

**SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK**



SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

Algoritmika a Sakktáblán

DIPLOMADOLGOZAT

Témavezető:
Dr. Kátai Zoltán,
Egyetemi docens

Végzős hallgató:
Török Csongor

2023

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



UNIVERSITATEA
SAPIENTIA

Algoritmă pe Tabla de Sah

LUCRARE DE DIPLOMĂ

Coordonator științific:
Dr. Kátai Zoltán,
Conferențiar universitar

Absolvent:
Török Csongor

2023

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



SAPIENTIA
HUNGARIAN UNIVERSITY
OF TRANSYLVANIA

Algorithms on the Chessboard

BACHELOR THESIS

Scientific advisor:
Dr. Káta Zoltán,
Associate professor

Student:
Török Csongor

2023

Declarație

Subsemnatul/a Török Csongor, absolvent(ă) al/a specializării
Informatică, promoția 19-22, 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,

Data:

Tg Mureș
2023.06.13

Absolvent

Semnătura

Török Csongor

LUCRARE DE DIPLOMĂ

Coordonator științific: Dr. Kátai Zoltán

Candidat: Török Csongor
Anul absolvirii: 2023

a) Tema lucrării de licență:

Algoritmica pe tabla de șah

b) Problemele principale tratate:

Probleme de șah de dominare

Probleme de șah de independență

Combinatia celor două probleme menționate anterior

Parcurgerea tabloului cu un cal (Amibițiile cavalerului)

Joc interactiv de logică pe tabla de șah

c) Desene obligatorii:

Diagrama cazurilor de utilizare

Diagramă de secvență

Diagramă de arhitectură

d) Softuri obligatorii:

Aplicație care ilustrează soluțiile algoritmice a problemelor susmenționate și face posibil studierea lor printr-un mod de lucru interactiv.

e) Bibliografia recomandată:

Burger, A. P. (1998). The queen's domination problem (Doctoral dissertation).

Squirrel, D., & Çull, P. (1996). A Warnsdorff-Rule Algorithm for Knight's Tours.

Gibbons, P. B., & Webb, J. A. (1997). Some new results for the queens domination problem. Australasian Journal of Combinatorics, 15, 145-160.

f) Termene obligatorii de consultații: lunar

g) Locul și durata practicii: Universitatea „Sapientia” din Cluj-Napoca,
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș.


Primit tema la data de: 20.06.2022

Termen de predare: 02.07.2023

Semnătura Director Departament



Semnătura responsabilului
programului de studiu



Semnătura coordonatorului



Semnătura candidatului



Kivonat

A történelem során, kevés olyan dolog maradt meg, amely apró bővítéseken kívül, szinte változatlan maradt, ugyanazt a táblán lezajló játékot jelentette a korai középkorban élő embereknek, mint amit napjainkban is millióknak. Nincsenek rejtett részei vagy bármi ami véletlenre lenne bízva, minden a játékosok előtt áll a kezdettől egészen a mattig.

Mindezek ellenére mégis, az egyik legnagyobb változást az elmúlt évtizedek informatikusai hozták, a különböző sakkos algoritmusokkal, programokkal. Mélyebb belátást nyújtottak a játék világába, kiszámítva bármelyik pozícióból a legjobb lépéseket, számunkra felfoghatatlan mennyiségű lehetőségek közül. Ezek fejlődésével egy teljesen új dimenziót nyert a játék, minden részletét elemezni tudjuk, a leoptimálisabb stratégiát másodpercek alatt képesek kigenerálni.

Dolgozatomban ezen programok sorait szeretném én is bővíteni, megoldásokat keresve sakkos problémákra, rejtvényekre amelyek akár évezredekkel ezelőtt is foglalkoztatták az embereket.

Az algoritmusaim lépésről lépésre mutatják miképpen keresnek megoldást az adott esetekre, mindezeket magyarázatokkal kiegészítve. Céлом ezzel a sakk komplexitásán kívül, ezen programok egyértelművé alakítását bárki számára, interaktív keretek között.

A sakkos algoritmusaim többféle módon keresnek megoldást ezekre a problémákra, adott példában a jelenlegi helyzetet kielemezve a legjobb lépéseket választják vagy a lehetséges eseteket végigtesztelve kapnak megoldást. Ezek futási ideit elmentve, törekedtem hogy a lehatékonyabb módszerek segítségével, látványosan és könnyen érthetően mutassam be a sakkos problémákat.

Nem utolsó sorban, a sakk népszerűsítését, ez iránti érdeklődés felkeltését is célomnak tűztem ki, hiszen bárki lehet mestere a királyok játékanak, hasonló programok nagyban segíthetnek ezen elsajátításában.

Rezumat

De-a lungul istoriei, doar câteva lucruri, în afară de îmbunătățirile sale minore, au rămas aproape neschimbate de la Evul Mediu până în prezent, reprezentând încă același joc de societate pentru milioane de oameni. Nu are părți ascunse sau elemente de șansă, totul stă în fața celor doi jucători de la bun început până la șah-mat.

Cu toate acestea, una dintre cele mai mari schimbări a fost adusă de informaticienii din ultimele decenii, cu diverși algoritmi și programe de șah. Au oferit o perspectivă mai profundă asupra jocului, calculând cele mai bune mișcări din orice poziție, dintr-o cantitate insondabilă de posibilități pentru noi. Odată cu dezvoltarea lor, jocul a căpătat o dimensiune complet nouă, putem analiza fiecare detaliu, putem genera cea mai optimă strategie în câteva secunde.

În teza mea, aș dori, de asemenea, să extind rândurile acestor programe, căutând soluții la problemele de șah și puzzle-urile care au ocupat oamenii chiar și cu mii de ani în urmă.

Algoritmii mei arată pas cu pas cum găsesc soluții pentru cazuri specifice, complet cu explicații. Pe lângă complexitatea șahului, scopul meu este să fac aceste programe clare pentru oricine, într-un cadru interactiv.

Algoritmii mei de șah caută soluții la aceste probleme în mai multe moduri, într-un exemplu dat ei aleg cei mai buni pași analizând situația actuală sau obțin o soluție testând toate cazurile posibile. Salvând timpii de rulare a acestora, am încercat să prezint problemele de șah într-un mod ușor de înțeles și plăcut vizual cu ajutorul celor mai eficiente metode.

Nu în ultimul rând, mi-am propus popularizarea șahului și să trezesc interesul față de el pentru alții, deoarece oricine poate fi expert în jocul regilor, fiindcă programele similare pot ajuta foarte mult la stăpânirea lui.

Abstract

Throughout history, only a very few things, apart from its minor improvements, have remained almost unchanged from the early Middle Ages to our present times, still representing the same board game for millions of people. It doesn't have any hidden parts or any elements of chance, everything stands in front of the two players from the very beginning to until the checkmate.

In spite of all these, one of the biggest change brought to chess were the programs and algorithms made by computer scientist from the past decades. These offered a deeper insight into the world of chess, calculating the best steps from any given position, from an unfathomable number of choices. As these programs evolved, the game received a whole new dimension of depth, since we could analyze every single detail, the most optimal strategy could be generated in mere seconds.

With my thesis, I would like to expand the ranks of these programs, looking to solve chess problems and puzzles, which even thousands of years ago could have occupied people.

My algorithms show step-by-step how they find solutions for the given cases, with added explanations. My goal, in addition to chess' complexity, is to clarify these algorithms for anyone, with an interactive interface.

My chess algorithms look for solutions to these problems in several ways, in a given example they choose the best steps by analyzing the current situation or they get a solution by testing all the possible cases. By saving the running times, I tried to present the chess problems in easily understandable and visually pleasing way with the help of the most effective methods.

Last but not least, popularizing chess, arousing interest in it for others was another goal of mine, since anyone can be an expert of the game of kings, similar programs can help a lot in mastering it.

Tartalomjegyzék

1. Bevezető	10
2. Programok, technológiák bemutatása	12
2.1. Felhasznált szoftverek, könyvtárak	12
2.1.1. Visual Studio	12
2.1.2. C++	12
2.1.3. Felhasznált könyvtárak	13
2.2. Rendszer követelmények	15
2.2.1. Funkcionális követelmények	15
2.2.2. Nem funkcionális követelmények	15
2.3. Általam megvalósított szoftver	17
2.4. A felhasználói felület	17
2.4.1. Megoldott problémák illetve interaktív részek	20
2.5. Problémák algoritmusai	20
2.5.1. N Királynő probléma	24
2.5.2. Különálló királynő lefedés	26
2.5.3. Királynő lefedés	27
2.5.4. Huszár turné	30
2.5.5. Bátya lefedés	35
2.5.6. Futár lefedés	37
2.5.7. Huszár lefedés	38
2.6. Sakk kihívás	41
3. Eredmények, következtetések	43
3.1. Problémák megoldási módszerei	43
3.2. Futási idők és megfigyelések	45
3.3. Összességében	47
Összefoglaló	48
Ábrák jegyzéke	49
Táblázatok jegyzéke	50
Irodalomjegyzék	51

1. fejezet

Bevezető

A sakk szabályait minden átlag ember ismeri. Egy kontinenseket, kultúrákat, időt áthidaló játék, alap elméletének és elérhetőségének köszönhetően. Bárki, kortól függetlenül élvezheti, legyen ő kezdő, vagy világszintű bajnok, a játék nyolcszor nyolcas tábla keretein belül az elmék csatáját több millióan vívják élőben és online is. Első ránézésre nem tűnik túl nehéznek, 6 különböző figura nem túl eltérő módon járhatja be a táblát, azzal a céllal hogy az ellenfél királyát legyőzze.

Valódi komplexitására már a középkorban is rájöttek, könyvekbe foglalva a háttérben rejlő elméleteket, kezdő lépéseket, végjátékairól és a szinte megszámlálhatatlan variációkat, (kb. 2^{64}) amelyek lejátszhatók a 32 bábuval.

Mindez a komplexitás akkor se szűnik meg, amikor részeire törjük, különböző rejtvényekben, átlagos játékokban nem előforduló eseteket tanulmányozunk, és próbálunk megoldani.

