Szakdolgozat

Miskolci Egyetem

Texas Hold'Em stratégiai vizsgálata webes környezetben

Készítette:

Laboda Dániel Balázs

Programtervező informatikus

Témavezető:

Dr. Földvári Attila József

Miskolc, 2022

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Laboda Dániel Balázs (H7PG8U) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: kombinatorika, webfejlesztés

A szakdolgozat címe: Texas Hold'Em stratégiai vizsgálata webes környezetben

A feladat részletezése:

A Texas Hold'Em napjaink legismertebb póker játéka. A dolgozat bemutatja a játék elemeit. Ezt követően egy olyan webalkalmazás kerül kifejtésre, amely a Texas Hold'Em póker játékban nyújt segítséget a felhasználónak. A koncepció az, hogy az alkalmazás a felhasználó lapjai alapján kiszámolja, hogy matematikailag érdemes-e megadnia az összeget vagy sem, ha a játékok száma tartana a végtelenbe, akkor nyerjen. Mindehhez frontend környezetnek a javascript egyik divatos könyvtára, a VueJS-t kerül felhasználásra, a backend a Node JS lesz, a stílusért és a weboldal külsejéért pedig a CSS felel.

Témavezető: Dr. Földvári Attila József (egyetemi adjunktus)

A feladat kiadásának ideje: 2021. szeptember 28.

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Laboda Dániel Balázs**; Neptun-kód: `H7PG8U` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Texas Hold'Em stratégiai vizsgálata webes környezetben* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

A póker a világ egyik legnépszerűbb és legismertebb kártyajátéka. 2021-ben a póker volt a harmadik legnépszerűbb kártyajáték, azon belül is a legtöbbet játszott, a Texas Hold'Em-nek a No Limit változata, vagy ahogy Dan Harrington pókervilágbajnok fogalmaz „a póker Cadillac-je" [5].

A pókerben közrejátszik a szerencse, így hivatalosan szerencsejátékként jegyezték be. Ennek ellenére, ha egy tapasztalt játékos leül játszani egy kezdő játékossal, akkor több játszmából, hosszú távon átlagosan a tapasztalt játékos fog nyerni. A tapasztalt játékosok remekül meg tudják figyelni az asztalnál zajló eseményeket, melyeket logikusan tudnak felhasználni, hiszen a póker az információ játéka. Ezeket az információkat az arckifejezésekből, gesztikulációból, a játékosok döntéseiből, emeléseinek méretéből, és sok más egyéb összetevő mellett, a lapokból számolt matematikai esélyekből nyerik, amely az egyik legfontosabb részlete a játéknak, ha sikeresek szeretnénk lenni.

Szakdolgozatomban ezeket az esélyeket vizsgálom egy játékos szempontjából anélkül, hogy a saját lapjaimon, és az asztalon lévő lapokon kívül bármilyen információ rendelkezésemre állna, tehát csak a matematikai háttérrel foglalkozom. A profi játékosok általában fejben ki tudják számolni az adott esélyeket különböző leegyszerűsített módszerekkel, viszont ezek csak közelítő eredményeket fognak adni, valamint csak egy számot kapnak, hogy hány százalék esélyük van. Szakdolgozatomban bemutatom, hogy hogy kaphatunk pontos esélyeket, amelyeket nemcsak sémákra illesztve számolunk, hanem minden lehetőséget külön vizsgálunk. Ezen kívül nem csak egyetlen számot, hanem több információt is láttatni szeretnék, amely elősegíti a játékos megfelelő döntését, hogy minél több esetben legyen eredményes.

Az alkalmazást webes környezetben valósítom meg, azon belül is a Vue JS keretrendszert használom. A stíluselemeket CSS-el hozom létre, a backend oldalon lévő szerver alkalmazást pedig a Node JS szolgáltatja. Ezeken kívül más keretrendszereket, könyvtárakat és kisegítő lehetőségeket is használok, melyeket részletezni fogok, továbbá bemutatom az alkalmazás létrejöttének lépéseit, az előforduló problémákat, ezekre adott megoldásaimat és magát az alkalmazást.

2. fejezet

Témaköri felvezetés

2.1. A póker alapjai

Szakdolgozatom könnyebb értelmezése érdekében, mindenképp tisztázni kell néhány, a pókerrel kapcsolatos alapfogalmat, valamint a játék menetét. Ebben a szakaszban ezeket ismertetem.

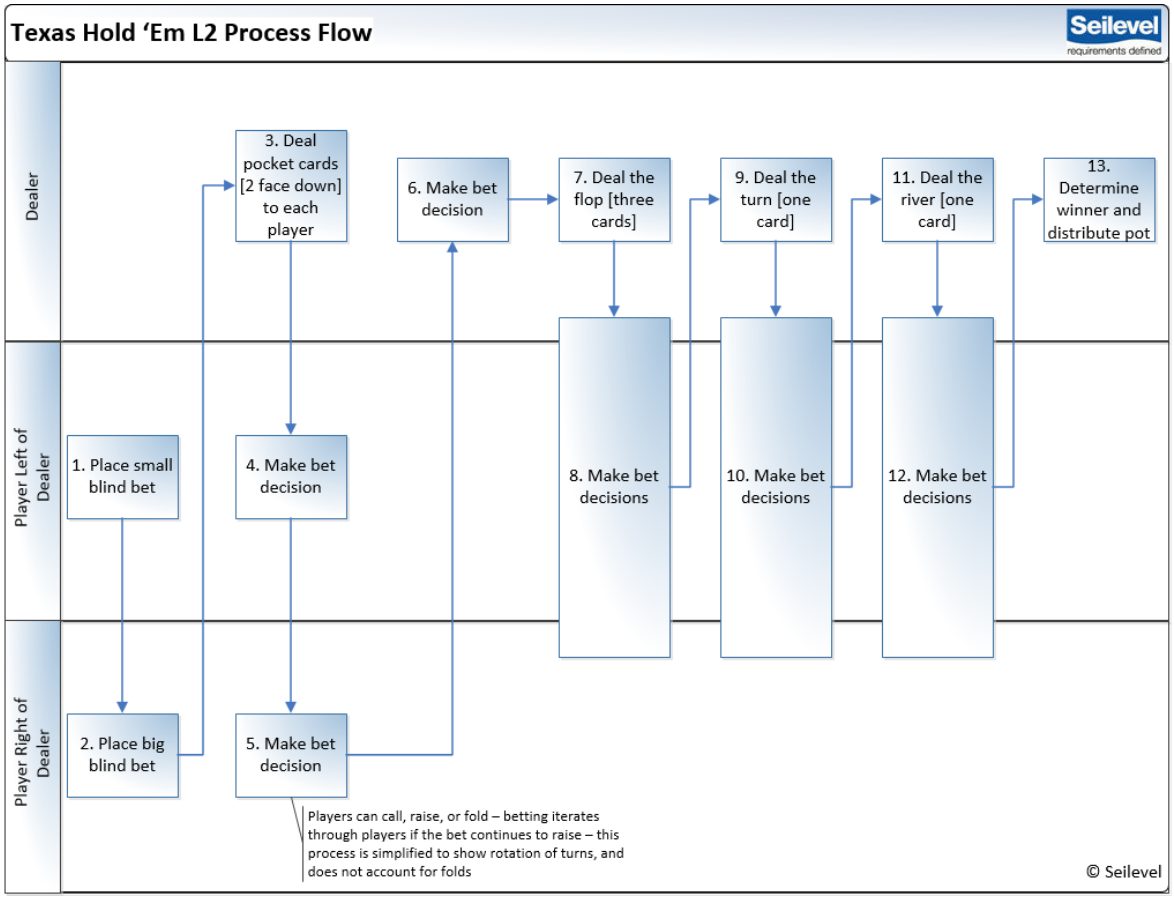
2.1.1. A játék menetének kifejtése

A játék menetét illetően próbálok csak a számunkra releváns részletekre koncentrálni. (A játékmenet áttekintését a 2.1. ábrán láthatjuk.) Egy játék több leosztásból áll. A játékosok száma eltérő lehet, általában 5 és 10 közötti létszámmal szokták játszani. Minden játékosnak kezdetben egyenlő összegű zsetonkészlete van, amivel játszhat. Minden leosztásnál van egy osztó, akit az első leosztásnál sorsolnak, a tőle bal oldalon ülő játékos a kis vak, az osztótól balra a második játékos pedig a nagy vak. A többi leosztásnál az osztó pozíció mindig az előző osztó bal oldalán ülő játékos lesz, valamint a kisvak és nagyvak pozíció is csúszik egyet. A játékot az óramutató járásával megegyező irányban játsszák.

Amint be van rakva a kis vak és a nagy vak, kezdődhet a leosztás. Az osztó a kis vaknak oszt először egy lapot, majd körbe mindenkinek, ezután megint egy lapot mindenkinek. A "beszédet" a nagy vak mellett ülő játékos kezdheti, vagyis ő nyilatkozhat először lapjairól. Alapvetően három választási lehetősége van, ahogy mindenkinek az asztalnál. Bedobja lapjait, beszáll, vagyis megadja az alaptétet, vagy pedig emel. Minden licitkör addig tart, amíg minden játékos befejezte a licitálást, megadta a tétet vagy dobta lapjait. Miután az utolsó licitkör lement, akkor az osztó egy lapot éget, vagyis letesz az asztalra fejjel lefelé, ami már nincs játékban. Ez után három lapot kirak egymás után, hogy mindenki lássa. Ekkor következik a második licitkör. Ebben az esetben már a kis vak kezdi a "beszélést", ha még játékban van. Ugyan az a három lehetősége van, majd ha ismét lement az utolsó licitkör, akkor még egy lap égetése, majd immáron csak egy lap asztalra helyezése következik. Megint jön egy licitkör, ahol ha játékban van, akkor ismét a kisvak kezd, amennyiben nincs, akkor a tőle bal oldalra ülő legközelebbi játékban lévő játékos. Ezt követően még egy égetés, és az utolsó felfordított lap az asztalra. Maradt még egy utolsó licitkör, és elérkeztünk a játék végéhez.

Abban az esetben nyer valaki, ha a bent maradt játékosok közül neki van a legerősebb lapja, vagy ha nem maradt más játékban csak egy valaki. A lent lévő 5 lapból és a kezünkben lévő 2-ből, összesen 5-öt választhatunk ki, így alakul ki a "kezünk".

Ezzel lement egy parti a játékból. A játékban valakinek gyarapodott a zsetonkészlete, valakinek pedig csökkent, de az is előfordulhat, hogy mindenki előtt ugyanannyi marad.



2.1. ábra. Folyamatábra a póker menetéről [8]

2.1.2. Alapfogalmak

Elengedhetetlen, hogy tisztázzunk néhány alapfogalmat. Ezek a következők [5]:

- Flop: Három kártyalap, ami egyszerre kerül az asztalra, színével fölfelé. Ezek a lapok a játékosok közös lapjai. A flop után egy újabb licitkör következik.
- Gomb vagy osztó (Button): A kis vak jobb oldalán ülő játékos. A flop után ő kerül utoljára sorra minden licitkörben. A gombot egy fehér korong jelzi, ami az óramutató járásával megegyező irányban körbejár az asztalon.
- Jó (Out): Olyan lap(ok), melyekkel kezünk nyerővé alakulna.
- Kezdeti bank (Initial pot): A vaktét és az esetleges alaptétek összege a licitálás megkezdése előtt.
- Kis vaktét (Small blind): Az osztó bal oldalán ülő játékos által a licitsorozat megkezdéseként kötelezően betett tét.
- Nagy vaktét (Big blins): A kis vak bal oldalán ülő játékos által kötelezően betett tét, a további akciók motiválására.

- Turn: A negyedik közös lap, ami színével fölfelé az asztal közepére kerül. A turn után újabb licitkör következik.
- River: Az ötödik és egyben utolsó közös lap, ami az asztal közepére kerül színével fölfelé. A river után az utolsó licitkör következik.
- Saját lapok (Hole cards): Az egyes játékosoknak a parti elején színével lefelé kiosztott két-két kártyalap. Ezeket a lapokat a többi játékos nem láthatja.
- Kéz: Egy játékos által, a lent lévő és a kezében lévő lapok közül kiválasztott öt lap.
- Nuts: A lehető legjobb kombinációval rendelkező játékos keze.
- Zsetonkészlet (Stack): Egy adott játékos előtt az asztalon lévő zsetonmennyiség.

