

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK



SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

Geometriai egyensúlypontok vizualizációja

ÁLLAMVIZSGA DOLGOZAT

Témavezető:
Olteán-Péter Boróka,
Tanársegéd
Dr. Farkas Csaba,
Egyetemi tanár

Végzős hallgató:
Fodor Lehel

2023

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,
SPECIALIZAREA INFORMATICĂ



UNIVERSITATEA
SAPIENTIA

Vizualizarea punctelor de echilibru geometrice

LUCRARE DE LICENȚĂ

Coordonator științific:
Olteán-Péter Boróka,
Asistent universitar
Dr. Farkas Csaba,
Profesor universitar

Absolvent:
Fodor Lehel

2023

**SAPIENTIA HUNGARIAN UNIVERSITY OF
TRANSYLVANIA
FACULTY OF TECHNICAL AND HUMAN SCIENCES
INFORMATICS SPECIALIZATION**



SAPIENTIA
HUNGARIAN UNIVERSITY
OF TRANSYLVANIA

Visualization of geometric equilibrium points

DIPLOMA THESIS

Scientific advisor:
Olteán-Péter Boróka,
Assistant professor
Dr. Farkas Csaba,
Full professor

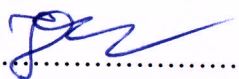
Student:
Fodor Lehel

2023

Declarație

Subsemnatul/a FODOR LEHEL, absolvent(ă) al/a specializării
INFORMATICA, promoția 2023 cunoscând
prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională a
Universității Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare
de licență/proiect de diplomă/disertație se bazează pe activitatea personală, cercetarea/proiectarea
este efectuată de mine, informațiile și datele preluate din literatura de specialitate sunt citate în
mod corespunzător.

Localitatea, TÂRGU-MUREȘ
Data: 14-06-2023

Absolvent
Semnătura.....

UNIVERSITATEA „SAPIENTIA” din CLUJ-NAPOCA
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș
Programul de studii: Informatică

Viza facultății:

LUCRARE DE DIPLOMĂ

Coordonator științific:

Dr. Farkas Csaba

Îndrumător:

Oltean-Péter Boróka

Candidat: Fodor Lehel

Anul absolvirii: 2023

a) Tema lucrării de licență:

Vizualizarea punctelor de echilibru geometrice

b) Problemele principale tratate:

Problemele principale tratate sunt următoarele: înțelegerea conceptelor fundamentale ale teoriei jocurilor din perspectivă matematică, cu accent special pe punctele de echilibru Nash. Utilizarea algoritmului Lemke și Howson, și necesitatea vizualizării rezultatelor sale.

c) Desene obligatorii:

Vizualizarea rezultatelor algoritmului Lemke and Howson

Grafice despre structura codului

d) Softuri obligatorii:

Implementarea algoritmului Lemke and Howson în Python.

Un software care include vizualizarea datelor, iar datele sunt generate de algoritmul Lemke and Howson.

e) Bibliografia recomandată:

Kreps, D. M. (1989). Nash equilibrium. Game theory, 167-177.

Liu, K. (2013). Lemke and Howson Algorithm.

Zagare, F. C. (1984). Game theory: Concepts and applications.

<https://github.com/s3rvac/lemke-howson.git>

f) Termene obligatorii de consultații:

g) Locul și durata practicii: Universitatea „Sapientia” din Cluj-Napoca,

Facultatea de Științe Tehnice și Umaniste din Târgu Mureș, sala / laboratorul.....

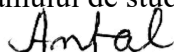
Primit tema la data de: 20.06.2022

Termen de predare: 07.07.2023

Semnătura Director Departament



Semnătura responsabilului
programului de studiu



Semnătura coordonatorului



Semnătura candidatului



Kivonat

Egy kétszemélyes játék esetében mindkét játékosnak nyilvánvalóan az a célja, hogy betartva a játék szabályait és előírásait megnyerje a játékot. Ennek érdekében mindkét játékos ahhoz a stratégiájához folyamodik, amelyik segítségével a legnagyobb hasznot tudja megszerezni magának. Viszont vannak olyan játékok, amikor ha egyik játékos ha alkalmazza a statégiáját, az a másiknak is a javára válik, ami nyilván az előbbi játékosnak kedvezőtlen. Ezért mindkét játékos meg kell határozza, hogy a rendelkezésére álló stratégiákat az esetek mekkora relatív gyakoriságában kell alkalmaznia, hogy maximalizálja az esélyeit a győzelem elérésében. Erre a problémára választ ad **John Forbes Nash** amerikai matematikus nevéhez fűződő **Nash-egyensúlypont**. Ugyanis kétszemélyes játékok esetén azt a stratégiaegyüttest, amely esetében egyik játékos sem tudja a stratégiája változtatásával növelni a nyereségét, ha az ellenfele sem vált stratégiát Nash-egyensúlypontnak nevezzük.

Dolgozatomban a fentebb említett egyensúlypont meghatározása a főcél úgy matematikailag levezetve, mint programozás szempontjából a **fogolydilemma** problémát felhasználva. Emellett kitérek a játékelméleti alapfogalmak ismertetésére, a fogolydilemma tárgyalására, a felhasznált algoritmus bemutatására, valamint az eredmény értelmezésére is. Eme egyensúlypont kiszámítására a **Lemke-Howson** algoritmust használom. A megkapott egyensúlypontot pedig Python programozási nyelv segítségével vizualizálom. A célokhoz tartoznak még a következők: az olvasó bevezetése a játékelmélet világába illetve segítségnyújtás az átlagos matematikai-és informatikai tudással rendelkező felhasználóknak a Nash-egyensúlypont megértésében.

Kulcsszavak: kétszemélyes játék, stratégia, John Forbes Nash, Nash-egyensúlypont, fogolydilemma, Lemke-Howson algoritmus, vizualizáció.

Rezumat

Într-un joc cu doi jucători, scopul ambilor jucători este evident să câștige jocul respectând regulile și regulamentele jocului. Pentru a obține asta, ambii jucători recurg la strategia cu care pot obține cel mai mare beneficiu pentru ei înșiși. Totuși, există jocuri în care dacă un jucător își aplică strategia, îl avantajează pe celălalt jucător, ceea ce este evident nefavorabil pentru jucătorul menționat anterior. Deoarece, ambii jucători trebuie să determine frecvența relativă de utilizare a strategiilor disponibile pentru a-și maximiza șansele în obținerea victoriei. Pentru această problemă oferă răspuns **punctul de echilibriu Nash**, care este în relație cu numele lui **John Forbes Nash**, un matematician de origine americană. Pentru jocuri de două persoane acel set de strategie la care niciun jucător nu își poate mări profitul schimbându-și strategia sa, dacă nici adversarul său nu schimbă strategie numim punct de echilibriu Nash.

Scopul principal în lucrarea mea este determinarea punctului de echilibriu menționat de mai sus atât din punct de vedere matematic cât și punct de vedere al programării, folosind problema **dilema prizonierului**. În plus, scriu și despre următoare subiecte, și anume: explicarea concepte de bază ale teoriei jocurilor, prezentarea dilema prizonierului și algoritmului folosit și interpretarea rezultatului. Pentru calcularea acestui punct de echilibriu aplic algoritmul **Lemke-Howson**. Punctul de echilibriu obținut vizualizez folosind limbajul de programare Python.

Fac parte în scopuri și următoarele: introducerea cititorului în lumea teoriei jocurilor și ajutarea utilizatorii cu abilități medii de matematică și informatică în înțelegerea echilibrului Nash.

Cuvinte de cheie: joc de două persoane, strategie, John Forbes Nash, punct de echilibriu Nash, algoritmul Lemke-Howson, vizualizare.

Abstract

In a two-person game the purpose of both players is obviously to win the game by following the game's rules and regulations. Therefore, both of the players resorts to the strategy that results their the gratest benefit. However, there are games, when if one of the players applies their strategy, it also benefits the other, which is obviously unfavorable for the first mentioned player. For this reason, both players must determine the relative frequency of using their available strategies to maximize their chances for the victory. This problem is answered by the **Nash equilibrium point**, named after the American mathematician, **John Forbes Nash**. In two-person games, that set of strategy in which neither of the players can increase their profit by changing their strategy, in case of their opponent doesn't change their strategy, is called Nash equilibrium point.

In my thesis the main purpose is to determine the equilibrium point mentioned above, mathematically and also in programming point of view, using the **prisoner's dilemma** problem. Besides that I will cover the following topics as well: the introduction of basic concepts of game theory, the prisoner's dilemma, the presentation of applied algorithm and the interpretation of the result. To compute this equilibrium point I apply the **Lemke-Howson** algorithm. For the visualization of obtained equilibrium point I use Python as programming language.

Also belong to the goals to introduce the reader to the world of game theory and to help users with average mathematical and information technology skill to understand the Nash equilibrium.

Keywords: two-person game, strategy, John Forbes Nash, Nash equilibrium point, Lemke-Howson algorithm, visualization.

Tartalomjegyzék

1. Bevezető	10
2. Játékelméleti alapfogalmak	12
2.1. Stratégia	12
2.2. Kifizetési-mátrix	12
2.3. Zéró összegű- és pozitív összegű játékok	12
2.4. Kooperatív- és nem kooperatív játékok	13
2.5. Degenerált-és nem degenerált játékok	13
3. John Nash és a Nash-egyensúlypont	14
3.1. John Nash	14
3.2. Nash-egyensúlypont	15
3.3. A fogolydilemma	15
4. A Lemke-Howson algoritmus elméleti- és matematikai leírása	18
4.1. Az algoritmus ismertetése	19
4.2. A eredmény kiszámítása és értelmezése	25
5. A felhasznált algoritmus megközelítése programozás szempontjából	28
6. Az egyensúlypont vizualizációja	39
6.1. Adatok és eredmény vizualizációja	39
6.2. A három pont közötti összefüggés ábrázolása	41
6.3. Az egyensúlypont összefüggése a derékszögű háromszöggel	42
7. Összegzés és továbbfejlesztési lehetőségek	44
Köszönetnyilvánítás	45
Irodalomjegyzék	46

1. fejezet

Bevezető

Az emberiséget nemcsak napjainkban, de a régebbi időkben is kikapcsolódás, szórakozás gyanánt foglalkoztatta a "játék" fogalma; különösképp a kiskorúak/tinédzserek/fiatalkorúak korosztályát illetően, de egy 40-es éveit töltő felnőtt vagy egy idősebb egyén esetében is helyet kap a játszásra szánt idő.

A játék egy olyan tevékenység, amelyet egy vagy több résztvevő tud végezni, valamint igényelhet bizonyos kellékeket. Szórakoztató, közösségépítő funkciója miatt kedvelt szabadidős tevékenység, fejleszti a logikát, képzelőerőt, memóriát, gondolkodóképességet. Vannak ugyan olyan játékfajták (pl. a puzzle), amihez elegendő pusztán egy játékos, de véleményem szerint sokkal érdekesebbek a többszemélyes játékok. Egyrészt ha csapatok között zajlik a mérkőzés, csapaton belül tudnak segíteni a résztvevők egymásnak, mindenki elképzeléseit/taktikáit tudják alkalmazni. Másrészt pedig, ha mindenki csak saját magára számíthat, mérlegelnie kell, hogy érdemes-e "szövetséget" kössön valamilyik résztvevővel vagy milyen stratégiát alkalmazzon, hogy a lehető legtöbbet hozza ki a lehetőségekből.

Ugyanakkor azt is figyelembe kell venni a résztvevők többszemélyes játék esetén, hogy érdemes-e stratégiát váltani, ha a többiek is ehhez az opcióhoz folyamodnak vagy épp ellenkezőleg: továbbra is a megszokott módon kell eljárni.

Minden játékos célja, hogy a játék szabályait tiszteletben tartva a lehető legjobban teljesítsen. Azonban megfontolandó mindegyikük számára, hogy végezetül milyen "optimumot" kell kihozni, két típust tudunk megemlíteni ebben az esetben: egyéni szinten teljesíteni a legjobban vagy csoport szintjén.

Amikor minden résztvevő végső pontszáma kizárólag a saját teljesítményétől függ, evidens, hogy ebben az esetben mindegyiküknek az a célja, hogy a többieket hátráltatva, minden lehetőséget kihasználva arra törekszik, hogy maximalizálja az esélyeit a nyeléshez, több feladatot teljesítsen, sok "pontot" zsákoljon be, tehát úgy mond egy "mohó" (angol megfelelőjeként: greedy) módszert kell alkalmazzon.