Vegyük az egyik alapvetőbb problémát, amelyre megoldást keresek a dolgozatomban, az N Királynős probléma, miszerint egy $N \times N$ -es táblán szeretnénk elhelyezni N darab királynőt, úgy hogy 2 királynő ne legyen ugyanabban a sorban, oszlopban vagy átlóban, vagyis ne üssék egymást. Első ránézésre egy egyszerű problémának tűnik, úgy gondolnánk hogy nem helyezhetünk el túl sok variációt. Ellenkezőleg viszont, mindez a tábla méretétől függően exponenciálisan növekvő számú megoldásokat eredményez. Sorok és oszlopok bővítéssel, a legnagyobb lefedéssel rendelkező bábuval is elérhetjük az alap játék komplexitását.

Méret	Megoldások száma
5×5	10
8×8	92
10×10	742
15×15	2, 279, 184
20×20	39, 029, 188, 884
25×25	2, 207, 893, 435, 808, 352

1.1. táblázat. Egyedi megoldások az N Királynős problémára

A számokon és más adatok megjelenítésén kívül, egyértelműen fontosnak tartottam hogy vizualizálva is legyen egy alapabb, két dimenziós sakktábla felületén mindez. Egyértelmű jelöléssel a tábla kereteit, négyzeteit ábrázoltam, sakkos koordinátákkal kiegészítve,

Interaktív oldala a projektemnek az algoritmusok összetettségéből következett, mivel csupán a végső eredmények kigenerálása, az algoritmus működésének megértésében egyáltalán nem tűnt hasznosnak. Elérhetőbbé akartam változtatni, az emberi gondolkodást összekötni, a játék alap elméletén keresztül, a gépi algoritmussal, didaktikai céllal is. Eből kifolyólag, debuggolás szerűen, minden fontosabb lépésnél, ellenőrzésnél, elhelyezésnél vagy törlésnél magyarázattal bővítettem ki az alap probléma megoldó programokat.

	A	B	C	D	E	F	G	H	
	Q	#####	#####	#####	#####	#####	#####	#####	1
		#####	#####	#####	#####	#####	#####	#####	
		#####	#####	#####	#####	#####	#####	#####	
<hr/>									
	#####	#####	Q	#####	#####	#####	#####	#####	2
	#####	#####		#####	#####	#####	#####	#####	
	#####	#####		#####	#####	#####	#####	#####	
<hr/>									
	#####	#####	#####	#####	Q	#####	#####	#####	3
	#####	#####	#####	#####		#####	#####	#####	
	#####	#####	#####	#####		#####	#####	#####	
<hr/>									
	#####		#####	#####	#####	#####			4
	#####		#####	#####	#####	#####			
	#####		#####	#####	#####	#####			
<hr/>									
	#####		#####		#####	#####	#####		5
	#####		#####		#####	#####	#####		
	#####		#####		#####	#####	#####		
<hr/>									
	#####	#####	#####		#####	#####	#####	#####	6
	#####	#####	#####		#####	#####	#####	#####	
	#####	#####	#####		#####	#####	#####	#####	
<hr/>									
	#####		#####		#####		#####	#####	7
	#####		#####		#####		#####	#####	
	#####		#####		#####		#####	#####	
<hr/>									
	#####		#####		#####			#####	8
	#####		#####		#####			#####	
	#####		#####		#####			#####	
<hr/>									

A felhasználó változtathat ezen, annak függvényében hogy szeretné-e lépésenként tanulmányozni, vagy csak a végső eredmény érdekli bármelyik implementált problémánál. Továbbá, a problémák algoritmusain kívül egy Sakkos kihívás menüpont is található, amelyben különböző feladatokra saját magunknak kell megoldást találnunk.

2. fejezet

Programok, technológiák bemutatása

2.1. Felhasznált szoftverek, könyvtárak

Dolgozatom ezen részében először beszélnék a felhasznált programról, a nyelvről, és ezen belül a könyvtárakról amelyek segítségével a problémákat és a felhasználói felületet sikerült megvalósítanom.

2.1.1. Visual Studio

Projektemet a Visual Studio integrált fejlesztői környezetében írtam meg, amely tartalmaz minden szükséges eszközt a software fejlesztéshez : egy kód szerkesztőt, fájlkezelőt, kompilálást és futtatást egyszerűsítő funkciókat illetve egy debuggoló rendszert. Több programozási nyelvet is támogat a fejlesztői környezet, a C++-t is, amiben a projektemet írtam.

2.1.2. C++

A C++ nyelvre esett a választásom projektem kivitelezésénél, több okból kifolyólag is:

- **Teljesítmény és sebesség szempontjából** : alacsony szintű programozási nyelv amely lehetővé teszi a közvetlen hozzáférést a hardverhez és ezek hatékony kihasználtságát, amely kritikus lehet nagy mennyiségű adatok feldolgozásánál, jelen esetben akár egy 25x25 mátrix tárolásánál és többszörös ellenőrzésénél
- **Rugalmasság és könnyen irányíthatóság** : széleskörű lehetőségeket nyújt objektum orientált és procedurális programozási paradigmákba egyaránt
- **Cross-Platform támogatottság** : bármilyen operációs rendszeren futtatható kód
- **Széleskörű és kiterjedt aktív közössége** : adott problémákra számos megoldás, dokumentáció található cikkekben, fórumokon, elterjedtségének, illetve C-re épülő kompatibilitásának köszönhetően

```

void basic_print(int table[50][50])
{
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            std::cout << table[i][j] << ' ';
        }
        std::cout << '\n';
    }
}

```

2.1. kódrészlet. C++ példakód egy alap 2D tömb kiíratására



C++ kompilálási folyamata

2.1.3. Felhasznált könyvtárak

- `iostream` :
 - C++ alapvető könyvtára, amely a beolvasás és kiíratás parancsait implementálja, a standard felületen belül
- `fstream` :
 - fájlokból való beolvasás, illetve ezekbe a kiíratásra használt parancsok könyvtára
 - futási idők elmentésére, a beállítások eltárolására használtam

- `conio.h` :
 - ugyancsak egy I/O könyvtár, amely a konzolba írt adatok direkt olvasására használók
 - az `iostream`-el szemben, a billentyűzeten bármelyik karakter lenyomása után, Enter megnyomása nélkül beolvassa a karaktert a `getch()` parancs
- `chrono` :
 - idő intervalumok mérésére
 - az algoritmusok futási ideinek kiszámítására, miliszekundumokban
- `Windows.h` :
 - egy Windows operációs rendszer specifikus könyvtár
 - nagyobb felhozatalából legtöbbször a `Sleep()` parancsnak veszem hasznát, amely megállítja a programot a paraméterében megadott miliszekundumig

```
#include <iostream>
#include <Windows.h>
#include <conio.h>

using namespace std;

int main()
{
    char c;
    c = _getch();

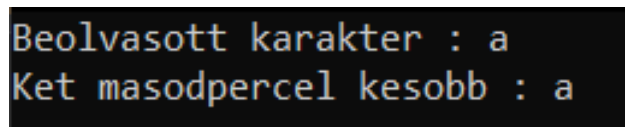
    cout << "Beolvasott karakter : " << c;
    cout << endl;

    Sleep(2000);

    cout << "Ket masodpercel kesobb : " << c;
    cout << endl;
    return 1;
}
```

2.2. kódrészlet. Példa a `Sleep()` és `getch()` parancs használatára

A két kiiratás között 2 másodperc (2000 ms) telt el.



```
Beolvasott karakter : a
Ket masodpercel kesobb : a
```

2.1. ábra. A kódrészlet eredménye az 'a' karakter lenyomása után

2.2. Rendszer követelmények

2.2.1. Funkcionális követelmények

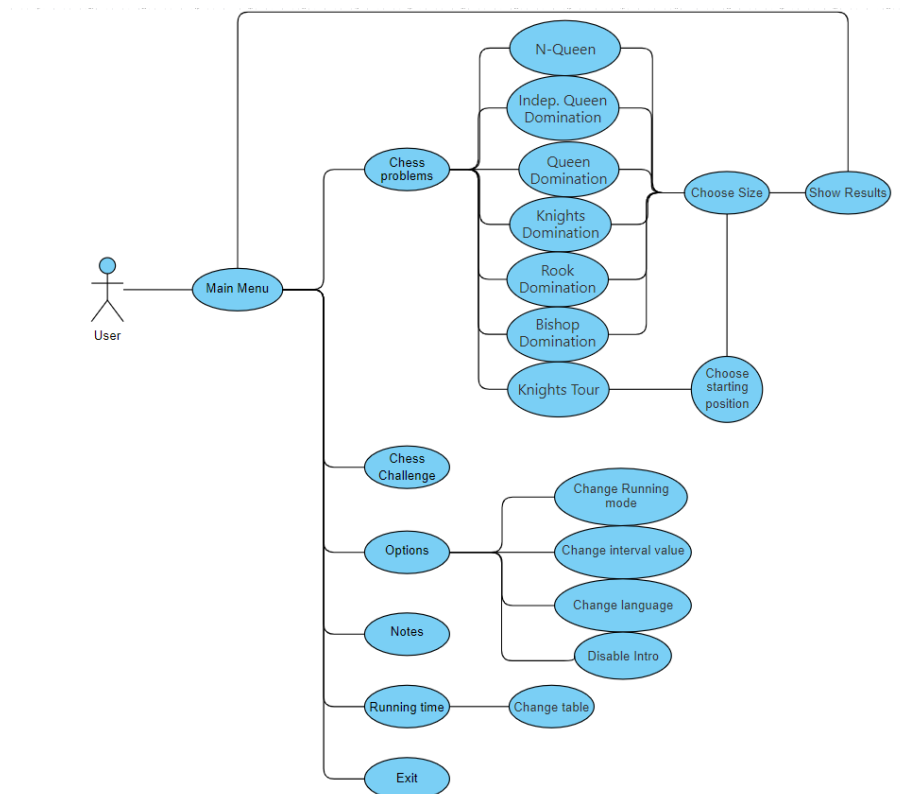
A program használatához nélkülözhetetlen egy billentyűzet, mivel egy terminál felületen jelenik minden része. A képernyőn megjelenő számok lenyomásával navigálhatunk az adott menüpontokba. Más esetben, a program értesít a használandó gombokról, visszalépéseknél bármely leütött billentyűvel tovább lépünk.