2.2. Hasonló célú szoftverek áttekintése

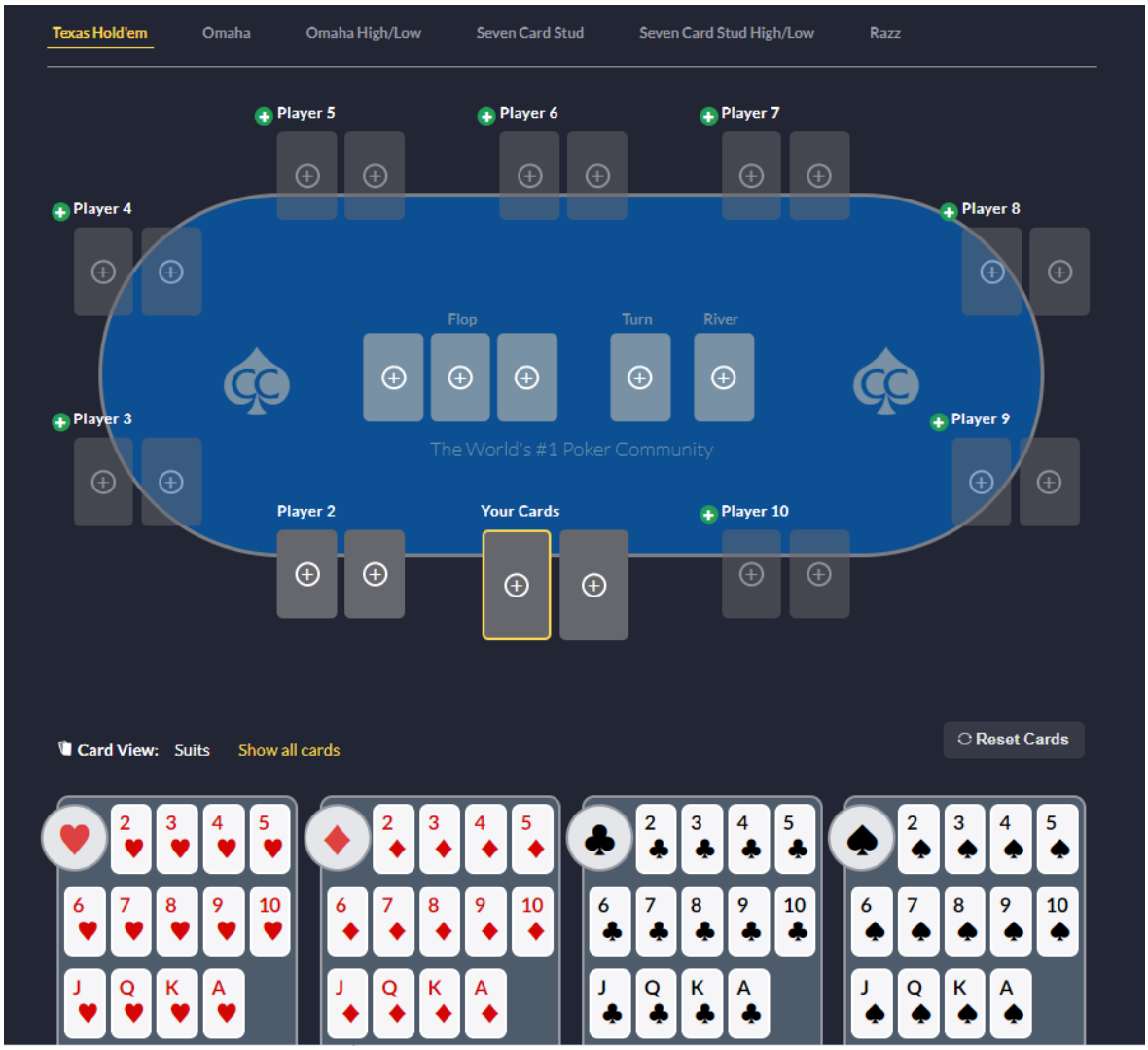
Mielőtt belekezdtem volna az alkalmazás elkészítésébe, piackutatást végeztem, milyen platformok/programok érhetők el, amelyek a témával foglalkoznak.

Nem kellett sokáig kutatni, hogy az elsőre rábukkanjak, hiszen a póker közvetítéseken láthatjuk, hogy ki van írva a játékosok nyerési esélye, egy-egy leosztásnál. Ott viszont a program ismeri a játékban lévő játékosok lapjait, valamint az asztalon lévő lapokat. Így könnyebb meghatározni melyik játékos nyer, vagy legalábbis kevesebb számítást igényel. Esetemben az alkalmazás csak a saját lapomat, valamint az asztalon lévőket ismeri.

Létezik egy technika a témával kapcsolatban, amivel a profi póker játékosok az esélyeiket számolják. A lényege csupán annyi, hogy kiszámolják a lehetséges out-okat, amivel javul a kezük, ezeket megszorozzák kettővel, majd annyival, ahány lapra még várunk. Flop esetén kettővel, turn esetén eggyel. Így kapnak egy számot, ami ha nagyobb, mint a banki esélyük, akkor matematikailag, vagy éppen gazdaságilag, kifizetődő tartani a tétet. Ezzel a megoldással szemben, az én elképzelésem alapján nem csak egy közelítő értéket fogunk kapni, hanem specifikusan minden lehetséges leosztásra egy pontos értéket kapunk, emellett még más adatokat is, melyek segítenek eldönteni a felhasználónak mi legyen a következő lépés.

Az Interneten két hasonló programot találtam a póker esélyek számítására. Mindkettő webalkalmazás és külalakra hasonlítanak az én elképzelésemre. Az elsőnél csak úgy lehet használni az alkalmazást, ha legalább két játékosnak megadjuk a lapjait. Ezzel a probléma ugyan csak az, mint a fent említett televíziós közvetítéseknél ismert megoldással. Ezt az alkalmazást a *chardchat.com* weboldalon találtam, ami a világ elsőszámú pókeres online közössége. Az alkalmazásról láthatunk egy képet a 2.2. ábrán.

A másik szoftver, ami talán a legjobban megközelíti a megoldásomat, a *pokernews.com* oldalán lévő webalkalmazás. Érdekesség, hogy ezt szintúgy egy óriás, pókerrel foglalkozó cég szponzorálta. A világ talán leghíresebb online póker platformja, a legnagyobb póker versenyek rendezője, a *PokerStars*. Ebben az esetben tudunk esélyeket számolni úgy is, hogy csak egy játékost választunk ki, csakúgy, mint az én megoldásomnál. Le tudunk rakni 1 és 9 között tetszőleges számú játékosokat, az alkalmazás pedig ezt figyelembe véve számolja az esélyeket. Ezen felül még további esélyeket is látunk, hogy hány százalék a valószínűsége annak, hogy párunk, drillünk, pókerünk stb.



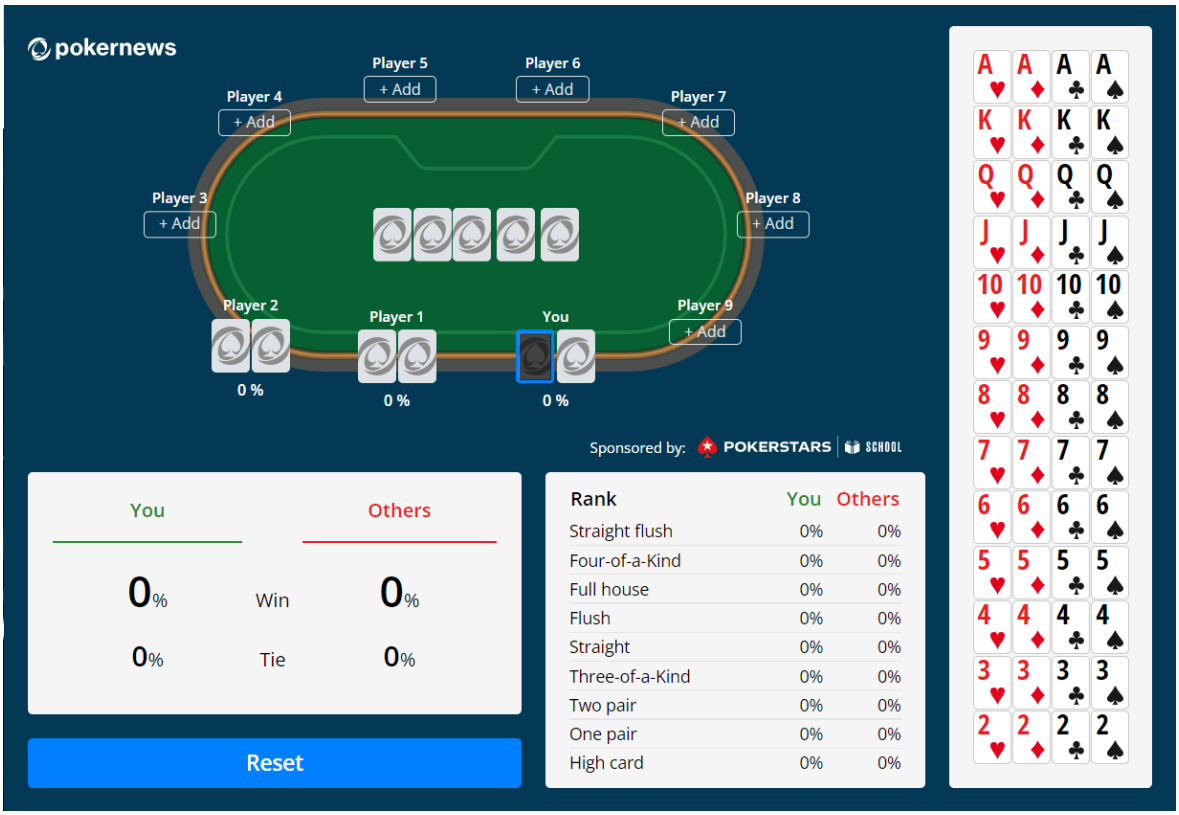
2.2. ábra. Cardschat Poker Odds Calculator [6]

alakuljon ki. Ez a megoldás nagy részben lefedi az általam kigondolt megvalósítást. A különbség, hogy én csak azt szeretném látni, hogy mennyi az esélyem a nyerésre. Az, hogy különböző kezek milyen esélyben alakulnak ki, számomra lényegtelen. Ezzel szemben olyan matematikai mutatókat, mint az átlag, a medián vagy a relatív gyakoriság sokkal fontosabb tudni, hogy segítsen a felhasználónak bemutatni az esélyeit [5]. Az alkalmazás grafikus felületéről láthatunk egy képet a 2.3. ábrán.

2.3. Az esélyek számítása

A tervezés első lépése a módszer meghatározása volt, amellyel az esélyeket számoljuk. Ezt hátulról előre haladva közelítettem meg, azaz a river-nél kezdtem, a turn-ön át, egészen a preflop-ig. Azért választottam ezt a megoldást, mert rekurzívan használtam fel a hátul alkalmazott módszert, azaz a river-nél megvalósított módszert a turn-nél és a flop-nél is felhasználom kisebb kiegészítésekkel.

Az alapötlet az volt, hogy legenerálom az összes lehetséges 7 lap együttesét, amelyekhez egy értéket rendelek, ami az adott lap erejét jelzi. Természetesen a legerősebb a royal flush, a leggyengébb pedig a 7-es magaslap. Csak, hogy szemléltessem a számo-



2.3. ábra. Pokerstars Texas Hold'Em Poker Odds Calculator [10]

lások bonyolultságát és nagyságát, ez

$$\binom{52}{7} = 133'784'560$$

eset. Turn esetében ismerem az én 7 lapomat, tehát csak egy keresést kell végezni ebben az adathalmazban, ami visszatér a kezem értékével. Az ellenfél esélyeinek kiszámítása már egy kicsit komplikáltabb. Az 52 lapból ismerünk 7-et, tehát a pakliban 45 lap marad. Ebből választhat további 2 lapot az ellenfél, ez

$$\binom{45}{2} = 990$$

lehetőség az ellenfél kezeire. Ez a szám fontos szerepet játszik a dolgozat történetében, és végig jelen lesz a dokumentációban. Erre a 990 lehetőségre mind el kell végezni a keresést a kb. 133 milliós adathalmazban, így kapunk 990 értéket. Innentől kezdve már csak az a kérdés, hogy hány százalékban kisebb a mi kezünk értéke az ellenfél lehetséges kezeinél.

Tovább haladva a turn esetében vizsgáljuk az esélyeket, amikor is csak 4 lap van az asztalon, egy lapra pedig még várunk, ami egyelőre ismeretlen. Itt kissé bonyolultabb a helyzet, mivel a saját kezünk értékét sem ismerjük. Természetesen vannak szélsőséges esetek, mikor már a turn-nél pókerünk vagy royal flush-ünk alakul ki, ezekben az esetekben nem releváns az utolsó lap, mindenképp nyertünk. Ilyen esetek azonban ritkán fordulnak elő. Először ki kell számolnunk a saját esélyeinket, amit majd össze tudunk hasonlítani az ellenfél esélyeivel. Két lap a kezünkben, négy lap az asztalon, 46 lap maradt, ami érkezhet az asztalra utolsóként. Mind a 46 lapra meg kell keresnünk

az adathalmazban a kezünk értékét, így a saját kezeinkre lesz 46 értékünk. Ezután mind a 46 lehetséges utolsó lapra meg kell keresnünk az ellenfél 990 lehetséges kezére az értékeket, ez 45'540 eset. Amint ezek megvannak, össze tudjuk hasonlítani az értékeket.

A flop esetében lévő értékek számításánál még több számítást kell végeznünk. A saját két lapunkon kívül itt már csak három lent lévő lapot ismerünk. Ez azt jelenti, hogy turn-re jöhet 47 féle lap, river-re pedig további 46 féle. A saját lapjaink

$$\binom{47}{2} = 1'081$$

féleképpen jöhetnek le. Ezek alapján az ellenfélnek

$$1081 \cdot 990 = 1'070'190$$

lehetséges keze van. Ezekre a lehetséges kombinációkra mind keresést kell végeznünk az adathalmazban, hogy megkapjuk a saját és az ellenfél értékeit, amivel tovább tudunk dolgozni.