Viszont, ha csapatszinten szeretnének győzedelmeskedni, nem feltétlenül célszerű ha az egy csoporton belüli résztvevők kizárólag a saját érdekeiket veszik figyelembe. (Mivelhogy a játéknak lehetnek olyan szabályai, hogy egyik játékos a csapatból ugyan szép pontszámot hoz össze, viszont ezt a többi csapattársa kárára, így a csapat nem nyeri meg a játékot.)

Az esetek nagy részében magától értetődőnek tűnik, hogy melyik játékos, hogyan kellene cselekedjen a győzelem érdekében. Viszont több játékos esetén, sok stratégiával ez kissé követhetetlenné válhat.

Dolgozatom célja egy olyan algoritmus implementálása, amely megad egy olyan "egyensúlypontot", amikor már egyetlen játékos sem érdemes stratégiát váltson, ha a többi résztvevő sem választ más stratégiát. Ennek az algoritmusnak az eredménye egy úgymond globális optimumot ad meg minden játékos számára.

A célokhoz hozzátartozik még a játékosok stratégiáinak és nyereségének rendszerezése, valamint eredmény vizualizálása, annak értelmezése a játékra nézve. Tehát lényegében a játék menete lesz modellezve Python programozási nyelvet használva (úgy az algoritmus implementálásához, mint az ábrázoláshoz).

Véleményem szerint a dolgozat megkönnyíti azoknak az embereknek értelmezni az eredményt, akik nem rendelkeznek játékelméleti-, matematikai- és informatikai ismeretekkel, és rávilágítja őket (hogya legközelebb társasjátékra kerülne a sor), hogy miként tudják alkalmazni a megértett információt.

A dolgozat felépítése a következőképpen néz ki: a bevezető után a második fejezetben sor kerül a szükséges játékelméleti fogalmak ismertetésére. A következő fejezetben John Nash munkásságáról, a Nash-egyensúlypontról valamint a felhasznált problémáról, a fogolydilemmáról írok. Ezek után az algoritmus (Lemke-Howson) elméleti-és matematikai modelljének taglalása történik. Levezetem matematikailag az algoritmust a fogolydilemmára, kiszámítom az eredményt és értelmezem. Ezt követően sor kerül az algoritmus implementálására, a lényegesebb kódsorok megmagyarázására, valamint az eredmény vizualizálására. Végezetül pedig az eredmény és az ábrák/rajzok értelmezése, összegzés valamint továbbfejlesztési lehetőségek zárják a dolgozat dokumentációját.

2. fejezet

Játékelméleti alapfogalmak

Ahhoz, hogy megértsük az algoritmus matematikai hátterét, először is nélkülözhetetlen az, hogy tisztában legyünk a játékelméleti alapfogalmakkal, szakszavakkal, jelölésekkel. Ebben a fejezetben ezek ismertetésére kerül sor.

2.1. Stratégia

Az egyik legalapabb fogalom, amivel kezdünk, az a **stratégia**. A stratégián egy olyan opciót, lépést értünk, amit a játékosnak a szabályok figyelembe vételével jogában áll alkalmazni.

Megjegyzés: Mindegyik játékosnak rendelkeznie kell legalább egy stratégiával.

Jelölése: $S = \{i | i = 1..n, n \in \mathbb{N}\}$

2.2. Kifizetési-mátrix

A következő tényező, amit értelmeznünk kell, az a **kifizetési-mátrix** [Eld] (a magyar nyelvű megfelelője az angol nyelvű "payoff-matrix"-nak). A payoff-matrix a következőképpen épül fel: sorainak száma megegyezik az első játékos (az angol terminológia szerint *row player*, magyarul pedig *sor-játékos*) stratégiának számával, míg oszlopainak száma a második játékos (hasonlóképpen *column player*, magyarul pedig *oszlop-játékos*) stratégiáinak számával egyenlő.

A mátrix elemeit számpárok alkotják, mégpedig a következőképpen: minden elem azt reprezentálja, hogyha a sor-játékos az i -edik opcióját váltja ki, míg az oszlop-játékos a j -edik stratégiáját játsza ki, akkor mekkora hasznuk/fizetésük lesz ebben az esetben külön-külön. A számpár első száma a sor-játékos, második száma az oszlop-játékos hasznát jelenti. (A következőkben ezt P -vel fogjuk jelölni.)

2.3. Zéró összegű- és pozitív összegű játékok

A kifizetéseket (pontokat) figyelembe véve megkülönböztetünk zéró összegű- és nem zéró összegű játékokat [oEBb]. **Zéró összegű játék** az a játék, amelyben a nyerések és a veszteségek összege nullát tesz ki, vagyis az egyik játékos a másik játékos közvetlen kárára profitál. **Pozitív összegű játék** alatt pedig azokat a játszmákat értjük, amik esetében

a nyereségek és veszteségek összege 0-nál nagyobb lesz, vagyis nem csak egymás kárára tudják növelni a pontszámaikat a játékosok.

2.4. Kooperatív- és nem kooperatív játékok

A **kooperatív**[Eld] játékok esetében a játékosoknak lehetőségük van kötelező érvényű megállapodásokat kötni egymással, ilyenkor a harc úgymond a szövetkezetek között zajlik. **Nem kooperatív** játszmáknál pedig a fentebb említett lehetőség nem adott a felek számára, így a játszma során mindenki saját magára számíthat csak.

2.5. Degenerált-és nem degenerált játékok

Ha olyan játékról van szó, amely szabályrendszere alapján nem marad játszmában mindegyik résztvevő az utolsó körig, hanem szerre kiesnek, ez sok következményt von maga után. Először is, ha például kártyajátzmák esetében, létezik olyan stratégia, hogy egyik játékos húzhat egy-egy lapot két másik résztvevőtől. Azonban, ha kiesnek a játékosok, és már csak egy marad rajta kívül, akkor ezt a képességet nem tudják érvényesíteni a még játékban levők.

Egy másik ok[PDP06], hogy a játékban való lépések és stratégiák száma redukálódik, az nem más, mint amikor kevés játékos marad talpon az utolsó körökre, és kialakul egy domináns stratégia, vagyis . Ezt ha nem követik, akkor megfosztják magukat a pontoktól vagy akár veszíthetnek is.

Amit még érdemes megemlíteni, az az erőforrások elapadása. Erőforrások lehetnek például: zsetonok, játékpénz, kártyalapok, karakterek, figurák (például a sakkban: király, ló, bástya). Ezt a szempontot főképp olyan esetekben fogja befolyásolni a játék menetét, amikor ha egyik résztvevő kiesik a játékból, a megszerzett pontjai és fentebb említett erőforrásai is kimennek a játékból. Így azok, akik még küzdenek a győzelemért, kevesebb erőforrás beszerzésével és felhasználásával kell érvényesülniük.

Az előbb megemlített és részletezett tényezők ismertetésére azért volt szükség, mivelhogy egy játékot ezek alapján is meg lehet különböztetni: lehet degenerált és nem degenerált.

A játékelméletben **degenerált**[PDP06] játéknak nevezzük az olyan játékokat, amikor a játékosok számára használható stratégiák száma a játék menete során csökken.

Ennek okát már ismertettük a ebben a fejezetben, pl. a résztvevők számának csökkenése, a domináns stratégiák gyakori alkalmazása vagy az erőforrások apadása. Az ilyen típusú játszmákban a végkifejlet kiszámíthatóbbá válik, ezáltal a játék veszít az összetettségéből és stratégiai jelentőségéből.

Ezzel szemben a **nem degenerált**[PDP06] játék olyan játékot takar, amely esetében a felhasználható lépések száma a játék előrehaladtával megmarad vagy növekszik.

3. fejezet

John Nash és a Nash-egyensúlypont

3.1. John Nash

Teljes nevén John Forbes Nash, 1928 június 13-án született Bluefield-ben (West Virginia-ban, az Amerikai Egyesült Államokban) és Nash egy amerikai származású matematikus volt, akit 1994-ben Közgazdasági Nobel-díjjal tüntettek ki mérőföldkönek számító, az 1950-es években megkezdett, a játékelmélet matematikájával kapcsolatos munkájáért. 2015-ben Abel-díjban részesült Louis Nirenberg-gel együtt, mivel közreműködött a parciális differenciálegyenletek tanulmányozásában. [oEBa]

Egyetemi éveit vegyészmérnöki szakon kezdte a pittsburgi "Carnegie Institute of Technology"-ban, mielőtt kémiával, majd matematikával foglalkozott volna. A fentebb említett intézményben tett szert alapfokú-és mesterfokú diplomára is 1948-ban. Doktori címét 1950-ben szerezte meg a "Princeton University"-ben.

Kutatásokat is végzett, 1951-ben csatlakozott a "Massachusetts Institute of Technology (MIT)" karához, itt a parciális differenciálegyenletek területével foglalkozott. Az '50-es évek végére sajnos mentális betegségei miatt lemondott ebben az intézményben végzett munkájáról. 1995-ben viszont a "Princeton University"-vel egy nem hivatalos egyesületet hozott létre, ahol egy vezető kutató matematikussá vált.

Ugyanabban az időben, amikor még egyetemi éveit töltötte, 1950 áprilisában sikerült publikálnia az első dolgozatát "The Bargaining Problem," címen (a cím magyar megfelelője "Az Alkudozás Problémája" lenne), az *Econometrica* folyóiratban. A "Non-Cooperative Games" című doktori disszertációjában kibővítette az Alkudozás témakör matematikai modelljét, amely 1951 szeptemberében az *Annals of Mathematics* folyóiratban jelent meg.

A differenciálegyenletekkel kapcsolatos kutatása (amelyet a MIT-en végzett) vezetett a "Real Algebraic Manifolds" című alapművéhez, amely szintén helyet kapott az *Annals of Mathematics* folyóiratban 1952 novemberében.

Habár mint olvashatjuk, számos kutatásban, tanulmányban és díjban volt része, mindenképp meg kell említenünk a következő nagyhatású matematikai tételeit.

Az egyik a Nash-Moser **inverz függvény** tétel, a másik a **Nash-De Giorgi** tétel (ami egy megoldást kínál David Hilbert XIX. századi problémájára; ezt Nash Nirenberg javaslatára vállalta).

Nash kutatása a játékelméletben és a hosszas szenvedése a paranoiás skizofrénia nevű betegségével eléggé ismertté vált az emberek számára az Oscar-díjas "A Beautiful Mind"

filmnek köszönhetően, amelynek alapját Sylvia Nasar azonos című, 1998-as életrajza képezte.

3.2. Nash-egyensúlypont

Most pedig térjünk át a Nash-nek köszönhetően felfedezett Nash-egyensúlypont taglalására. Ez a fogalom a fontos szerepet játszik a játékelméletben. Már 1838-ban is tanulmányozták ezt a témát, de nem kizárt, hogy még régebb is foglalkoztak vele [Kre89].

Egy játék esetén minden résztvevő arra törekszik, hogy a játék szabályainak eleget téve minél több pontot szerezzen és dobogós helyre kerüljön [Unka]. A Nash-egyensúlypont viszont egy olyan optimális "megoldást" nyújt a játék végkimenetelének illetően, amikor egyik játékosnak sem éri meg stratégiát váltani, ha a többi játékos sem vált stratégiát, így mindegyikük számára a játék a lehető legkedvezőbb végkimenetellel fog lezárulni.

Ahhoz, hogy egy játéknak létezzen Nash-egyensúlypontja, a következő feltételek kell érvényesüljenek: többszemélyes kell legyen, nem kooperatív és nem degenerált. A dolgozat keretén belül a fogolydilemma problémára támaszkodva fogjuk tárgyalni a Nash-egyensúlypontot.

3.3. A fogolydilemma

A klasszikus fogolydilemma az egyike azon problémáknak, amely segítségével könnyen be tudunk vezetni egy - a fogalmakkal, ismeretekkel, matematikai-és programozói tudással nem rendelkező - személyt a játékelmélet rejtelseibe.