A program futtatható a projekt Debug folderjében levő ChessApp.exe fájl által. Ellenkező esetben, a Visual Studio felületén keresztül is futtatható, a projekt fájl megnyitásával és kompilálásával.

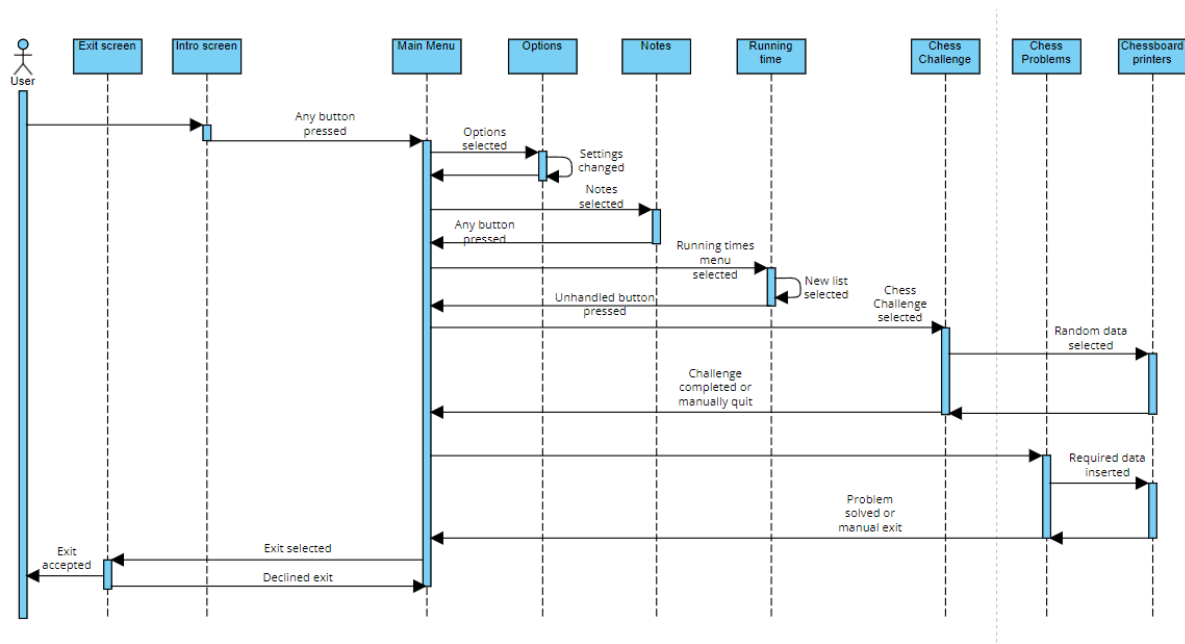
2.2.2. Nem funkcionális követelmények

Révén hogy a Windows.h könyvtárat is alkalmazom, kizárólag Windows alapú operációs rendszereken működik a program. A 2.1.3 fejezetben említett könyvtárak szükségesek, egy C++ kompiláló illetve a Visual Studio minimális rendszerkövetelményei.

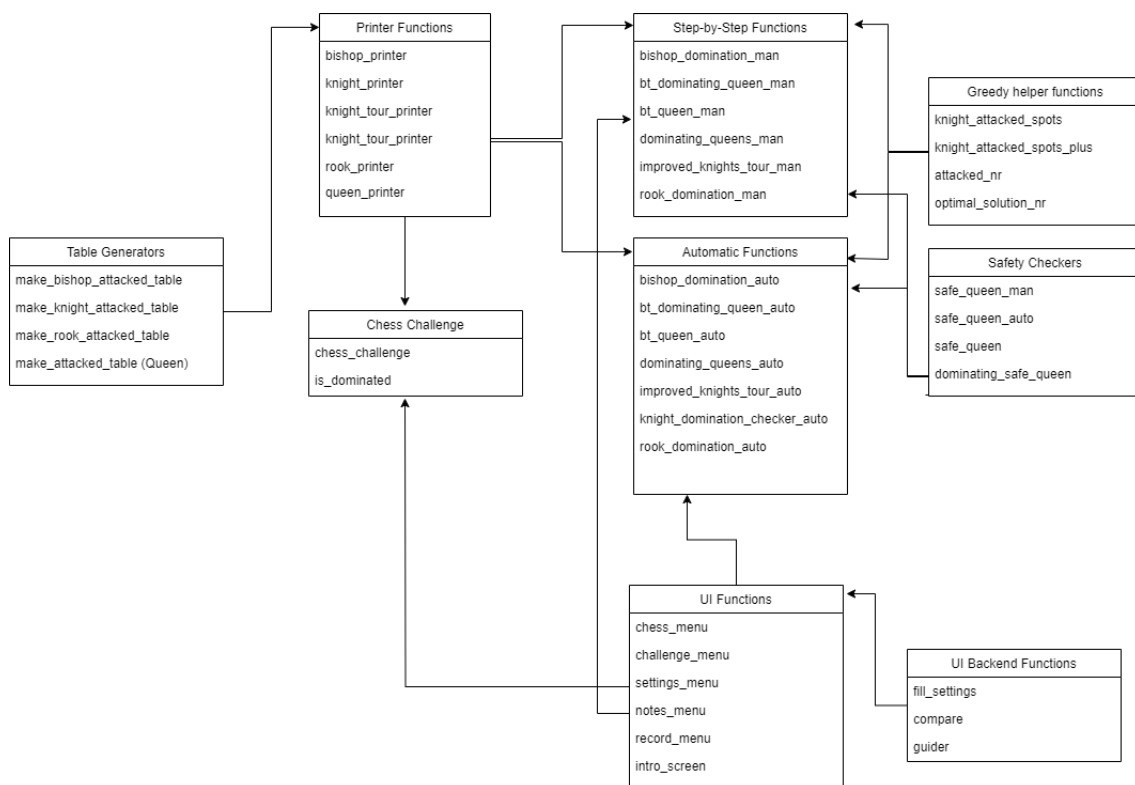
- 1.8 GHz vagy gyorsabb processzor
- 2 GB RAM
- Minimum 800 MB tárhely



2.2. ábra. A projekt Use Case diagramja



2.3. ábra. A projekt Sequence diagramja



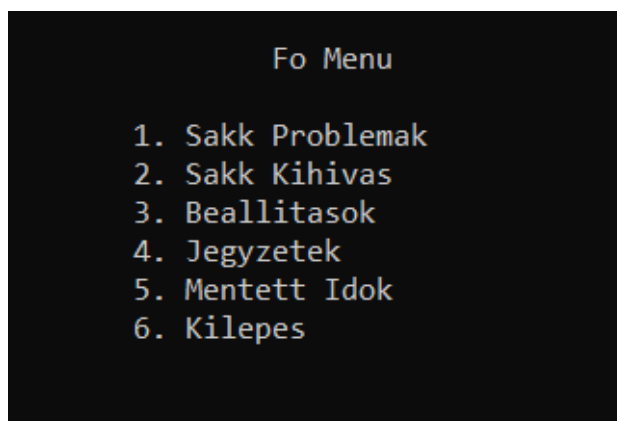
2.4. ábra. A projekt függvényeinek architektúrája

2.3. Általam megvalósított szoftver

A sakkos problémák előtt, a felhasználói felület fontosabb részeit mutatnám be részletesebben, a keretrendszert amellyel elérhetjük ezeket és változtathatjuk beállításait, futási módjait.

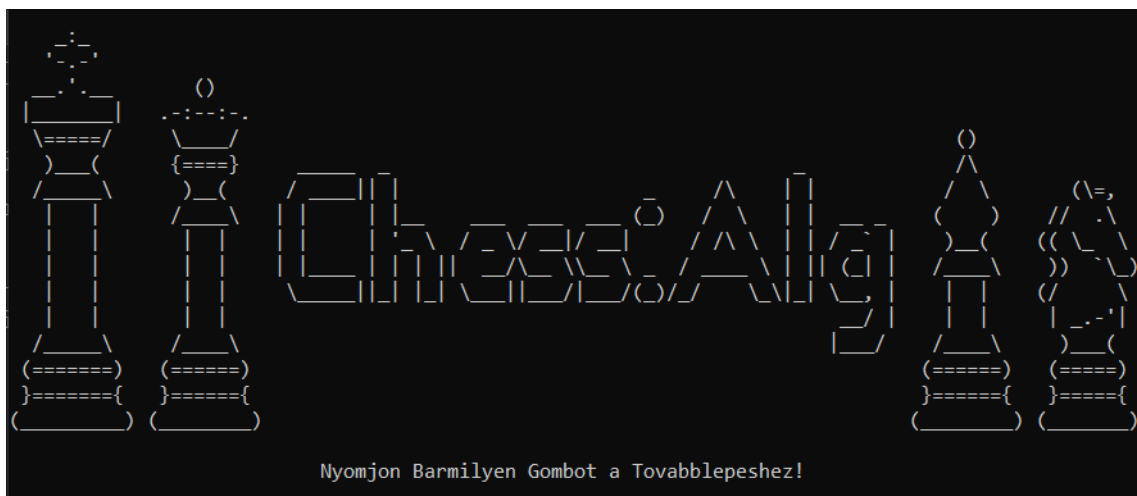
2.4. A felhasználói felület

Program elindításakor alapértelmezetten egy bevezető kép fogad, amely tartalmazza a projektem stilizált nevét, ahonnan továbblépve a fő menübe jutunk.



2.5. ábra. A fő menü, magyar nyelven

Az adott menüpontnak megfelelő szám megnyomásával történik a navigálás, használva a `getch()` parancsot a lenyomott karakter beszerzésére. Ezt követően switch-ek segítségével, a terminál tartalma minden kiírása előtt törlődik, és a lenyomott gombnak megfelelő újabb menüpont szövege veszi át a helyét. Adott esetekben, különböző kódrészletek átugrásához vagy előző kiírásokhoz való visszalépéshez, a "goto" parancsot használom. A felület szövegét a `system("cls")` parancs segítségével törlöm.



2.6. ábra. A bevezető grafika a program elindításakor

Minden menüpontban, ha megadott számokon kívüli karaktert ütünk be, ugyanazon menü újra töltődik, hibák elkerülése érdekében.

