

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM  
MAROSVÁSÁRHELYI KAR,  
SZOFTVERFEJLESZTÉS SZAK



SAPIENTIA  
ERDÉLYI MAGYAR  
TUDOMÁNYEGYETEM

Közelítő algoritmusok NP-teljes feladatok megoldására

**MESTERI DISSZERTÁCIÓ**

Témavezetők:  
dr. Kása Zoltán,  
egyetemi tanár  
dr. Kupán A. Pál,  
egyetemi docens

Végzős hallgató:  
Berecki Zoltán

**2023**

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA  
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,  
SPECIALIZAREA DEZVOLTAREA APLICAȚIILOR  
SOFTWARE



UNIVERSITATEA  
SAPIENTIA

Algoritmi de aproximare pentru rezolvarea problemelor  
NP-complete

**TEZĂ DE MASTERAT**

Coordonatorii științific:

dr. Kása Zoltán,  
profesor universitar  
dr. Kupán A. Pál,  
conferențiar universitar

Absolvent:

Berecki Zoltán

**2023**

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



**SAPIENTIA**  
HUNGARIAN UNIVERSITY  
OF TRANSYLVANIA

Approximation algorithms to solve NP-complete problems

**MASTER THESIS**

Scientific advisors:

dr. Kása Zoltán,  
full professor  
dr. Kupán A. Pál,  
associate professor

Student:

Berecki Zoltán

**2023**

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

Viza facultății:

### LUCRARE DE DISERTAȚIE

Coordonator științific:  
**conf. Dr. Kupán A. Pál**  
**prof. Dr. Kása Zoltán**

Candidat: **Berecki Zoltán**  
Anul absolvirii: 2023

a) Tema lucrării de licență:

Algoritmi de aproximare pentru rezolvarea problemelor NP-complete

b) Problemele principale tratate: Bin Packing, Knapsack, Colorarea grafurilor, CNF Satisfiabilitate, Clasa P, Clasa NP, algoritmi de aproximare

c) Desene obligatorii: imagini jpg ca ilustrații

d) Softuri obligatorii: Aplicația realizată

e) Bibliografie recomandată:

Sara Baase (1978) Computer Algorithms, Introduction to Design and Analysis, San Diego State University, Addison-Wesley Publishing Company

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (fordítók: Iványi Antal., Benczúr András) (2003). Új algoritmusok, Sclolar Informatika, Budapest

f) Termene obligatorii de consultanță: săptămânal

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

Primit tema la data: 10/10/2022

Termen de predare: 06/07/2023

Semnătura Director Departament  
**conf. Dr. Kátai Zoltán**

Semnătura coordonatorului  
**conf. Dr. Kupán A. Pál**  
**prof. Dr. Kása Zoltán**

Semnătura responsabilului  
programului de studiu  
**conf. Dr. Kupán A. Pál**


Semnătura candidatului  
**Berecki Zoltán**

## Declarație

Subsemnatul/a Berecki Zoltan, absolvent(ă) al/a specializării Dezvoltarea aplicațiilor software, promoția 2022, 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, Îzpe Mureș  
Data: 7. VI. 2023

Absolvent

Semnătura Berecki Zoltan 

# Kivonat

Ezen dolgozat tárgya Közelítő algoritmusok vizsgálata NP-teljes feladatok megoldására különböző algoritmusok segítségével. A mindennapi feladataink nagy részét le tudjuk írni határidőnapló szerint, ami jelentheti a napi teendőink algoritmusait. Ezeket az algoritmusokat át tudjuk írni számítógépes programokra. Ezek között viszont vannak olyan teendők is, amelyekre nincs pontos képlet, amivel gyorsan és pontosan meg tudjunk oldani egy feladatot. Ilyen esetekben keresünk, egy, az elvárthoz minél közelebbi megoldást, ami majdnem ugyanolyan jól megoldja az adott helyzetet.

Viszont fontos, hogy ne csak pontos megoldást kapjunk, hanem elég gyorsan tudjuk megoldani ezeket a feladatokat. Ilyen feladat lehet, például kirándulásra készüléskor, hogy minél gyorsabban tudjuk bepakolni a legfontosabb dolgokat, vagy a legrövidebb útvonal megtervezése. Ilyen esetben fontos lehet a szempontok alapos mérlegelése és jó, ha tudjuk milyen módszerekkel lehet ezeket a feladatokat gyorsan és hatékonyan megoldani.