Egyrészt, mivel egy konkrét (akár a valós életben is előforduló) eseményt dolgoz fel tudományosabb szempontok alapján. Ennek a problémának a modellje a politikában és a mindennapi életben szintén alkalmazható döntési szituációk elemzésére.

Másrészt pedig, mivel az eset nem rendelkezik sok végkimenetellel a foglyok számára, így nem veszíti el a fonalat az a személy, aki még nem mozog annyira otthonosan a játékelméletben, nem lesz komplikált az algoritmus alkalmazása, lépéseinek levezetése.

A fogolydilemma a következő esetet foglalja magába: két személyt őrizetbe vesz a rendőrség, mivel egy bűntény elkövetésével vádolják őket, tehát gyanúsítottak. Mindkettőjüket egy-egy külön zárkába helyezik, hogy egymással ne tudjanak kommunikálni, és egy ajánlatot kínálnak fel nekik. [Unka]

Az ajánlat a következő: ha mindketten bevallják a bűntettet, akkor mindkettőjük 5-5 év fogságot kap. Ha az első vallomást tesz, míg a második tagad, akkor az első szabadlábra helyezik, míg a másodikra 10 év letöltendő szabadságvesztés vár.

Hasonlóképpen alakul az eljárás, ha a második beismeri a bűncselekmény elkövetését, de az első tagadja. Viszont ha mindketten tagadják az elkövetést, akkor 2-2 év börtönbüntetéssel részesülnek egyaránt.

Most pedig térjünk rá a dilemma tárgyalására és megoldására. Nyilván mindkét gyanúsítottnak az a célja, hogy a lehető legkevesebbet - pontosabban egy napot sem - kelljen rács mögött töltsön, viszont a Nash-egyensúlypont nem csak az egyik fél számára nyújt nyereséget, hanem az a célja, hogy mindenkit érjen jutalom és kár egyaránt.

Ellenőrizzük, hogy a fogolydilemma eleget tesz-e a Nash-egyensúlypont feltételeinek. Többszemélyes, mivel két fogvatartottról van szó. Nem kooperatív, ugyanis a felek el

vannak szigetelve egymástól, nem tudnak egyességet vagy üzletet kötni egymással. Nem-degenerált, mivel a felhasználható lépések nem csökkennek.

Tehát ez a probléma rendelkezik Nash-egyensúlyponttal.

Ahhoz, hogy a tárgyalás érthetőbb legyen, az adatok és eredmények pedig szemelőtt legyenek, a második fejezetben ismertetett játékelméleti alapfogalmakat és jelöléseket használjuk. Első sorban, mivel nem tudjuk a gyanúsítottak nevét, az elsőszámút "A"-val, míg a másods számút "B"-vel fogjuk jelölni.

Tehát az "A" a sor-játékos, míg a "B" résztvevő az oszlop játékos.

A következő fogalom, az nem más mint a stratégiák halmaza. A sor-játékos opcióinak halmazát a következőképpen reprezentáljuk:

$$S_A = \{1, 2\}$$

Megjegyzés: Az egyszerűség kedvéért az 1-es szám a vallomást jelenti a továbbiakban, míg a 2-es szám a tagadást.

Az oszlop-játékosnak is ugyanazok a stratégiái, tehát a jelölés is nagyon hasonló, mégpedig:

$$S_B = \{1, 2\}$$

Ezek után az egyensúlypont megállapításához legszükségesebb paraméter, vagyis a payoff-matrix kerül meghatározásra.

Figyelembe véve a játékosok döntéseik után kapott hasznukat, a payoff-matrix így alakul:

$$\begin{pmatrix} (5, 5) & (0, 10) \\ (10, 0) & (2, 2) \end{pmatrix}$$

A fenti mátrixból egyértelműen lehet következtetni, hogy ha globálisan (mindkét foglyot figyelembe véve) vizsgáljuk az esetet, akkor a legkedvezőbb megoldás az lesz, ha mindketten tagadják a büntett elkövetését. Így megúsznák 2-2 év börtönnel. Viszont, mivel azt is számításba kell venniük, hogy egy ilyen "élet-halál" helyzetben nem bízhatnak meg egymásban, akkor egyének szintjén a tagadás nem adja a legkedvezőbb megoldást. Mivelhogy, ha az egyikőjük tagad, de a másik vall, akkor az előbbi 10 év fogságot kap. Hasonlóan alakul a büntetés a fordított esetben is. Tehát a tagadással vagy 10 évet kapnak vagy 2 évet, nyilván mindkét fél számára a leoptimálisabb (globálisan nézve), ha tagadnak, de túl nagy a kockázatvállalás, mert rosszabb esetben 10 évet is kaphatnak.

Ha egyének szintjén vizsgáljuk az esetet, akkor viszont a vallomás lesz a domináns stratégia. Ugyanis, ha mindketten bevallják, akkor 5-5 évet kell rácsok mögött tölteniük, míg ha egyikőjük beismeri a büntettet, a másik pedig nem, akkor akkor az előbbi szabadlábra kerül, míg az utóbbi kegyetlen büntetésben részesül. Tehát, ha csak az egyik játékosra levetítve szemléljük a következményeket, akkor ha vallomást tesz, rosszabbik

esetben 5 év, szerencsés esetben 0 év a büntetése, vagyis nincs akkora kockázatvállalás, mint a tagadás esetében.

Ez a tény, hogy az uralkodó stratégia a vallomás, a Nash-egyensúlypont értelmének is eleget tesz.

Egy kétszemélyes, nonkooperatív, nem degenerált játékban - mint ahogy a fogolydilemmában, mert eleget tesz ezeknek a kritériumoknak- a Nash-egyensúlypont azt a stratégiaegyüttest jelenti, amely esetén a játékosoknak nem éri meg más opciót kijátszani, ha az ellenfelük sem vált más stratégiára. Ugyanis, ha az egyikük a vallomás mellett teszi le a voksát, akkor a másiknak is célszerűbb a vallomás, mert ha nem vall, akkor kemény büntetés vár rá, ha vall, akkor 5-5 évet kapnak.

A tagadás viszont nem lesz domináns stratégia, mivelhogy ha a sor-játékos tagad és az oszlop-játékos is ehhez az alternatívához folyamodik, akkor 2-2 évet kapnak. Tehát ha egyikőjük tagad, akkor a másik jobb, ha vall, merthogy akkor nem kerül börtönbe. Így belátható, hogy sem a sor játékos, sem az oszlop játékos számára nem domináns stratégia a vallomás.

Ezen szavakkal végigvezetett tárgyalás után sejthető, hogy erre a dilemmára a Nash-egyensúlypont az lesz, ha mindketten a vallomás mellett döntenek.

A soron következő fejezetben ennek a sejtésnek a bizonyítása végett egy algoritmus matematikai levezetésével ezt fogjuk megkapni eredményül.

4. fejezet

A Lemke-Howson algoritmus elméleti- és matematikai leírása

A Lemke-Howson algoritmus többszemélyes játékok esetében szolgáltatja a Nash-egyensúlypontot, vagyis egy olyan stratégiaegyüttest, amely esetén egyik játékos sem tudja a stratégiája változtatásával növelni a nyereségét, ha a másik játékos sem változtat stratégiát.

Megjegyzés: Játéknak nem csupán a táblajátékok (pl. Sakk, Rummy) vagy a szabadtéri játékok bizonyulnak ebben az esetben. Amikor a "játék" kifejezést használjuk, az minden olyan helyzetre utal, amire igazak a következők: szükség van hozzá legalább két játékosra és a játékosok érdekei összefüggenek vagy kölcsönösen függenek egymástól. [Zag84]

Például játéknak minősül ebben az értelemben az a helyzet is, amikor két cég reklámozni szeretne, hogy népszerűsítse termékeit vagy szolgáltatásait. Ha egyikük sem foglalkozik a marketing résszel, akkor maradnak az eddigi bevételükkel, mivel nem jutnak újabb vevőkhöz. [Unkb]

Ha mindketten reklámoznak (viszont nem lépnek be új vevők a piacra), akkor ugyan ki kell fizessék a reklámozás díját, viszont így ugyanakkora veszteséggel szenvednek mindketten. Ha egyik cég befektet a termékei népszerűsítésébe a másikkal ellentétben, akkor nagy eséllyel elcsalja a másik társaság vevőinek jelentős hányadát, ezáltal bevételre tesz szert.

Ahogy az előbbi bekezdésben bemutatott példából belátható, hogy játékelmélet szempontjából a valós élet különböző területeiről (pl. marketing; de ide lehet sorolni a politikát is) származó esetek játékként minősíthetők, így az előbbi fejezetben bemutatott fogolydilemma is ide sorolható.

A Lemke-Howson algoritmus a fogolydilemma adataira támaszkodva kerül bemutatásra.

Mielőtt rátérnénk a bemutatására, elengedhetetlen, hogy az algoritmus hátrányai és kikötései egyaránt ismertetve legyenek, ugyanis nem minden esetben nyújt segítséget a Nash-egyensúlypont megtalálásában.

Elsősorban kikötés van a játékosok számát illetően: kizárólag kétszemélyes játékokra értelmezhető az eljárás.

Másodsorban, az eredményt megvizsgálva, az algoritmus egy játék esetén csak egyetlen egyensúlypontot talál meg. (Vannak olyan játékok, amelyek több Nash-egyensúlyponttal rendelkeznek.) [Liu13]

Ami még a megszorításokhoz tartozik, hogy a játék nem degenerált kell legyen.

Ez a szakkifejezés a dolgozat második fejezetében már ismertetésre került. A fogolydi-

lemma nem degenerált, mivel: a résztvevők létszáma nem csökken, vagyis ha eggyel is csökkenne, akkor már értelmét veszítené a játék, vége szakadna; a stratégiák száma nem csökken, igazából az a játék egyetlen körből áll, mert a foglyok döntenek, hogy melyik opciót választják, ezután pedig a szabályoknak megfelelően kapják meg a büntetésüket vagy előnyösebb esetben a nyereségüket; különösebb képességek, kártyalapok, erőforrások nem szerepelnek a játékban, úgyhogy ezek elapadásától sem kell tartaniuk.

4.1. Az algoritmus ismertetése

Így, hogy minden feltételnek eleget tesz a probléma, elkezdődik az algoritmus bemutatása.

A legmeghatározóbb paraméter, ami befolyásolja a végeredményt, az a payoff matrix. Ennek felírásával kezdődik az algoritmus.

$$P = \begin{pmatrix} (5, 5) & (0, 10) \\ (10, 0) & (2, 2) \end{pmatrix}$$

Ahhoz, hogy érthetőbbé váljon a leírás, két közönséges mátrixra bontódik az előbbi mátrix (az elemei természetes számok, nem számpárok):

$$P_A = \begin{pmatrix} 5 & 0 \\ 10 & 2 \end{pmatrix}$$

$$P_B = \begin{pmatrix} 5 & 10 \\ 0 & 2 \end{pmatrix}$$

Eme probléma kifizetési mátrixában viszont van egy kis csavar. Mégpedig az, hogy itt a jutalom nem hasznot jelent, hanem börtönben letöltendő éveket, azaz kárt.

Tehát minél nagyobb a "jutalom", az annál rosszabbat jelent a feleknek. Azért, hogy ne legyen ennyire bonyolult ennek a felfogása - mivel itt a nyereség tulajdonképpen rosszat jelent - ezt úgy lehet megoldani, hogy a P_A és P_B mátrixok minden elemét negáljuk.

Ennek köszönhetően érthetőbbé válik a payoff-mátrixok értelmezése. Ugyanis, ha összehasonlítjuk például a 10-et az 5-tel, akkor a 10 az nyilvánvalóan nagyobb, de a raboknak az a kedvezőbb, ha minél kevesebb évet kell tölteniük a rácsok mögött. Ezért, ha negáljuk az értékeket, akkor már a kisebbik szám fogja jelenteni a csekélyebb nyereséget, míg a nagyobbik a többet.

Tehát a P_A és P_B mátrixok minden elemét kicseréljük az ellentétéseire:

$$P_A = \begin{pmatrix} -5 & 0 \\ -10 & -2 \end{pmatrix}$$

$$P_B = \begin{pmatrix} -5 & -10 \\ 0 & -2 \end{pmatrix}$$

Az algoritmus legegyszerűbb leírása és Nash tételének legmegérthetőbb bizonyítása kétszemélyes játékokra két politóppal van összefüggésben, ami hamarosan meg lesz határozva.

