*

2.2 Неке особине карташких игара

Карташким играма сматра се било која друштвена игра која подразумева коришћење горе описаног шпила. Захваљујући дугогодишњој историји карташког шпила, до данас постоји широк спектар различитих игара. Многе од игара имају светски прихваћена правила, док се у другима правила разликују од културе до културе.

Већина карташких игара има предефинисан број играча који могу истовремено играти једну партију. Разликујемо игре са једним играчем (солитер и пасијанс игре), игре са два играча (таблић обично сврставамо у ову категорију) и игре са више од два играча. Многе игре омогућавају проширење са два на четири играча поделом у партнерске парове. Обично ови парови седе један наспрам другог тако да једни другима не могу видети карте. Партнери могу размењивати информације које се не односе на карте које имају тренутно у руци. Други начин проширења игре јесте да сваки играч игра за себе и овакво проширење је скоро увек могуће и лимитирано је само на број карата у шпилу. Током партије која укључује два играча, оба играча углавном имају у руци само одређени број карата из шпила, јер би у случају поседовања свих карата играчи могли да знају све противникове карте.

У већини игара играчи играју по један потез у унапред одређеном смеру (нпр. редослед казаљке на сату уколико седе за столом). Обично је један играч, *дилер*, одређен да меша шпил и дели карте осталим играчима. У зависности од игре ово може бити предност или мана. Део игре између два дељења назива се *рука*, а тако се назива и скуп карата које један играч има код себе након дељења. Једна игра се састоји од неколико рука и завршава се обично када се потроши комплетан шпил или када неки играч пређе унапред дефинисан број поена. Многе игре захтевају да шпил буде добро промешан због подједнаких вероватноћа победе свих играча. На професионалним турнирима постоје дилери који нису играчи и који су тренирани да праведно мешају карте и вешто их поделе.

Након што дилер измеша карте, наредни играч по редоследу игре *сече* шпил. Сечење је потез у коме се шпил подели на две целине и пребаци се горња испод доње. Током дељења сваки играч добија карте једну по једну или више одједном у редоследу игре. Битно је да током дељења карте остану окренуте лицем ка столу тако да нико не може видети туђе карте. У неким играма се такође одређени број карата ставља на сто лицем окренутим на горе и те карте формирају *талон*. Остатак карата које нису подељене остављају се по страни за наредни круг дељења.

Скоро све игре настале су у малом кругу људи и одатле се ширили у зависности од интересантности. Таблић је игра која је настала на Балкану и углавном није позната у осталим деловима света. Постоје и светски

познате игре које имају интернационална правила и организације које воде рачуна о њима. Игра бриџ има интернационалне турнире које организују Светска бриџ федерација [6], као и локалне организације попут Бриџ савеза Србије [7] или Америчке контракторске бриџ лиге [5]. Још један пример светски познате игре је покер који се јавља са различитим модификацијама широм света. Постоје интернационални турнири са устаљеним правилима као Светски серијал у покеру [8] и Светски покерски турнир [9]. Правила ових организација морају се поштовати само унутар турнира које оне организују. Углавном се правила покера који се игра на осталим местима разликује у неком детаљу.

Већина правила за професионалне турнире има дефинисан скуп понашања играча који се сматра *варањем*. Играчи бивају кажњени у случају да буду ухваћени у варању. Постоје и ситуације где играч случајно види туђу карту или одигра потез када није ред на њега. Ове радње обично захтевају поништавање партије. Сматра се да играч који случајно види туђу карту а не пријави то такође вара.

2.3 Типови игара

Карташке игре могу се поделити у породице сличних игара које поседују неке заједничке особине.

Трик игре. У сваком потезу играчи извуку једну карту из своје руке и у зависности од вредности извучене карте један играч осваја појене. У зависности од игре победник је онај који освоји највише руку, најмање руку или тачно оређен број руку. Бриџ је пример овакве игре.

Игре комбиновања. Циљ оваквих игара је сакупити одређену колекцију карата пре осталих играча. Популарни представник ових игара је реми, где је циљ сакупити низ карата истог знака или све знакове једног броја. Победник је играч који се први ослободи свих карата из руке.

Игре паковања. Као што име налаже, циљ оваквих игара је „спаковати“ све карте из руке на сто. Блекџек и уно су популарни представници ове породице.

Игре сакупљања. Победник је играч који сакупи цео шпил. Познати представник је игра рат где играчи извлаче по једну карту и играч са већом картом преузме обе карте код себе.

Игре пецања. У овим играма играчи траже са талона карту која одговара карти у њиховој руци. Ове игре су најраспрострањеније у Италији где се скопа сматра националном карташком игром и Кини, где постоји доста различитих игара из ове породице.

Игре поређења. У овим играма победник је играч који је има највреднију руку. Представници ове групе су покер и блекџек.

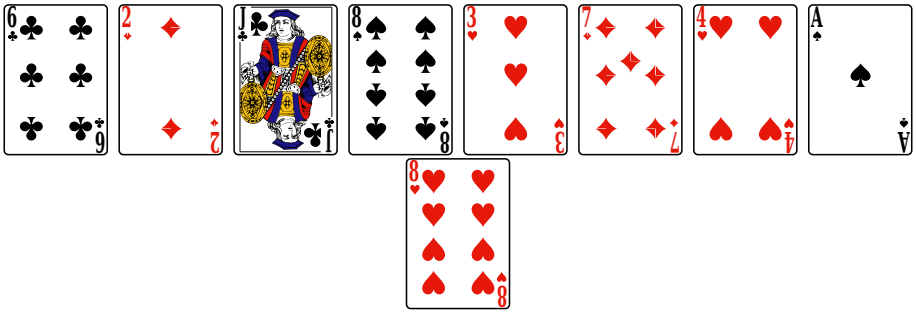
Пасијенс и солитер игре. Ове игре намењене су за једног играча који има циљ да на основу задатих правила сложи све карте из руке на талон.

2.4 Правила игре таблић

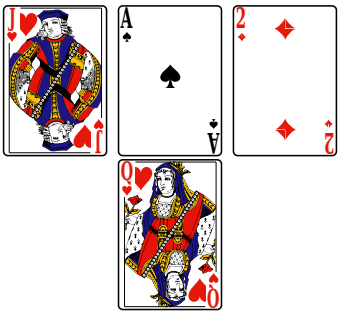
Таблић се најчешће игра у два играча и правила која следе односе се на такву игру [10]. У свакој партији одреди се дилер који дели карте током целе партије. У свакој наредној партији дилери се смењују. На почетку игре, након мешања и сечења шпила, играч који дели остави четири карте на талон. Након што остави карте на талон, дилер подели противнику и себи по једну руку која се састоји од 6 карата. Дилер дели руку тако што наизменично даје по три карте, најпре противнику па себи. Једна рука састоји се од по 6 потеза сваког играча, где потезе играчи повлаче наизменично. Обично први игра играч који није дилер. Игра се завршава када се истроши цео шпил, односно након четири руке.