Ebben a dolgozatban arra keressük a választ, hogy milyen módszert érdemes választani ahhoz, hogy egy NP-teljes feladatot elég jól meg tudjunk oldani megadott határidőn belül. Az egyes feladatokra két különböző algoritmust fogunk mutatni egyet, ami nem hatékony és egy másikat, ami elég jó megoldást ad.

Mindkét algoritmussal ugyanazokat a teszteseteket oldattam meg és mértem az algoritmusok futási idejét. Ezekre a tesztesetekre átlagokat számoltunk. A kapott eredményeket értelmezni fogjuk.

A munkánkat elemezni fogjuk és levonjuk a tanulságot belőle, hogy milyen módszerekkel érdemes az egyes feladattípusokat megoldani, valamint azt is, hogy milyen programozási nyelvekkel érdemes tovább fejleszteni ezeket az algoritmusokat.

**Kulcsszavak:** Np-teljes problémák, Közelítő algoritmusok, binpacking, hátizsák probléma, gráf színezés.

# Rezumat

Subiectul acestei teze este Algoritmi de aproximare pentru rezolvarea problemelor NP-complete folosind diferiți algoritmi. Cele mai multe dintre sarcinile noastre zilnice pot fi scrise într-un jurnal, care pot fi algoritmi pentru sarcinile noastre zilnice. Putem transcrie acești algoritmi în programe de calculator. Cu toate acestea, unele dintre aceste sarcini nu au o formulă precisă pentru a le rezolva rapid și precis. În astfel de cazuri, căutăm o soluție cât mai apropiată de cea așteptată, care rezolvă situația aproape la fel de bine.

Cu toate acestea, este important nu numai să obținem o soluție exactă, ci și să putem rezolva aceste probleme suficient de repede. O astfel de sarcină ar putea fi, de exemplu, atunci când ne pregătim pentru o călătorie, să împachetăm cât mai repede lucrurile cele mai importante sau să planificăm cel mai scurt traseu. În astfel de cazuri, poate fi important să se analizeze cu atenție problemele și este bine să se știe ce metode pot fi folosite pentru a rezolva aceste sarcini rapid și eficient.

În această lucrare, vom analiza ce metodă ar trebui aleasă pentru a rezolva o sarcină NP-completă suficient de bine într-un interval de timp dat. Pentru fiecare sarcină, vom prezenta doi algoritmi diferiți, unul care este inefficient și altul care oferă o soluție suficient de bună.

Am rezolvat aceleași cazuri de test cu ambii algoritmi și am măsurat timpul de execuție al algoritmilor. S-au calculat mediile pentru aceste cazuri de testare. Rezultatele vor fi interpretate.

Ne vom analiza activitatea și vom afla din ea ce metode ar trebui folosite pentru a rezolva fiecare tip de problemă, precum și ce limbaje de programare ar trebui folosite pentru a dezvolta în continuare acești algoritmi.

**Cuvinte de cheie:** probleme Np-complete, algoritmi de aproximare, binpacking, problema rucsacului, colorarea grafurilor.

# Abstract

The subject of this paper is Approximation algorithms to solve NP-complete problems using different algorithms. Most of our everyday tasks can be written down in a time diary, which can be the algorithms for our daily tasks. We can rewrite these algorithms into computer programs. Among these, however, there are tasks for which there is no exact formula that can be used to solve a task quickly and accurately. In such cases, we normally look for a solution as close as possible to what is expected, which solves the given situation almost as well.

However, it is important not only to get an exact solution, but also to be able to solve these problems quickly enough. Such a task might be, for example, when preparing for a trip, to pack the most important things as quickly as possible, or to plan the shortest route. In such cases, it can be important to consider the issues carefully and it is good to know what methods can be used to solve these tasks quickly and efficiently.

In this paper, we will analyse which method should be chosen in order to be able to solve a complete NP task well enough within a given timeframe. For each task, we will present two different algorithms, one that is inefficient and another one that gives an acceptable solution.