A politóp¹ az elemi geometriában egy lapos oldalakkal rendelkező mértani objektum, amely bármely dimenziószám esetén létezhet. Például a sokszög (poligon) a kétdimenziós politópnak felel meg, míg a poliéder a háromdimenziózt reprezentálja.

Egy másik megfogalmazás, amely egyenértékű az előbbi értelmezéssel, a következő: a politóp egy olyan geometriai alakzat, amelyet le lehet írni egyenlőségek és egyenlőtlenségek rendszerével.

Jelölje B_j a oszlop-játékos j -edik oszlopát, ami megfelel a második játékos j -edik stratégiájának, míg a sor-játékos (vagyis az első résztvevő) i -edik sorát, ami tulajdonképpen az i -edik opciójának felel meg A_i jelöléssel illetjük.

Ekkor a két politópot [Pri] formálisabban a következő halmazok segítségével tudjuk meghatározni:

$$P_1 = \{x \in \mathbb{R}^M \mid (\forall i \in M : x_i \geq 0) \ \& \ (\forall j \in N : B_j^\top \cdot x \leq 1)\}$$

$$P_2 = \{y \in \mathbb{R}^N \mid (\forall i \in N : y_i \geq 0) \ \& \ (\forall i \in M : A_i \cdot y \leq 1)\}$$

Ezen a halmazok és összefüggéseiből egy egyenlőtlenségrendszer írható fel, ezekből fogjuk megkapni a Nash egyensúlypontot. Mivel az egyenlőtlenségekkel kicsivel nehezebb dolgozni, mint az egyenletekkel, ezért az a célunk, hogy valamilyen módon az előbb említett egyenlőtlenségrendszert egyenletrendszerre alakítsuk át. Ez a következőképpen fog történni: mivel az algoritmus szempontjából lényeges egyenlőtlenségek "kisebb vagy egyenlő", azaz \leq jelet tartalmaznak, ez azt teszi lehetővé, hogyha mindegyik egyenlőtlenség bal oldalához hozzá lehet adni egy változót, úgy, hogy az így kapott kifejezés ezen az oldalon egyenlő legyen a jobb oldali kifejezéssel. Ezeket a változókat az angol terminológia szerint *slack variable*-nek (magyarul: laza változónak) nevezzük.

Ennek a módszernek a segítségével alakul át az egyenlőtlenségrendszer egyenletrendszer formájába [Pri].

Most pedig, az egyenletek, vagyis az egyenletrendszer felírására kerül sor. A halmazokban szereplő X és Y vektorok az egyenletrendszer ismeretleneit tartalmazzák. Az X^\top annyi elemet tartalmaz, mint az első játékos stratégiáinak száma, míg az Y vektor annyit, amennyi a második résztvevő stratégiáinak száma és mindkét vektor oszlopvektor.

Ahogy a politópokban szereplő egyenlőtlenségek is szemléltetik, x és y , vagyis a kifizetési mátrix elemei nem lehetnek nullánál kisebbek. E feladat "payoff-matrix"-a ennek a

¹Politóp értelmezése

feltételnek nem tesz eleget, mivelhogy sok negatív értéket tartalmaz.

Erre a következő a megoldás: mindkét játékos kifizetési mátrixának minden eleméhez hozzá lehet adni egy tetszőleges természetes számot úgy, hogy ez az eljárás végkimenetelét nem befolyásolja.[Liu13]

Jelen esetben a legkisebb elem a -10, tehát elegendő lenne 10-et hozzáadni minden elemhez. Viszont előfordulhatnak olyan esetek, amikor nem szolgáltat helyes eredményt ilyenkor az algoritmus. Ezért, a biztonság kedvéért mindig legalább egy akkora számot kell hozzáadni a mátrixok elemeihez, hogy ezek 0-nál nagyobbak legyenek.

Tehát így a mi esetünkben elegendő 11-et hozzáadni a P_A és P_B mátrixok mindenik eleméhez.

Az így kapott mátrixok a következők:

$$P_{A'} = \begin{pmatrix} 6 & 11 \\ 1 & 9 \end{pmatrix}$$

$$P_{B'} = \begin{pmatrix} 6 & 1 \\ 11 & 9 \end{pmatrix}$$

Mivel mindkét játékosnak 2 stratégiája van, ezért az X és Y vektorok is 2-2 ismeretlent fognak tartalmazni.

A politópoknál ugyan Y -nal van jelölve a második vektor, viszont a az egyszerűség és a könnyebben átláthatóság miatt mindkét vektor X -el lesz jelölve ezentúl és az ismeretlenek is. (Természetesen indexek segítségével jelezve lesz, hogy melyik vektorról és melyik ismeretlenről van szó.)

Ezek alapján:

$$X_A = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$X_B = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$$

A rendszer felírásához az P_B mátrixot transzponálni kell. (Át kell írni a sorait oszlopokba.)

$$P_{B'}^T = \begin{pmatrix} 6 & 11 \\ 1 & 9 \end{pmatrix}$$

A politópokban szereplő feltételeknek megfelelően az egyenletrendszer jelöléseket használva a következőképpen alakul:

$$P_{A'} \cdot X_B \leq \mathbf{1}$$

$$P_{B'}^T \cdot X_A \leq \mathbf{1}$$

Megjegyzés: A megvastagított 1-es mindkét egyenlet jobb oldalán egy két soros, 1-eseket tartalmazó oszlop mátrixra utal.

Ahogy néhány bekezdéssel ezelőtt szó volt róla, bevezetésre kerül a **slack variable**. Ezek is oszlop mátrixokban foglalnak helyet.

$$SV_A = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix}$$

$$SV_B = \begin{bmatrix} s_3 \\ s_4 \end{bmatrix}$$

Az SV_A -t az első-, míg az SV_B -t a második egyenlőtlenséghez adjuk hozzá, ezáltal két egyenletté alakul át a rendszer.

$$P_{A'} \cdot X_B + SV_A = \mathbf{1}$$

$$P_{B'}^T \cdot X_A + SV_B = \mathbf{1}$$

Innen kifejezzük a **laza változókat** tartalmazó vektorokat²:

$$SV_A = \mathbf{1} - P_{A'} \cdot X_B$$

$$SV_B = \mathbf{1} - P_{B'}^T \cdot X_A$$

A feltételeket és a feladat adatait felhasználva az egyenletrendszer a következőképpen alakul:

$$\begin{cases} s_1 = 1 - 0x_1 - 0x_2 - 6x_3 - 11x_4 \\ s_2 = 1 - 0x_1 - 0x_2 - 1x_3 - 9x_4 \\ s_3 = 1 - 6x_1 - 11x_2 - 0x_3 - 0x_4 \\ s_4 = 1 - 1x_1 - 9x_2 - 0x_3 - 0x_4 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases}$$

Megjegyzés: A fenti egyenletrendszerben azért szerepelnek a 0-sok, mert a rendszer át fog íródni mátrix alakba.

Az 1-esek pedig azért vannak feltüntetve, mivel az algoritmus során arányokat kell számítani és összehasonlítani az x ismeretlenek együtthatói felhasználásával. Ha az 1-esek szerepelnek a rendszerben, akkor könnyebben észrevehető az együttható, mintha nem lenne oda írva.

Most pedig a rendszer együtthatói átíródnak mátrix alakba. (Az összes **slack variable**

²slack variable

indexe elé egy "-" (mínusz) előjel kerül - a későbbiekben ismertetjük ennek okát.)

$$\begin{pmatrix} -1 & 1 & 0 & 0 & -6 & -11 \\ -2 & 1 & 0 & 0 & -1 & -9 \\ -3 & 1 & -6 & -11 & 0 & 0 \\ -4 & 1 & -1 & -9 & 0 & 0 \end{pmatrix}$$

Most pedig térjünk rá az algoritmus bemutatására. Szükség van még néhány fogalom tisztázására, a gondolatmenet és szabályok bemutatására.

A rendszer együtthatóiból alkotott mátrix második oszlopát (vagyis az 1-esekből álló oszlopmátrixot) **bázisnak** nevezzük. Az egyenletek bal oldalán elhelyezkedő változók a laza változók, míg rendszer ismeretlenek (x_1, x_2, x_3, x_4) pedig az **aktuális változók**. Azok a változók, amelyek nincsenek benne a bázisban, **bázison kívüli változók**-nak nevezzük.

A rendszer bal oldalán levő változókat **bázisban levő változóknak** fogjuk tekinteni. Kezdetben minden laza változó a bázisban van, minden aktuális változó pedig a bázison kívül foglal helyet.

Az együtthatókból felírt mátrix második oszlopában levő 1-eseket **kezdeti bázisértékeknek** fogjuk hívni³. (Eme oszlop elemei fogják alakítani a Nash-egyensúlypontot a számítások végén.)

Megjegyzés: A két politóp egyenletei együtthatóit fel lehetett volna írni két külön mátrixba, viszont ha egy nagyobb mátrixot használunk egyszerre lehet követni mind a négy egyenlet változását.

Az egyensúlypont meghatározása érdekében használt metódus iteratív jellegű és az angol terminológia a **"Pivoting"** névvel illeti.

Az eljárás⁴ lényege abban áll, hogy minden egyes lépésben egy változó (amelyik persze nincs benne a bázisban) be fog kerülni a bázisba (a "pivot"), és egy változó - azok közül, amelyek a bázisban vannak - ki el fogja hagyni a bázist.

Ha egy változót (például x_1 -et) be szeretnénk vinni a bázisba, akkor ezt a következőképpen lehet elérni: minden olyan egyenletben, ahol a szóbanforgó változó szerepel, számítunk egy arányt a kurrens bázisérték ellentéteséből (negáltjából) és a bázisba bekerülendő változó együtthatójából. Ezek közül az arányok közül kiválasztjuk a legkisebbiket, és abból az egyenletből kifejezzük a kurrens változót (átvisszük az egyenlet jobb oldalára, tehát bekerül a bázisba).

Megjegyzés: Ha a két arány közül valamelyik értéke negatív, akkor nyilvánvaló, hogy az lenne a kisebb, de az algoritmus nem szolgáltat helyes eredményt ha ezt figyelembe vesszük. Tehát a negatív arányt ki kell zárni, és a másik arány fogja tovább vinni az algoritmust.

Így, hogy a szóbanforgó ismeretlen már ki van fejezve, a többi egyenletbe (ahol szerepelt az a változó, ami már a bázisban van) ezt behelyettesítjük, ezáltal már az egyenletek jobb oldalán nem fog szerepelni ez a változó.

Megjegyzés: Nem befolyásolja az algoritmus végkimenetelét az, hogy melyik változót vesszük be a bázisba először. Nem kötelező x_1 -gyel kezdeni.

³elnevezések

⁴pivoting

Azt, hogy melyik aktuális változó kerül be következőleg a bázisba, mindig az előzőleg a bázisba bevitt változó határozza meg. Ugyanis ha beviszünk egy aktuális változót, akkor ki kell hoznunk egy "slack variable"-t. Ennek a "slack variable"-nek az indexe határozza meg, hogy mi kerül be a bázisba legközelebb.

Formálisan írva, ha s_i kiment a bázisból, akkor x_i -t kell bevinni ezután, és ezt a módszert kell alkalmazni fordított esetekben is.

Ha az első bázisba bevitt változó az x_i volt, akkor ezt az iteratív eljárást addig kell ismételni, amíg x_i vagy s_i elhagyja a bázist.

Így, hogy minden szükséges elméleti fogalom, szabály és az algoritmus lépései is ismertetve vannak, alkalmazásra kerülhet az eljárás a tárgyalta problémára.

A következőkben lépésről-lépésre levezetjük az algoritmust,⁵ egy-két mondatot is hagyunk az iterációk között magyarázat gyanánt.