Az utolsó licitkör, aminél számításokat kell végeznünk, az tulajdonképpen a játék menetét illetően az első licitkör. A preflop esetben nem ismerünk semmilyen lapot, csak azt a kettőt, amit osztottak nekünk. Az 52 lapos pakliból 50 lap érkezhet az asztalon lévő első helyre, a másodikra 49, az ötödikre, vagyis az utolsóra pedig 46 lehetséges lap. Itt kell a legtöbb számítást végezni, amikor is

$$\binom{50}{5} = 2'118'760$$

féleképpen alakulhat ki a saját kezünk. Ezt megszorozva a már ismert rivern-nél lévő ellenfél kezeinek kombinációjával, az ellenfél lehetséges kezeinek száma 2'097'572.400, ami több, mint kétmilliárd eset. Ennyi értékkel kell majd számolnunk a továbbiakban. Ez jól mutatja, hogy milyen összetett a problémakör és hogy mennyi számítást igényel. Az alapötletben ezt nem számoljuk valós időben, hiszen rengeteg időt venne igénybe. Ehelyett egyszer legeneráljuk ezeket a kombinációkat az összes lehetséges kézre preflop, ami 1'326 lehetőség, tehát ennyivel kell megszoroznunk még a fent említett számot.

2.4. A grafikus felület elemei

Az alapvető elképzelésem az alkalmazás külsőjéről az volt, hogy az oldalra érkezéskor rögtön egy autentikációs felület fogadja a felhasználót. Ezzel egy amolyan privát jelleget szerettem volna kölcsönözni az alkalmazásnak, hogy csak az használhassa, aki regisztrált. A regisztráció/bejelentkezés után egy főoldal fogad minket. Ezen az oldalon a Texas Hold'Em póker változatról egy rövid leírást mutatok be, valamint az alkalmazás lényegét szemléltetem. Az oldal tetején van egy egyszerű navigációs felület, ahol továbbmehetünk arra az oldalra, ahol magát a játékot találjuk.

Itt is az egyszerűséget tartom szem előtt. Középen helyezkedik el egy pókerasztal felülnézetből. Alatta jelenítem meg a négy színt (treff, káró, kör, pikk). Ezekre rákattintva előjön az adott színből az összes (13) kártya, ami közül kiválasztunk először egyet, ez a lap pedig eltűnik a választható lehetőségek közül és lekerül az asztalra saját

lapként. Megint a négy színt látjuk, amiből ismét kiválasztunk egyet, és még egy lapot. Ezek után ugyanezzel a módszerrel tudjuk lehelyezni az asztal közepére is a közös lapokat.

Minden licitkör előtt az asztal mellett láthatja a felhasználó az összesített nyerési esélyét, ami segít neki a döntés meghozásában. Ha több lapot szeretne a felhasználó az asztalra tenni, mint a 2 (saját) + 5 (közös), akkor egy felugró ablak jelenik meg, ami jelzi neki, hogy már több lapot nem használhat. Két választása van ebben az esetben, az egyik, hogy figyelmen kívül hagyja, ilyenkor elemezheti tovább az esélyeket. A másik, hogy új leosztást kezd, ilyenkor eltűnik minden lap az asztalról és folytathatja a játékot.

2.5. A becslés eredményeinek megjelenítése

A piackutatás során derült fény számomra arra, hogy a legtöbb ehhez hasonló alkalmazás csak egy átfogó százalékos esélyt ad az éppen aktuális leosztásról. Ez volt az egyik motivációm, hogy az alkalmazásomban ennél több információ álljon rendelkezésre a felhasználónak.

Azon kívül, hogy egy összefogó százalékos esélyt számítok minden licitkörnél, létrehozok egy oszlopdiagrammot is, amelyben minden lehetséges következő lap(ok)ra kirajzolom, hogy annál a losztásnál milyen százalékban nyer a felhasználó. Rivernél ennek nincs sok jelentősége, mivel ismerjük az összes lapot. Itt csak két oszlop lesz, az én, illetve az ellenfél nyerési esélye. Turn-nél 46 lap érkezhet még, így 46 oszlopunk lesz, aminél láthatjuk, hogy melyik esetben mennyi esélyünk van nyerni, ezek mellett természetesen az összesített esélyt is megjelenítem. Erre azért van szükség, hogy tisztább képet nyújtsak a felhasználó számára az esélyeiről. Egy példán keresztül megvilágítva, lehet, hogy a turn-ön 70% esélyünk van a nyerésre, viszont, ha ez úgy adódik össze, hogy 9 esetben 100% az esélyünk, 11 esetben 90%, 6 esetben 80%, a maradék 20 esetben pedig 40%, akkor jobban át kell gondolnunk a következő lépésünket, mintha ez a 70% egymáshoz közel álló számokból alakulna ki.

Emellett, a diagrammon több konstans értéket is megjelenítek. A fent említett oszlopdiagramm sokat segít a döntés meghozásában, viszont nehezen elemezhető. Éppen ezért szemléltetem az ellenfél nyerési esélyének az átlagát, a mediánját, valamint a szórását is. Ezek további segítséget nyújtanak a felhasználónak.

A diagramm alatt található egy beviteli mező, ahova egy számot adhatunk meg. Itt számolja az alkalmazás a realtív gyakoriságot, vagy más néven az empirikus valószínűséget. Tulajdonképpen az ide beütött érték előfordulásának számát kapjuk meg. Ha kíváncsiak vagyunk arra, hogy az ellenfélnek mennyi esetben van több esélye nyerni, mint 50%, akkor beütjük azt, hogy 50, és ki fogja írni a relatív gyakoriságot. Ez az érték 0 és 1 közé eshet.

2.6. Felhasznált technológiák

Ebben a szakaszban bemutatom az összes olyan technológiát, programozási nyelvet, keretrendszert és könyvtárat, amit felhasználtan a szakdolgozatom elkészítése során. Törekszem a tömör magyarázatra. Mindegyiknél próbálom elmagyarázni miért választottam, valamint mik az előnyei, amik számomra kedvezőek voltak.

2.6.1. Egyoldalas webalkalmazások

Ha figyelemmel követjük a JavaScript keretrendszerek fejlődését, akkor láthatjuk, hogy egyre nagyobb teret kapnak az egyoldalas alkalmazások (*SPA*, *Single Page Application*), az olyan többoldalas alkalmazásokkal (*Multi Page Application*) szemben, mint a jQuery vagy a Laravel.

Három elterjedt JavaScript keretrendszert különböztethetünk meg. A Facebook által fejlesztett React-ot, az Angular-t, amit a Google hozott létre és a Vue JS, amit Evan You fejlesztett [4].

2.6.2. Vue JS

Mindhárom SPA-nak megvan a maga előnye és hátránya. Ezek közül inkább csak a Vue előnyeire koncentrálok a másik kettővel szemben. A három közül ez a legkönnyebben tanulható. Használatát aránylag könnyen el lehet sajátítani.

A maga 18 KB-os méretével rendkívül kicsinek számít, amit könnyű letölteni és feltelepíteni, ennek ellenére pozitívan hat a felhasználói élményre és jó keresőmotor optimalizálással is rendelkezik. Saját virtuális DOM-ot (*Document Object Model*) renderel, ami jobban teljesít, mint a React vagy az Angular. Könnyen olvasható a kódja, könnyen integrálható, jól dokumentált, és ezen kívül sok más előnye is van.

Többek között a Vue JS egyik fő tulajdonsága – ami megvan a többi SPA-ban is – hogy komponensekből épül fel, amiket újra felhasználhatunk, ezzel is elősegíti a rövidebb programkódot, átláthatóságot, egyszerűséget, amiket már fent említettem.

Ezek miatt esett a választásom a Vue JS-re, amire több forrás keretrendszerként, mások pedig könyvtárként hivatkoznak [4].

2.6.3. Node JS és az Express

Amellett, hogy a legnépszerűbb programozási nyelv, a JavaScript az egyik leginkább univerzális szoftverfejlesztési technológia. Hagyományosan frontend fejlesztésre használták, viszont az utóbbi időben elterjedt a szerver oldali (backend) használata is. Az egyik eszköz – és talán a legismertebb – ami ezt az elmozdulást elősegítette az a Node JS.

A Node JS tulajdonképpen nem egy keretrendszer, nem is egy könyvtár, hanem egy futtató környezet, ami a Chrome V8-as motorján alapszik. A technológiát először 2009-ben mutatta be Ryan Dahl az Európai Javascript Konferencián. Annak ellenére, hogy ezen a konferencián rögtön elnyerte a legizgalmasabb szoftver díjat, nem volt használatos széles körben. A technológia 2017-ben csúcsosodott ki, amikor is először használta egy ismertebb cég, a LinkedIn, vagy, hogy még néhányat említsek, a Netflix, eBay és az Uber.

Az egyik hatalmas előnye a Node JS használatának, hogy automatikusan teljeskörű (full stack) web fejlesztővé válunk vele. Gondoljunk csak bele, két legyet ütünk egy csapásra. Egyszerűen nincs alternatívája a szerver oldali programozásnak a Javascriptben, csak a Node JS, ez teszi megkerülhetetlenné a technológiát.

Ezek mellett gyors feldolgozású, mivel közös nyelvet használ a frontend és a backend, így a szinkronizáció gyors. Esemény alapú modell-t (*Event-Based Model*) használ, ezért népszerű választás online játékok vagy videó konferenciák készítéséhez. Gazdag az ökoszisztémája, az NPM (*Node Package Manager*) – ami a Node JS alapértelmezett

csomagkezelője - paranccsal több, mint 800 ezer könyvtárat érhetünk el. Ezen felül számos előnye van még, amikre most nem térek ki.

Az Express a legnépszerűbb Node JS keretrendszer. Köztes szoftverként (middleware) hivatkoznak rá, amely annyit tesz, hogy tulajdonképpen a kliens és a szerver oldal közötti híd felépítéséhez biztosít eszközöket. Könnyű, rugalmas, és véleménymentes keretrendszer. Véleménymentes, mert semmilyen módon nem korlátozza a fejlesztőt, így nagy szabadságot ad. Emellett nagy teljesítményű, és hatalmas közössége van a sok felhasználó miatt [3].

2.6.4. Chart JS

A Chart JS egy nyílt forráskódú javascript könyvtár, adatok megjelenítésére, amely nyolc féle diagramm típust támogat. 2013-ban fejlesztették, mára a második legnépszerűbb diagramm készítő javascript könyvtár a GitHub-on. Meglepően egyszerű használni, ez volt a fő oka, amiért ezt választottam.

A Chart JS HTML5-be renderel. Az én célomhoz csupán kétféle diagramm típust kell használnom. Az oszlopdiagrammot és a vonaldiagrammot, ezeket kell egybefűznöm és úgy megjelenítenem [7].

2.6.5. HTML

A HTML-ről (*HyperText Markup Language*) nem szeretnék hosszasan írni. Talán egy laikus is tudja, hogy ha webalkalmazásról van szó, vagy akár csak egy honlapról, akkor megkerülhetetlen a HTML.

A HTML-t weboldalak készítésére hozták létre, amit később bárki elérhet, aki rendelkezik internet kapcsolattal (feltéve, hogy fel van töltve a weboldal egy domain-re). Használhatunk benne főcímeket, paragrafusokat, beépített képeket, videókat. Ezeket úgynevezett tag-ek határozzák meg. Az elején a kezdő tag, a végén pedig a záró tag.

Szakdolgozatomban a HTML5-öt használtam, amit 2014-ben mutattak be. Többek között olyan újításokat tartalmazott, mint a beépített audio és video tartalom [2].

2.6.6. CSS

A CSS (*Cascading Style Sheets*) egy stílusleíró nyelv. Alkalmazása a HTML elemek formai megjelenésére irányul, azaz hogyan szeretnénk láttatni a weboldalunkon megjelenő tartalmakat.

A CSS a HTML elemekre hat, a kommunikálás pedig többek között a *selector*-ok segítségével történik. Ezeket a CSS alapból tartalmazza, hivatkozhatunk egy megformálni kívánt paragrafusra, főcímre, valamint megadhatunk saját osztályokat is, amiket többször fel tudunk használni. A deklaráció tulajdonságokat és értékeket tartalmaz.

Itt szeretnék megemlíteni két dolgot, amit a szakdolgozatom készítése során tapasztaltam. Az egyik, hogy érdekes volt számomra megfigyelni, hogy leginkább ez a rész volt az, ami felkeltette az érdeklődésem, ebben tudtam maximálisan kiteljesedni. A másik pedig, hogy megjegyezném az egyre elterjedtebb felhasználási módját a CSS-nek. Ez pedig nem más, mint a Bootstrap, amit azért hoztak létre, hogy könnyedén készítsünk reszponzív weboldalakat, azaz minden képernyőméreten szépen megjelenő oldalakat.

Esetemben tudtam, hogy az én alkalmazásomat számítógép képernyőjén, esetleg más eszközökön, de kizárólag fektetett állapotban lehet használni. Éppen ezért nem láttam értelmét mélyebben beleásni magam ebbe a világba [2].

2.6.7. Google Firebase

A Firebase egy szoftverfejlesztő platform, ami 2011-ben indult és 2014-ben került a Google tulajdonába. Valós idejű adatbázisként (*RealTime Database*) indult, mostanra viszont 18 szolgáltatása és dedikált API-ja (*Application Programming Interface*) van. Az egész platform egy úgynevezett Backend-as-a-Service megoldást kínál mobil és webalapú alkalmazásokhoz, amely szolgáltatást tartalmaz az alkalmazások fejlesztésére, tesztelésére, és kezelésére.

Számomra a legfőbb előnyei a technológiának, amiért ezt választottam a következők. Teljesen ingyenesen lehet használni a legtöbb szolgáltatását. Könnyű hozzáférést biztosít az adatokhoz, a Firebase console-on keresztül. Könnyű az integrálása is, és minimális programozási ismereteket kíván, tehát majdnem bárki be tudja építeni az alkalmazásába.