I solved the same test cases with both algorithms and I have also measured the running time of the algorithms. The averages for these test cases were also calculated. The results are to be interpreted.

We will analyze our work and learn from it what methods should be used to solve each type of problem, as well as what programming languages should be used for a further development of these algorithms.

**Keywords:** Np-complete problems, approximate algorithms, binpacking, knapsack problem, graph coloring.



# Tartalomjegyzék

<b>1. Bevezető</b>	<b>11</b>
<b>2. Gráfszínezésekkel kapcsolatos feladatok bemutatása</b>	<b>12</b>
2.1. Gráf színezés algoritmus backtracking módszerrel megoldva . . . . .	12
2.2. Az elkészített program bemutatása: . . . . .	14
2.3. A kód összetettsége a következőképpen elemezhető: . . . . .	15
2.4. Gráf színezés algoritmus branch and bound módszerrel megoldva . . . . .	15
2.5. Íme az elkészített c++ program működésének vázlatos leírása: . . . . .	16
2.6. A kód összetettsége a következőképpen elemezhető: . . . . .	17
2.7. A c++ program futtatása során a következő eredményeket kaptam . . . . .	17
<b>3. Bin packing feladat bemutatása</b>	<b>20</b>
<b>4. Hátizsák feladat bemutatása</b>	<b>28</b>
<b>5. A P osztály</b>	<b>38</b>
<b>6. Az NP osztály</b>	<b>39</b>
6.1. NP-Teljes problémák . . . . .	39
6.2. CNF kielégíthetőség . . . . .	40
<b>7. Közelítő algoritmusok</b>	<b>41</b>
7.1. Minimális lefedő csúcshalmaz . . . . .	41
7.2. Az utazóügynök feladat . . . . .	42
7.3. Az utazóügynök feladat háromszög - egyenlőtlenséggel . . . . .	43
<b>8. A rendszer specifikációja</b>	<b>49</b>
8.1. Felhasználói követelmények . . . . .	49
8.2. Rendszerkövetelmények . . . . .	49
8.3. Nem funkcionális elvárások . . . . .	49
<b>9. Tervezés</b>	<b>50</b>
<b>10. Kivitelezés</b>	<b>51</b>
10.1. Bin packing vizuális ábrázolása . . . . .	52
10.2. Christofides algoritmus vizuális ábrázolása . . . . .	54
10.3. Gráf színezés vizuális ábrázolása . . . . .	54
10.4. Hamiltoni körnek a vizuális ábrázolása . . . . .	55

10.5. Hátizsák feladat vizuális ábrázolása . . . . .	56
10.6. TSP körút vizuális ábrázolása . . . . .	56
10.7. Dodekaéder vizuális ábrázolása . . . . .	57
<b>11.Mérések</b>	<b>59</b>
11.1. Bin Packing, láda pakolással kapcsolatosan, n=100 elem, közelítő megoldás esetén . . . . .	59
11.2. Bin Packing, láda pakolással kapcsolatosan, n=1000 elem, közelítő megol- dás esetén . . . . .	59
11.3. Bin Packing, láda pakolással kapcsolatosan, n=10000 elem, közelítő meg- oldás esetén . . . . .	60
11.4. Gráfszínezés backtracking és branch and bound módszerekkel, n=10 elem esetén . . . . .	60
11.5. Gráfszínezés backtracking és branch and bound módszerekkel, n=100 elem esetén . . . . .	60
11.6. Gráfszínezés backtracking és branch and bound módszerekkel, n=1000 elem esetén . . . . .	60
11.7. Bruteforce és dinamikus módszerrel hátizsák feladat megoldása, n=100 elem esetén . . . . .	61
11.8. Bruteforce és dinamikus módszerrel hátizsák feladat megoldása, n=1000 elem esetén . . . . .	61
11.9. Bruteforce és dinamikus módszerrel hátizsák feladat megoldása, n=10000 elem esetén . . . . .	61
11.10 TSP Bruteforce és Christofides megoldásának összehasonlítása, n=10 elem esetén . . . . .	61
11.11 Szoftverek összehasonlítása . . . . .	61
<b>Összefoglaló</b>	<b>62</b>
11.12 Következtetések . . . . .	62
<b>Köszönetnyilvánítás</b>	<b>63</b>
<b>Ábrák jegyzéke</b>	<b>64</b>
<b>Táblázatok jegyzéke</b>	<b>65</b>
<b>Irodalomjegyzék</b>	<b>67</b>