A teljes felület két nyelven van, alapértelmezetten magyarul vagy átállítható angolra minden szöveg, amihez hozzá tart a problémák magyarázata is.

Mindezt a Beállítások menüpontban változtathatjuk, ahol még dinamikusan cserélhetők a következő változók is :

- **A programok futási módja** : itt tudjuk megváltoztatni hogy lépésről-lépésre, vagy automatikus módon fussanak a probléma megoldó algoritmusok
 - **Lépésről-lépésre** : a program csak akkor iterál, ha a felhasználó lenyom egy gombot a billentyűzetén, vagy megáll 'a' beütése után
 - **Automatikus** : a megadott intervalum értéke szerint lépked a program
- **A lépések közötti intervalum** : automatikus módban használt intervalum miliszekundumban, amit ha 0-ra állítunk csak a végső megoldást fogja kigenerálni
- **Bevezető** : A program elindításakor megjelenő grafika be- vagy kikapcsolásához

Ahogy bármelyik értéket változtatjuk ezek közül, a settings.txt fájlba elmentett értékek felülíródnak, és elmentődnek. Ha a programot újraindítjuk, az általunk hagyott beállítás szerint fut a program. Minden állítható pont egy globális változóban van elmentve, annak érdekében hogy bármelyik függvény számára elérhető legyen.

```
system("cls");
if (lan)
{
    std::cout << "\n\t\tMain Menu\n\n";
    Sleep(40);
    std::cout << "\t1. Chess Problems\n";
    Sleep(40);
    std::cout << "\t2. Chess Challenges\n";
    Sleep(40);
    std::cout << "\t3. Settings\n";
    Sleep(40);
    std::cout << "\t4. Notes\n";
    Sleep(40);
    std::cout << "\t5. Recorded Times\n";
    Sleep(40);
    std::cout << "\t6. Exit\n";
}

...

btn = _getch();
switch (btn)
```

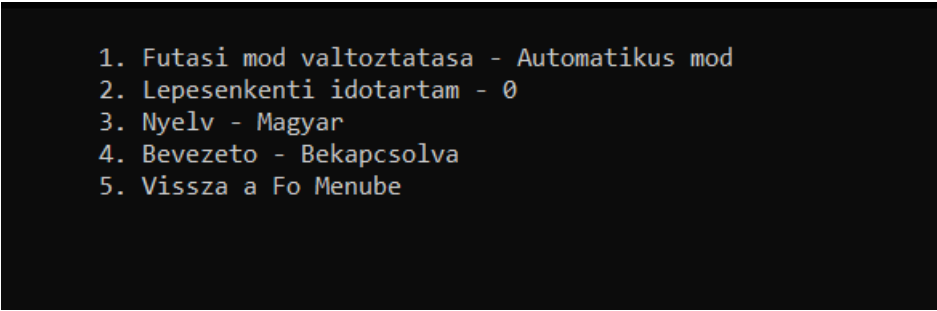
2.3. kódrészlet. Részlet a fő menü kódjából

A Leírás menüpontban egy rövid összefoglalója van a projektemnek, magyarázatokat adva a program használatára. Végül de nem utolsó sorban a Mentett időknél minden automatikus módban futtatott algoritmusok idejei találhatók, a tábla méretei szerint és a problémák típusai szerint felosztva, amiket a data.txt mentünk el minden futtatás után.

A Kilépést választva lezárhatjuk a programot.

Az előbb említett szöveges fájl soraiban az első szám a futási módot (1 = manuális, 0 = automatikus), ezt követően a probléma sorszámát, tábla méretét (N, ami NxN) illetve futási idejét találjuk ms-ben.

Hasonlóan a setting.txt-ben, az első 3 számjegy a futási módot, nyelvet illetve a bevezetőnek megfelelő bool-okat jelképezik, amit követ a lépések közötti intervalum értéke.



```
1. Futasi mod valtoztatasa - Automatikus mod
2. Lepesenkenti idotartam - 0
3. Nyelv - Magyar
4. Bevezeto - Bekapcsolva
5. Vissza a Fo Menube
```

2.7. ábra. A beállítások menü tartalma

A megfelelő opciók számjegyeit lenyomva az adott pontok értékét változtatni tudjuk, változtatva sorrendben automatikus és lépésről lépésre mód között, időtartamot növelve 100 miliszekundumokkal, váltva magyar és angol nyelvek között, illetve ki- és bekapcsolni a bevezető képernyőt.

2.4.1. Megoldott problémák illetve interaktív részek

Belépve a Sakk Problémák menüpontba, összesen 7 feladat közül választhatunk:

- N Királynős probléma
- Királynő különálló lefedés
- Királynő lefedés
- Huszár turné
- Bástya lefedés
- Futár lefedés
- Huszár lefedés

A menüpontban meghívódik a guider nevű függvény, amelybe a problémától függetlenül kiolvassa a jelenlegi beállításokban levő adatokat, vagyis hogy milyen futási módban, milyen értékű az intervalum változó, továbbá a tábla méretét kéri billentyűzetről. Miután ezek ki lettek választva, a feltételeknek megfelelően, meghívja a probléma függvényét.

Az előbb említett problémákon kívül, a **Sakk kihívás** menüpontban játékszerű feladványok jelennek meg, amelyekre a felhasználónak kell megoldásokat találnia. Összesen 8 különböző feladat van, mindegyikbe változó tábla méretek, különböző maximális megengedett darab számok, ahol célunk a tábla lefedése a megadott bábuval.

Első indításakor egy leíró szöveg jelenik meg, ami leírja a játék lényegét. Egy kurzor fog megjelenni a táblán, amelyet a WASD betűkkel irányíthatunk, a Space lenyomásával pedig a jelenlegi pozícióra helyezhetünk vagy levehetünk egy bábút. A feladvány mindig a tábla alatt van, a megengedett maximális számnál nem helyezhetünk többet, vagy ha a feladat különálló megoldást kér, akkor csak üres vagy nem támadott mezőkre helyezhetünk figurákat. Ha sikeresen megoldjuk, visszalépünk a fő menübe, ahonnan visszanavigálva ugyanide, egy újabb feladatot kapunk, egészen addig amíg egy gratuláló üzenet fogad minket. Bármikor visszaléphetünk a fő menübe a Q gomb lenyomásával.

2.5. Problémák algoritmusai

Mindegyik problémának 2 függvényt implementáltam, annak függvényében melyik futási mód szerint indítjuk el. Továbbá, mindegyik bábunak saját tábla kiírató függvénye, szabályainak megfelelő lefedést kigeneráló algoritmusokat alkalmaztam.

Először egy alap két dimenziós tömböt adok át mindegyiknek, 1-el jelölve ha bármelyik a bábuk közül el van helyezve egy adott pozícióban, 0-val ha üres. Ezt a "table" nevű globális változóba mentem el, amely minden probléma megoldásánál újra inicializálok. Következő lépésben, egy újabb 2D tömbbe, 1-est helyezek minden bábuk által lefedett pontra, majd 2-essel jelzem magát a bábút. Az "attacked-table" nevű változóba mentem le ezeket, amit átadok a megfelelő tábla kirajzoló függvényeknek.

```

void knight_printer(int table[50][50])
{
    char abc = 'A';
    make_knight_attacked_table(table);
    bool larger = false;
    int temp = 0;
    int number = 1;
    if (n < 39)
    {
        int rows = 0;
        while (rows < n)
        {
            if (abc > 'Z' && !larger)
            {
                abc = 'a';
                larger = true;
            }
            std::cout << " " << abc << " ";
            abc++;
            rows++;
        }
        std::cout << "\n";
        rows = 0;
        while (rows < n)
        {
            std::cout << "-----";
            rows++;
        }
        std::cout << "-\n";
        ...
    }
}

```

2.4. kódrészlet. Részlet a huszár kiírató függvényéből

A fenti kódrészletben létrehozom az alap lefedettségi táblát a huszárok lépési szabályai szerint. Ezt követően, annak függvényében milyen tábla méretet adtunk meg, a vízszintes sakkos koordináták jelölésére kerül sor, a sorok végső négyzeteinek megjelenítése után pedig a sorszámok.

Tábla mindegyik négyzetei 3x3 méretű mezőket foglalnak el, melyeknek kereteiket "-" és "|" karakterek segítségével rajzolom ki. Abban az esetben ha egy négyzet le van fedve bármilyen bábú által, a kereten belüli sorait hashtag-el töltöm ki.

```

void make_knight_attacked_table(int table[50][50])
{
    int k = 0;
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            if (table[i][j] == 1)
            {
                xcoord[k] = i;
                ycoord[k] = j;
                k++;
            }
        }
    }
    int x_add[9] = { -1,1,-2,-2, -1,1,2,2 };
    int y_add[9] = { -2,-2,-1,1, 2,2,-1,1 };
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            attacked_table[i][j] = 0;
            for (int l = 0; l < k; ++l)
            {
                for (int h = 0; h < 9; ++h)
                {
                    if (attacked_table[xcoord[l] + x_add[h]][ycoord[l] +
                        y_add[h]] != 2)
                    {
                        attacked_table[xcoord[l] + x_add[h]][ycoord[l] + y_add[h]]
                            = 1;
                    }
                }
                if (xcoord[l] == i && ycoord[l] == j)
                {
                    attacked_table[i][j] = 2;
                }
            }
        }
    }
}

```

2.5. kódrészlet. A huszár támadási mátrixát létrehozó függvény

Első lépésben elmentem mindegyik báb pozícióját, segítségével létrehozom a támadási mátrixot. A függvény végig próbálja mindegyik elmentett koordinátán az összes huszár által megtehető lépéseket, külön odafigyelve arra, hogy ne írja felül a bábuk pozícióit.

A bábuk jelölései :

- Q - Queen, vagyis Királynő
- T - Tower, vagyis Bástya
- B - Bishop, vagyis Futár
- H - Hussar, vagyis Huszár

Ezek folytonos újboli kiírása és a terminál törlése által, dinamikus vizualizálom a táblára helyezett bábukkal az algoritmusok lépéseit, egy valódi táblát szimulálva szöveges formában.

A kiírások a Sakk kihívásnál csupán annyiban változnak, hogy a mozgatható kurzor is megjelenik, amely a 3x3-as négyzetek keretein belül "+" karakterek jelzik.