Legelső bázisba bekerülendő aktuális változónak válasszuk az x_1 -et. Ez a változó a harmadik és negyedik egyenletben van benne. Kiszámítjuk az arányokat. A harmadik egyenlet alapján az arány $-1/(-6)=1/6$, míg a negyedik összefüggésből a $-1/(-1)=1$ arányt kapjuk. $1/6$ kisebb mint 1, tehát a harmadik egyenletből kell kifejezni x_1 -et, így a következőt kapjuk:

$$x_1 = 1/6 - 1/6 \cdot s_3 - 11/6 \cdot x_2$$

A kapott egyenlőség jobb oldalát kell behelyettesíteni a negyedik egyenletbe x_1 helyére:

$$s_4 = 1 - 1/6 + 1/6 \cdot s_3 + 11/6 \cdot x_2 - 9 \cdot x_2$$

Elvégezzük a műveleteket:

$$s_4 = 5/6 + 1/6 \cdot s_3 - 43/6 \cdot x_2$$

Így a rendszer együtthatóiból alkotott mátrix a következőképpen alakul:

$$\begin{pmatrix} -1 & 1 & 0 & 0 & -6 & -11 \\ -2 & 1 & 0 & 0 & -1 & -9 \\ 1 & 1/6 & -1/6 & -11/6 & 0 & 0 \\ -4 & 5/6 & 1/6 & -43/6 & 0 & 0 \end{pmatrix}$$

Azokat a sorokat, amelyek nem tartalmazzák a bevitt változót, változatlanul lemásoljuk. s_3 helyére bekerült az x_1 , ezért a harmadik sor első oszlopában levő index átalakul 1-essé.

Mivel s_3 kiment a bázisból, a szabály szerint x_3 -at kell behozni a következő lépésben. x_3 az első és második egyenletben szerepel. Kiszámítjuk az arányokat: $-1/(-6)=1/6$ és $-1/(-1)=1$. Mivel $1/6$ kisebb mint 1, ezért x_3 -at az első egyenletből fogjuk kifejezni:

$$x_3 = 1/6 - 1/6 \cdot s_1 - 11/6 \cdot x_4$$

Ezt pedig a második egyenletbe helyettesítve, az alábbi kapjuk:

$$s_2 = 1 - 1/6 + 1/6 \cdot s_1 + 11/6 \cdot x_4 - 9 \cdot x_4$$

⁵algoritmus matematikailag

Összevonjuk az egynevű tagokat:

$$s_2 = 5/6 + 1/6 \cdot s_1 - 43/6 \cdot x_4$$

Ismét átírjuk a mátrixot, ott, ahol indokolt:

$$\begin{pmatrix} 3 & 1/6 & 0 & 0 & -1/6 & -11/6 \\ -2 & 5/6 & 0 & 0 & 1/6 & -43/6 \\ 1 & 1/6 & -1/6 & -11/6 & 0 & 0 \\ -4 & 5/6 & 1/6 & -43/6 & 0 & 0 \end{pmatrix}$$

Mivel az első egyenletben volt kisebb az arány, így s_1 hagyta el a bázist, emiatt a mátrix első oszlopának első sorában az index átíródik 3-ra.

s_1 kilépett a bázisból, tehát x_1 -et kellene bevinni a bázisba. Ami azt jelenti, hogy az algoritmus itt véget ér, mivel x_1 volt az a változó, ami legelőször bekerült a bázisba, és s_1 kijött a bázisból.

4.2. A eredmény kiszámítása és értelmezése

A dolgozat eme részében kiszámításra kerül a Nash-egyensúlypont az előbb végigvezetett Lemke-Howson algoritmusra alapozva.

Mint ahogy akkor is meg volt említve, az eredményt az algoritmus utolsó iterációja után kapott mátrixból kell meghatározni, pontosabban annak a mátrixnak a második oszlopából - vagyis a kezdeti bázisértékek oszlopából.

Az egyensúlypont⁶ kiszámítása a következőképpen történik: figyelembe véve az eljárás végén kapott mátrix első oszlopát is, meg kell vizsgálni, hogy milyen előjele van az első oszlopban levő elemeknek.

Ennek érdekében kiírjuk egy külön mátrixba ezeket az elemeket, mivel a többi elem már úgysem befolyásolja a végeredmény alakulását.

$$\begin{pmatrix} 3 & 1/6 \\ -2 & 5/6 \\ 1 & 1/6 \\ -4 & 5/6 \end{pmatrix}$$

Ahogy meg volt már említve, a Nash-egyensúlypont minden játékos minden stratégiájára egy relatív gyakoriságot mutat meg - vagyis, hogy az esetek hány százalékában (hányad részében) kell ezzel a lehetőséggel élni minden játékos, hogy ez az egyensúly kialakuljon.

Ezekből az arányokból lehet majd következtetni a domináns stratégiákra.

Mivel ezek a számok azt kell megmutassák, hogy a sor-játékos és a oszlop-játékos az adott stratégiáit az esetek hány százalékában kell alkalmazza, ezért ezek a számok a $[0:1]$ intervallum elemei kell, hogy legyenek.

⁶kiszámítás

Jelen esetben az a feltétel teljesül is, mert az elemek vagy $1/6$ -dal vagy $5/6$ -dal egyenlőek. *Megjegyzés:* Ha valamelyik érték nem eleme a $[0;1]$ intervallumnak, akkor az annak megfelelő stratégia relatív gyakorisága 0 lesz.

Az első oszlopban levő számok a játékosok stratégiáit jelentik. Az 1-es és 2-es a sor-játékosét, míg a 3-as és 4-es az oszlop-játékosét.

Az 1 és 3 a vallomást jelképezik, a 2 és 4 pedig a tagadást. Amelyik értékek negatívak az első oszlopban, annak a stratégiának a relatív gyakorisága is 0 lesz. Tehát a Nash-egyensúlypont a következő lenne: $((1/6,0),(1/6,0))$.

Ezzel azonban az a probléma, hogyha a két külön vektorként tekintünk a játékosok stratégiáinak relatív gyakoriságaira, akkor ezek nem lesznek sztochasztikusak. Akkor **sztochasztikus** egy vektor, ha az elemei 0-nál nagyobb vagy egyenlőek, 1-nél pedig kisebb vagy egyenlőek, és az összegük kiadja az 1-et. (Másképpen fogalmazva, a vektor elemei lényegében egy valószínűségi eloszlást adnak meg. A relatív gyakoriságokat is fel lehet úgy fogni, mint valószínűségi értékek.)

Ahhoz, hogy ezek a vektorok sztochasztikusok legyenek, normalizálni kell őket. A normalizálás a következő képlet⁷ alkalmazásával történik:

$$normalized(x) = \left(\sum_{i=1}^{|x|} x_i \right)^{-1} \cdot x$$

Ahol az x jelöli a normalizálandó vektort, $|x|$ az elemeinek számát, x_i pedig a vektor i -edik elemét.

Tehát lényegében össze kell adnunk az elemeket, majd az összeg inverzével meg kell szorozzuk a vektort. Jelen esetben mindkét játékos relatív gyakoriságainak elemei és a sorrendje is megegyezik. Mindkét esetben az elemek összege $1/6$.

$1/6$ inverze egyenlő $1/(1/6)$, ami 6. Ha mindkét vektort beszorozzuk 6-tal, akkor a következőt kapjuk eredményül: $((1,0),(1,0))$.

Tehát ez lesz a fogolydilemma Nash-egyensúlypontja.

Így, hogy megkaptuk az eredményt, értelmeznünk is kell azt. Mivel mindkét játékos esetében a relatív gyakoriságok értékekben és sorrendileg is megegyeznek, valamint ha figyelembe vesszük a relatív gyakoriságokat is, akkor arra lehet következtetni, hogy mindkét játékos az esetek 100%-ában az első stratégiáját kell alkalmazza és 0%-ban kell élnie a második lehetőségével.

Vagyis minden esetben a vallomás mellett kellene döntsön és egyetlenegy esetben sem folyamodhat tagadáshoz. Mivel a fogolydilemma egy egyfordulós játék - vagyis a végki-menetel egyetlen kör után megvan (miután mindkét játékos eldöntötte, hogy tagad vagy vall) - ezért ha a Nash-egyensúlypont alapján akarnak dönten, akkor ebben az egy fordulóban(körben) a vallomás mellett kell döntenek mindketten. Tehát a domináns stratégia a vallomás.

Ez a tény, hogy a fentebb kiszámított értéket kaptuk egyensúlypontnak, ugyanazt jelenti (szoros összefüggésben van) mintha csak döntések és következmények alapján vizsgálnánk meg a játékot.

A Nash-egyensúlypont[Unka], mint ahogy már említettük, azt a helyzetet adja meg, amikor nem érdemes a játékosoknak stratégiát váltani, ha az ellenfelek sem tesznek így.

⁷normalizálás

Ebben a problémában két ugyanolyan stratégiával rendelkezik mindkét játékos: vallomás és tagadás.

Ha az egyik játékos tagad, akkor nem érdemes a másik résztvevőnek tagadnia, mert akkor mindketten 2-2 évet kapnak büntetésül. Tehát itt érdemes a stratégiaváltás, hiszen azzal elkerülhető a börtön.

Viszont ha egyik játékos a vallomás mellett dönt, de a másik tagad, akkor az előzőt szabadlábra helyezik, míg a másikra egy 10 éves büntetés vár. Ha a második is bevallja bűnösségét, akkor mindketten 5-5 évet kapnak. Következésképpen ebben az esetben, ha valamelyikük vall, akkor a másik fogoly is jobban teszi, ha vall. Vagyis itt a stratégia megváltoztatása nem vezet célra, ami azt jelenti, hogy a domináns stratégia a vallomás.

Összegezve az eddig leírtakat, konklúzióként annyit tudunk levonni, hogy mivel nem bíznak meg egymásban a fogvatartottak, így akkor kerülnek ki a ebben az esetben a helyzetből, ha mindketten a vallomás mellett döntenek.

Tehát akkor alakul ki a Nash egyensúlypont, ha mindketten bevallják a tetteiket.

5. fejezet

A felhasznált algoritmus megközelítése programozás szempontjából

Ahogy a többi fejezet esetében is meg volt említve, a keresett egyensúlypont kiszámítása érdekében a Lemke-Howson algoritmust alkalmazzuk.

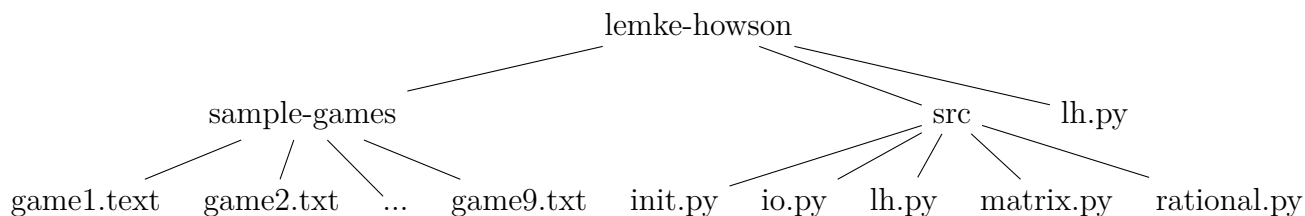
Az előző fejezetben eme algoritmus matematikai levezetését végeztük el, most pedig ennek az informatikai modellezésével, azaz implementációjával fogunk foglalkozni és az eredmény vizualizációjával.

Az algoritmus leprogramozásához egy már régebb implementált programot használok fel. Erre a programra egy GitHub repository-n keresztül szereztem tudomást¹.

Ez a program Python nyelvben íródott, 2.5-ös verziót használva. Én ugyanúgy Python programozási nyelvet fogok alkalmazni, csak nem a 2.5-ös verziószámút, hanem a 3.11-est. Fejlesztői környezetnek a PyCharm-ot használok.

Az én feladataim közé tartoznak: a felhasznált programkód struktúrájának és működésének elmagyarázása, a verziószámokból adódóan a hibák és inkompatibilis programrészletek kijavítása valamint a megkapott egyensúlypont vizualizációjának implementációja.

A felhasznált algoritmus programkódjának struktúráját egy felbontási fa segítségével fogom szemléltetni. (Csak azokat a programegységeket magyarázom el, amelyek elengedhetetlenek az algoritmus implementációjának megértéséhez.)



Így, hogy mostmár átlátható a projekt felépítése a felbontási fának köszönhetően, elmagyarázom a csomagok és fájlok szerepét és lényegét.