# 1. fejezet

## Bevezető

Sarah Baase, 1978-ban a Számítógépes algoritmusok, bevezetés a tervezésbe és elemzésbe című könyvében a következő módon határozza meg az algoritmus fogalmát: létezik egy olyan program, amely a megfelelő típusú, szerkezetű bemeneti adatra, helyes eredményt ad, ha megfelelő mennyiségű időt adunk neki és biztosítjuk a szükséges tárhelyet.[Sar78]

A gyakorlati alkalmazásokkal számos probléma van, ami megoldható (lehet rá írni programot), viszont az idő- és tárhelyigény túlságosan nagy, így ezek a programok használhatatlanok. A komplexitás mértékére vonatkozó axiómákat kidolgozták, ez vonatkozhat a végrehajtott utasítások számára, vagy a tárolóbitek számára. Ezen axiómák segítségével be lehet bizonyítani olyan feladatok létezését, amelyekre nincs legjobb program.

Sara Baase könyvében az első hat fejezet során elemezte a számára legfontosabbnak tartott algoritmusokat, az algoritmusok által elvégzett munkát, átlagos eset és legrosszabb eset elemzéseket készített.[Sar78]

Mindezt a felhasznált tárhely, egyszerűség, optimális megoldás, valamint alkalmazhatóság és programozás szempontjából próbálta megközelíteni.[Sar78]

Ezek az algoritmusok keresés, rendezés, irányítatlan és irányított gráfokkal, karakterlánc illesztési mintákról, polinomokkal és mátrixokkal kapcsolatos feladatokról szóltak.[Sar78]

Ezen algoritmusok komplexitása  $O(n^3)$  pedig a megfelelően meghatározott bemeneti méret, ami alacsony időigényre utal.[Sar78]

Jelen dolgozat célja olyan feladatok vizsgálata, amelyre még nem alkottak meg optimális megoldást generáló algoritmusokat. Ezek közül sok olyan optimalizálási feladat, amelyeket gyakran használnak mindennapi tevékenységeink során.

Sarah Baase könyvében olyan meghatározásokat ad meg, amelyek segítségével a feladatokat csoportosítja a megoldásukhoz létfontosságú idő alapján, így különbséget tud tenni a „nehéz” (jelentősen sokáig tartó) és „nem túl nehéz” feladatok között. Ezek a feladatok megfogalmazhatóak kérdések formájában úgy, hogy csak igen vagy nem választ kelljen rá adjunk. Néhányat optimalizálási feladatként is szavakba foglalhatunk.[Sar78]

Sarah Baase könyvének a hetedik fejezetében a következő feladatok megoldásait mutatja be: gráfszínezés; munkák ütemezése büntetésekkel; Bin packing; hatizsak (táska); CNF kielégíthetőség; Hamilton utak és Hamilton-áramkörök.[Sar78]

## 2. fejezet

# Gráfszínezésekkel kapcsolatos feladatok bemutatása

Egy gráfban minimum hány szín szükséges az egyes csúcsok kiszínezéséhez, azzal a feltétellel, hogy két szomszédos nem lehet ugyanolyan színű. Ez a feladat például egy térkép vagy sakktábla kiszínezésénél. A csúcsok az egyes mezők, az élek a két mező között futnak.[CLR<sup>+</sup>03]

Általában NP teljes probléma eldönteni, hogy egy gráf kevesebb színnel kiszínezhető-e? Feltéve, hogy igaz a  $P \neq NP$  sejtés, nem is lehetséges hatékony algoritmus ennek eldöntésére.[CLR<sup>+</sup>03]

A feladat egy olyan optimális színezés előállítása, olyan színezés, amely csak megadott színeket használ. Alternatív megoldásként megadhatunk egy  $A$  gráfot és egy adott  $n$  számú színt és megkérdezhetjük, létezik-e olyan  $A$  gráfnak olyan színezése, amely  $n$  színt használ. (Ha igen, akkor  $A$  gráfot  $n$  színezhetőnek mondjuk).[CLR<sup>+</sup>03]

Például a térképek színezése egy gráfszínezési probléma.