Annak ellenére, hogy ez egy nagyon összetett platform, az alkalmazásom kizárólag a Real Time Database-t használja ezekből. A Google Firebase végzi az autencikációt, vagyis a regisztrációt, és a bejelentkezést az oldalra [1].

2.6.8. JSON

A JSON (Javascript Object Notation) egy szövegalapú nyílt szabvány, amit emberek számára is könnyen olvasható adatátvitelre terveztek. JavaScript alapú alkalmazásokhoz, valamint hálózati kapcsolaton keresztüli továbbításra használják.

Egyik és talán legfontosabb előnye, hogy a legtöbb dinamikus nyelvben közvetlenül megfelel az alapvető adattípusoknak, továbbá különbséget tesz a **string**, **number** és **boolean** értékek között, tehát könnyedén lehet használni a webfejlesztés során.

Objektumokat hozhatunk benne létre, melyekre kulcsokkal tudunk hivatkozni. Számomra ez volt a fő szempont, mert nagy adathalmazzal kell dolgoznom és ezekben keresést végezni, ami ezzel a módszerrel gyorsnak tekinthető.

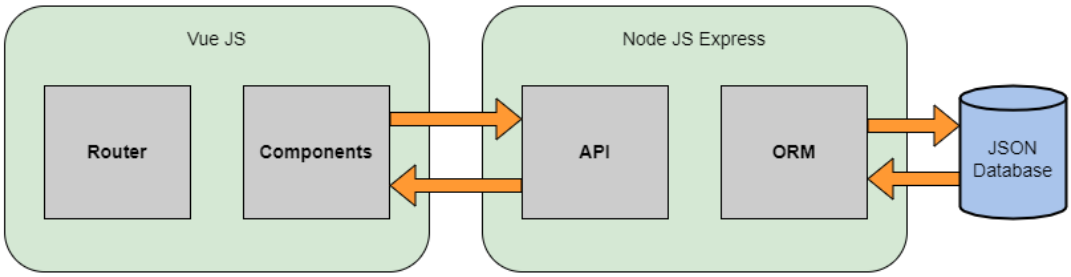
3. fejezet

Tervezés és implementáció

Ebben a fejezetben bemutatom az elkészült programot a számomra legérdekesebb programrészletekkel. Nem térek ki minden funkcióra és programkódra, csak a lényegesebb egységekre. Részletezem a frontend kivitelezését, amelybe tartozik a Vue JS, HTML és CSS kódok. Ezután a backend számításokat mutatom be, amelyet Node JS környezetben valósítottam meg.

3.1. A projekt felépítése

Mielőtt belekezdek a megoldásaim, függvények, programkódok részletezésébe, először bemutatom a projekt felépítését, hogy egyszerűbben érthető legyen, kezdve a webalkalmazás felépítésével (3.1. ábra).



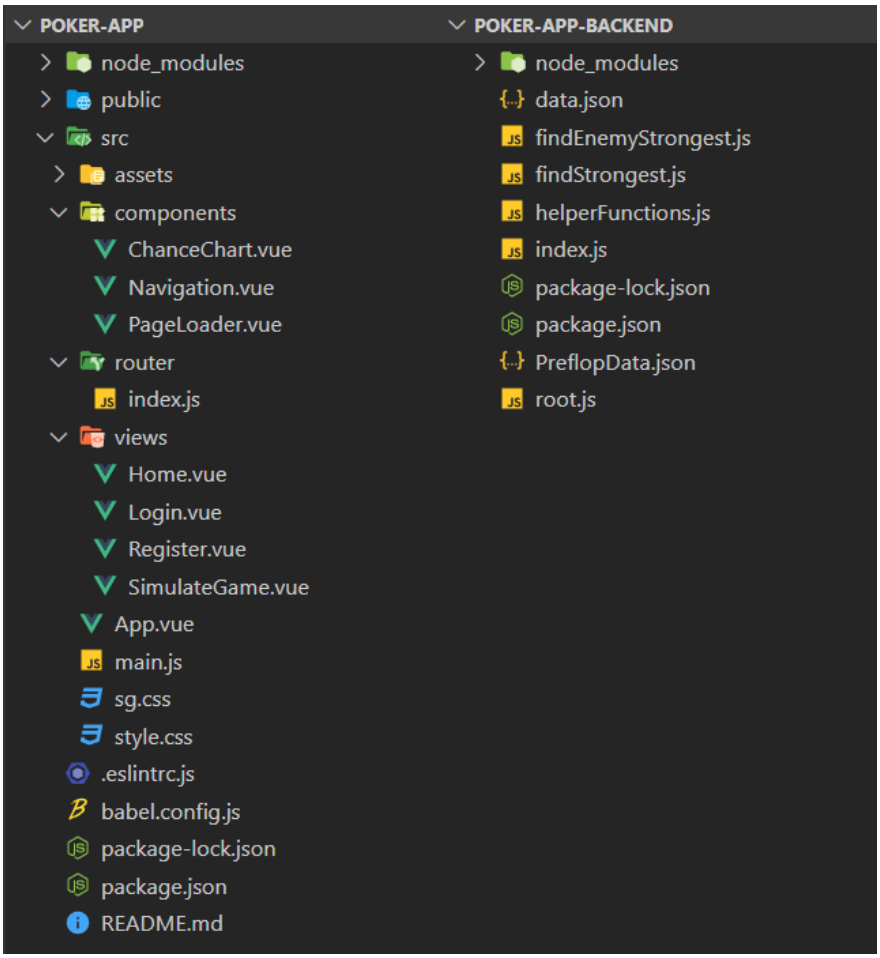
3.1. ábra. A webalkalmazás architektúrális ábrája

A webalkalmazás frontend részét a már említett Vue JS-ben valósítottam meg. Ebben vannak a komponensek és a vue-router, amely az oldalak közti navigációt végzi el. A frontend és a backend közötti kommunikációt egy API adja meg. A JSON adatbázis lokálisan helyezkedik el és csak a backend kommunikál vele, illetve használja a számításokhoz.

A 3.2. ábrán látható az elkészült projekt struktúrája fájl rendszer szinten.

Az ábra bal oldalán a frontend felépítése látható. A `components` mappában vannak a komponensek, amelyeket többször fel lehet használni az oldalakon.

A `router` mappában lévő `index.js` fájlban vannak definiálva a `router`-ek. Itt meg kell adni az elérési útvonalat, az oldal nevét, amelyre irányítani szeretnénk a felhasz-



3.2. ábra. A projekt struktúrája file rendszer szinten

nálót, illetve importálni az adott vue file-t. Ezeket az oldalakat a views mappában találjuk, itt az autentikációt végző oldalak, a home fül és a *SimulateGame* helyezkedik el, amelyben maga a póker alkalmazás valósul meg.

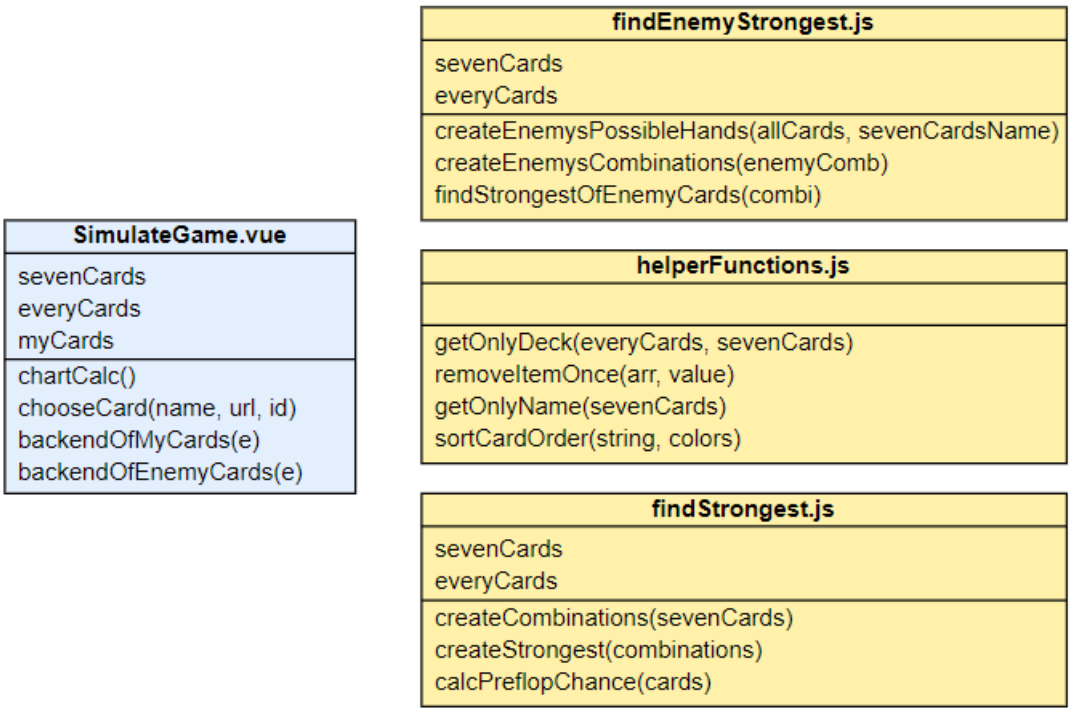
Ezeken kívül a `main.js`-ben importálni kell minden olyan kiegészítőt, melyet használok az alkalmazásban, valamint a Firebase config része is itt van definiálva. Fontosak még az CSS fájlok. Ezeket importálom be az egyes oldalakon, hogy a megfelelő megjelenést kölcsönözzem nekik. Ezeken kívül a `package.json` és `package-lock.json` file-okban meg vannak adva a felhasznált technológiák verziója, hogy ne legyen köztük ütközés.

A 3.2. ábra jobb oldalán látható az alkalmazás back end része. Itt az architektúra jóval egyszerűbb. Két JSON adathalmazt hoztam létre; az egyik a kártyák erősségét tárolja, a másik a preflop esélyeket, mind a kettőben egy-egy objektum helyezkedik el. Itt is megtalálható a `package.json` és a `package-lock.json`, ezek ugyanazt a szerepet töltik be, mint a front end-nél. Az `index.js` is hasonló mint, a front end-en, itt is be kell importálni minden felhasznált technológiát, valamint megadni, hogy milyen porton fusson a szerver. Erre azért van szükség, hogy a localhost-on futtatott kliens és szerver ne kavarodjon össze.

A `findStrongest.js`-ben azok a kódok vannak, melyek a felhasználó esélyeit vizsgálják, a `findEnemyStrongest.js`-ben pedig azok, amik az ellenfél esélyeit. A `helperFunctions.js` arra szolgál, hogy a több helyen meghívott függvényeket eltárol-

ja, így azokat nem kell többször megírni. Végül a `root.js`-ben két POST `response` helyezkedik el, így történik az adatátvitel a kliens részére.

Az osztály diagrammon bemutatom, hogy nagy vonalakban milyen adatokkal, függvényekkel és a hozzájuk kapcsolódó paraméterlistákkal dolgoztam. Ez a 3.3. ábrán látható.



3.3. ábra. Az alkalmazás osztály diagrammja

A kék elem a klienst szemlélteti, a sárgák pedig a szervert hivatottak bemutatni.

Az adatok mindenhol tartalmazzák a `sevenCards` és az `everyCards` adatokat, kivéve a `helperFunctions`-nál, hiszen onnan csak meghívom a függvényeket. A frontend oldalon továbbá jelen van a `myCards` is, erre az esélyek ábrázolása miatt van szükség. Itt jelenítem meg a diagrammokat a `chartCalc` függvénnyel. A `chooseCard` függvény veszi ki a lapokat a paklból, amit majd továbbadok a szervernek, valamint a lapok megjelenítésében is szerepet játszik. A két backend függvény pedig két POST `request`-et tartalmaz.

A sárga részeken találhatóak azok az adatok és függvények, amik az esélyek számolásáért felelősek. A teljesség igénye nélkül, itt számolom a lehetséges érkező lapokat, ezekre az értékeket, sorba rendezem a lapokat, leszűkítem csak a nevekre, és egyéb lent látható eseményeket.

3.2. A frontend kivitelezése

Egy webalkalmazás frontend része az, amit a felhasználó érzékel az oldalból. Itt részletesen bemutatom a főbb egységeket, melyek Vue JS-ben íródtak. A HTML és CSS

részekre csak minimálisan térek ki, itt pedig igyekszem azokat a megoldásokat bemutatni, melyek a legizgalmasabbak voltak a kivitelezés során.

3.2.1. Autentikáció

Az alkalmazás első oldala a bejelentkező/regisztrációs felület. A két oldal kisebb eltérések mellett ugyanaz, ezért nem térek ki külön csak a változásokra. A register oldalon meg kell adnunk egy e-mail címet, egy hozzá tartozó jelszót, valamint egy becenevet is. Ha regisztráltunk, akkor a login oldalon csak megadjuk az e-mail címünket és a hozzá tartozó jelszót, a sign in gombra kattintunk és már be is enged minket az oldal. A bejelentkezést és a regisztrációt a Firebase végzi, valamint a validációt is. Az e-mail címnek tartalmaznia kell egy kukac jelet, illetve pontot és egy domain nevet. A jelszónak legalább hat karakter hosszúnak kell lennie.