У једном потезу играч изабере карту из своје руке и покушава да нађе један или више скупова карата са талона који у збиру дају његову карту. То значи да играч може узети све карте са талона које имају вредност као и његова одабрана карта из руке. Поред тога, играч може узети и све комбинације карата са талона које у збиру дају његову извучену карту. Ове комбинације морају бити дисјунктне, односно никоје две комбинације не смеју имати заједничку карту. Вредности карата одређују се њиховим бројем, са додатком да жандар има вредност 12, краљица 13, а краљ 14. Ас карта може у сваком потезу произвољно имати вредности 1 или 11 у зависности од играчеве одлуке. Може се десити да играч не може наћи одговарајуће карте на талону и у том случају треба да изабере произвољну карту из руке и одложи је на талон. Након одиграног потеза играч све сакупљене карте (рачунајући и карту из руке) ставља на своју гомилу. Када се заврши цела игра, преостале карте са талона сакупља играч који је последњи нешто носио са талона.

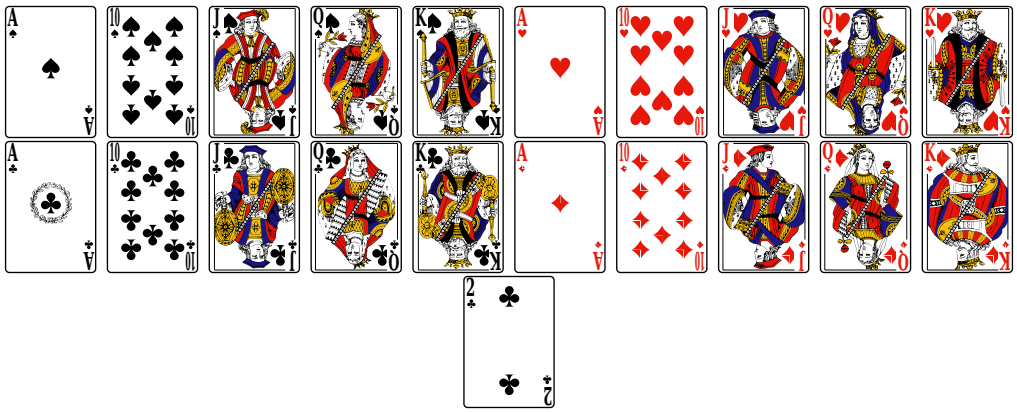
Победник је играч који на крају сакупи највећи број поена. Најзначајнији фактор за број поена јесте број освојених *штихова*. Штихови су карте A, 10, J, Q и , где свака од њих носи по један поен уз додатак да десетка каро (дупла десетка) носи два поена. Посебна карта у игри је двојка треф (мала двојка) која такође доноси један поен. Други фактор који утиче на поена је број „очишћених“ табли. Када играч након потеза остави талон празан (очисти талон), добија додатни поен. Уколико је резултат нерешен играч са већим бројем карата на гомили (уколико посотји) добија три додатна поена (три на карте). Победник је играч са већим бројем поена.



Слика 2: *Изглед табле представљен је у горњем реду, а корисник жели одиграти 8 срце. На талону се налазе следеће групе које одговарају осмици: шест детелина и два каро, 8 лист, 7 каро и ас лист, док је последња одговарајућа група 3 херц, 4 херц и ас лист. Последње две групе имају заједничког аса тако да се играч мора одлучити за само једну од њих, док су преостале две групе без заједничких чланова тако да их може носити обе. Играч се такође може одлучити и да не носи неку од група, али то обично није оптимална стратегија (видети наредни део).*



Слика 3: *Играч у руци има краљицу а на табли се налазе жандар, ас и двојка. У овом случају одговарајуће комбинације су жандар и ас (посматрамо га као да има вредност 1), или двојка и ас (посматрамо га као да има вредност 11). Није могуће носити све три карте.*



Слика 4: *Штихови су битне карте које играчу доносе један поен. Десетка каро (дупла десетка) носи два поена, а поред штихова ту је и двојка треф (мала двојка) која носи један поен.*

3 Преглед стратегија

У овом поглављу детаљно се описују сви коришћени алгоритми. Први део описује стратегије које обично људи користе играјући таблић. Те стратегије су саставни део похлепног алгоритма који се описује у другом делу. Последњи део описује бектрек алгоритме који имају више информација о игри него што би играч требало да има.

3.1 Стратегије које користе играчи

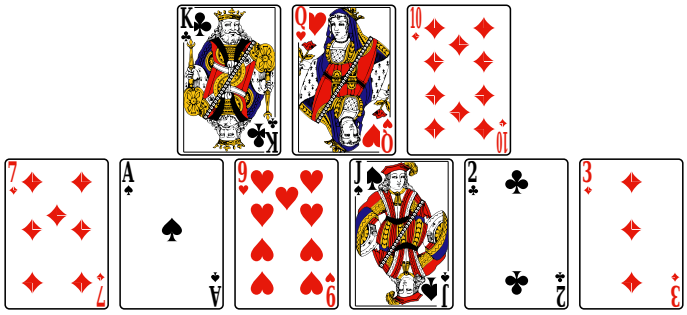
У овом делу разматрају се нека од стратегија које најчешће сви играчи користе. Оне варирају од једноставнијих до компликованијих и битно их је размотрити због наредног поглавља која су инспирисана њима.

На слици 2 описана је ситуација када корисник може са талона покупити више од једне групе карата. Пошто систем бодовања налаже да корисник треба да сакупи што више карата, поготово штихова, то је сасвим јасно да је скоро увек оптимално узети све групе које је могуће однети тренутном картом. Изузетак за ово правило биће описан мало касније, а може га направити само играч који памти карте које су до сада прошле.

Наредна слика (слика 3) представља случај где корисник може узети уз своју карту из руке (даму) или жандара и аса или аса и двојку. Уколико се одлучи за прву варијанту, играч свом скору доприноси 3 поена, док у другом случају доноси само 2. Јасно је да ће се играч увек одлучити да одигра потез који му тренутно доноси више поена.