A **lemke-howson** címke felel meg a fa gyökerének, ami tulajdonképpen a projekt neve. Az első package, vagyis a **sample-games**, a bemeneti állományokat tartalmazza,

¹[Algoritmus](#)

innen fogja kapni az algoritmus a bemeneti adatokat. Több fájlt is tartalmaz, ami lehetővé teszi az algoritmus helyességének tesztelését. Ahhoz, hogy a program olvasni tudjon egy bemeneti állományból, annak két egyforma méretű mátrixot kell tartalmaznia, a mátrixok elemei egy-egy szóközzel kell legyenek elválasztva illetve a két mátrix között szerepelnie kell egy üres sornak.

A következő csomag, vagyis a **src** foglalja magába tulajdonképpen az algoritmushoz legszükségesebb fájlokat. Az **io.py** tartalmazza azokat az I/O metódusokat (író-és olvasó függvényeket), amelyek kommunikálnak a projekt felhasználójával. Ez a modul négy függvényt tartalmaz, a következőkben ezek közül az algoritmus szempontjából lényegesebbeket fogom bemutatni.

A `parseInputMatrices(text)` függvénynek egyetlen paramétere van, így kapja meg, hogy melyik állományból kell olvassa az adatokat. Az eljárás első része a megadott állományt felhasználva, annak tartalmát átalakítja két mátrixba.

```
def parseInputMatrices(text):
    """Parses two matrices from the selected text and returns
    them in a tuple (m1, m2).

    text - text from which the matrices will be parsed (string)

    Preconditions:
        - text must contain two matrices (see matrix.Matrix.__repr__())
          separated by an extra new line (there might be additional
          newlines after the second matrix, which are ignored)
        - both matrices must have the same number of rows and cols

    Raises ValueError if some of the preconditions are not met.
    """
    try:
        mTexts = text.split('\n\n')
        if len(mTexts) > 2:
            # Check whether there are only newlines in the rest of the text
            for mText in mTexts[2:]:
                for c in mText:
                    if c != '\n':
                        raise ValueError('Redundant characters at the end ' + \
                                         'of the input.')
            mTexts = (mTexts[0], mTexts[1] + '\n')

        m1 = matrix.fromText(mTexts[0] + '\n')
        m2 = matrix.fromText(mTexts[1])
```

A függvény további részében pedig a kivételkezelések következnek, ezek akkor fognak kiváltódni, ha egy olyan mátrix keletkezik, ami nem felel meg a **sample-games** csomag állományainál említett feltételeknek.

```

except IndexError:
    raise ValueError('Input text does not contain two valid matrices.')
except ValueError:
    raise ValueError('Input text does not contain two valid matrices.')
except matrix.InvalidMatrixReprError:
    raise ValueError('Input text does not contain two valid matrices.')

if m1.getNumRows() != m2.getNumRows() or m1.getNumCols() != m2.getNumCols():
    raise ValueError('Input text contains two matrices with different ' + \
        'number of rows or columns.')

return (m1, m2)

```

A következő állomány, aminek a főbb metódusait prezentálnám, az nem más mint a *matrix*. Ebben helyet kap az **objektumorientált** programozási paradigma alkalmazása, a *Matrix* nevű osztály bevezetése révén.

Ennek az osztálynak (vagy angol terminológia szerint class-nek) az a rendeltetése, hogy modellezni lehessen az algoritmushoz szükséges mátrixokat a kétdimenziós térben (kétdimenziós tömbökben tárolva).

```

class Matrix(object):
    """This class represents a matrix in a two dimensional space.

    Objects of this class are mutable."""

    def __init__(self, rows, cols):
        """Creates a matrix with the selected number of rows and columns.

        rows - number of rows
        cols - number of columns

        All elements are initialized to zero.

        Preconditions:
        - rows > 0
        - cols > 0

        Raises ValueError if some of the preconditions are not met.
        """
        if rows <= 0:
            raise ValueError('Number of matrix rows must be greater than zero.')
        if cols <= 0:
            raise ValueError('Number of matrix cols must be greater than zero.')

```

```

self.__rows = rows
self.__cols = cols

# Create the matrix and initialize all elements to zero
self.__m = []
for i in xrange(1, self.__rows + 1):
    row = []
    for j in xrange(1, self.__cols + 1):
        row.append(0)
    self.__m.append(row)

```

Az *init(self, rows, cols)* metódus felel meg lényegében az osztály konstruktorának; három paraméterrel rendelkezik. A *self* az hasonló szerepet tölt be, mint Java-ban a *this*, vagyis lehetővé teszi, hogy tudjunk hivatkozni azokra az objektumokra, amiket az osztály példányosítása révén kaptunk. A **rows** és **cols** attribútumokban pedig megadjuk, hogy hány soros és hány oszlopos mátrixot szeretnénk létrehozni.

A többi metódus pedig a mátrixműveletek miatt felel: pl. két mátrix összehasonlítása, adott pozíciójú elem visszatérítése, stb.

A *rational.py*-ban szintén egy osztály szerepel, mégpedig a **Rational**. Ennek az osztálynak a szerepe a racionális számok (tört alakba írható számok) modellezése.

Az *init(self, a, b=1)* függvény az osztály konstruktor, a *self* szerepét az előző osztály esetében már részleteztük. Az *a* paraméter fogja jelenteni a tört számlálóját, míg a *b* a nevezőnek felel meg.

A *b=1* azt jelenti, hogy ennek a paraméternek az alapértelmezett értéke 1; ha nem adjuk meg ezt, akkor a konstruktor automatikusan 1-es értéket tesz a nevezőbe.

A metódus további részeiben ellenőrizzük, hogy: a nevező ne legyen 0 (mert akkor nincs értelme a törtnek); mi a teendő, ha *a* alaphoz racionális illetve ha az egyik szám negatív. A *gcd* függvény megadja a két szám legnagyobb közös osztóját, amit arra használunk, hogy a törtet irreducibilis alakban tudjuk használni.

```

def _gcd(a, b):
    """Returns the greatest common divisor of a and b."""
    if a < b:
        a, b = b, a
    while b != 0:
        a, b = b, a % b
    return a

```

```

def __init__(self, a, b=1):

    if b == 0:
        raise ValueError('b must be nonzero.')

    try:
        # Lets suppose that a is a rational number
        self.__a = a.nom()
        self.__b = a.denom()

        if b != 1:
            raise ValueError('If a is a rational number, b must be 1.')
    except AttributeError:
        # a and b must be ordinary numbers
        self.__a = a
        self.__b = b

        # Sign normalization
        if self.__b < 0:
            self.__a = -self.__a
            self.__b = abs(self.__b)

        # Commensurability normalization
        d = _gcd(abs(self.__a), abs(self.__b))
        self.__a //= d
        self.__b //= d

```

ional > __init__ > try

Az osztály további függvényei a törtekkel kapcsolatos alapvető műveleteket modellezik: összeadás, összehasonlítás, inverz, stb.

A `src` package-ben található `lh.py`-ban találhatók meg az algoritmus implementációjához legszükségesebb függvények.

A `normalizeMatrices(m1,m2)` a két játékos payoff mátrixaiban megvizsgálja, hogy vannak-e 0-nál kisebb elemek. Ha vannak, akkor hozzáad egy olyan konstanst mindkét mátrix minden eleméhez, hogy azok pozitívak legyenek; ha nincsenek, akkor a két mátrix marad az eredeti formájában. A függvény egy tuple-t térít vissza, amely a két "normalizált" mátrixot tartalmazza.


```

def normalizeMatrices(m1, m2):

    ms = (m1, m2)

    # Check for the least value in both matrices
    lowestVal = m1.getItem(1, 1)
    for m in ms:
        for i in range(1, m.getNumRows() + 1):
            for j in range(1, m.getNumCols() + 1):
                if m.getItem(i, j) < lowestVal:
                    lowestVal = m.getItem(i, j)

    normMs = (matrix.Matrix(m1.getNumRows(), m1.getNumCols()),
              matrix.Matrix(m2.getNumRows(), m2.getNumCols()))

    # Copy all items from both matrices and add a proper constant
    # to all values
    cnst = 0 if lowestVal > 0 else abs(lowestVal) + 1
    for k in range(0, len(normMs)):
        for i in range(1, ms[k].getNumRows() + 1):
            for j in range(1, ms[k].getNumCols() + 1):
                normMs[k].setItem(i, j, ms[k].getItem(i, j) + cnst)

    return normMs

```

A *createTableaux($m1, m2$)* eljárásnak két argumentuma van, mindkettő egy mátrix. A függvény feladata a **tableaux** létrehozása, vagyis a rendszer együtthatóit, a bázisban levő változókat és az indexeket tartalmazó mátrix felépítése.

A tableaux-nak annyi sora lesz, mint amennyi az $m1$ mátrix sorainak és oszlopainak az összege. (Általánosabban az $m2$ mátrix oszlopainak számát kellene hozzáadni, de a tableaux létrehozásához szükséges az a feltétel, hogy a két mátrix oszlopainak száma megegyezzen. Így mindegy, hogy melyiket adjuk hozzá.)

```

def createTableaux(m1, m2):
    if m1.getNumRows() != m2.getNumRows() or m1.getNumCols() != m2.getNumCols():
        raise ValueError('Selected matrices does not have the same number ' + \
                           'of rows and columns')

    # The total number of strategies of both players
    S = m1.getNumRows() + m1.getNumCols()

    # The tableaux will have S rows, because there are S slack variables
    # and S + 2 columns, because the first column is the index of the basis
    # in the current column and the second column is initially all 1s
    t = matrix.Matrix(S, S + 2)

    # Initialize the first column (index of the current basis variable).
    # Because there are only slack variables at the beginning, initialize
    # it to a sequence of negative numbers starting from -1.
    for i in range(1, t.getNumRows() + 1):
        t.setItem(i, 1, -i)

    # Initialize the second column to all 1s (current value of all basis)
    for i in range(1, t.getNumRows() + 1):
        t.setItem(i, 2, 1)

    # Initialize indices from the first matrix
    for i in range(1, m1.getNumRows() + 1):
        for j in range(1, m1.getNumCols() + 1):
            t.setItem(i, m1.getNumRows() + j + 2, -m1.getItem(i, j))

    # Initialize indices from the second matrix
    for i in range(1, m2.getNumRows() + 1):
        for j in range(1, m2.getNumCols() + 1):
            t.setItem(m1.getNumRows() + j, i + 2, -m2.getItem(i, j))

    return t

```

A már részletezett függvények után a *makePivotingStep*(*t*, *p1SCount*, *ebVar*) függvény ismertetésével folytatjuk. Ennek a függvénynek a feladata, hogy bevigyen egy aktuális változót a bázisba.

Három paraméterrel rendelkezik, a **t** az a tableaux-t jelenti, a **p1SCount** az első játékos stratégiájának sorszámát, míg az **ebVar** a bevinni óhajtott változó indexét az első oszlopból.

```

def makePivotingStep(t, p1SCount, ebVar):
    # 1st precondition
    if abs(ebVar) <= 0 or abs(ebVar) > t.getNumRows():
        raise ValueError('Selected variable index is invalid.')
    # 2nd precondition
    if p1SCount < 0 or t.getNumRows() <= p1SCount:
        raise ValueError('Invalid number of strategies of player 1.')

    # Returns the column corresponding to the selected variable
    def varToCol(var):
        # Apart from players matrices values, there are 2 additional
        # columns in the tableaux
        return 2 + abs(var)

    # Returns the list of row numbers which corresponds
    # to the selected variable
    def getRowNums(var):
        # Example (for a game 3x3):
        #   -1,-2,-3,4,5,6 corresponds to the first part of the tableaux
        #   1,2,3,-4,-5,-6 corresponds to the second part of the tableaux
        if -p1SCount <= var < 0 or var > p1SCount:
            return range(1, p1SCount + 1)
        else:
            return range(p1SCount + 1, t.getNumRows() + 1)

    # Check which variable should leave the basis using the min-ratio rule
    # (it will have the lowest ratio)
    lbVar = None