Munkák ütemezése büntetésekkel. Tegyük fel, hogy van  $n$  darab  $J_1, \dots, J_n$  feladat, ezeket külön-külön kell elvégezni. Adottak a  $t_1, \dots, t_n$  végrehajtási idők, a  $d_1, \dots, d_n$  határidők (a teljes eljárás indulási idejétől mérve) és a  $p_1, \dots, p_n$  holtidők elmulasztásáért járó büntetések, mind pozitív egész számok. A feladatok konkrét ütemezése a  $\pi$  permutációja  $\{1, 2, \dots, n\}$ , ahol  $J_\pi(1)$  az elsőként elvégzett feladat,  $J_\pi(2)$  a következő feladat, és így tovább.[Sar78]

$$P_\pi = \sum_{j=1}^n [if\ t_\pi(1) + \dots + t_\pi(j) > d_\pi(j)\ then\ p_\pi(j)\ \text{egyébként}\ 0.]$$

A feladat egy olyan ütemtervet keresni, amely minimálisra csökkenti a teljes büntetést, vagy alternatívaként, egy nemnegatív  $k$  egész számot adva, meghatározzuk, hogy létezik-e olyan ütemterv, ahol teljesül a  $P_\pi \leq k$ . [Sar78]

### 2.1. Gráf színezés algoritmus backtracking módszerrel megoldva

A következő ábra bemutatja a gráf színezési algoritmust backtracking módszerrel.

```
grafSzinezes():  
    MAXN = 10100  
    n = 0  
    adj = [[] for _ in range(MAXN)]  
    szin = [0] * MAXN
```

```

fuggveny megFelelo(v, c):
    for (int i = 0; i < adj[v].size(); ++i):
        int u = adj[v][i];
        ha szin[u] == c:
            return False
    return True

grafSzinezesiEszkoz(v) függvény:
    ha v == n + 1:
        return True

    c tartományban (1, n + 1):
        ha biztonsagos(v, c):
            szin[v] = c
            if grafSzinezesiEszkoz(v + 1):
                return True
            szin[v] = 0 // visszalepes

    return False

szamoldMegSzint() függvény:
    for (int i = 1; i <= n; i++)
        szineket.insert(color[i])
    return szineket.size()

Fo függvény():
    Indítsa el az időzítót
    Bemenet olvasása fajlból
    m = 0
    n, m = egész számok olvasása fajlból
    A szomszedsagi lista adj és a szintomb inicializálása
    minden elre 1-től m-ig:
        u, v = egész számok olvasása a fajlból
        Hozza-fuza v-t az adj[u]-hoz és u-t az adj[v]-hez
    A szinekSzama inicializálása 1-re
    míg a grafSzinezesiEszkoz(1) hamis:
        Novelje a szinek szamat
    Nyomtassa ki a szineket
    SzinSzamlalo kiszámítása
    Nyomtassa ki a szükséges számú szint
    Nyomtassa ki az eltelt időt

```

[Knu11][K03]