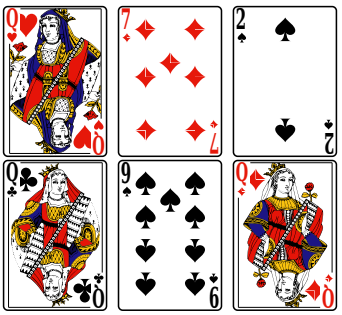
Још једна интересантна ситуација која се често догађа јесте она у којој играч не може носити ништа и треба да се одлучи коју ће карту из руке ставити на талон. Слика 5 илуструје овај пример. На талону се налазе краљ, краљица и десетка. Играч у руци има седмицу, аса, деветку, жандара, двојку и тројку. У случају да на талон избаци аса, играч пружа противнику могућност да уколико има краља покупи три штиха или уколико има аса покупи два. Избацивање аса у раној фази руке је обично лоша стратегија јер пружа више могућности противнику да покупи штих. Уколико играч избаци жандара, то такође оставља могућност да противник покупи два штиха уколико и он има жандара. Избацивање мале двојке оставља могућност да противник има жандара и покупи два штиха или обичну двојку и покупи један штих. Ако се играч одлучи да избаци тројку, то оставља могућност да противник има краљицу и покупи 10, 3 и Q, односно три штиха (рачунајући и његову карту из руке). Избацивање деветке и седмице не оставља никакву додатну могућност противнику да покупи штих и у овом случају делује као најбољи избор.

Још један битан аспект у стратегији игре јесте памћење досадашњих карата. Претходни пример бавио се разматрањем стратегије на основу минимизације противникових могућности. Слика 6 објашњава један случај када је памћење досадашњих карата помаже у одлучивању о тренутном потезу. Једна сигурна опција за играча је да једном својом дамом из руке покупи даму са талона и освоји два поена. Међутим, уколико је играч праћењем досадашњих карата установио да је једна дама већ прошла, тада је боља стратегија да најпре избаци једну своју даму из руке на талон, а тек у наредном потезу преосталом дамом из руке покупи две даме са талона. Памћење карата може бити корисно и у многим другим ситуаци-



Слика 5: *На талону се налазе краљ, краљица и десетка. Играч у својој руци нема ниједну карту којом може носити било шта са талона и треба да се одлучи коју ће карту одиграти.*

јама и обично играчи који памте досадашње карте имају знатну предност у односу на оне који не памте.



Слика 6: *Ако корисник зна да је до сада прошла већ једна дама, тада може бити сигуран да противник у својој руци нема карту којом може покупити даму са талона. У овом случају оптимално му је да најпре избаци једну даму на талон, а тек у наредном потезу покупи обе даме одједном (и трећу из руке).*

3.2 Похлепни алгоритам

3.2.1 Опис парадигме

Похлепни алгоритми представљају парадигму у којој се у сваком тренутку бира опција која даје тренутно најбољи исход. Основна карактеристика ових алгоритама јесте да су обично брзи јер у обзир узимају само тренутно најбољу опцију. Међутим, често је случај да похлепни алгоритми не дају оптимално решење, о чему ће касније бити речи.

Пример похлепног алгоритма јесте такозвани проблем избора активности [11]. Овај проблем се може представити једном свакодневном ситуацијом. Нека је дат амфитеатар и списак часова које тог дана треба одржати. Сваки час има своје време почетка и време завршетка и потребно је сместити што већи број часова у амфитеатар тако да се никоја два часа не поклапају. Још један услов је да су часови задати сортирани монотоно растуће по временима завршетка, односно:

$$f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_{n-1} \leq f_n$$

Овај услов се лако може постићи сортирањем.

Проблем 1. *Дат је скуп часова S, где сваки час има време почетка s_i и време завршетка f_i, где је $0 \leq s_i < f_i < \infty$. Ако се активност a_i изабере, тада онда траје у интервалу $[s_i, f_i)$. Часови a_i и a_j се не преклапају ако важи да је $s_i \geq f_j$ или $s_j \geq f_i$. Потребно је изабрати скуп активности највеће кардиналности у коме се никоје две активности не преклапају.*

Нека су часови дати као у табели 1. У овом примеру могуће је примера ради изабрати часове са редним бројевима 3, 9 и 11, за које је јасно да задовољавају услов да се никоја два не преклапају. Такође је лако проверити да се за скуп 1, 4, 8 и 11 не дешава да се нека два преклапају. Још један скуп за који то важи јесте 2, 4, 9 и 11. Испоставља се да последња два избора представљају оптимално решење проблема, односно максималан број часова које је могуће сместити у амфитеатар.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	7	9	6	11	9	11	12	14	14	8	18
f_i	10	11	12	13	15	15	16	17	18	20	22

Табела 1: *Пример листе активности, где је i индекс активности, s_i време њеног почетка, а f_i време завршетка.*

Интуитивно је бирати часове који остављају највише простора осталим часовима. Од тако изабраних часова мора постојати један који се најраније

завршио. Да би било простора за што више других часова, најбоље је да се први час заврши што пре, односно за први час је оптимално изабрати час 1. Посматрањем преосталих часова лако се елиминишу сви они часови који почињу пре s_1. Нека је скуп преосталих часова $S_k = \{a_i \in S : s_i \geq f_k\}$. Од свих часова из скупа S_k изабере се онај који се најраније завршава. Објашњење интуиције иза овог избора слично је објашњењу избора првог часа - бира се час који оставља највише простора за преостале. Остало је још само показати да је овакав избор оптималан.

Теорема 1. *За дати непразан потпроблем S_k, активност a_m из S_k која има најмање време завршетка припада бар једном решењу са највећим бројем активности које се међусобно не преклапају.*

Доказ. Нека је A_k скуп активности из S_k који је највећи могућ и у коме се никоје две активности међусобно не преклапају. Нека је a_j активност из A_K која има најраније време завршетка. Ако је $a_j = a_m$ доказ је завршен јер a_m припада неком највећем скупу непреклапајућих активности из S_k. Ако је $a_j \neq a_m$, потребно је посматрати скуп $A'_k = A_k \setminus \{a_j\} \cup \{a_m\}$, односно скуп A_k где се уместо a_j налази a_m. Активности у A'_k се не преклапају јер су активности у A_k, a_j је активност са најмањим временом завршетка у A_k и из поставке теореме важи да је $f_m \leq f_j$. Пошто скупови A_k и A'_k имају исту кардиналност, таде је A'_k такође највећи скуп непреклапајућих догађаја и у њему се налази a_m. □

Овиме је показано да похлепно решење води до оптималног решења проблема избора активности. Потребно је напоменути да похлепни алгоритми не доводе увек до оптималног решења, али се доста често користе за хеуристике које доводе до решења које је јако блиско оптималном.