```

Az alprogram fentebbi látható sorai ellenőrzik, hogy s stratégia sorszáma és az index megfelelő legyen, a *varToCol()* metódus visszaadja annak az oszlopnak a sorszámát, amelyben az aktuális változó jelen van. A *getRowNums()* alprogram pedig visszatéríti azon sorok sorszámából alkotott listát, amelyekben szerepel az illető aktuális változó. A függvény további részében meghatározzuk, hogy melyik változó kell elhaggya a bázist (ami az ezelőtti fejezetben részletezett "legkisebb arány" alapján dől el). Itt alkalmazzuk a *getRowNums()*-t, ami a hatékonyságban játszik szerepet; csak azokban a sorokban számítja ki az arányt, ahol a bázisba bevinni óhajtott változó szerepel. Miután megvan, hogy melyik változó került be a bázisba, azt az annak megfelelő egyenletből kifejezzük, utána pedig a többi egyenletbe helyettesítjük, ahol még szerepelt. A visszatérítési érték az a változó lesz, ami elhagyta a bázist.

```

lbVar = None
minRatio = None
# Check only rows in the appropriate part of the tableaux
for i in getRowNums(ebVar):
    if t.getItem(i, varToCol(ebVar)) < 0:
        ratio = -rational.Rational(t.getItem(i, 2)) /\
            t.getItem(i, varToCol(ebVar))
        if minRatio == None or ratio < minRatio:
            minRatio = ratio
            lbVar = t.getItem(i, 1)
            lbVarRow = i
            lbVarCoeff = t.getItem(i, varToCol(ebVar))
# Update the row in which the variable that will leave the basis was
# found in the previous step
t.setItem(lbVarRow, 1, ebVar)
t.setItem(lbVarRow, varToCol(ebVar), 0)
t.setItem(lbVarRow, varToCol(lbVar), -1)
for j in range(2, t.getNumCols() + 1):
    newVal = rational.Rational(t.getItem(lbVarRow, j)) / abs(lbVarCoeff)
    t.setItem(lbVarRow, j, newVal)
# Update other rows in the appropriate part of the tableaux
for i in getRowNums(ebVar):
    if t.getItem(i, varToCol(ebVar)) != 0:
        for j in range(2, t.getNumCols() + 1):
            newVal = t.getItem(i, j) + t.getItem(i, varToCol(ebVar)) *\
                t.getItem(lbVarRow, j)
            t.setItem(i, j, newVal)
        t.setItem(i, varToCol(ebVar), 0)
return lbVar

```

Ezt követően a *getEquilibrium(t, p1SCount)* metódus meghatározza az egyensúlypontot. A **t**-ben a tableaux-t kapja meg, míg a **p1SCount**-ban az első játékos stratégiának számát. A függvény első részében ellenőrizzük, hogy a stratégiák száma legyen 0 és a tableaux sorainak száma között, illetve a tableaux első sorában levő számok moduluszai 1-től a sorok számáig mindegyik számot kell tartalmazzák. (Azért van szükség a moduluszra, mivel vannak negatív számok is.)

A továbbiakban pedig a rendszer egyenleteiből alkotott mátrix második oszlopát felhasználva megkapjuk mindkét játékos minden stratégiájának relatív gyakoriságát. A függvény ezt téríti vissza egy tuple formájában.

```

def getEquilibrium(t, p1SCount):

    # 1st precondition
    if p1SCount < 0 or t.getNumRows() <= p1SCount:
        raise ValueError('Invalid number of strategies of player 1.')
    # 2nd precondition
    firstColNums = []
    for i in range(1, t.getNumRows() + 1):
        firstColNums.append(abs(t.getItem(i, 1)))
    for i in range(1, t.getNumRows() + 1):
        if not i in firstColNums:
            raise ValueError('Invalid indices in the first column of the tableaux.')

    # I decided to use a list instead of a tuple, because I need
    # to modify it (tuples are immutable)
    eqs = t.getNumRows() * [0]

    # Equilibrium is in the second column of the tableaux
    for i in range(1, t.getNumRows() + 1):
        # Strategy
        strat = t.getItem(i, 1)
        # Strategy probability
        prob = t.getItem(i, 2)
        # If the strategy index or the probability is lower than zero,
        # set it to zero instead
        eqs[abs(strat) - 1] = rational.Rational(0) if (strat < 0 or prob < 0) else prob

    # Convert the found equilibrium into a tuple
    return (tuple(eqs[0:p1SCount]), tuple(eqs[p1SCount:]))

```

Az utolsó részfeladat, aminek megoldásához szükségeltetik egy alprogram, a *normalizeEquilibrium(eq)*. Ennek feladata normalizálni az **eq** paraméteren keresztül megkapott egyensúlypontot. Hasonlóan a többi alprogramhoz, itt is ellenőrzésekkel kezdünk. Az **eq**-nek egy tuple-nek kell lennie, ami 2 elemet tartalmaz, elemei számpárok kell legyenek, valamint a számpárokból szereplő számok kötelező módon racionális számok. Ezután megtörténik a normalizálás, majd a függvény visszatéríti a normalizált egyensúlypontot egy tuple formájában.

```

def normalizeEquilibrium(eq):

    # 1st precondition
    if len(eq) != 2 or (len(eq[0]) == 0 or len(eq[1]) == 0):
        raise ValueError('Selected equilibrium is not valid.')
    # 2nd precondition
    for i in range(0, 2):
        for j in range(0, len(eq[i])):
            if not isinstance(eq[i][j], rational.Rational):
                raise ValueError('Selected equilibrium contains a ' + \
                                   'non-rational number.')

    # Normalizes a single part of the equilibrium (the normalization
    # procedure is the same as with vectors)
    def normalizeEqPart(eqPart):
        probSum = reduce(lambda x, y: x + y, eqPart, 0)
        return tuple(map(lambda x: x * probSum.recip(), eqPart))

    return (normalizeEqPart(eq[0]), normalizeEqPart(eq[1]))

```

A *lemkeHowson(m1,m2)* függvény egybefoglalja az előbbieket, két mátrix argumentuma van, ezeket normalizálja, elkészíti a tableaux-t, beviszi a megfelelő aktuális változókat a bázisba, kiszámítja a normalizált Nash-egyensúlypontot és visszatéríti.

A másik *lh.py* névvel ellátott file(amelyik nincs benne a **src** package-ben) a main-nek felel meg, itt olvassuk ki a bemeneti állományból az adatokat, meghívjuk a *lemkeHowson(m1,m2)* függvényt és kiírjuk az eredményt.

6. fejezet

Az egyensúlypont vizualizációja

A fogolydilemmához tartozó Nash-egyensúlypont vizualizációjához, valamint az észrevételek és adatok ábrázolásához egyaránt egy YouTube videó anyagát használtam fel¹. Ennek mintájára fogom ábrázolni a kifizetéseket és a problémához tartozó egyensúlypontot.

Amint beláthattuk, a fogolydilemmához tartozó Nash-egyensúlypont a $((1,0),(1,0))$ tuple volt. Ez azt jelenti hogy mindkét fogvatartottnak az esetek 100%-ában a vallomás mellett kell döntenie, vagyis, a kifizetési mátrixot vizsgálva a $(6,6)$ pont lesz az egyensúlypont, mivel ha mindketten vallanak, akkor ezek az értékek a "hasznok".

6.1. Adatok és eredmény vizualizációja

Az alábbi képen a vizualizáció implementálása található a *visualizationEquilibrium* függvényben. Először megadjuk listákban a kifizetéseket, ezt követően kirajzoljuk a 4 színes egyenest. Mindegyik egyenes mellé fel van tüntetve, hogy mit jelöl, például a piros színű azt mutatja meg, amikor a második játékos az első stratégiáját alkalmazza, az elsőről pedig nem tudjuk, hogy hogyan döntött. Ábrázoljuk pontokkal a kifizetés-párokat, megadjuk, hogy a tengelyeken az értékek milyen intervallumban legyenek feltüntetve.

Mivel a kifizetési mátrixban levő értékek minél nagyobbak, annál rosszabbat jelennek a feleknek, így a Nash egyensúlypont ott fog kialakulni, ahol az x és y változók szorzata minimális; ez pedig azon az egyenesen lesz, amely átmegy a $(6,6)$, $(1,11)$ és $(11,1)$ pontokon.

Nem nehéz belátni, hogy ennek az egyenesnek az egyenlete $x + y = 12$ alakban írható fel. Innen kifejezve y -t, megkapjuk, hogy $y = 12 - x$, tehát: $x \cdot y = 12 \cdot x - x^2$. A $n(x)$ függvény ezt a kifejezést téríti vissza, majd a főfüggvényben meghatározzuk a kifejezés deriváltjának zérushelyét, az lesz az x koordináta, y -t pedig az $y = 12 - x$ alapján számítjuk ki.

Így meghatároztuk az egyensúlypontot, amit ábrázolunk.

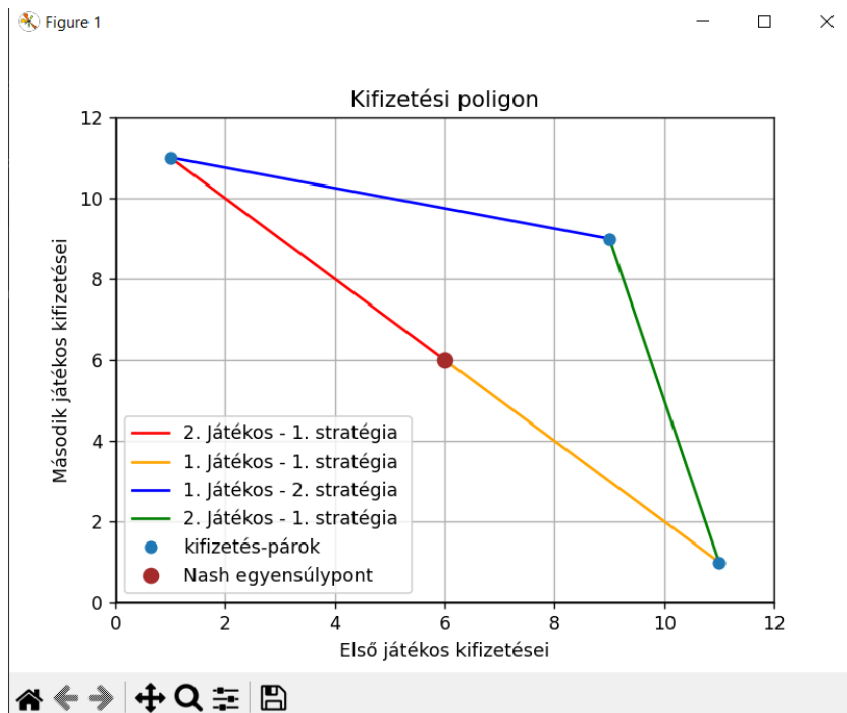
¹Ábrázolás

```
def visualizationEquilibrium():
    first_player = [6, 11, 1, 9]
    second_player = [6, 1, 11, 9]
    fig1 = plt.figure()
    ax1 = fig1.add_subplot(111)
    plt.plot([1, 6], [11, 6], 'red', label='2. Játékos - 1. stratégia')
    plt.plot([6, 11], [6, 1], 'orange', label='1. Játékos - 1. stratégia')
    plt.plot([1, 9], [11, 9], 'blue', label='1. Játékos - 2. stratégia')
    plt.plot([11, 9], [1, 9], 'green', label='2. Játékos - 1. stratégia ')
    plt.plot(first_player, second_player, 'o', label='kifizetés-párok')
    plt.xlim(min(first_player) - 1, max(first_player) + 1)
    plt.ylim(min(second_player) - 1, max(second_player) + 1)
    plt.axhline(color='black')
    plt.axvline(color='black')
    plt.title('Kifizetési poligon')
    plt.xlabel('Első játékos kifizetései')
    plt.ylabel('Második játékos kifizetései')

    x = sy.Symbol('x')
    dn = sy.diff(n(x), x)
    soln = sy.solve(dn)
    plt.plot(soln, 12 - soln[0], 'o', markersize=8, color='brown', label='Nash egyensúlypont')
    plt.legend()
    plt.grid()
    plt.show()

def n(x):
    return 12 * x - x ** 2
```

Az alprogram az alábbi grafikont eredményezi:



6.2. A három pont közötti összefüggés ábrázolása

Amit észrevettem a számítások végén illetve a grafikonok tanulmányozása után, hogy a Nash-egyensúlypont, a valószínűségi eloszlás és a normalizálatlan valószínűségi eloszlás között szoros kapcsolat van.