A problémák algoritmusai lefutása után megjelenik a teljes futási idő, ha sikeresen lefutott vagy félbe lett szakítva, majd ezt követően, bármely gombot megnyomva, visszalépünk a fő menübe.

	A	B	C	D	E	F	G	H	
		#####	#####	#####	#####	#####	#####	#####	
	Q	#####	#####	#####	#####	#####	#####	#####	1
		#####	#####	#####	#####	#####	#####	#####	
		#####	#####	#####	#####	#####	#####	#####	
	#####	#####		#####	#####	#####	#####	#####	
	#####	#####	Q	#####	#####	#####	#####	#####	2
	#####	#####		#####	#####	#####	#####	#####	
	#####	#####	#####	#####		#####	#####	#####	
	#####	#####	#####	#####	Q	#####	#####	#####	3
	#####	#####	#####	#####		#####	#####	#####	
	#####		#####	#####	#####	#####			
	#####		#####	#####	#####	#####			4
	#####		#####	#####	#####	#####			
	#####		#####		#####	#####	#####		
	#####		#####		#####	#####	#####		5
	#####		#####		#####	#####	#####		
	#####	#####	#####		#####	#####	#####	#####	
	#####	#####	#####		#####	#####	#####	#####	6
	#####	#####	#####		#####	#####	#####	#####	
	#####		#####		#####		#####	#####	
	#####		#####		#####		#####	#####	7
	#####		#####		#####		#####	#####	
	#####		#####		#####			#####	
	#####		#####		#####			#####	8
	#####		#####		#####			#####	

Program megallitva.
Teljes futasi ido : 1604 ms

2.8. ábra. Példa egy félbe szakított algoritmusra

2.5.1. N Királynő probléma

A problémába egy N-szer N méretű táblára N darab királynőt szeretnénk elhelyezni, úgy hogy különálló legyen mindegyik, ne támadják egymást. Megoldást Backtracking segítségével keresek, a program addig fut, amíg minden oszlopba elhelyezett egy királynőt, azzal a feltétellel hogy ezeknek a bal felső és bal alsó átlói, illetve sorai üresek.

```
bool safe_queen(int table[50][50], int s, int o) // Backtracking N-Queen
    safety checker
{
    for (int i = 0; i < o; ++i) // megnézzük a sorat ha üres
    {
        if (table[s][i] == 1)
        {
            return false;
        }
    }
    int i = s;
    int j = o;
    while (i >= 0 && j >= 0) // ellenőrizzük a bal felső átlót
    {
        if (table[i][j] == 1)
        {
            return false;
        }
        --i;
        --j;
    }
    i = s;
    j = o;
    while (i < n && j >= 0) // ellenőrizzük a bal alsó átlót
    {
        if (table[i][j] == 1)
        {
            return false;
        }
        ++i;
        --j;
    }
    return true;
}
```

2.6. kódrészlet. Az ellenőrző függvény N királynős elhelyezésre

Abban az esetben ha nem érte el az utolsó oszlopot, vagyis nem tud helyezni minden oszlopba egy királynőt, Backtracking segítségével visszavesz királynőket, amíg végül sikerül egy teljes megoldást találnia.

A fő függvénybe második paraméterként az oszlop számát adom át amelyen el szeretném helyezni a következő királynőt. Mindez addig tart amíg ez egyenlő lesz N értékével.

Kiíratásánál a "queen-printer" függvényt hívom meg, lépésről lépésre módon pedig egy magyarázatokkal kiegészített változatát használom.

```
bool bt_queen_auto(int table[50][50], int j)
{
    if (j == n)
    {
        if (interval)
            system("cls");
        queen_printer(table);
        return true;
    }
    for (int i = 0; i < n; ++i)
    {
        if (safe_queen(table, i, j))
        {
            if (interval)
                system("cls");
            table[i][j] = 1;
            if (interval)
            {
                queen_printer(table);
                Sleep(interval);
            }
            if (bt_queen_auto(table, j + 1))
                return true;
            table[i][j] = 0;
            if (interval)
            {
                system("cls");
                queen_printer(table);
                Sleep(interval);
            }
        }
    }
    return false;
}
```

2.7. kódrészlet. N Királynő megoldására használt függvény

2.5.2. Különálló királynő lefedés

Célunk a tábla teljes lefedése királynőkkel, úgy hogy ezek ne támadják egymást, legoptimálisabb számú darab elhelyezésével. A minimális darab számokért lásd [Bur98] Hasonló ellenőrzést használunk mint az előző problémánál, viszont pár bővítéssel kiegészítve. Megegyező módon, itt nézem mind a négy átlót, sort és oszlopot is, ha egyiken se talál 1-et, vagyis egy előzőleg elhelyezett királynőt, akkor True-t térít vissza a "safe-queen" függvény.

Az algoritmus második paramétereként megadott változó amikor meghaladja az optimális minimum darabszámot, visszatérít egy False-t, backtrack-elést követően pedig visszavesz egy királynőt. A legelső optimális megoldásnál leáll az algoritmus.

```
...
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < n; j++)
    {
        if (dominating_safe_queen(table, i, j))
        {
            if (nr == limit)
            {
                return false;
            }
            table[i][j] = 1;
            if (bt_dominating_queen_auto(table, nr + 1))
                return true;
            table[i][j] = 0;
        }
    }
}
return false;
```

2.8. kódrészlet. Részlet a különálló királynő lefedés kigenerálására

Minden rekurziót követően, legelőször ellenőrizzük a lefedési mátrixban hogy nem maradt egy üres mező se, vagyis minden pontja támadva van. Ha az elhelyezett darabok száma megegyezik az optimális megoldásával, akkor véget ér az algoritmus és a "queen-printer"-el ezt kiíratom.

Ellenkező esetben, a kódrészletbe teszteli tovább a lehetséges pozíciókat, a megengedett darabszámmal.

Futási módtól függően, automatikus módban csak a táblát, míg a lépésről lépésre módnál magyarázó szöveget is megjeleníték.

A	B	C	D	E	F	G	H	
	#####			#####			#####	1
	#####			#####			#####	
	#####			#####			#####	
		#####		#####		#####		2
		#####		#####		#####		
		#####		#####		#####		
			#####	#####	#####			3
			#####	#####	#####			
			#####	#####	#####			
#####	#####	#####	#####	+++	#####	#####	#####	4
#####	#####	#####	#####	+ Q +	#####	#####	#####	
#####	#####	#####	#####	+++	#####	#####	#####	
			#####	#####	#####			5
			#####	#####	#####			
			#####	#####	#####			
		#####		#####		#####		6
		#####		#####		#####		
		#####		#####		#####		
	#####			#####			#####	7
	#####			#####			#####	
	#####			#####			#####	
#####				#####				8
#####				#####				
#####				#####				

2.9. ábra. Királynő lefedése ábrázolva 8x8 táblán

2.5.3. Királynő lefedés

Hasonlóan az előző problémához, itt is le szeretnénk fedni a táblát királynőkkel, viszont a különállóság megkötése nélkül, ami annyit jelent, hogy a királynők üthetik egymást. Az adott feladatnál Backtracking helyett Greedy módszerrel próbáltam megoldást találni a problémára.

```
int attacked_nr(int table[50][50], int s, int o)
{
    int count = 0;
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            if (i == s && table[i][j] == 0)
            {
                count++;
            }
            else if (j == o && table[i][j] == 0)
            {
                count++;
            }
        }
    }
}
```

```

        {
            count++;
        }
        else if (std::abs(s - i) == std::abs(o - j) && table[i][j] == 0)
        {
            count++;
        }
    }
}
return count;
}

```

2.9. kódrészlet. Greedy függvény a támadott pontok számára

A fenti függvény számára átadom a táblát és egy adott pont sor és oszlop koordinátáit, amelyen ellenőrzi, hogy összesen hány királynők által nem lefedett pontot támad. A függvény végig megy a mátrix összes pontján, számolva a soraiban és oszlopaiban az üres és nem támadott pontokat.

Felhasználva egy képletet, miszerint ha kivonjuk a megadott pont sor és oszlop értékéből a jelenleg ellenőrzött mező ugyanazon koordinátáit, ezek abszolút értékeknek megegyezésénél, garantáltan valamely elhelyezett királynő átlóján helyezkedünk el.

Ahogy végig haladt a mátrixon, visszatéríti hány üres pontot fedett le a megadott koordinátájú királynő és a legnagyobb értékűre elhelyezünk egyet.

```

void dominating_queens_auto(int table[50][50], int count)
{
    make_attacked_table(table);
    int nul = false;
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            if (attacked_table[i][j] == 0)
            {
                nul = true;
                break;
            }
        }
    }
    if (!nul)
    {
        if (interval)
            system("cls");
        return;
    }
    int xcoord = 0, ycoord = 0;
    int max = -1;
    for (int i = 0; i < n; ++i)
    {

```

```

for (int j = 0; j < n; ++j)
{
    if (max < attacked_nr(attacked_table, i, j))
    {
        max = attacked_nr(attacked_table, i, j);
        xcoord = i;
        ycoord = j;
    }
}
table[xcoord][ycoord] = 1;
if (interval)
{
    system("cls");
    queen_printer(table);
    Sleep(interval);
}
dominating_queens_auto(table, count + 1);
}

```

2.10. kódrészlet. Királynő lefedés Greedy módszerrel

Minden újrahívásnál, ellenőrzi a mátrix teljes lefedettségét, ugyanúgy mint az előző algoritmusnál. Amint ez teljesült, a program véget ér és kiiratás történik.

	A	B	C	D	E	F	G	H	
1	#####	Q	#####	#####	#####	#####	#####	#####	
2	#####	#####	#####	#####	#####	#####	Q	#####	
3	#####	#####	#####	#####	#####	#####	#####	#####	
4	#####	#####	#####	Q	#####	#####	#####	#####	
5	#####	#####	Q	#####	#####	#####	#####	#####	
6	#####	#####	#####	#####	Q	#####	#####	#####	
7	#####	#####	#####	#####	#####	#####	#####	#####	
8	#####	#####	#####	#####	#####	#####	#####	#####	
Teljes futási idő : 21 ms									

2.10. ábra. 8x8 táblán a Greedy királynős lefedés megoldása

2.5.4. Huszár turné

Feladatunk a problémánál a tábla teljes bejárása egy huszár segítségével, azzal a feltétellel hogy mindegyik mező csak egyszer látogatható.