3.2.2 Похлепна стратегија за таблић

У претходном поглављу било је речи о стратегијама које људи користе док играју таблић. Све стратегије које не рачунају бројење карата коришћене су као идеја за похлепни алгоритам чији опис следи. Претпоставка је да су улаз у алгоритам тренутне карте играча и изглед талона, а излаз треба да буде једна карта из руке као и све карте са талона које играч носи, ако таквих има.

Алгоритам се сматра похлепним јер узима решење које је оптимално само у оквиру тренутног потеза. За сваку карту из руке проба се сакупити што већи број штихова са талона. Бира се она карта из руке која носи највећи могући број штихова. У случају да постоји више таквих карата, бира се произвољна.

Предности овог алгоритма јесу брзина извршавања и једноставност имплементације. Псеудокод похлепног бота дат је у алгоритму 1, док се детаљна имплементација у C++језику може наћи у додатку А. Карте у руци и карте на талону представљене су помоћу низова *ruka* и *tabla*, редом. Први део алгоритма броји колико асова постоји на табли. Овај број је неопходан јер се сваки ас може посматрати као 1 или као 11, те је број различитих могућности њихових вредности 2^{broj_asova}. За сваку карту из руке и сваки могући распоред вредности асова испробавају се могућности купљења карата са табле. Најједноставнији начин за проверу шта се највише може сакупити са табле јесте испробавање свих могућих подскупова са табле. Уколико је сума неког подскупа дељива са вредношћу карте из руке, а да притом ниједна карта из скупа није већа од карте из руке, тада је скуп могуће покупити том картом. Јасно је да је број свих подскупова карата са табле управо $2^{tabla.len}$, где је *tabla.len* дужина низа *tabla*. У конкретној имплементацији су коришћене битовне операције за генерисање свих могућих подскупова, јер величина табле односно никада не може премашити број бита у регистру целобројног типа (у имплементацији је коришћен 32-битни тип због једноставности).

Алгоритам 1 Похлепни алгоритам

```
function PohlepniAlgoritam(tabla, ruka)
    najbolji_skor ← (0, 0)
    najbolja_maska ← 0
    najbolji_indeks ← 0
    broj_asova ← 0
    for i ∈ {0, 1, 2, ... tabla.len − 1} do
        if tabla[i] = 1 then
            broj_asova ← broj_asova + 1
    for all karta ∈ ruka do
        for maska_asova ∈ {0, 1, 2, ... 2^(broj_asova−1)} do
            for maska_table ∈ {0, 1, 2, ... 2^(tabla.len−1)} do
                karta ← ruka[i]
                skor ← ProbajPotez(karta, maska_asova, maska_table, ruka, tabla)
                if skor > najbolji_skor then
                    najbolji_skor ← skor
                    najbolji_indeks ← i
                    najbolja_maska ← maska_table
    OdigrajPotez(ruka[najbolij_indeks], najbolja_maska, ruka, tabla)
```

Псеудокодови за рачунање скора за неки потез, као и за играње потеза представљени су у алгоритмима 2 и 3, редом. Алгоритам 1 проверава најп-

ре да ли од изабраних карти са талона нека има вредност већу од карте из руке. У том случају пријављује се да је немогуће покупити тренутни скуп карти. У исто време рачуна се и сума свих изабраних карти због касније провере да ли је та сума дељива тренутном картом. Алгоритам за играње потеза најпре обрише карту из низа који представља играчеву руку, а затим обрише све карте са талона које је играч изабрао. Оба алгоритма су линеарна по броју карата на талону.

Алгоритам 2 Рачунање броја поена за тренутни потез

```
function ProbajPotez(karta, maska_asova, maska_table, ruka, tabla)
    skor ← (0, 0)                          ▷ број штихова и број карата, редом
    suma ← 0
    broj_asova ← 0
    for i ∈ {0, 1, 2, ... tabla.len − 1} do
        if ProveriBit(maska_table, i) = FALSE then
            continue
        vrednost ← tabla[i]
        if vrednost = 11 then
            if ProveriBit(maska_table, i) then
                vrednost ← 1
            broj_asova ← broj_asova + 1
        if vrednost > karta then
            return IMPOSSIBLE
        suma ← suma + vrednost
        if vrednost = 1 or vrednost ≥ 10 then
            skor.stihovi ← skor.stihovi + 1
    if suma mod karta ≠ 0 then
        return IMPOSSIBLE
    if karta = 1 or karta ≥ 10 then
        skor.stihovi ← skor.stihovi + 1
    if BrojBita(maska_table) = table.len then
        skor.stihovi ← skor.stihovi + 1
    return skor
```

Укупна сложеност похлепног алгоритма може се изразити као

$$O(2^{table.len} \cdot 2^{broj\_asova} \cdot ruka.len \cdot tabla.len)$$

, али пошто величина руке не премашује 6, а број асова не премашује 4, израз се може упростити на само

$$O(2^{table.len} \cdot tabla.len)$$

.

Алгоритам 3 Играње потеза са изабраном картом из руке и картама са талона

```
function OdigrajPotez(karta, maska_table, ruka, tabla)
    Obrisi(ruka, karta)
    nova_tabla ← []
    for i ∈ {1, 2, 3, ... tabla.len − 1} do
        if ProveriBit(maska_table, i) = FALSE then
            Dodaj(nova_tabla, tabla[i])
        if maska = 0 then
            nova_tabla ← karta
    tabla ← nova_tabla
```

3.3 Бектрек алгоритам

3.3.1 Опис парадигме

Бектрек парадигму најлакше је објаснити кроз неколико примера. Два проблема која следе описују све главне карактеристике бектрек алгоритама.

Проблем 2. *Нека је дата шаховска табла димензија $N \times N$. Потребно је поставити N краљица на таблу тако да се оне међусобно не нападају. За две краљице каже се да се нападају уколико се налазе у истом реду, истој колони или на истој дијагонали.*

Размотримо број комбинација које треба испитати да би се дошло до решења. Најпре, треба приметити да се у сваком реду сме налазити тачно једна краљица. Број стања које треба испитати је N^N, јер се краљица у једном реду може поставити на N начина, а има N редова.