Ennek a két oldalnak a HTML része nagyon egyszerű. Egy űrlapot, belül két szöveges beviteli mezőt, ezen kívül egy gombot találunk. Mindezt közbe zárja egy `div` elem, ami tartalmaz még két címet is, melyek a *login* és a *sign up* feliratok. Ezekkel tudunk váltani a két oldal között. A `login` rész még kiegészül egy harmadik beviteli mezővel, ami a becenév.

```
1  <div class="login">
2    <h2 class="active"> Login </h2>
3    <h2 class="nonactive"><router-link to="/register"> Sign up </
        router-link></h2>
4    <form @submit.prevent="Login">
5      <input type="text" class="text" name="E-mail" v-model="email">
6      <span>E-mail</span>
7      <input type="password" class="text" name="password" v-model="
          password">
8      <span>password</span>
9      <button class="signin" value="Login">
10       Sign In
11     </button>
12   </form>
13 </div>
```

Az autentikációt viszonylag egyszerű elvégezni a Firebase segítségével. Regisztrációnál a `createUserWithEmailAndPassword` függvényt használjuk, amivel létrehozunk egy felhasználót. Ehhez egy e-mail cím és egy jelszó tartozik, jelen esetben kiegészülve egy becenévvel. Ezt el is tárolja nekünk a Firestore Database-ben. Ezután bejelentkezésnél a `signInWithEmailAndPassword` függvényt használjuk, amely megkapja az e-mail és jelszó párost, és ha ez egyezik, már bent is vagyunk a főoldalon.

```
1  const Register = () => {
2    firebase
3      .auth()
4      .createUserWithEmailAndPassword(email.value, password.value)
5      .then((user) => {
6         db.collection("users")
7           .doc(user.uid)
8           .set({nickname: nickname.value})
9       })
10      .catch((err) => alert(err.message));
11 };
```

Az autentikációs oldalak stílusa talán a leglátványosabb az alkalmazásban. Igyekeztem egységesen megformázni az összes oldalt. Ahol lehet lekerekítést használok, és mindenhol azonos kék és piros színekkel kombinálok. A CSS legérdekesebb része számomra ebben a részben a két címsor formázása volt. A felhasználónak ez csupán színek váltakozása, viszont annál sokkal érdekesebb. Mikor melyik cím aktív, úgy annak a színe változik fehérré és kap egy aláhúzást, a másik címsor pedig elhalványodik. Ezt a `login` és a `register` oldal váltásával jelenítem meg.

```
1 .active {
2   border-bottom: 2px solid #1161ed;
3 }
4 .nonactive {
5   color: rgba(255, 255, 255, 0.2);
6 }
```

3.2.2. Főoldal és navigációs fejléc

A főoldalról röviden összefoglalva szeretnék írni, mert tartalmilag is elég rövid. Mindössze két block elemben két paragrafus található benne, formázva. Az egyik röviden leírja a póker játék lényegét, a másik pedig megfogalmazza, miről is szól az alkalmazás.

A Vue JS egyik előnye, hogy komponensekből épül fel, melyeket többször fel tudunk használni. Nekem az egyik ilyen komponensem a navigációs fejléc. Ezt mindegyik oldalon alkalmazom, hiszen a felhasználó ezen keresztül tud váltani az oldalak között. Másik sajátossága a `view-router`, ami megvalósítja az oldalak közötti ugrást, ezek az `index.js` fájlban vannak definiálva.

```
1 <nav class="navbar">
2   <ul>
3     <li class="stand"><router-link to="/" class="must">Home</
        router-link></li>
4     <li class="stand"><router-link to="/simulategame" class="must"
        >Simulate Game</router-link></li>
5     <li @click="Logout" class="log"><router-link to="/login"
        class="must">Logout</router-link></li>
6   </ul>
7 </nav>
8 </template>
```

A Vue-nak köszönhetően ebben a komponensben nincs szükség JavaScript-re, anélkül tudunk váltani az oldalak között. Természetesen ezt is formázom CSS-el.

3.2.3. Játék szimuláció

Az alkalmazás nagy része frontend szempontból a `SimulateGame.vue` komponensben került megvalósításra. Itt találjuk magát a játékot. Az oldalra érkezéskor középen egy pókerasztalt látunk, alatta pedig a 4 színt. Ezek közül egyre rákattintunk, akkor felugrik abból a színből mind a 13 lap, itt tudunk kiválasztani egyet. Ha ez megtörtént, akkor az a lap eltűnik a választási lehetőségek közül, többször már nem választhatjuk, valamint hozzáadja egy tömbhöz, majd ugyanezt megismételjük 6-szor. Az összes lapot előre definiáltam. A lapokra való kattintáskor meghívódik a `chooseCard` függvény, amely a fentieket végzi el.

```
chooseCard(name, id, url) {
  if (this.isThere === 8) {
    this.cardsFull = true;
    return;
  }
  this.sevenCards.push({ name, url });
  this.isThere = this.isThere + 1;
  this.showPic = false;
  this.isHidden = false;
  this.cards[id - 1] = false;
}
```

A függvény elején rögtön egy feltétel van, ami ellenőrzi, hogy ne rakhassunk le 7 lapnál többet az asztalra. Amennyiben így szeretnénk tenni, egy felugró ablak jelenik meg, amely jelzi a felhasználónak, hogy több lapot már nem tehet le.

Ezen kívül a kiválasztott lapot belerakjuk a `sevenCards` globális objektumba, melyet majd továbbadunk a backend-nek, hogy végezze el a számításokat. Továbbá tartalmaz pár egyéb külső változót is, melyek a színek és lapok megjelenítéséért felelnek.

A másik lényeges rész ezen az oldalon az esélyeket megjelenítő diagramm. Ez is egy külön komponens, viszont itt töltöm fel adatokkal. A backend-től megkapjuk az összes következő lapra számolt esélyt. Az ellenfélnek mindig 990-szer több esete lesz, hiszen nem ismerjük a lapjait. Emiatt a mi esélyeink első esetét össze kell hasonlítani az ellenfél első 990 esetével, ebből kapunk egy százalékos esélyt, amely majd az első oszlopunk lesz a diagrammon, és így tovább.

Ezek mellett még olyan matematikai értékeket is számolok, mint az átlag, a medián vagy a szórás, melyek közül szerintem az utóbbi a legérdekesebb.

```
calculateDeviation(arr) {
  let mean =
    arr.reduce((acc, curr) => {
      return acc + curr;
    }, 0) / arr.length;
  arr = arr.map((k) => {
    return (k - mean) ** 2;
  });
  let sum = arr.reduce((acc, curr) => acc + curr, 0);
  return Math.sqrt(sum / arr.length);
}
```

A felhasználó továbbá meg tud adni egy értéket is, amelyre az alkalmazás kiszámolja a relatív gyakoriságot, így többlet információhoz juthat, hogy mennyi esetben van például 50% felett az ellenfél nyerési esélye.

3.3. A backend kivitelezése

A backend rész megvalósítása véleményem szerint a legkiemelkedőbb eleme a szakdolgozatomnak. A következőkben az ehhez tartozó Node JS programkódok fognak szerepelni magyarázattal, valamint bemutatom az adathalmazomat is.

3.3.1. Adathalmaz bemutatása

Úgy gondolom, átláthatóbb a dokumentáció úgy, ha először az adathalmazomat mutatom be, ugyanis erre többször fogok hivatkozni a későbbiekben.

Mint azt az alapötlet szakaszban is említettem, először egy kb. 133 millió soros adathalmazzal kezdtem el a megvalósítást, melyben az összes lehetséges 7 lap kombinációja benne volt. Ezt Python programozási nyelv segítségével magamnak generáltam le. A generáció futási ideje 7,5 óra volt, az adathalmaz mérete pedig 6 GB.

```
from itertools import combinations

cards = combinations(["2c", "3c", "4c", ... , "As"], 7)
sevenCards = {}
sevenCards["cards"] = []

with open("data_new.json", "w") as outfile:
    for c in cards:
        c = list(c)
        outfile.write(f"{c}, \n")
```

Hamar kiderült, hogy ez akkora adathalmaz, amivel nagyon nehezen tudtam volna a továbbiakban dolgozni. Rengeteg keresést kell végeznem ebben, ami nagyon sok időbe telne, ezért más megoldást kellett találnom.

Az Interneten találtam egy másik adathalmazt, amely lényegesen kisebb volt, 7462 soros, ezt használtam fel [9]. Ez a lehetséges 5 lap kombinációinak száma úgy, hogy nem vizsgálunk külön minden színt. Így a számításaim annyival bővültek, hogy a kiválasztott 7 lapból

$$\binom{7}{5} = 21$$

keresést kell végeznem, hogy megtaláljam a legerősebb 5 lapot, amit a felhasználó magától is ki tud választani, viszont az alkalmazásnak is tudnia kell, tehát ezzel kell majd tovább számolnia.

Az adathalmazt JSON-ben tároltam el. Egyetlen objektumot tartalmaz, aminek az értéke a kombinációk, a kulcsa pedig a kéz erőssége. Azoknál az eseteknél, ahol számításba kell vennünk, hogy a lapok színe azonos, ott egy F betűt szúrtam az értékekbe. Ezeket abc sorrendbe rendeztem az egyszerűbb felhasználás végett.

```
{
  "cardStrenght": {
    "AFJKQT": 1,
    "9FJKQT": 2,
    "89FJQT": 3,
    "789FJT": 4,
    "6789FT": 5,
    .
    .
    .
    "23457": 7462
  }
}
```

Az első 10 érték a színsorok, royal flush-el kezdve, a 11-dik az ász póker, király kísérővel, a 7462-dik pedig a 7-es magaslap.

A projekt vége felé közeledve, a preflop esélyek számolása az addig alkalmazott módszer szerint még mindig túl sok időt vett volna igénybe. Mivel két lap ismeretében nincs is annyi értelme több matematikai értéket szemléltetni, így arra jutottam, hogy csak egy százalékos esélyt fogok mutatni a többi lap ismerete nélkül. Ehhez létrehoztam egy hasonló adathalmazt, mint a kártyaerősséget tartalmazó, ezt elneveztem

`PreflopData.json`-nek. A tartalmát az alábbi táblázat adja. A párok alkotta átló fölött lévő kezdőlapok az egyszínűek esélye, alatta pedig a különböző színű lapoké.

	A	K	Q	J	T	9	8	7	6	5	4	3	2
A	85%	68%	67%	66%	66%	64%	63%	63%	62%	62%	61%	60%	59%
K	66%	83%	64%	64%	63%	61%	60%	59%	58%	58%	57%	56%	55%
Q	65%	62%	80%	61%	61%	59%	58%	56%	55%	55%	54%	53%	52%
J	65%	62%	59%	78%	59%	57%	56%	54%	53%	52%	51%	50%	50%
T	64%	61%	59%	57%	75%	56%	54%	53%	51%	49%	49%	48%	47%
9	62%	59%	57%	55%	53%	72%	53%	51%	50%	48%	46%	46%	45%
8	61%	58%	55%	53%	52%	50%	69%	50%	49%	47%	45%	43%	43%
7	60%	57%	54%	52%	50%	48%	47%	67%	48%	46%	45%	43%	41%
6	59%	56%	53%	50%	48%	47%	46%	45%	64%	46%	44%	42%	40%
5	60%	55%	52%	49%	47%	45%	44%	43%	43%	61%	44%	43%	41%
4	59%	54%	51%	48%	46%	43%	42%	41%	41%	41%	58%	42%	40%
3	58%	54%	50%	48%	45%	43%	40%	39%	39%	39%	38%	55%	39%
2	57%	53%	49%	47%	44%	42%	40%	37%	37%	37%	36%	35%	51%

3.4. ábra. Preflop kezek esélyei [11]

3.3.2. A felhasználó esélyeinek számítása

Az egyszerűbb része az esélyek becslésének a felhasználó esélyeinek a kiszámolása, ugyanis itt kevesebbet kell számolni.

A backend megkapja a frontend-től a felhasználó által kiválasztott lapokat. Itt is definiálva van a pakli minden lapja, viszont objektum helyett csak egy tömbbe, mivel csak a lapok nevére van szükségünk. A `default` függvényben egy ellenőrzést végzek, hogy milyen hosszú az objektum, amit megkap. Erre azért van szükség, mert különböző számításokat kell végezni ha a flop-nál, turn-nél vagy river-nél szeretnénk megtudni esélyeinket. Először minden esetben a `getOnlyName` és a `getJustDeck` függvényeket használom. Az előbbivel az objektumot egy tömbre szűkítem, hogy csak a kártyák neveivel dolgozhassak tovább. Az utóbbival az összes lapot tartalmazó tömbből kitörlöm azokat, amelyeket a felhasználó már kiválasztott, így megkapom a pakliban maradt lapokat.