Ehhez a *src* package *lh.py* file-jában, felhasználva a régebbi függvényeket írtam még egy újabb metódust, ami a normalizálatlan egyensúlypontot téríti vissza.

```
def unnormalizedEquilibrium(m1,m2):  
  
    (normM1, normM2) = normalizeMatrices(m1, m2)  
    t = createTableaux(normM1, normM2)  
    p1SCount = normM1.getNumRows()  
    initBasisVar = 1  
    leftBasisVar = makePivotingStep(t, p1SCount, initBasisVar)  
    while abs(leftBasisVar) != initBasisVar:  
        leftBasisVar = makePivotingStep(t, p1SCount, -leftBasisVar)  
    return getEquilibrium(t, p1SCount)
```

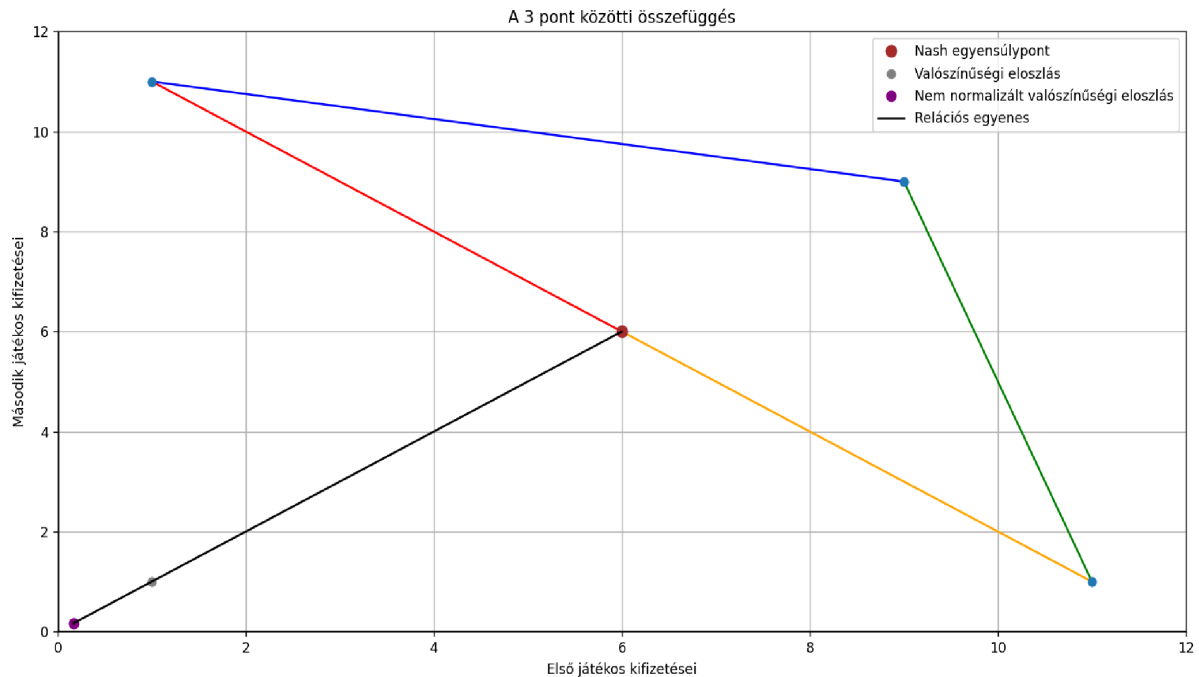
Ezt a függvényt is meghívom a main-ben. A *visualizationCorrelation()* nevű függvényben pedig ábrázolom ezt a 3 pontot. Az egyensúlypont a (6,6), a valószínűségi eloszlásból kiválasztjuk az elemek első elemeit, ez lesz az (1,1) pont, a normalizálatlan eloszlásnál hasonlóan járunk el, ezáltal megkapjuk az (1/6,1/6) pontot.

Könnyen észrevehető, hogy ezek a pontok kollineárisak, mégpedig az $x = y$ egyenletű egyenesen helyezkednek el (első szögfelező). Ami még érdekes, hogy ha az egyensúlypont x koordinátáját megszorozzuk a normalizálatlan eloszlás x koordinátájával és az y koordinátákra hasonlóan, megkapjuk a normalizált eloszlás koordinátáit ($6 \cdot 1/6 = 1$).

Az alábbi kép ennek ezt az összefüggést ábrázolja. (Csak a lényeges kódsorok szerepelnek a képen, mert a függvény nagy része hasonlít a *visualizationEquilibrium* függvényre.)

```
plt.plot(soln, 12 - soln[0], 'o', markersize=8, color='brown', label='Nash egyensúlypont')  
plt.plot(1,1,'o', markersize=6, color='grey', label='Valószínűségi eloszlás')  
plt.plot(1/6,1/6, 'o', markersize=7, color='purple', label='Nem normalizált valószínűségi eloszlás')  
plt.plot([1/6,6],[1/6,6],color='black',label='Reláció egyenes')
```

Az alprogram megjeleníti a 3 említett pontot illetve az egyenest, aminek ezek elhelyezkednek.



6.3. Az egyensúlypont összefüggése a derékszögű háromszöggel

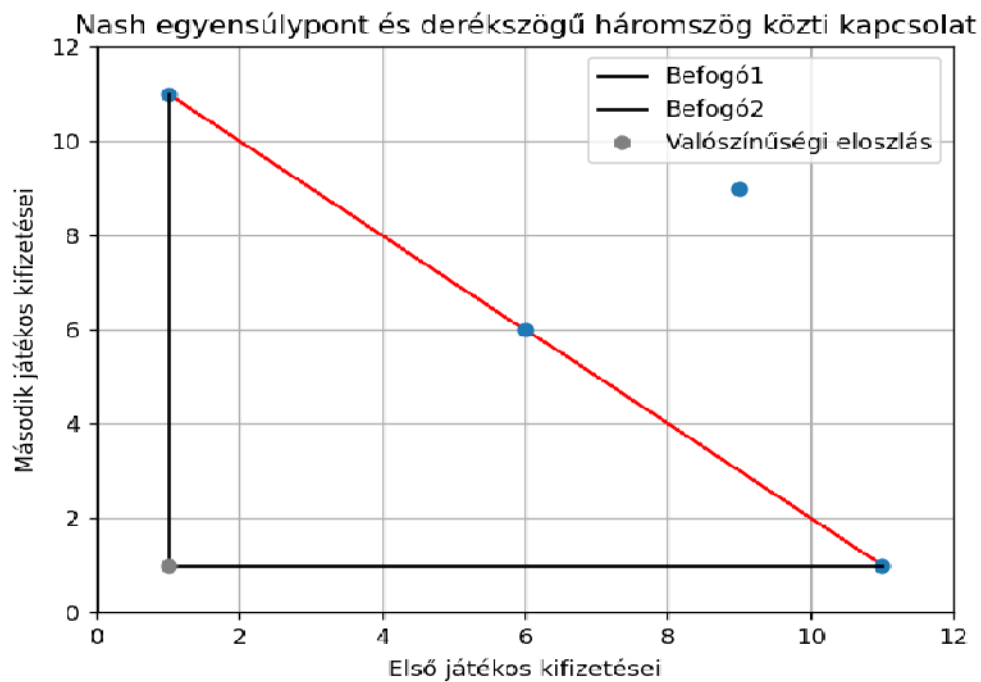
Az előbb bemutatott összefüggés mellett, egy másik érdekességre figyeltem fel az ábrák tanulmányozása során. Az észrevételem az, hogyha szerkesztünk egy derékszögű háromszöget, amelynek az átfogója a feladatnak megfelelő politóp legalacsonyabban elhelyezkedő oldala (ezt az oldalt az ábrán pirossal fogom színeezni), akkor a háromszög harmadik csúcsa (vagyis ahol létrejön a derékszög) a normalizált valószínűségi eloszlás koordinátaival egybeesik.

Vagyis a harmadik csúcs x koordinátája az átfogó magasabban levő csúcsának x koordinátájával egyenlő, míg az y koordináta az átfogó alacsonyabban elhelyezkedő csúcsának y koordinátájával egyezik meg.

Hasonlóan, mint az előbb, ebben az esetben is csak a lényeges kódsorokat fogja tartalmazni a soron következő kép. Ábrázolom a `rightTriangle()` függvényben a politóp legalacsonyabban levő oldalát, vagyis az átfogót, a befogókat és a normalizált valószínűségi eloszlásnak megfelelő pontot.

```
plt.plot([1, 6], [11, 6], 'red')
plt.plot([6, 11], [6, 1], 'red')
plt.plot([1, 1], [11, 1], color='black', label='Befogó1')
plt.plot([11, 1], [1, 1], color='black', label='Befogó2')
plt.plot(1, 1, 'o', markersize=6, color='grey', label='Valószínűségi eloszlás')
```

A következő grafikont kapjuk a függvény meghívása után.



A fenti ábrán a késsel jelölt pontok a kifizetés-párokat jelentik, a piros színű szakasz a már fentebb részletezett ábrákon látható politóp legalacsonyabban elhelyezkedő oldala (a keletkezett derékszögű háromszög átfogója).

A fekete szakaszok pedig a háromszög befogói lesznek. Az ábrán jól látható, hogy a két befogó metszéspontja és a szürke színnel ábrázolt, valószínűségi eloszlásnak megfelelő pont egybeesik.

7. fejezet

Összegzés és továbbfejlesztési lehetőségek

Dolgozatomban egy különleges geometrai egyensúlypont, mégpedig a játékelmélethez kapcsolódó Nash-egyensúlypont vizualizációjával foglalkoztam.

Ahhoz, hogy ez megoldható legyen, először is szükség volt a játékelméleti alapfogalmak ismertetésére. Ezen kívül egy játéknak minősíthető helyzetre, aminek megfelelt a fogoly-dilemma probléma. Miután ezt a problémát részleteztem, bemutattam a Lemke-Howson algoritmust elméleti, matematikai és programozás szempontból egyaránt.

Ezt követően, az eredmény értelmezése után, PyCharm fejlesztői környezetben vizualizáltam a játék adatait, eredményeit, összefüggéseit.

Továbbfejlesztési lehetőségnek egy interaktív weboldalt képzelnék el, ahol a felhasználónak lehetőségében áll megszabni a játék adatait, szabályait. Ezenkívül tetszőleges vizualizációkat készíthet, változtatva bizonyos paramétereket, tanulmányozva így az eredmény módosulását.

Mivel a Lemke-Howson algoritmus kizárólag kétszemélyes játékok esetében alkalmazható, ezért a kibővítést lehetne fokozni még olyan algoritmus vagy eljárás matematikai levezetésével illetve implementációjával, ami 3 vagy még több játékos esetén határozza meg a Nash-egyensúlypontot.

A programozáshoz a GitHub verziőkövető alkalmazást használtam, az én publikus GitHub repositorym elérhetősége: <https://github.com/Lehel812/State-exam>.

Köszönetnyilvánítás

Meg szeretném köszönni a témavezető oktatóimnak, Dr. Farkas Csabának és Olteán-Péter Borókának, hogy a dolgozatomban segítségemre voltak és támogattak, bármilyen jellegű probléma, kérdés vagy elakadás merült fel.

Általuk betekintést nyerhettem a játékelmélet csodálatos és érdekes világába, valamint újabb hasznos matematikai-és Python programozással kapcsolatos ismeretekkel illetve készségekkel gazdagodtam. Nélkülük nem jöhetett volna létre ez a dolgozat.

Irodalomjegyzék

- [Eld] Stephen Eldridge. Nash equilibrium.
- [Kre89] David M Kreps. Nash equilibrium. In *Game Theory*, pages 167–177. Springer, 1989.
- [Liu13] Kunpeng Liu. Lemke and howson algorithm. 2013.
- [oEBa] The Editors of Encyclopaedia Britannica. John nash.
- [oEBb] The Editors of Encyclopaedia Britannica. positive sum game.
- [PDP06] Irene Parrachino, Ariel Dinar, and Fioravante Patrone. Cooperative game theory and its application to natural, environmental, and water resource issues: 3. application to water resources. *Application to Water Resources (November 2006)*. *World Bank Policy Research Working Paper*, (4074), 2006.
- [Pri] David Pritchard. Game theory and algorithms lecture 6: The lemke-howson algorithm.
- [Unka] Unknown. Fogolydilemma.
- [Unkb] Unknown. Nash egyensúly.
- [Zag84] Frank C Zagare. Game theory: Concepts and applications. 1984.