Простор претраге се може још сузити. Краљица која се налази у првом реду може се произвољно поставити на N начина. Након што се прва краљица постави у неку од колона, тада се ниједна краљица не може поставити у ту колону. Ово оставља могућност да се краљица у другом реду може поставити у неку од преосталих $N-1$ колона. За краљицу у трећем реду остаје $N-2$ слободне колоне, итд. Долази се до $N \cdot (N-1) \cdot (N-2) \cdot 3 \cdot 2 \cdot 1 = N!$ стања које треба претражити.

Асимптотска сложеност алгоритма се не може спустити испод овога, али постоји начин да се током извршавања нека стања елиминишу. Алгоритам 4 рекурзивно тражи све валидне рапсореде краљица, успут водећи рачуна и о заузетости дијагонала на којима се покуша ставити краљица. У пракси се показује да се оваквим смањивањем могућих стања претрага знатно убрза.

Други проблем који се може решити бектреком је бојење графа помоћу три боје [14]. Формална поставка задатка наведена је као проблем 3.

Проблем 3. *Дат је граф са n чворова. Обојити сваки чвор графа тачно једном од три задате боје, тако да не постоји ивица графа која спаја два истобојна чвора.*

Пошто је сваки чвор могуће обојити на три различита начина, јасно је да је величина простора претраге 3^n. Претрага се може убрзати елиминацијом неких стања која имају бар једну ивицу са истобојним чворовима. Алгоритам 5 описује бектрек стратегију за бојење графа. Изабере се произвољан чвор v који није у скупу обојених чворова. За сваку од три могуће боје испроба се да ли је она адекватна за чвор v. Боја је адекватна ако

Алгоритам 4 Бектрек алгоритам за распоређивање краљица на шаховску таблу

```
function KRALJICE(red)
    if red = N then
        ISPISI(tabla)                    ▷ Тренутни распоред је задовољавајући
    for i ∈ {0, 1, 2, ... N − 1} do
        if ZAUZETAKOLONA(i) then
            continue
        if ZAUZETAGLAVNADIJAGONALA(red, j) then
            continue
        if ZAUZETASPOREDNADIJAGONALA(red, j) then
            continue
        ZAUZMIKOLONU(j)
        ZAUZMIGLAVNUDIJAGONALU(red, j)
        ZAUZMISPOREDNUDIJAGONALU(red, j)
        tabla[red] ← j
        KRALJICE(red + 1)
        OSLOBODIKOLONU(j)
        OSLOBODIGLAVNUDIJAGONALU(red, j)
        OSLOBODISPOREDNUDIJAGONALU(red, j)
```

ниједан сусед чвора v није обојен у ту боју. Уколико је боја адекватна претрага се рекурзивно наставља даље. Претрага је готова када скуп обојених чворова постане једнак скупу свих чворова графа, односно $U = V$.

Иако је у оба наведена примера асимптотска сложеност иста као и код најједноставније претраге, стратегија у пракси знатно убрзава решење.

3.3.2 Бектрек стратегија за таблић

У претходном поглављу описан је похлепни алгоритам који бира најбољу могућност посматрајући само тренутно стање руке и талона. Бектрек стратегија посматра неколико потеза унапред, али је то могуће само уз додатне информације о игри (које играч у стандардној игри нема).

Додатна информација о игри коју је најлакше искористити јесте садржај руке противника. Наиме, ако је играчу на почетку познат садржај руке противника и садржај талона, бектрек претрагом се може испробати сваки могући редослед потеза (има их 6!) те се изабрати онај који доноси најбољи скор. Да би се одредио комлетан исход, потребно је знати стратегију којом противник игра.

Алгоритам 6 доста је сличан похлепном алгоритму из претходног поглавља. Наиме у сваком кораку се испробавају све могуће карте и сви

Алгоритам 5 Бектрек алгоритам за бојење графа у три боје

```
function BOJENJE(boja, G, U)  ▷ G представља граф који треба обојити
                              ▷ U је скуп обојених чворова
    if U = G.V then
        for i ← 1 → N do
            ISPISI(boja[i])
        return
    for v ∈ {0, 1, 2, ... N − 1} do
        if v ∉ U then
            break
    for C ∈ {0, 1, 2} do
        moguce_bojenje ← TRUE
        for u ∈ adj[v] do
            if boja[u] = C then
                moguce_bojenje ← FALSE
                break
        if moguce_bojenje then
            boja[v] ← C
            U ← U ∪ {v}
            BOJENJE(boja, G, U)
```

могући скупови карата са талона, идентично као у похлепном алгоритму. Разлика је у томе што се у обзир још рачунају и противников потез, као и рекурзивни позив бектрек алгоритма након противниковог потеза. Скор за сваки потез рачуна се као сума скора који се тада оствари као и најбољег могућег резултата у свим наредним потезима. Детаљна имплементација у језику C++може се наћи у додатку А.

Као што је већ речено, број могућих начина да се одигра 6 карата из руке је 6! па је сложеност бектрек алгоритма

$$O(6! \cdot 2^{table.len} \cdot tabla.len),$$

а пошто је 6! константа може се занемарити. Добија се да је асимптотска временска сложеност бектрек алгоритма иста као и за похлепни алгоритам:

$$O(2^{table.len} \cdot tabla.len),$$

Иако су асимптотске сложености исте, у пракси је очекивано да ће бектрек алгоритам због свих рекурзивних позива бити знатно спорији. О брзинама имплементације бектрек алгоритма биће речи у наредном поглављу.