River esetében a tömb hosszúsága 7. Itt van a legegyszerűbb dolgom, mivel minden lap ismert, csak vissza kell térni a felhasználó kezének értékével. Először legenerálom a 7 lapból az összes 5 lap kombinációját. Ezt a `createCombinations` függvénnyel teszem. Ez paraméterként megkap egy 7 elemű tömböt és visszatér 21, 5 elemű tömbbel. A generációhoz a JavaScript *generatorics* nevű kombinatorikai könyvtárát használom.

```
function createCombinations(sevenCards) {
  let allCombinations = [];
  for (let comb of G.combination(sevenCards, 5)) {
    allCombinations.push(comb.slice());
  }
  return allCombinations;
}
```

Ha megvan a 21, 5 elemű tömbünk, akkor már csak meg kell keresni mindegyiknek az értékét az adathalmazban, majd kiválasztani a legkisebbet és azzal visszatérni. Ezt a `createStorngest` függvény valósítja meg, amely megkapja az előbb említett 5 lapos kombinációkat és visszatér a legerősebb 5 lap értékével.

```
function createStrongest(combinations){
  let rawdata = fs.readFileSync("data.json");
  let strenghtOrder = JSON.parse(rawdata);
  let result = [];
  for (let combination of combinations) {
    let nameString = "";
    let colors = { C: 0, S: 0, H: 0, D: 0 };
    for (let card of combination) {
      nameString += card[0];
      colors[card[1]] += 1;
    }
    let ordered = sortCardOrder(nameString, colors);
    result.push(strenghtOrder.cardStrenght[ordered]);
  }
  return Math.min(...result);
}
```

A `strengthOrder` változóban eltárolom az adathalmazt. Egy *foreach* ciklussal végig megyek mind a 21 kombináción. A cikluson belül ellenőrzöm, hogy a lapok között hány egyforma színű lap van, hiszen ha mind az 5 egyforma színű, akkor az olyan értékek közt is keresni kell, ahol számít a szín, tehát fulsh, esetleg színsorok. Itt a `sortCardOrder` függvény is szerepet játszik, itt rendezem abc sorrendbe a lapokat, illetve szúrok közé egy F betűt is, amennyiben figyelembe kell vennünk, hogy azonos színűek a lapok. A megkapott 21 lap értékével feltöltök egy tömböt, ezek után a függvény visszatér ennek a minimumával, ami a felhasználó lapjának az értéke lesz, ezt adom át a forntend-nek.

Turn-nél a kapott tömb hosszúsága 6, azaz még egy lapra várunk, amely nem ismert. Ebben az esetben ugyanazt kell tenni, mint a river esetében, egy kiegészítéssel. Mivel egy lapra még várunk, így végig kell menni egy ciklussal az összes pakliban maradt kártyán, melyekkel kiegészítem a kapott tömböt, és elvégzem ugyanazokat a számításokat, amiket a river-nél is. Mivel 46 lap maradt a pakliban, így 46 értékkel tér vissza a függvény, ezeket adom át a frontend-nek.

Abban az esetben, hogy ha három közös lap van az asztalon (azaz a flop), akkor egy 5 hosszúságú tömböt kap a backend, tehát még két lapra várunk. Hasonló a teendő, mint az előző esetben. Mivel most két lapra várunk, így két egymásba ágyazott ciklusra van szükség, majd mindkettőnek az aktuális elemét hozzá kell fűzni a kapott tömbhöz, ezután meghívni ezekre a már ismert függvényeket. Ezzel megkapjuk az összes lehetséges lapra az esélyünket. A belső *for* ciklusnak mindig eggyel a külső *for* ciklus léptető változója előtt kell járnia, mivel így elkerülhetem, hogy egy adott kombináció kétféleképpen is előforduljon. Számunkra irreleváns, hogy a negyedik utcán érkezik egy treff király, az ötödiken pedig egy kör ász, vagy fordítva, a végeredmény ugyanaz.

Mikor csak a saját lapjainkat ismerjük, csupán annyit tesz az alkalmazás, hogy megkeresi a megadott két lapos kombinációt a `PreflopData.json` adathalmazban, és visszatér az értékével. Ez lesz a százalékos esélye a felhasználónak megnyerni a partit flop előtt. Ehhez a program a `calcPreflopChance` függvényt használja. Ebben szintén eltárolom egy változóba az adatokat, majd elvégzem a keresést benne. Ezen kívül annyi feladata van még, hogy ellenőrizze a két lap azonos színű-e. Ezt úgy teszi, hogy a lapok nevének a második karakterét, amely jelöli a színét, összehasonlítja. Amennyiben ezek egyenlőek, hozzáfűz egy F betűt, majd elvégzi a keresést az összefűzött sztringgel, valamint annak fordítottjával is. Erre azért van szükség, mert csak egy sorrendben szerepelnek a lapok a JSON dokumentumban. Például, az azonos színű ász-király "AKf" jelölésű, viszont, ha mi először királyt kapunk, utána ászt, akkor a "KAf" jelölésre nem fog találni semmit.

```
1  function calcPreflopChance(cards) {
2    let rawdata = fs.readFileSync("PreflopData.json");
3    let chances = JSON.parse(rawdata);
4    let cardCheck;
5    let reversed;
6
7    if (cards[0][1] == cards[1][1]) {
8      cardCheck = cards[0][0] + cards[1][0] + "f";
9      reversed = cards[1][0] + cards[0][0] + "f";
10   } else {
11     cardCheck = cards[0][0] + cards[1][0];
12     reversed = cards[1][0] + cards[0][0];
13   }
14
15   if (chances["preflopStrenght"][cardCheck]) {
16     return chances["preflopStrenght"][cardCheck];
17   } else {
18     return chances["preflopStrenght"][reversed];
19   }
20 }
```

3.3.3. Ellenfél esélyeinek számítása

Az ellenfél esélyeinek számítása egy kicsit bonyolultabb folyamat, mivel itt minden műveletnél 990-szer többet kell számolni, mint a saját esélyeinknél, hiszen az ellenfél kezében lévő két lapot nem ismerem.

Ez a rész sokban megegyezik az előző alszakaszban kifejtett részekkel. Itt is definiáltam a paklit, megkapjuk a frontend-től a kiválasztott lapokat, valamint szintén a `getOnlyName` és `getJustDeck` függvényekkel kezdünk. Mivel többek között ezeket a függvényeket ez a modul és az előzőekben tárgyalt is használja, így egy külön `helperFunctions.js` fájlban tárolom el, hogy ne kelljen mindkettőt kétszer megírni.

Ebben az esetben is az elágazás ágainak működését mutatom be. Az első eset, amikor 7 hosszúságú a kapott tömb, tehát minden lapot ismerünk. Itt szükségem van az ellenfél 990 lehetséges lapjának az értékére. Ezt a `createEnemysPossibleHands` nevű függvény végzi.

```
function createEnemysPossibleHands(allCards, sevenCardsName) {
  let eCombination = [];
  let everyCards = [...allCards];
  let allCombinations = [];
  for (let comb of G.combination(everyCards, 2)) {
    allCombinations.push(comb.slice());
  }
  for (let comb of allCombinations) {
    eCombination.push(comb.concat(sevenCardsName.slice(2, 7)));
  }
  return eCombination;
}
```

Ez a függvény megkapja az összes kártyát, ami még a pakliban maradt, valamint a kiválasztott hét lapot. A maradék lapokon végigmegy egy ciklussal és kiválasztja belőle az összes 2-es lap kombinációját, szintén a `generatorics` JavaScript könyvtár segítségével. Ezután ehhez a 2 laphoz hozzáfűzi a kapott tömb utolsó 5 elemét, hiszen ezek a közös lapok, amikkel keze alakulhat ki ellenfelünknek. Ezekkel az értékekkel visszatér.

A `default` függvényben egy *foreach* ciklussal végig megyünk ezeken a 7 lapos kombinációkon. A cikluson belül hasonló módon, mint a felhasználó esélyeinek számításánál, legenerálom a 7-es kombinációkból az 5-ös kombinációkat, majd a legkisebb értékűvel számolok tovább. Így 990 értéket fogok kapni, az összes lehetséges 2 lapra, ami az ellenfélnél lehet. Ezután frontend-en már csak annyi a teendő, hogy összehasonlítsam a felhasználó lapjainak értékét ezzel a 990-el, és megkapom, hogy hány százalék esélye van nyerni a felhasználónak.

Ha még egy lapra várunk az asztalon, akkor az előbb említett lépéseket közrefogom egy *for* ciklussal, mely végigmegy a pakli lapjain és az adott lappal kiegészíti a tömböt. Hasonló módon, mint a felhasználó esélyeinek számításánál, annyi különbséggel, hogy itt ügyelni kell a lap tömbből való kitörlésére is, hiszem a számításokat végző függvények megkapják a 7 lap mellett a pakliban maradt lapokat is. A törlést a `removeItemOnce` függvény végzi, mely szintén a `helperFunctions`-ben található, hiszen mindkét modulban felhasználom.

```
export function removeItemOnce(arr, value) {
  let arrCopy = arr;
  var index = arrCopy.indexOf(value);
  if (index > -1) {
    arrCopy.splice(index, 1);
  }
  return arrCopy;
}
```

Flop esetében csak úgy mint eddig, két egymásba ágyazott *for* ciklussal valósítom meg a maradék két lap érkezését. Szintén a belső ciklus eggyel előrébbről indul, mint a külső az ismétlődés elkerülése végett. Ilyenkor az 5 laphoz mindkét ciklusnál hozzáfűzöm az adott kártyát, valamint kitörlöm a tömbből. Ezekre a tömbökre elvégzem ugyanazokat a lépéseket, mint eddig és megkapom az ellenfél összes lehetséges lapjára az értéket. Ugyanígy összehasonlítom a felhasználó várható értékeivel laponként, majd ábrázolom diagramon.

3.4. Kapcsolat a backend és a frontend között

A backend és a frontend közötti kapcsolatot két függvénnyel valósítom meg, amelyekben egy-egy POST kérés szerepel. Mindkét esetben a *body*-ban átadom a `sevenCards` objektumot, amely tartalmazza a felhasználó által kiválasztott lapokat. Az egyik esetben a `fetch`-ben a `mystrongest`, a másikban az `enemystrongest` szerepel. A backend mindig az 5000-es porton kommunikál a frontend-del.

```
backendOfMyCards(e) {
  e.preventDefault();
  fetch("http://localhost:5000/mystrongest", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ sevenCards: this.sevenCards }),
  })
    .then((res) => {
      return res.json();
    })
    .then((json) => {
      (this.myCards = json);
    });
}
```

A függvény visszatér a backend-től kapott adatokkal, valamint a `myCards` tömbbe kerülnek a felhasználó esélyei. Az ellenfél esélyeinél természetesen az ő esélyeivel tér vissza a függvény, és az `enemyCards` tömböt tölti fel.

Szerver oldalról a `root.js` fájlban található az a rész, amely a kapcsolatot végzi a kliens oldallal. Ide importáltam a `findStrongest.js` és `findEnemyStrongest.js` fájlokat. Itt két POST `response` található. A különbség a kettő között, hogy míg az egyik a fent említett fájlok közül ez előbbit, a másik az utóbbit valósítja meg és küldi vissza. Egy változóba eltárolom a frontend kérés body részét, erre meghívom az adott függvényt és ezzel vissza is küldöm a frontend-nek.

```
router.post("/mystrongest", (req, res, next) => {
  let cards = req.body;
  res.setHeader("Content-Type", "application/json")
  res.json(findStrongest(cards)).send();
});
```

4. fejezet

Teljesítmény optimalizálás

Miután az alkalmazás hozzávetőlegesen elérte végső fázisát, és a funkciók nagy része működött, akkor derült fény arra a problémára, amit már előre lehetett sejteni, ez pedig a túlságosan hosszú futási idő. Mivel nagyon sok számítást kell elvégeznie a programnak, így elkerülhetetlen, hogy megfelelő optimalizálás nélkül lassan működjön az alkalmazás. Ebben a fejezetben ezt a problémakört fejtem ki.

A lassúság funkcionális problémát nem okoz, mivel a célként kitűzött feladatokat a program elvégzi. Ez inkább a felhasználási mód rovására megy. Az alapvető elképzelés ugyanis az volt, hogy egy olyan végeredményt kapok, amit egy asztalnál, élő pókerezés közben, úgynevezett csaló szoftverként tudok használni, ami elvégzi a gondolkodást helyettem. Hogyha 30 percet kell várni arra, hogy a flop-on kiszámolja a program, milyen esélyeim vannak, akkor azt a gyakorlatban nem lehet effektíven felhasználni.

4.1. Optimalizálási módok

Ahhoz, hogy optimalizálni tudjam az alkalmazást, először meg kell vizsgálni, mi okozhatja a lassulást, melyek a szűk keresztmetszetek, továbbá, hogy hol lehet változtatni a kódon ezek kiküszöböléséhez. A leginkább akkor szembetűnően lassú a program, mikor az ellenfél esélyeit vizsgálom, ezért optimalizálás szempontjából kizárólag a back end-re, azon belül is leginkább a `findEnemyStrongest.js` forráskódjára fókuszáltam.

Az alábbiakban felsorolom azokat a tényezőket, amiket változtatni terveztem a program gyorsítása érdekében. Ezeket később pontról pontra kifejtem, bemutatva az eredeti és a módosított megoldásokat, valamint megpróbálom szemléltetni az elért gyorsulást, amennyiben tapasztalható egyáltalán.

- *Adathalmaz átszervezése*: Mivel az adathalmazban az elemek kártya értékek szerint voltak sorba rendezve, így eszerint kellett rendeznem az asztalon lévő lapokat is. Ehelyett ABC sorrendbe rendezem a gyorsabb keresés érdekében.

- *Tömbök előre allokálása és indexelése*: Mivel az, hogy adott esetben hány értéket kapunk az ellenfél lehetséges lapjaira ismert, ezért ahelyett, hogy minden esetben hozzá fűznék egy értéket a tömb végéhez, inkább előre definiálom azt, és az aktuális elemet csak beállítom.

- *Függvények kiszervezése*: A profilozást, és ezzel a szűk keresztmetszetek felderítését segíti, hogy ha a forráskód megfelelően tagolt. Ezzel közvetve több lehetőség adódik a teljesítmény javítására.

A következő szakaszokban ezen optimalizálási lehetőségek megvalósítása kerül bemutatásra.