Az előző algoritmusokhoz képest, ez bővül egy újabb bemeneti résszel, melyben a tábla méretének megadása után, a kiinduló pontot és kéri a feladat, sakkos koordinátákban megadva. Erre a pontra lesz először a huszár elhelyezve, ahonnan célja a tábla összes többi mezejének bejárása.

A megoldásra vezető utat Warnsdorff szabály alapját alkalmaztam [DS96], ami azt feltételezi hogy a huszárt mindig arra a mezőre helyezzük, ahonnan a lehető legkevesebb lépést teheti meg.

A	B	C	D	E	F	G	
							1
		####		####			2
		####		####			
		####		####			
	####				####		3
	####				####		
	####				####		
			+++ + H + +++				4
	####				####		5
	####				####		
	####				####		
		####		####			6
		####		####			
		####		####			
							7

2.11. ábra. Egy huszár lehetséges lépései

Legjobb esetben, egy üres táblán, nyolc választási lehetősége van a huszárnak, minimum helyzetben pedig kettő, a tábla sarkain levő pontoknál, ha $N \geq 3$.

A kiinduló ponttól kezdve, ellenőrizzük mindegyik szabályos pozíciót ahova elhelyezhető, és megszámoljuk ezeknek hány pontra léphetnek következőre. Greedy módszernek megfelelően, mindig a legkisebb pontot választjuk ezek közül, és léptetjük a huszárt.

```

int knight_attacked_spots(int table[50][50], int x, int y)
{
    int x_add[9] = { -1,1,2,2,-1,1,-2,-2 };
    int y_add[9] = { 2,2,1,-1,-2,-2,-1,1 };
    int count = 0;
    for (int k = 0; k < 8; ++k)
    {
        if (x + x_add[k] >= 0 && x + x_add[k] < n)
        {
            if (y + y_add[k] >= 0 && y + y_add[k] < n && table[x + x_add[k]][y + y_add[k]] == 0)
            {
                count++;
            }
        }
    }
    return count;
}

```

2.11. kódrészlet. Huszár támadható pontjait számoló függvény

Az x-add illetve y-add tömbökbe a huszár L betű alakú mozdításához szükséges számokat tároljuk, ezeken végig haladva, minden olyan mező amely a tábla keretein belül van és nem látogatott, növeli a támadott pontok számát.

```

bool improved_knights_tour_auto(int table[50][50], int x, int y, int count)
{
    bool found = false;
    if (count == n * n - 1)
    {
        system("cls");
        knight_tour_print(table);
        if (lan)
        {
            std::cout << "\nEvery point has been visited.\n";
        }
        else
        {
            std::cout << "\nMinden pont meglátogatva.\n";
        }
        return true;
    }
    while (!found)
    {
        for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
        if (table[i][j] == 0)
        {
            found = true;

```

```

    }
    if (!found)
    {
        return true;
    }
    else
        break;
}
int min = 99;
int x_add[9] = { -1,1,-2,-2, -1,1,2,2 };
int y_add[9] = { -2,-2,-1,1, 2,2,-1,1 };
for (int l = 0; l < 8; ++l)
{
    if (x + x_add[l] >= 0 && x + x_add[l] < n && table[x + x_add[l]][y +
        y_add[l]] == 0)
    {
        if (y + y_add[l] >= 0 && y + y_add[l] < n && table[x + x_add[l]][y
            + y_add[l]] == 0)
        {
            int temp = knight_attacked_spots(table, x + x_add[l], y +
                y_add[l]);
            if (temp < min)
            {
                min = temp;
            }
        }
    }
}
for (int k = 0; k < 8; ++k)
{
    if (x + x_add[k] >= 0 && x + x_add[k] < n && table[x + x_add[k]][y +
        y_add[k]] == 0)
    {
        if (y + y_add[k] >= 0 && y + y_add[k] < n && table[x + x_add[k]][y
            + y_add[k]] == 0)
        {
            if (knight_attacked_spots(table, x + x_add[k], y + y_add[k]) ==
                min)
            {
                count++;
                table[x + x_add[k]][y + y_add[k]] = count + 1;
                if (interval)
                {
                    system("cls");
                    knight_tour_print(table);
                    Sleep(interval);
                }
                if (improved_knights_tour_auto(table, x + x_add[k], y +
                    y_add[k], count))
            }
        }
    }
}

```



```

        return true;
        count--;
        table[x + x_add[k]][y + y_add[k]] = 0;
        if (interval)
        {
            system("cls");
            knight_tour_print(table);
            Sleep(interval);
        }
    }
}
}
}
return false;
}

```

2.12. kódrészlet. Huszár turné függvénye

Kiíratáshoz a saját "knight-tour-print" függvényt használom. Ebben a meglátogatott pontra a "count" változó értékét helyezem, az kiíratásoknál lényegében a lépések sorszámát mutatva.

Előfordulhat a bejárás során, hogy egy adott pozícióra érve, a megtehető lépések száma több következő lépésnél megegyezik a minimum értékkel. Ezekben az esetekben, a kettő vagy több egyenlő értékű minimum közül a helytelen utat választva rossz megoldáshoz, vagyis a tábla bejárásánál rossz útra vezethet.

Egyik módszere ennek megoldására a huszár megtehető lépéseinek sorrendjének a felcserélése, annak függvényében milyen pozíción vagyunk a táblán, lásd [DS96].

Ez azonban folytonos ellenőrzést igényelne, a koordináták képletekbe helyezését, és az x-add illetve y-add konstans változtatását.

A	B	C	D	E	F	
1	28	9	20	3	30	1
10	21	2	29	16	19	2
35	8	27	18	31	4	3
22	11	36	15	26	17	4
7	34	13	24	5	32	5
12	23	6	33	14	25	6
Minden pont meglátogatva.						

2.12. ábra. 6x6 táblán A1 pontról indított huszár turné eredménye

Algoritmusomba elmentem mindegyik olyan pontot amely egyelő a minimum támadási pontok számával. Ahogy elér egy zsákutcába, úgy hogy nem járta még be teljesen a táblát, Backtracking-et alkalmazva, visszafele lépked amíg eléri a tiebreak pontját, és a következő minimum ponton keresztül próbál megoldást találni.

Lépésről lépésre módban, minden pozíció váltást kiírat, illetve a következő pozícióból megtehető lépések számát is közli.

Ugyanazon kiinduló pont és tábla méret esetében ugyanazt a kizárólag nyílt, vagyis nem a kezdeti ponthoz visszajutó, megoldást kapjuk.

Abban az esetben ha nem sikerül a turné, nem létezik szabályos bejárása a táblának.

2.5.5. Bástyá lefedés

A tábla lefedése bástyák segítségével, úgy hogy ezek különállóak és minden esetben a leoptimálisabb darab szám legyen elhelyezve.

```
bool empty_row(int table[50][50], int row)
{
    for (int i = 0; i < n; ++i)
    {
        if (table[row][i] == 1)
        {
            return false;
        }
    }
    return true;
}
bool empty_column(int table[50][50], int col)
{
    for (int i = 0; i < n; ++i)
    {
        if (table[i][col] == 1)
        {
            return false;
        }
    }
    return true;
}
```

2.13. kódrészlet. Bástyák ellenőrző függvényei

Egyszerűbb lépési szabályai miatt, csupán egy adott sorát és oszlopát kell ellenőrizniük a táblának, hogy mindkettő üres legyen. Abban az esetben ha ez teljesül elhelyezünk egyet a legelső elérhető pozícióra.

```
bool rook_domination_auto(int table[50][50])
{
    srand(time(NULL));
    int count = 0;
    while (count != n)
    {
        for (int i = 0; i < n; ++i)
        {
            if (empty_row(table, i))
            {
                int temp = rand() % n;
                if (empty_column(table, temp))
                {
                    table[i][temp] = 1;
                    count++;
                    if (interval)
```

```

    {
        system("cls");
        rook_printer(table);
        std::cout << '\n';
        Sleep(interval);
    }
}
}
}
}
return true;
}

```

2.14. kódrészlet. Kaotikus bástya lefedés függvénye

Hasonlóan az N királynős problémához, amikor N darab bábút elhelyeztünk, csak akkor ér véget az algoritmus.

Felhasználva egy véletlenszerű szám generátort, elhelyezünk N darab bástyát a random szám értékének megfelelően, a tábla teljes lefedettségéig.

Sajátos kiírató függvényével, a "rook-printer"-el minden megfelelő, nem támadott pozícióra helyezett bástyát ábrázolom, amíg végső megoldást talál az algoritmus.

	A	B	C	D	E	F	G	H	
	####	####	####	####	####	####	####		
	####	####	####	####	####	####	####	T	1
	####	####	####	####	####	####	####		
	####	####		####	####	####	####	####	
	####	####	T	####	####	####	####	####	2
	####	####		####	####	####	####	####	
	####	####	####	####	####		####	####	
	####	####	####	####	####	T	####	####	3
	####	####	####	####	####	####	####	####	
	####	####	####	####	####	####		####	
	####	####	####	####	####	####	T	####	4
	####	####	####	####	####	####	####	####	
	####	####	####		####	####	####	####	
	####	####	####	T	####	####	####	####	5
	####	####	####	####	####	####	####	####	
	####		####	####	####	####	####	####	
	####	T	####	####	####	####	####	####	6
	####		####	####	####	####	####	####	
	T	####	####	####	####	####	####	####	
		####	####	####	####	####	####	####	7
		####	####	####	####	####	####	####	
	####	####	####	####		####	####	####	
	####	####	####	####	T	####	####	####	8
	####	####	####	####	####	####	####	####	

2.13. ábra. Egyike bástya lefedés 40320 megoldásainak 8x8 tábla esetén

2.5.6. Futár lefedés

Az előző problémákhoz hasonlóan, itt is a feladat optimális megoldás számmal, a tábla teljes lefedése különállóan futárokkal.