Претходно описан алгоритам посматра само тренутне руке играча и противника. Остаје још размотрити бектрек алгоритам који посматра све

Алгоритам 6 Бектрек алгоритам за играње таблића

```
function BEKTREK(ruka, protivnikova_ruka, tabla)
    if ruka.empty then
        return (0, 0)
    najbolji_skor ← (0, 0)
    najbolja_maska ← 0
    najbolji_indeks ← 0
    broj_asova ← 0
    for i ∈ {0, 1, 2, ... tabla.len − 1} do
        if tabla[i] = 1 then
            broj_asova ← broj_asova + 1
    for all karta ∈ ruka do
        for maska_asova ∈ {0, 1, 2, ... 2^(broj_asova−1)} do
            for maska_table ∈ {0, 1, 2, ... 2^(tabla.len−1)} do
                karta ← ruka[i]
                skor ← PROBAJPOTEZ(karta, maska_asova, maska_table, ruka, tabla)
                n_prot_ruka ← protivnikova_ruka
                n_tabla ← tabla
                PROTIVNIKOVASTRATEGIJA(n_tabla, n_prot_ruka)
                n_skor ←BEKTREK(ruka \ karta, n_prot_ruka, n_tabla)
                skor ← skor + n_skor
                if skor > najbolji_skor then
                    najbolji_skor ← skor
                    najbolji_indeks ← i
                    najbolja_maska ← maska_table
    ODIGRAJPOTEZ(ruka[najbolij_indeks], najbolja_maska, ruka, tabla)
    return najbolij_skor
```

могуће исходе игре уз познавање целог шпила. Алгоритам је суштински исти, уз додатак да треба да води рачуна о дељењу карата играчима у тренуцима када се то иначе дешава у игри. Број начина на који играч може одиграти једну партију таблића је $(6!)^4 = 268738560000$, јер се редослед потеза може мењати само унутар руке, а у партији има 4 руке. Овај алгоритам је имплементиран, али се испоставило да је простор претраживања сувише велики чак и уз бројне оптимизације.

3.4 Генетски алгоритам

3.4.1 О генетским алгоритмима

Генетски алгоритми представљају хеуристику која се често користи у оптимизационим проблемима. Обично је потребно наћи локални минимум или максимум неке фунције више променљивих где је превише скупо користити постојеће нумеричке методе. Основни кораци алгоритма настали су по узору на генетске процесе у природи.

На почетку се генерише група насумичних решења проблема (популација) која касније еволуирају ка бољем решењу. Свако решење из тог скупа назива се јединка и представља један члан домена функције које треба оптимизовати. Јединка се обично представља бинарним кодом или једноставно као низ реалних бројева из домена функције.

Популација се итеративно побошљава и у свакој итерацији настаје нова генерација популације. Први корак у свакој итерацији је рачунање функције прилагођености која представља колико је израчуната вредност функције близу оптималној. У наредном кораку врши се селекција, односно одабир јединки које имају најбоље вредности фитнес функције. Обично се селекција врши избацивањем дела популације који има прилагођеност мању од неке вредности или једноставно избацивањем неког фиксираног броја јединки. Након селекције обично следе генетске операције мутирања и укрштања. Током мутирања сваки члан популације или насумично изабрани чланови популације мењају свој генетски материјал са предефинисаном вероватноћом. Процес укрштања састоји се од избора две јединке које размењују генетски материјал. У случају да су гени представљени као реални бројеви, укрштање се најчешће врши као размена вредности или као узимање тежинске средине два броја. Након што се генетске операције заврше, може се наставити даље са наредном итерацијом. Процес еволуције се завршава обично након одређеног броја генерација или након што фитнес функција најприлагођеније јединке пређе неки праг.

3.4.2 Генетски алгоритам за играње таблића

4 Резултати

Симулиране су партије између два различита алгоритма и мерена су времена извршавања. Детаљан код који симулира партију може се наћи у додатку А. Решења су тестирана на рачунару са четворојезгарним процесором Интел Кор И7 који ради на 2,5 GHz.

На симулираних 50 партија, бектрек алгоритам је био бољи у 35 партија, док је похлепни алгоритам био бољи у 15. Просечно време извршавања по партији бектрек алгоритма била је 29,23 секунде, док се похлепни алгоритам извршавао просечно мање од једне милисекунде по партији. Објашњење за партије које је освојио похлепни алгоритам јесте распоред карата. Наиме, након анализе игара утврђено је да је похлепни алгоритам у партијама које је освојио у свакој руци имао знатно већи број штихова и тиме имао предност у односу на бектрек алгоритам.

5 Закључак

Додаци

A Имплементације алгоритама

Фајл 1: *Главни програм*

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>

#include "utils.h"
#include "backtrack6.h"
#include "backtrack.h"
#include "greedy.h"

using namespace std;

int first_timer, second_timer;

pair<pair<int, int>, pair<int, int>> play_game(vector<int> deck
    ) {
    vector<int> first;
    vector<int> second;
    vector<int> table;
    int curr_player = 0;
    pair<pair<int, int>, pair<int, int>> score = make_pair(
        make_pair(0, 0), make_pair(0, 0));
    for (int i = 0; i < 4; i++) {
        table.push_back(deck.back());
        deck.pop_back();
    }
    int last_player_scored = 0;
    for (int move = 0; move < 48; move++) {
//        cout << first.size() << " " << second.size() << endl;
//        cout << "TABLE: ";
//        for (int i = 0; i < table.size(); i++) {
//            cout << table[i] << " ";
//        } cout << endl;
        if (curr_player == 0 && first.empty()) {
            deal(deck, first, second);
        }

        pair<int, int> tmp_score;

        if (curr_player == 0) {
            clock_t begin = clock();
            tmp_score = backtrack6(table, first, second);
            first_timer += clock() - begin;

            score.first.first += tmp_score.first;
```

```
            score.first.second += tmp_score.second;
            if (tmp_score > make_pair(0, 0)) {
                last_player_scored = 0;
            }
        } else {
            clock_t begin = clock();
            tmp_score = greedy_simple(table, second);
            second_timer += clock() - begin;
            score.second.first += tmp_score.first;
            score.second.second += tmp_score.second;
            if (tmp_score > make_pair(0, 0)) {
                last_player_scored = 1;
            }
        }
        curr_player ^= 1;

    }

    // zadnja karta
    int remaining_valuables = 0;
    for (int i = 0; i < table.size(); i++) {
        if (table[i] >= 10) {
            remaining_valuables++;
        }
    }
    if (last_player_scored == 0) {
        score.first.first += remaining_valuables;
        score.first.second += table.size();
    } else {
        score.second.first += remaining_valuables;
        score.second.second += table.size();
    }
    return score;
}

int main() {
    int games_played = 50;
    int first_wins = 0;
    int second_wins = 0;
    int draws = 0;
    for (int game = 0; game < games_played; game++) {
        vector<int> deck;
        for (int i = 2; i <= 14; i++) {
            for (int j = 0; j < 4; j++) {
                deck.push_back(i);
            }
        }
        random_shuffle(deck.begin(), deck.end());
        pair<pair<int, int>, pair<int, int>> score = play_game(
            deck);
```