4.1.1. Adathalmaz átszervezése

Az adathalmazom alapvetően kártyaértékek szerint volt rendezve. Egyrészt azért, mert a forrásban is így volt tárolva, másrészt az emberi szem számára is könnyebben értelmezhető így. Tehát a *royal flush* megfelelője a "AKQJTF" karakterlánc volt, amelyben az F betű azt jelöli, hogy figyelembe kell venni, hogy egyező színűek a lapok. Mivel az adathalmazban így voltak rendezve az elemek, az aktuálisan kiválasztott lapokat is ilyen módon kellett sorba állítanom. Első körben logikusnak tűnt ez a megoldás, viszont a megvalósítása kicsit hosszú programkódot eredményezett. Ennek néhány részlete a következőképpen néz ki.

```
1  function sortCardOrder(string, colors) {
2    let ordered = "";
3    let timesArray = [
4      (string.match(/A/g) || []).length,
5      (string.match(/K/g) || []).length,
6      .
7      .
8    for (let i = 0; i < timesArray[0]; i++) {
9      ordered += "A";
10   }
11   for (let i = 0; i < timesArray[1]; i++) {
12     ordered += "K";
13   }
14   .
15   .
16   if (colors["C"] == 5) {
17     ordered += "F";
18   } else if (colors["H"] == 5) {
19     ordered += "F";
20   } else if (colors["S"] == 5) {
21     ordered += "F";
22   } else if (colors["D"] == 5) {
23     ordered += "F";
24   }
25
26   return ordered;
27 }
```

Látható, hogy mind a 13 lapot külön kellett vizsgálnom, és úgy összefűzni egy szöveglánccá. A függvény végén az egyező színeket számolom össze. Amennyiben létezik olyan lehetőség, hogy 5 egyforma színű kártya van, akkor hozzáfűzök a szöveg végére egy "F" karaktert.

Először az adathalmazt kellett módosítanom. Mindössze annyi történt, hogy az elemek nem értékek szerint lettek sorba rendezve, hanem ABC sorrend szerint. A fenti példánál maradva, így a "AKQJTF"-ből "AFJKQT" lett. Ennek köszönhetően nem volt szükség az összes kártyát külön vizsgálnom.

A módosított függvény elején szintén vizsgálom a színeket, viszont utána már csak beépített függvényeket használok. A sorba rendezésre a `sort()` függvény használható, viszont az csak egy tömb elemeit rendezi sorba, ebben az esetben viszont egy szöveg karaktereit kell rendezni. Éppen ezért használom még a `split()`, illetve a `join()` függ-

vényeket. Az előbbi feldarabolja az elemeket jelen esetben karakterekre, hogy azokat tudjam rendezni. Az utóbbira azért van szükség, hogy a feldarabolt, sorba rendezett értékeket újra összefűzze egy szöveglánccá. Végeredményben ugyanannyi elemű tömböt kapunk vissza, mint ahány elemű a függvények hívása előtt volt. Az optimalizált változatból egy kódrészlet az alábbi.

```
function sortCardOrder(string, colors) {
    let ordered = string;
    if (colors["C"] == 5){
      ordered += "F";
    } else if (colors["H"] == 5) {
      ordered += "F";
    } else if (colors["S"] == 5) {
      ordered += "F";
    } else if (colors["D"] == 5) {
      ordered += "F";
    }
    return ordered.split('').sort().join('');
}
```

4.1.2. Tömbök előre allokálása és indexelése

A tömbök létrehozására a `push` függvényt használom. Minden lehetséges lapkombináció kiszámításánál ezzel a függvénnyel töltöm fel a tömböket, amikkel majd visszatérek és átadom a front end-nek. Ezt a függvényt, például *flop*-on az ellenfél esélyeinek számításánál 1'070'190-szer hívom meg. Az volt a feltevésem, hogy ez a módszer lényegesen lassítja a szoftver működését.

Mivel minden vizsgálatnál adott, hogy hány elemű lesz a tömb, így meg lehet oldani, hogy a tömböket előre allokálom. Az elágazások elején definiálom a tömböket, hogy hány eleműek legyenek, majd a feltöltéskor mindig az éppen aktuális elemet állítom be. A vizsgálatoknál több helyen *foreach* ciklust használok, ennek a hátránya viszont, hogy nem tudom a tömböket indexelni. Hogyha előre allokálom a tömböket, akkor szükségem van az indexelésre. Ennek érdekében azokon a helyeken, ahol meg szeretném ezt valósítani át kell alakítanom a *foreach* ciklusokat egyszerű *for* ciklusokká.

Hogy minél kevesebb programkódot kelljen bemutatnom, kizárólag azokat a részleteket szemléltetem, amikor mindegyik kártya ki van osztva, ugyanis ilyenkor a legrövidebb a kód. Alább látható ennek az optimalizálás előtti változata.

```
let result = [];
if (sevenCards.length == 7) {
  let eCombination =
    createEnemysPossibleHands(everyCards, sevenCardsName);
  for (const comb of eCombination) {
    result.push(findStrongestOfEnemysCards(comb));
  }
  return result;
}
```

Ezek alapján az elágazás előtt nem csak definiáltam a `result` tömböt, hanem meg is határoztam a méretét. Mikor ez megvolt, át kellett alakítanom a belső *foreach* ciklust, hogy az indexelést el tudjam végezni. Ennek a két feltételnek kellett teljesülnie, hogy megvalósítsam az optimalizálásnak ezt a tényezőjét.

```
let result = new Array(990);
```

```
2  if (sevenCards.length == 7) {
3    let eCombination = createEnemysPossibleHands(everyCards,
       sevenCardsName);
4    for (let i = 0; i < eCombination.length; i++) {
5      result[i] = findStrongestOfEnemysCards(eCombination[i])
6    }
7    return result;
8  }
```

4.1.3. Függvények kiszervezése

A program forráskódjában számos helyen nagyon hosszú programkód blokkok szerepeltek. A terv az volt, hogy ezeket külön függvényekbe szervezem ki. Esztétikai és átláthatósági problémákat okoz, ha egy nagy, ömlesztett kódból áll az egész program.

Próbáltam minél rövidebb kódot kapni, ennek érdekében az első lépésem az volt, hogy megvizsgáltam melyek azok a programrészletek, amelyeket a `findStrongest.js` és a `findEnemyStrongest.js` állományokban is felhasználok. Létrehoztam a `helperFunctions.js` fájlt, amelybe kiszerveztem ezeket a részleteket külön függvénybe, majd a többi fájlba már csak importálnom kellett és fel is tudtam használni. Az alábbi kódokat szerveztem ki külön függvénybe és helyeztem a `helperFunctions.js` fájlba.

- `removeItemOnce`: Ez a függvény megkap egy tömböt és egy értéket, majd kitörli a kapott tömbből a kapott értéket. Ezután visszatér a szűkített tömbbel.
- `sortCardOrder`: Ez az adathalmaz átszervezésénél tárgyalt függvény. Eleinte ez is ömlesztve, ismétlődve szerepelt mind a két állományban. Az optimalizált változatot is kiszerveztem.
- `getOnlyName`: Front end-en egy objektumban tárolom azokat a kártyákat, melyeket kiválasztott a felhasználó. Back end-en az objektumból csak egy értékre van szükségem, a kártyák neveire, azt viszont egy tömbben is el tudom tárolni. Ez a függvény pont ezt hivatott megvalósítani. Szintén a `helperFunctions.js`-be került.
- `getJustDeck`: Ez egy gyakran használt részlet. Megkapja az összes kártyát, valamint a kiválasztott lapokat. Visszatér ennek a különbségével, tehát a pakliban maradt kártyákkal.

A közös részek kiszervezésével lényegesen rövidebb lett a programkód. Ezután az összes fájlban igyekeztem a függvények hosszát a minimumra csökkenteni. A leginkább szembetűnő változás a `findStrongest.js` függvényében történt. Az optimalizálás előtt 80 soros volt, az optimalizálás során pedig sikerült lecsökkenteni 30 sorosra. Ezt az összesen 110 sor programkódot nem szemléltetném, inkább csak a leglényegesebb részt. Az eddigi 2 külön függvényből lett 5, amiből a legtöbb változást a `createStrongest` függvény hozta.

```
1  function createStrongest(combinations){
2    let rawdata = fs.readFileSync("data.json");
3    let strenghtOrder = JSON.parse(rawdata);
4    let result =  [];
5    for (let combination of combinations) {
```

```
6       let nameString = "";
7       let colors = { C: 0, S: 0, H: 0, D: 0 };
8       for (let card of combination) {
9         nameString += card[0];
10        colors[card[1]] += 1;
11      }
12      let ordered = sortCardOrder(nameString, colors);
13      result.push(strenghtOrder.cardStrenght[ordered]);
14    }
15    return Math.min(...result);
16  }
```

Ez a részlet valósítja meg azt, amit a felhasználó magától is el tud dönteni. Megkap 25, 5 elemű tömböt, ezek a 7 lerakott lapból kiválasztható 5 lap. Ezeket mind megkeresi az adathalmazban, majd a legkisebb értékűvel tér vissza. Tehát megadja, hogy a választható 7 lap közül melyik 5 lap az, amelyik a legerősebb, ez lesz a játékosé. Ezt a keresést akkor is el kell végezni, amikor ismerjük az összes lapot, valamint akkor is, mikor csak 5-öt vagy 6-ot. Ezek alapján a `default` függvényben ez a részlet háromszor szerepelt. Éppen ezért sikerült ezzel elérni szembetűnően rövidebb kódot.

4.2. Eredmények kiértékelése

Annak érdekében, hogy a futási időről készült méréseim hitelesek legyenek, kitűztem néhány előfeltételt. Mivel nem ugyanannyi időbe telik a programnak elvégezni a számításokat különféle lapok esetében, így minden mérésnél azonos lapokat választottam ki, hogy a mérést pontosnak tekinthessem. Ezek a következőek voltak. Saját lapjaimnak a pikk ászt, és pikk királyt választottam. A flop a pikk ász, pikk 10-es, és pikk 2-es volt. Turn-ön a treff 10-es, majd végül river-nél a treff 7-es. Minden mérésnél ezeket a lapokat választottam ki, illetve mindig az éppen aktuálisakat. Természetesen a flop vizsgálatánál a saját lapjaimon kívül csak három lapot raktam le az asztalra.

Másik tényező, amit fontosnak tartottam egységesíteni az a hardver, amin futtatom az alkalmazást. A méréseket ugyan azon a számítógépen végeztem. Csak a kliens, a szerver, és a böngésző futott a gépen, ezzel is elősegítve azt, hogy más, háttérben futó alkalmazás nem befolyásolja a mérést.

4.1. táblázat. Futási idők módosításonként

Futási idő [s]	Flop		Turn		River	
	Eredeti	Optimalizált	Eredeti	Optimalizált	Eredeti	Optimalizált
Adathalmaz átszervezése	2143.5615	2008.4787	117.8541	95.9102	2.0827	2.5742
Tömbök előre allokálása és indexelése	2143.5615	2119.8991	117.8541	114.0629	2.0827	1.9105
Függvények kiszervezése	2143.5615	2787.2205	117.8541	122.7858	2.0827	2.8656

A 4.1. táblázatban láthatóak a futási idők mind a három módosításra levetítve, valamint a preflop utáni licitkörökre. Mint azt a táblázat is jól mutatja, sajnos nem sikerült a várt gyorsulást elérni, viszont érdekesek az eredmények, amiket kaptam.

A legnagyobb javulást az adathalmaz átszervezésével értem el. A river-nél egy fél másodperces lassulás tapasztalható, viszont a turn esetében már 22 másodperces javulást látunk. Futási idő szempontjából a flop a legkritikusabb, itt pedig több, mint két perces javulást eredményezett a módosítás. Ebből a szempontból elhanyagolható, a fél másodperces lassulást river-nél, hiszen az kevésbé feltűnő a felhasználónak, mint a flop számításának lassúsága.

A tömbök előre allokálása és indexelése minimális javulást hozott. River esetében megközelítőleg egy tizedmásodperces, turn-nél 3 másodperces, flop-nál pedig 24 másodperces gyorsulást tapasztaltam. Ezek alapján elmondható, hogy a `push` függvényhívás nem lassítja lényegesen a futási időt.

A függvények kiszervezése kifejezetten lassította a programot. A másik célom viszont a programkód jobban olvashatóvá és esztétikusabbá tétele volt. Flop esetében több, mint 10 perces lassulást hozott a függvények kiszervezése. Alap esetben nem kellene érezhetőnek lennie, hogy egy programrészletet közvetlen futtatunk, vagy egy külön függvénybe írjuk és úgy hívjuk meg. Az én esetemben viszont, a flop-nál maradva, ezt a függvényt 1'070'190-szer kell meghívnom. Valószínűleg ha ennyiszer hívunk meg egy függvényt, akkor ott már szembetűnő lesz ez az időtartam. Ez okozhatja jelen esetben is a lassulást. Ezeket figyelembevéve két lehetőségem volt, az egyik, hogy elvetem ezt a megoldást és a `default` függvényben marad ugyan az a programkód többször is. A másik, hogy elfogadom az ezzel járó lassulást, cserébe esztétikusabb lesz a kódsor. Mivel nem tervezem az alkalmazást publikálni, így inkább a mellett döntöttem, hogy a szakdolgozatom szempontjából hasznosabb, ha inkább az átlátható programkód mellett maradok.

4.3. Működés ellenőrzése

Az alkalmazás véleményem szerint elérte végső formáját. Nem maradt más hátra, mint a tesztelés, hogy megbizonyosodjak, az alkalmazás rendben működik. Ehhez, a szakdolgozat elején említett hasonló szoftverek egyikét hívtam segítségül. Azt választottam, amelyikben ki lehet választani azt, hogy hány játékossal játszunk. A szoftver alap működését meglehetősen egyszerűen lehet tesztelni ennek a segítségével. Mindössze annyit kell megvizsgálnom, hogy az általam készített program ugyanazt a százalékos végeredményt adja-e, mint a referenciaként használt.