Bármilyen esetben, a legjobb megoldást mindig az $N/2$ oszlopba, vagyis a tábla közepére, ennek minden egyes sorában helyezett futárok eredményezik.

```
bool bishop_domination_auto(int table[50][50])
{
    for (int i = 0; i < n; ++i)
    {
        if (n % 2 == 0)
            table[i][n / 2 - 1] = 1;
        else
            table[i][n / 2] = 1;
        if (interval)
        {
            system("cls");
            bishop_printer(table);
            Sleep(interval);
        }
    }
    return true;
}
```

2.15. kódrészlet. Futár lefedés függvénye

Mivel csak az átlókat támadják, és ugyanazon oszlopon halad végig a ciklus, ezért nincs szükségünk más ellenőrző függvényekre, kivétel nélkül mindig sikeresen lefedik N darab bábbal a teljes táblát.

Kiíratásához a bishop-printer-t használom, amely az elhelyezett bábuknak csak az átlóit fedi le.

2.5.7. Huszár lefedés

A végső probléma amelyre megoldást próbáltam találni, legkevesebb szükséges darab elhelyezésével, a teljes tábla lefedése huszárok által.

Akárcsak magában a sakkban is, a huszár a legkiszámíthatatlanabb bábu, komplexitása révén nem tudtam erre konklúzív, minden esetben jó megoldást találó algoritmust írni. Emiatt is hagytam a WIP (work in progress) szöveget mellette.

Külön kezeltem kisebb tábla méretekre :

- Ha kisebb vagy egyenlő mint három

Ebben az esetben minden üres sorába helyezek egy huszárt, amíg a végére érek, vagy nem maradt egy lefedetlen mező se

- Ha egyenlő négyvel

A táblán megkeresem a max pontokat, ahonnan a legtöbb mezőket támadhatják, és mindegyikre helyezek egy huszárt

Ezt követően, egyfajta mintát követve, felosztom a táblát negyedekre, és ezeken belül végzek hasonló lépéseket.

Mindegyik negyedben keresem a legtöbb pontot támadó pozíciót, és ezekre helyezek huszárokat, amíg teljes lefedettség lesz a táblának.

A negyedekben külön-külön megkeresem a helyi maximum pontokat, majd ezekre dinamikusan elhelyezem huszárokat, esetleges kettős lefedések esetén ne kerüljenek olyan darabok a táblára amelyek már csak lefedett mezőket támadnak.

```
for (int i = 0; i < n / 2 + 1; ++i) // Bal felső negyed
{
    for (int j = 0; j < n / 2 + 1; ++j)
    {
        int temp = knight_attacked_spots(new_table, j, i);
        if (temp == max)
        {
            table[j][i] = 1;
            top_left_x = j;
            top_left_y = i;
            done = true;
            break;
        }
    }
    if (done)
        break;
}
done = false;
for (int i = n; i > n / 2 - 1; --i) // Bal alsó negyed
{
    for (int j = 0; j < n / 2 + 1; ++j)
    {
        int temp = knight_attacked_spots(new_table, i, j);
        if (temp == max)
```

```

        {
            table[i][j] = 1;
            done = true;
            break;
        }
    }
    if (done)
        break;
}
done = false;
for (int i = n; i > n / 2 - 1; --i) // Jobb also negyed
{
    for (int j = n; j > n / 2 - 1; --j)
    {
        int temp = knight_attacked_spots(new_table, j, i);
        if (temp == max)
        {
            table[j][i] = 1;
            bottom_right_x = j;
            bottom_right_y = i;
            done = true;
            break;
        }
    }
    if (done)
        break;
}
done = false;
for (int i = 0; i < n / 2 + 1; ++i) // Jobb felső negyed
{
    for (int j = n; j > n / 2 - 1; --j)
    {
        int temp = knight_attacked_spots(new_table, i, j);
        if (temp == max)
        {
            table[i][j] = 1;
            done = true;
            break;
        }
    }
    if (done)
        break;
}
}

```

2.16. kódrészlet. Huszár lefedésnél a negyedeken belül az első báb elhelyezése

```
bool knight_domination_checker(int table[50][50])
{
    make_knight_attacked_table(table);
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            if (attacked_table[i][j] == 0)
                return false;
        }
    }
    return true;
}
```

Annak függvényében hogy páros vagy páratlan méretű a tábla, a negyedek keretei változnak, átfedettség elkerülése érdekében.

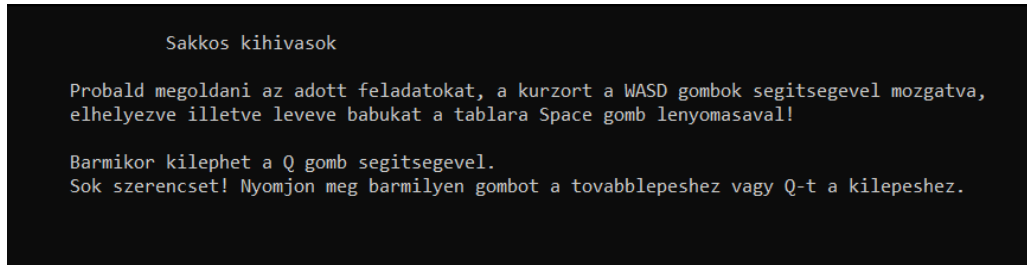
Amíg talál az algoritmus üres mezőket, egy goto segítségével újból elvégzi az előzőleg említett lépéseket, egészen a legelső megoldás kigenerálásáig.

[illegible]

40

2.6. Sakk kihívás

A fő menüből a második pontot választva beléphetünk az projektem interaktív, játékszerű részéhez. Itt a felhasználó feladata, adott kritériumok szerint, az előzőleg implementált problémák megoldása. Első indításkor a következő képernyő fogad bennünket :



2.15. ábra. Sakk kihívás fogadó szövege

Amint értesítve lett a felhasználó a játék alapjáról, továbblépve, véletlenszerűen kiválaszt az algoritmus egyet a nyolc előre beállított, megoldható probléma közül. Négy különböző változó segítségével létrehozza a feladványt :

- N, a tábla mérete
- Figure, a báb típusa
- Goal, kötelező darab szám
- Indep, ha különálló pozíciókat fogad csak el

Amint ezek az adatok ki lettek választva, ezeknek megfelelően a hozzáillő táblát ábrázolja az elvégzendő feladat szövegével.

Kezdetben a kurzort a bal felső sarokba helyezi, innen a W A S D gombok lenyomásával tudjuk fel, balra, le illetve jobbra irányítani. Ha a táblán kívüli irányba mozgatjuk, akkor az ellenkező felébe ugrik át a kurzor.

A Space lenyomásával a feladat bábuját helyezzük el, és kigeneráljuk ennek a lefedési mátrixát. Ha jelenleg egy olyan pozíción van a kurzor, ahova már helyeztünk egy bábút, újbóli Space nyomásával levesszük ezt.

Figyelmeztetést kapunk abban az esetben amikor több bábút akarunk elhelyezni a megengedett számnál, vagy abban az esetben amikor nem különálló mezőt választunk a táblán.

Ahogy teljesítjük a megadott kérést, értesítve leszünk, és visszalépünk a fő menübe. Újbol visszavigálva a Sakk Kihívás menübe, már átugorjuk a kezdeti bemutató szöveget és egy új feladatot kapunk.

Mindegyik feladat elvégzése után egy rövid üzenet fogja jelezni a kihívások teljesítését.

```

if (figure == 1)
{
    queen_printer(table);
    if (lan)
    {
        if (indep_int)
        {
            std::cout << "\nDominate the table with " << goal << " independent
                queens!\n";
        }
        else
            std::cout << "\nDominate the table with " << goal << " queens!\n";
    }
    else
    {
        if (indep_int)
        {
            std::cout << "\nFedd le a tablat " << goal << " darab kulonallo
                kiraly novel!\n";
        }
        else
            std::cout << "\nFedd le a tablat " << goal << " darab kiraly novel!\n";
    }
    if (indep_int)
    if (indep)
    {
        if (lan)
        {
            std::cout << "Not independent placement!\n";
        }
        else
        {
            std::cout << "Nem kulonallo pozicio!\n";
        }
    }
}

```

2.18. kódrészlet. Részlet a Sakk kihívás királynős feladatokat kezelő részéből

3. fejezet

Eredmények, következtetések

3.1. Problémák megoldási módszerei

A 2.5. fejezetben változtatva Backtracking illetve Greedy módszerek között sikerült megoldást találnom a kiszabott sakkos problémákra. Komplexebb esetekben, mint például a huszár turné és lefedésnél, ennek a kettőnek a keverékét alkalmaztam.

Bármely programozási paradigma segítségével megoldást találhatunk ezekre a sakkos problémákra, viszont legtöbb esetben ezek nem a legcélravezetőbbek. Projektem első prototípusában mindegyik problémát több paradigma segítségével szerettem volna megoldani és így ábrázolni az implementált tábla kirajzoló függvényekkel. A probléma kiválasztását követően egy újabb ablakban válaszhattunk volna Dinamikus, Greedy, illetve Backtracking módszerek közül.

Ezek viszont nagyrészt aluloptimalizáltak, lassú futási időkhöz vezettek, helytelen megoldásokkal, ami miatt csak a leghatékonyabb megoldásokat hagytam végül.

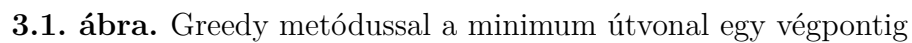
A két fő paradigma amelyekre végül nagyobb hangsúlyt fektettem, azok a Greedy és Backtracking módszerek voltak.

Greedy metódus előnyei:

- egyszerű és átlátható implementáció
- gyors futási idő, mivel minden pillanatban csak a számukra legjobb döntést választják
- egy követhető alap elv ismételt alkalmazásával talál megoldást
- alacsony idő és erőforrás igény

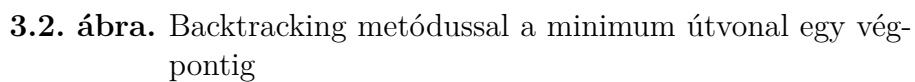
Greedy metódus hátrányai:

- Nem mindig optimális megoldást találnak
- Gyakran ragad lokális optimumba, csak rövid távon halad megfelelő úton



- mélységi bejárásával az összes lehetséges változaton végigiterál
- garantáltan talál megoldást vagy meghatározza hogy ez nem létezik

- magas idő és erőforrás igény
- nehezebben követhető, komplikáltabb implementáció, rekurzív hívások és a visszalépési logikája miatt



44

3.2. Futási idők és megfigyelések

Méret	Megtett idő (ms)
4×4	7
5×5	12
8×8	21
10×10	38
11×11	46

3.1. táblázat. N Királynő futási idejei

Az 2.5.1 algoritmus annak ellenére hogy kizárólag Backtracking módszerrel működik, gyorsan talál megoldást az adott táblaméretekre. Ezekhez az időkhöz hozzáadódik a tábla létrehozása és kiírása is, így lényegében elhanyagolható idő alatt elvégzik a feladatot.