```
        if (score.first > score.second) {
            cout << "First wins!" << endl;
            first_wins++;
        } else if (score.second > score.first) {
            cout << "Second wins!" << endl;
            second_wins++;
        } else {
            cout << "Draw" << endl;
            draws++;
        }
        printf("(%d, %d) - (%d, %d)\n", score.first.first,
           score.first.second, score.second.first, score.second
           .second);
    }
    cout << "Score: " << first_wins << " - " << second_wins <<
       endl;
    cout << (double) first_timer / games_played /
       CLOCKS_PER_SEC << " "
         << (double) second_timer / games_played /
            CLOCKS_PER_SEC << endl;
    return 0;
}
```

Фајл 2: *Хедер фајл за бектрек алгоритам*

```
#ifndef TABLIC_BACKTRACK_H
#define TABLIC_BACKTRACK_H

std::pair<int, int> backtrack(std::vector<int> &table,
                              std::vector<int> &hand,
                              std::vector<int> &other,
                              std::vector<int> &deck,
                              bool i_am_the_first_player);

#endif //TABLIC_BACKTRACK_H
```

Фајл 3: *Имплементација бектрек алгоритма*

```
#include <vector>
#include <iostream>
#include <map>

using namespace std;

#include "greedy.h"
#include "utils.h"

map<tuple<vector<int>, vector<int>, vector<int>>, tuple<int,
    int, pair<int, int>>> memo6;

pair<int, int> backtrack6_helper(vector<int> &table,
                                 vector<int> &hand,
                                 vector<int> &other,
                                 int depth = 0) {
    if (hand.empty()) {
        return make_pair(0, 0);
    }
    auto tpl = make_tuple(table, hand, other);
    if (memo6.find(tpl) != memo6.end()) {
        auto best_move = memo6[tpl];
        int idx = get<0>(best_move);
        int mask = get<1>(best_move);
        pair<int, int> score = get<2>(best_move);
        do_move(hand[idx], mask, hand, table);
        return score;
    }

    pair<int, int> best_score = make_pair(0, 0);
    pair<int, int> best_curr_score = make_pair(0, 0);
    int best_mask = 0, best_idx = 0;
    int aces = count(table.begin(), table.end(), 11);
    for (int i = 0; i < hand.size(); i++) {
        for (int aces_mask = 0; aces_mask < (1 << aces);
            aces_mask++) {
            // table size can possibly grow big
            for (int mask = 0; mask < (1LL << table.size());
                mask++) {
                int card = hand[i];
                pair<int, int> tmp_score = try_move(card, mask,
                    aces_mask, hand, table);

                vector<int> new_table = table;
                vector<int> new_hand = hand;
                vector<int> new_other = other;
                do_move(card, mask, new_hand, new_table);
```

```
                if (!new_other.empty()) {
                    greedy_simple(new_table, new_other);
                }

                pair<int, int> global_score = backtrack6_helper
                    (new_table, new_hand, new_other, depth + 1);
                global_score.first += tmp_score.first;
                global_score.second += tmp_score.second;
                if (global_score > best_score) {
                    best_score = global_score;
                    best_curr_score = tmp_score;
                    best_idx = i;
                    best_mask = mask;
                }
            }
        }
    }

    do_move(hand[best_idx], best_mask, hand, table);
    memo6[tpl] = make_tuple(best_idx, best_mask,
        best_curr_score);
    return best_curr_score;
}

pair<int, int> backtrack6(vector<int> &table,
                          vector<int> &hand,
                          vector<int> &other) {

    pair<int, int> score = backtrack6_helper(table, hand, other
        );
//    cout << score.first << " " << score.second << endl;
    return score;
}
```

Фајл 4: *Хедер фајл за похлепни алгоритам*

```
#ifndef TABLIC_GREEDY_H
#define TABLIC_GREEDY_H

std::pair<int, int> greedy_simple(std::vector<int> &table, std
    ::vector<int> &hand);

#endif //TABLIC_GREEDY_H
```

Фајл 5: *Имплементација похлепног алгоритма*

```
#include <vector>
#include <iostream>

#include "utils.h"

using namespace std;

pair<int, int> greedy_simple(vector<int> &table, vector<int> &
    hand) {
    pair<int, int> best_score = make_pair(0, 0);
    int best_mask = 0, best_idx = 0;
    int aces = count(table.begin(), table.end(), 11);
    for (int i = 0; i < hand.size(); i++) {
        for (int aces_mask = 0; aces_mask < (1 << aces);
            aces_mask++) {
            // table size can grow big
            for (int mask = 0; mask < (1LL << table.size());
                mask++) {
                int card = hand[i];
                pair<int, int> tmp_score = try_move(card, mask,
                    aces_mask, hand, table);
                if (tmp_score > best_score) {
                    best_score = tmp_score;
                    best_idx = i;
                    best_mask = mask;
                }
            }
        }
    }
//    cout << "GRDY _OUT " << endl;