Többféle leosztást is összehasonlítottam, mindenhol ugyanazt az eredményt kaptam, viszont jelen esetben csak egyet szemléltetnék. Próbáltam a lapokat véletlenszerűen kiválasztani. Jelen esetben a felhasználó lapjai, a káró 7-es, és 2-es. A flop a treff 7-es, kör 2-es, és a treff ász. Így tehát két párunk alakult ki, ami kifejezetten jó eset, azaz magas nyerési esélyre számíthatunk. A Pokerstars által szponzorált alkalmazás azt a végeredményt adta, hogy 84,87% esélyünk van nyerni, az ellenfélnek pedig 13,86%. A saját szoftveremnél ugyanezeket a lapokat választottam ki. Eredménynek ugyancsak 84,87%-ot kaptam a saját nyerési esélyeimre, így az ellenfélé 100%-84,87%=13,86%. Így tehát kijelenthetjük, hogy a százalékos esélyek számítása jól működik az alkalmazásomban flop esetében.

A következő helyzet, ahol vizsgálni kell a helyes működést, az a turn. Negyedik lapnak a pikk ászt választottam. Ebben az esetben a board-on kialakult egy pár, ami a felhasználó egyik párját értéktelenné teszi, mivel az ász pár és 7-es pár sokkal erősebb, mint a 7-es és 2-es párok. Ebben az esetben az várjuk, hogy jóval kevesebb lesz a felhasználónak a nyerési esélye, mint a flop-nál volt. Az internetes szoftver most 72.62%-

ot mutat. Kiválasztottam a saját alkalmazásomban is a pikk ászt, ami így 72,6%-ot írt ki. Látható, hogy a két program számítása között két százados különbség van. Egyrészről ez a felhasználó szemszögéből elhanyagolható, másrészt ezt a különbséget a feltételezett számítási módszereknek, valamint a lehetséges kerekítési különbségeknek tudom be. A többi esetben ilyen eltérést nem tapasztaltam.

Az elvégzett vizsgálatok alapján az alkalmazás az elvártaknak megfelelően működik. A flop-nál és a turn-nél is megbizonyosodtam arról, hogy a kapott esélyek helyesek. Már csak az utolsó lap van hátra, a river, ahol a treff királyt választottam. Így három egyforma színű lap van az asztalon, ami esélyt ad arra, hogy valakinek flush-e alakuljon ki. Ezért várhatóan a játék utolsó százalékos esélye még kevesebb lesz a korábbiaknál. Az interneten futtatva 63,33% nyerési esélyt kaptam, ami az általam elkészített verziónál is megegyezett. A fentieket figyelembe véve kijelenthető, hogy a szakdolgozatom webalkalmazása jól működik, tehát helyes értékeket számol. A 4.1. és a 4.2. ábrán szemléltetem a két alkalmazás számolt értékeit river esetén.

4.1. ábra. A Pokerstars alkalmazásának számítási eredménye river-nél [10]

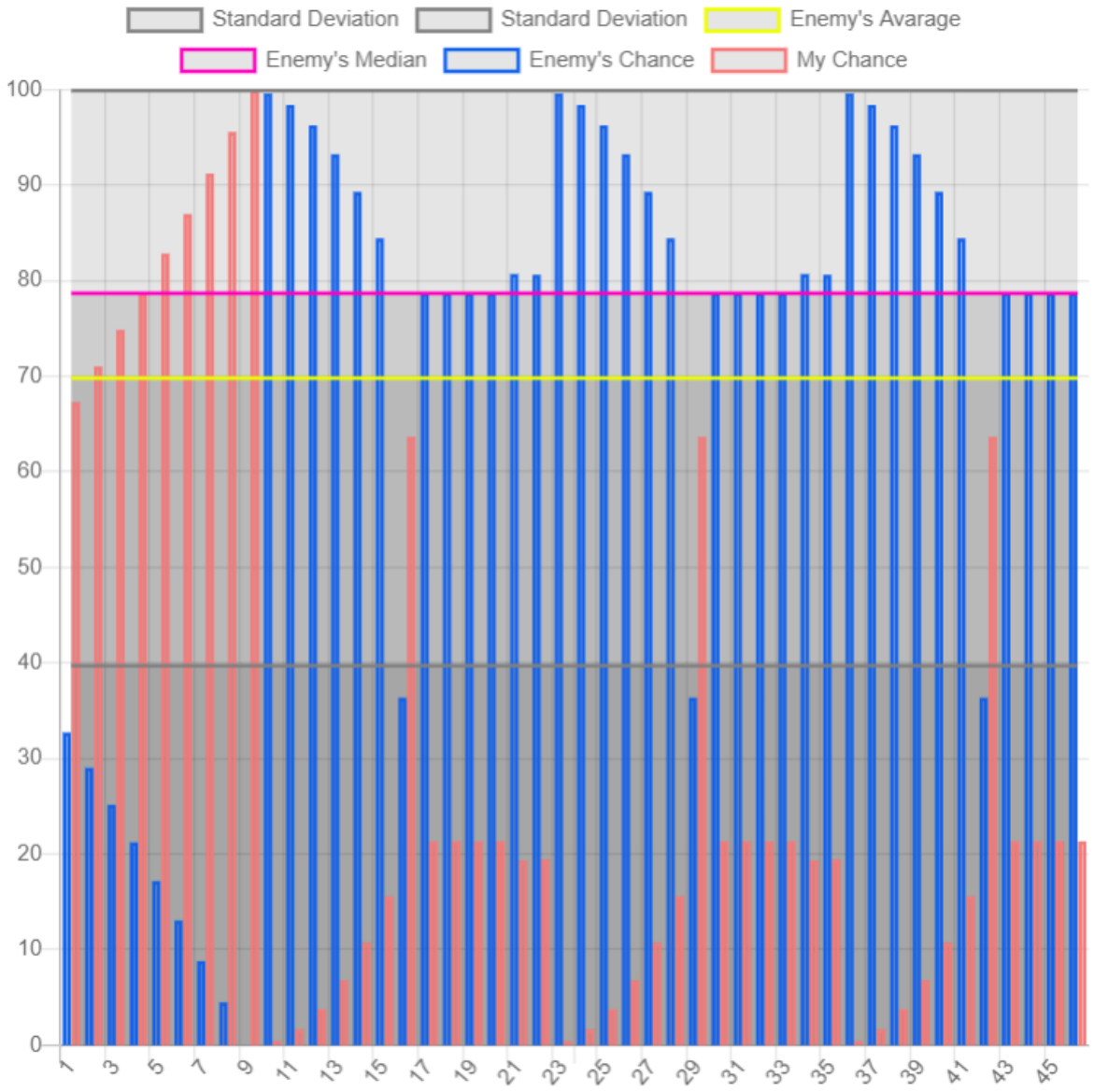
A diagramm helyes működésének a vizsgálata már bonyolultabb folyamat. Ebben az esetben azt a megoldást választottam, hogy egy olyan kártyakombinációt választok, amelynek viszonylag egyedi diagrammja lesz. Például turn-nél négy treff van az asztalon, a felhasználó kezében két káró van és sem párt, sem erősebb kezet nem alkotnak a lapok. Ebben az estben, ha még egy pikk érkezne river-nél, akkor a board-on lévő lapok flush-t alkotnának, amit a program nyereségként ábrázol. Tehát a diagrammon azt kell látnunk, hogy a maradék pikk lapok esetén nagyon magas a nyerési esélye a felhasználónak.

Ennek megfelelően kiválasztottam a felhasználó lapjainak a káró 2-est és 3-mast, az asztalra pedig a treff ászt, királyt, dámát, és bubit. Ezután legeneráltam a diagrammot, ami a 4.3. ábrán látható. Az `everyCards` tömbben növekő sorrendben helyezkednek el a lapok, először a treff, majd a káró, kör, végül pedig a pikk. A lapokból az látszik, hogy ha bármelyik treff következik a river-nél, akkor a board-on flush alakulna ki, ha a treff 10-es érkezne, akkor pedig royal flush. Ezen kívül, ha bármilyen 10-es érkezne, akkor sor alakulni ki a board-on, ami szintén a felhasználónak kedvez. A kézben lévő lapok a lehető legalacsonyabbak, illetve nem alakulhat ki belőle flush, sor, de még csak

4.2. ábra. Az alkalmazásom számítási eredménye river-nél

egy drill sem. Ha nem érkezik több treff, akkor a legmagasabb kezünk, ami lehet, az egy hármas pár, viszont ez sem mondható erősnek, mivel az asztalon csupa magas lap van.

Ezeket a tényeket mutatja be a 4.3. ábrán kirajzolt diagramm. A piros oszlop a felhasználó nyerési esélyeit mutatják, a kékek pedig az ellenfelét. Jól látszik, hogy az első kilenc kártyánál nagyon magas a nyerési esélye a felhasználónak. Ezek a maradék treff lapok a pakliban, az utolsó pedig a treff 10-es, aminél 100% az esély a győzelemre. A többi lapra három kivétellel mindegyiknél az ellenfélnek nagyobb az esélye. Ez a három lap a maradék 10-esek a pakliban, amelyekkel sor alakulna ki a board-on. A lila konstans mutatja az ellenfél esélyeinek mediánját, a sárga az átlagát, a szürkék pedig a szórását. Ezek mind segítik a felhasználót a döntéshozásban.



4.3. ábra. A kirajzolt diagramm turn-nél

5. fejezet

Összefoglalás

Számtalan tapasztalatot szereztem a szakdolgozatom megírása közben. Mivel még nem foglalkoztam webalkalmazás fejlesztéssel, így rengeteg tanulás előzte meg a program elkészítését. Megismerkedtem a Vue JS-el, a Node JS-el, és más nyelvekkel vagy keretrendszerekkel, amiket a szakdolgozatom készítése során felhasználtam. Szerencsére mindegyikük jól dokumentált környezet, így sok segítséget találtam hozzájuk az Interneten. Mivel napjainkban elterjedtek a webes alkalmazások, így remélem a jövőben is hasznomra válnak majd ezek az ismeretek.

Olyan témát sikerült választanom, amely az életben is közel áll hozzám, így nagy lelkesedéssel álltam hozzá az egész munkafolyamathoz. Megtaláltam a párhuzamot a póker és a matematika, statisztika között, amit véleményem szerint sikerült jól megvalósítani és bemutatni. A célomat, hogy egy "csaló programot" hozzak létre, csak megszorításokkal sikerült elérnem. Az alkalmazás tökéletesen megvalósítja az elképzeléseimet, kizárólag a futási idő akadályozó tényező az életben való felhasználásában. Hamar világossá vált számomra, hogy a fejlesztésnek egy nagyon fontos része a tervezés, hiszen az adja a program alapját. Érdekes volt számomra az is, hogy egy olyan alkalmazást sikerült létrehoznom, amelyhez hasonlót világhírű pókerrel foglalkozó társaságok alkottak meg.

A webalkalmazás gyakorlatban való felhasználásra a következők a megállapításaim. Mindenképpen tovább kellene optimalizálni a futási időt, hogy gördülékenyebben lehessen használni. Továbbá, hasznos lehet reszponzívvá tenni az alkalmazást, hogy mobilról, táblagépről és más eszközökről is felhasználóbarát megjelenést kapjon. Ha pedig a felhasználók körét szeretnénk bővíteni, akkor az általam használt módszerrel más pókerfajták szimulációját is el lehetne készíteni. Ha csak a Texas Hold'Em-nél maradunk, akkor a fix limit és a pot limit változatát is meg lehetne valósítani, vagy akár az Omaha-t, 5 lapos pókert is.

Irodalomjegyzék

[1] The good and the bad of firebase backend services. https://www.altexsoft.com/blog/firebase-review-pros-cons-alternatives/, [Online, 2020.11.06].

[2] What is front-end development: Key technologies and concepts. https://www.altexsoft.com/blog/front-end-development-technologies-concepts/, [Online, 2020.01.30].

[3] The good and the bad of node.js web app development. https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-node-js-web-app-development/, [Online, 2019.10.21].

[4] The good and the bad of vue.js framework programming. https://www.altexsoft.com/blog/engineering/pros-and-cons-of-vue-js/, [Online, 2019.09.19].

[5] Dan Harrington Bill Robertie. Hold'em versenystratégia, 2006.

[6] Poker odds calculator powered by cardschat. https://www.cardschat.com/poker-odds-calculator.php.

[7] Chart.js. https://en.wikipedia.org/wiki/Chart.js, [Online, 2022.01.20].

[8] Megan Jackson. Data flow diagram (dfd) tutorial: Texas hold 'em. https://seilevel.com/requirements/data-flow-diagram-dfd-tutorial-texas-hold-em, [Online, 2019.].

[9] Cactus Kev. Enumerating five-card poker hands. http://suffe.cool/poker/7462.html?fbclid=IwAR2c67leOJRIyx7pSq17k5KOLYxnCgIMjJaH04nsRI1OLl3qX9kCwGk5vQw, [Online, 2006.].

[10] Texas hold'em poker odds calculator sponsored by pokerstars. https://www.pokernews.com/poker-tools/poker-odds-calculator.htm.

[11] Preflop esélyek ábra. https://qph.cf2.quoracdn.net/main-qimg-1591e7d9bf5a26292b44fe33bc612908-pjlq, [Online, 2020.].

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).