Méret	Megtett idő (ms)
4×4	64
5×5	65
7×7	93
8×8	76
10×10	39519
11×11	1623945

3.2. táblázat. Különálló királynő lefedés futási idejei

	A	B	C	D	E	F	G	H	I	J	K	
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	1
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	Q	#####	#####	#####	#####	#####	#####	#####	2
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	3
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	#####	#####	#####	#####	Q	#####	4
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	5
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	Q	#####	#####	#####	#####	#####	6
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	7
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	Q	#####	#####	#####	#####	#####	#####	#####	#####	8
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	9
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	#####	#####	Q	#####	#####	#####	10
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	11
	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	#####	

Found an optimal solution!

Total running time : 1623945 ms

3.3. ábra. Különálló királynő lefedés eredménye 11x11 táblán

2.5.2 Ismerve az optimális megoldások számát [Bur98] [PG97] dominanciánál, észrevehetjük hogy gyorsabban sikerül 8x8 méretnél megoldást találnia, mint 7x7 esetében. Ez abból következik hogy 638 egyedi megoldás létezik 8x8 táblán, míg 7x7-nél csupán 13. Backtracking módszert ismerve, tudjuk hogy ez végigpróbál minden lehetséges variációt, és tábla mérettől függetlenül, hamarabb találja meg a nagyobb darabszámúra a megoldást.

Azokban az esetekben, amikor a tábla mérete 10 vagy 11, exponenciális megnő a futási idő, hiszen egyetlen egyedi megoldásuk van; 10 esetében 39 másodperc, 11-nél pedig kb. 27 percet véve igénybe ennek kigenerálására.

Méret	Megtett idő (ms)
4x4	6
6x6	15
8x8	29
10x10	45
12x12	69
14x14	88

3.3. táblázat. Királynő lefedés futási idejei

Mivel kizárólag egy Greedy metódust 2.5.3 alkalmaztam, futási idő szempontjából optimális, viszont nem mindig talál helyes megoldásokat. Legjobb esetekben N/2 lépésen belül sikerül végső kiíratáshoz jutnia. Viszont Backtracking metódust alkalmazva exponenciálisan több lépést tenne meg egy helyes megoldás kigenerálásához :

- 8x8 tábla esetén, tudva hogy 5 a legkisebb szükséges darabszám, a variánsok száma

$$\frac{(64)!}{5! * (54 - 5)!}$$

amely összesen 7624512 jelent.

Az előző problémában a különállóság megköötése nagyban csökkentette variánsok számát, minden elhelyezett királynő lezárta a tábla több négyzetét, emiatt is alkalmaztam a Backtracking metódust ott.

Méret	Megtett idő (ms)
4x4	0
6x6	74
8x8	100
10x10	120
12x12	150
14x14	187

3.4. táblázat. Huszár turné futási idejei

Keverve a Greedy és Backtracking metódusokat talál megoldásokat a turnés problémára 2.5.4, azokban az esetekben amikor nincs megoldás, pld. 4x4 táblánál, ezekre külön lekezelés nélkül, elhanyagolható idő alatt visszatéríti végső eredményét.

3.3. Összességében

Ismerve a metódusok előnyeit és hátrányait, és a problémák részletes tanulmányozását követően optimális megoldásokhoz vezethet a helyes módozat alkalmazása. [TED10]

Bár a Greedy módszer tipikusan nem az elsődleges megközelítése összetettebb sakkos problémák megoldására, néhány részlete kezelhető illetve kivitelezhető ezzel a technikával is. Adott esetekben csak egy lokális optimum adhatja a teljes megoldást is, rövid távon, illetve futási idő szempontjából kedvező megoldásokat ad.

Azonban a Backtracking módszer, a sakk komplexitása révén, a leggyakrabban használt technika, lépési szekvenciák végigpróbálására, minden pozíció leellenőrzésére amíg az adott feltétel teljesül. Hosszú távon, mélyebb keresési algoritmusok garantáltan optimális megoldásokhoz vezetnek, csupán nagyobb végrehajtási idő árán.

Ezt a két módszert keverve, a Backtracking visszalépési technikája, illetve adott helyzetekben lokális optimumok meghatározása Greedy stratégiával vezethet helyesség, optimum és idő szempontjából is a legjobb megoldásokhoz.

A bástya illetve futár lefedés problémák egyszerűségéből kifolyólag, nem tértem ki ezekre külön előző fejezetemben, $O(n)$ -t megközelítő komplexitásuk révén.

Összefoglaló

A sakk szabályait, alapjait vizsgálva, sikerült elmélyülnöm ennek évezredekbe visszanyúló múltjába a felmerült problémáknak köszönhetően. Mindazok ellenére hogy a kapcsolata az informatikával viszonylag nemrég jött létre, napjainkba szinte kulcsfontosságúvá vált ennek alaposabb megértéséhez, a játék és az ezt körülfogó tudományok fejlődéséhez.

Dolgozatomban sakkos problémák megoldásával foglalkoztam, amelyeket egy interaktív kereten belül vizualizáltam didaktikai, informáló céllal. Legelőször magáról a sakkról, ennek mélyre nyúló történelméről és komplexitásáról ejtettem pár szót, majd a programozási nyelvel és felhasznált software-kel, könyvtárakkal kapcsolatban tettem pár említést. Ezt követően a felhasználói felület részein, menüpontjain haladtam végig, majd a hátuk mögött rejlő kódot, annak működéséről, a terminálba megjelenő szöveg által szimulált menüt mutattam be részletesen. Az beállítások menüpontban levő pontok bemutatását követően felsoroltam az implementált problémákat, illetve felületesen említést tettem ezek működéséről és az interaktív Sakkos kihívás menüpont lényegéről is. Az sakktáblák ábrázoló függvényeit összefoglalva, mindegyik problémáról, azok eredményeiről és a felhasználói felület adott részeiről is több képet illesztettem jobb átláthatóság érdekében. Ezek fontosabb kódrészleteit magyaráztam, kiemelve a választható futási módot ezeknél, az algoritmusok működésének könnyebb megértéséhez. A problémák háttérben zajló működéseiken, megoldási módszereiken végighaladva a Sakkos kihívás alaposabb bemutatásához jutunk el. Végül, de nem legutolsó sorban az eredményeket, futási időket sorolom fel, amelyek a problémák megoldását követően jelenítek meg. A felhasznált módszerek előnyeit és hátrányait helyezem egymással szemben, következtetéseket levonva ezekből.

Számomra a dolgozatom mindenképp felkeltette érdeklődésem sakkos problémákban, ezek informatikai megoldásaival, a közeljövőben magának a sakknak algoritmizálását szeretném megvalósítani, ismerve megközelítéseit, előfeltételeit ehhez hasonló feladatoknak.

A projektet illetően, továbbfejlesztési pontok első helyén egy újabb felhasználói felület kidolgozása áll, illetve a tábla háromdimenziós ábrázolása, könnyebb elérhetőség érdekében. Az algoritmusok bármelyike tovább fejleszthető, javítható mélytanulási módszerekkel minták felismerésére, már bejárt állapotok dinamikus elmentésével, más programozási nyelvekben való implementálásával. Továbbá, az informatív oldala is bővíthető, több helyzet magyarázása, a problémák más módszerekkel való kidolgozása és ezek összehasonlítása.

[Projekt GitHub repository linkje.](#)

Ábrák jegyzéke

1.1. Egy 8x8 táblára három darab királynő (Q) és az általuk lefedett mezők	11
2.1. A kódrészlet eredménye az 'a' karakter lenyomása után	14
2.2. A projekt Use Case diagramja	15
2.3. A projekt Sequence diagramja	16
2.4. A projekt függvényeinek architektúrája	16
2.5. A fő menü, magyar nyelven	17
2.6. A bevezető grafika a program elindításakor	17
2.7. A beállítások menü tartalma	19
2.8. Példa egy félbe szakított algoritmusra	23
2.9. Királynő lefedése ábrázolva 8x8 táblán	27
2.10. 8x8 táblán a Greedy királynős lefedés megoldása	29
2.11. Egy huszár lehetséges lépései	30
2.12. 6x6 táblán A1 pontról indított huszár turné eredménye	34
2.13. Egyike bástya lefedés 40320 megoldásainak 8x8 tábla esetén	36
2.14. Huszár lefedés 8x8 tábla esetén	40
2.15. Sakk kihívás fogadó szövege	41
3.1. Greedy módszerrel a minimum útvonal egy végpontig	44
3.2. Backtracking módszerrel a minimum útvonal egy végpontig	44
3.3. Különálló királynő lefedés eredménye 11x11 táblán	45

Táblázatok jegyzéke

1.1. Egyedi megoldások az N Királynős problémára	10
3.1. N Királynő futási idejei	45
3.2. Különálló királynő lefedés futási idejei	45
3.3. Királynő lefedés futási idejei	46
3.4. Huszár turné futási idejei	46

Irodalomjegyzék

- [Bur98] Alewyn Petrus Burger. The queens domination problem. *University of South Africa, Pretoria*, pages 81–87, 1998.
- [DS96] Paul Cull Douglas Squirrel. A warnsdorff-rule algorithm for knight’s tours on square boards. pages 6–8, 1996.
- [PG97] J.A. Webb P.B. Gibbons. Some new results for the queens domination problem. *Department of Computer Science, University of Auckland*, 1997.
- [TED10] Manuela PĂNOIU Tatiana-Elena DUȚĂ, Alexandru OPREAN. Educational software methods and strategies for designing algorithms backtracking, greedy method and dynamic programming. *Department of Electrical Engineering and Industrial Informatics, “Politechnica” University of Timisoara*, 2010.