    do_move(hand[best_idx], best_mask, hand, table);
//    cout << "GRDY _OUT " << endl;
    return best_score;
}
```

Фајл 6: *Тренирање генетског алгоритма*

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>

#include "utils.h"
#include "greedy.h"
#include "genetic.h"

using namespace std;

int first_timer, second_timer;

int play_game(vector<int> deck, tuple<double, double, double,
    double> genome) {
    vector<int> first;
    vector<int> second;
    vector<int> table;
    int curr_player = 0;
    pair<pair<int, int>, pair<int, int>> score = make_pair(
        make_pair(0, 0), make_pair(0, 0));
    for (int i = 0; i < 4; i++) {
        table.push_back(deck.back());
        deck.pop_back();
    }
    int last_player_scored = 0;
    for (int move = 0; move < 48; move++) {
        if (curr_player == 0 && first.empty()) {
            deal(deck, first, second);
        }

        pair<int, int> tmp_score;

        if (curr_player == 0) {
            clock_t begin = clock();
            tmp_score = greedy_simple(table, first);
            first_timer += clock() - begin;

            score.first.first += tmp_score.first;
            score.first.second += tmp_score.second;
            if (tmp_score > make_pair(0, 0)) {
                last_player_scored = 0;
            }
        } else {
            clock_t begin = clock();
            tmp_score = genetic(table, second, first, genome);
            second_timer += clock() - begin;
```

```
            score.second.first += tmp_score.first;
            score.second.second += tmp_score.second;
            if (tmp_score > make_pair(0, 0)) {
                last_player_scored = 1;
            }
        }
        curr_player ^= 1;

    }

    // zadnja karta
    int remaining_valuables = 0;
    for (int i = 0; i < table.size(); i++) {
        if (table[i] >= 10) {
            remaining_valuables++;
        }
    }
    if (last_player_scored == 0) {
        score.first.first += remaining_valuables;
        score.first.second += table.size();
    } else {
        score.second.first += remaining_valuables;
        score.second.second += table.size();
    }
    int other = score.first.first;
    int my = score.second.first;
    if (score.first.second > score.second.second) {
        other++;
    } else if (score.second.second > score.first.second) {
        my++;
    }
    return my - other;
}

int tournament(tuple<double, double, double, double> genome) {
    int score = 0;
    for (int i = 0; i < 30; i++) {
        int tmp = play_game(generate_deck(), genome);
        if (tmp > 0) {
            score++;
        }
        if (tmp < 0) {
            score--;
        }
    }
    //cout << score << endl;
    return score;
}

void crossover(tuple<double, double, double, double> &a, tuple<
```

```
    double, double, double, double> &b) {
    get<0>(b) = (get<0>(a) + get<0>(b)) / 2;
    get<1>(b) = (get<1>(a) + get<1>(b)) / 2;
    get<2>(b) = (get<2>(a) + get<2>(b)) / 2;
    get<3>(b) = (get<3>(a) + get<3>(b)) / 2;
}

double rand01() {
    return (double) rand() / RAND_MAX;
}

int main() {
    vector<pair<int, tuple<double, double, double, double>>>
       population;
    int population_size = 50;
    for (int i = 0; i < population_size; i++) {
        auto tpl = make_tuple(rand01(), rand01(), rand01(),
           rand01());
        population.push_back(make_pair(0, tpl));
    }

    int iterations = 100;
    for (int _it = 0; _it < iterations; _it++) {
        for (int i = 0; i < population.size(); i++) {
            population[i].first = tournament(population[i].
               second);
        }
        sort(population.begin(), population.end());
        reverse(population.begin(), population.end());
        cout << "BEST SCORE: " << tournament(population[0].
           second) << endl;

        for (int i = population_size / 3; i < population.size()
           ; i++) {
            if (rand01() < 0.05) {
                population[i] = make_pair(0, make_tuple(rand01
                   (), rand01(), rand01(), rand01()));
            }
        }
        for (int i = 0; i < population.size(); i++) {
            for (int j = i + population_size / 3; j <
               population.size(); j++) {
                crossover(population[i].second, population[j].
                   second);
            }
        }
        auto tpl = population[0].second;
        cout << get<0>(tpl) << " " << get<1>(tpl) << " " << get
           <2>(tpl) << " " << get<3>(tpl) << endl;
    }
```

```
    auto tpl = population[0].second;
    cout << get<0>(tpl) << " " << get<1>(tpl) << " " << get<2>(
       tpl) << " " << get<3>(tpl) << endl;
    return 0;
}
```

Фајл 7: *Хедер фајл за похлепни алгоритам*

```
#ifndef TABLIC_GENETIC_H
#define TABLIC_GENETIC_H

std::pair<int, int> genetic(std::vector<int> &table,
                            std::vector<int> &hand,
                            std::vector<int> &other,
                            std::tuple<double, double, double,
                               double> genome);

#endif //TABLIC_GENETIC_H
```

Фајл 8: *Имплементација генетског алгоритма*

```
#include <iostream>
#include <vector>

#include "utils.h"
#include "greedy.h"

using namespace std;

pair<int, int> genetic(vector<int> &table,
                       vector<int> &hand,
                       vector<int> &other,
                       tuple<double, double, double, double>
                           genome) {
    double best_fit = -1e12;
    pair<int, int> best_score = make_pair(0, 0);
    int best_mask = 0, best_idx = 0;
    int aces = count(table.begin(), table.end(), 11);
    for (int i = 0; i < hand.size(); i++) {
        for (int aces_mask = 0; aces_mask < (1 << aces);
            aces_mask++) {
            // table size can grow big
            for (int mask = 0; mask < (1LL << table.size());
                mask++) {
                int card = hand[i];
                pair<int, int> my_score = try_move(card, mask,
                    aces_mask, hand, table);
                pair<int, int> other_score = make_pair(0, 0);
                vector<int> newtable = table;
                vector<int> newother = other;
                if (!other.empty()) {
                    other_score = greedy_simple(newtable,
                        newother);
                }
                double curr_fit = 0.0;
                curr_fit += get<0>(genome) * my_score.first;
                curr_fit += get<1>(genome) * my_score.second;
                curr_fit -= get<2>(genome) * other_score.first;
                curr_fit -= get<3>(genome) * other_score.second
                    ;
                if (curr_fit > best_fit) {
                    best_score = my_score;
                    best_fit = curr_fit;
                    best_idx = i;
                    best_mask = mask;
                }
            }
        }
```

```
    }

    do_move(hand[best_idx], best_mask, hand, table);
    return best_score;
};
```

Литература

[1] http://www.wopc.co.uk/, 2017

[2] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/recitation-videos/recitation-20-dynamic-programming-blackjack/, 2017

[3] https://github.com/emisaacson/Blackjack, 2015

[4] Ryan A. Dutsch *A risk-averse strategy for blackjack using fractional dynamic programming* Louisiana State University, Louisiana, 2003

[5] http://www.acbl.org/, 2017

[6] http://www.worldbridge.org/, 2017

[7] http://www.bridgeserbia.org/, 2017

[8] http://www.wsop.com/, 2017

[9] http://www.worldpokertour.com/, 2017

[10] http://www.igrajkarte.com/blog/post/tablic-pravila/, 2010

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms, 3rd edition.* MIT Press, Massachusetts, 2009.

[12] S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani *Algorithms.* Lecture notes.

[13] Jon Kleinberg, Eva Tardos *Algorithm Design, 1st edition.* Addison Wesley, Massachusetts, 2006.

[14] Udi Manber *Introduction to Algorithms - A Creative Approach.* Addison Wesley, Massachusetts, 1989.

[15] Scott McNeely *Ultimate Book of Card Games: The Comprehensive Guide to More than 350 Games.* Chronicle Books, California, 2009.

[16] Peter Norvig, Stuart J. Russell *Artificial Intelligence: A Modern Approach, 3rd edition* Prentice Hall, New Jersey, 2014

[17] http://en.cppreference.com/w/, 2017

[18] Melanie Mitchell *An Introduction to Genetic Algorithms* MIT Press, Massachusetts, 1998