## 2.2. Az elkészített program bemutatása:

1. A c++ kód tartalmazza a szükséges könyvtárakat a bemenethez/kimenethez, a vektorkezeléshez, a rendezéshez, a fájlműveletekhez, a rendezetlen halmazhoz és az idő méréséhez.
2. Meghatároz egy konstans MAXN értéket, amely a csúcsok maximális számát jelenti.
3. Deklarál egy n változót a csúcsok számának tárolására.
4. Adj szomszédsági listát deklarál a gráf reprezentálására, ahol minden csúcsnak van egy vektora a szomszédos csúcsokból.
5. Egy tömbszínt deklarál a csúcsokhoz rendelt színek tárolására.
6. Meghatároz egy megFelelo függvényt, amely a szomszédos csúcsok vizsgálatával ellenőrzi, hogy biztonságos-e c színt rendelni a v csúcshoz.
7. Meghatározza a GrafSzinezeseSegedprogram rekurzív függvényt, amely visszalépést hajt végre, hogy megtalálja a gráf érvényes színét.
8. Az alapeset: Ha minden csúcsot kiszíneztünk, adjuk vissza a true értéket az érvényes színezés jelzésére.
9. Ismételje meg a színeket 1-től n-ig, és próbáljon minden színt hozzárendelni a v csúcsához.
10. Ha biztonságos a c szín hozzárendelése a v csúcsához, rendelje hozzá a színt, és hívja meg rekurzívan a GrafSzinezeseSegedprogram-t a következő csúcsához.
11. Ha érvényes színezést találunk a fennmaradó csúcsokhoz, adjuk vissza a true értéket.
12. Ha nem található érvényes színezés, lépjen vissza úgy, hogy visszaállítja a v csúcs színét 0-ra, és próbálkozzon a következő színnel.
13. Ha az összes szín kipróbálása után nem található érvényes színezés, adja vissza hamis értékét.
14. Meghatároz egy szamoldMegSzint függvényt, amely visszaadja a csúcsokhoz rendelt egyedi színek számát.
15. A fő funkció elkezdődik.
16. Megnyit egy "grafSzinezes\_rand\_10000.in" nevű bemeneti fájlt olvasásra.
17. Megnyit egy "grafSzinezes\_rand\_10000.out" nevű kimeneti fájlt írásra.
18. Az eltelTime változót deklarálja a végrehajtási idő tárolására.
19. Beolvassa a bemeneti fájlból az n csúcsok számát és az m élek számát.
20. Kiírja n és m értékét a konzolra.
21. Beolvassa az éleket a bemeneti fájlból, és hozzáadja a szomszédsági listához.
22. Elkezd mérni a végrehajtási időt.
23. A színek számát 1-re inicializálja.
24. Amíg nem található érvényes színezés, növelje a színek számát, és próbálja újra a GrafSzinezeseSegedprogram függvény használatával.
25. Leállítja a végrehajtási idő mérését.
26. Kinyomtatja az egyes csúcsok színeit mind a konzolba, mind a kimeneti fájlba.
27. Kiszámítja a felhasznált színek számát.
28. Kinyomtatja a színek számát a konzolra és a kimeneti fájlra.
29. Kinyomtatja az eltel időt a konzolra és a kimeneti fájlra.
30. A program véget ér. [Knu11]

## 2.3. A kód összetettsége a következőképpen elemezhető:

- Az megFelelo függvény időbonyolultsága  $O(\text{degree}(v))$ , mivel ellenőrzi a szomszédos csúcsok színeit, ahol a fok( $v$ ) egy  $v$  csúcs fokát jelöli.
- A GrafSzinezeseSegedprogram függvény időbonyolultsága  $O(n * \text{fok}(v))$ , mivel minden csúcsra és minden szomszédos csúcsra meghívja az megFelelo-et.
- A szamoldMegSzint függvény időbonyolítása  $O(n)$ , mivel a szintömbön keresztül iterál az egyedi színek megszámlálásához.
- A teljes komplexitás az  $n$  csúcsok számától és az  $m$  élek számától függ. A domináns tényező a visszalépési folyamat a GrafSzinezeseSegedprogram függvényben, amelynek összetettsége  $O(n * \text{fok}(v))$ .

## 2.4. Gráf színezés algoritmus branch and bound módszerrel megoldva

GrafSzinezes() algoritmus:

```
adj <- ures 2D vektor
szinek <- ures vektor
minszinek <- ures vektor

fuggveny megFelelo(v, c):
  for each u in adj[v]:
    if u < v and szinek[u] == c:
      return false
  return true

fuggveny grafSzinezesiEszkoz(v):
  if v == size of adj:
    maxColor <- maximalis elem szinekből
    minszinek[0] <- minimum minszinek[0] es maxColor + 1
    return

  for c from 0 to size of adj:
    if megFelelo(v, c):
      szinek[v] <- c
      grafSzinezesiEszkoz(v + 1)
      return

  szinek[v] <- 0

fuggveny grafSzinezes():
  minszinek[0] <- infinity
  grafSzinezesiEszkoz(0)
  return minszinek[0]

Main fuggveny():
  fin <- nyissa meg a "grafSzinezes1000.in" bemeneti fajlt
```

```

fout <- nyissa meg a "grafSzinezes_branchandbound_1000.out" kimeneti
      fajlt

Olvassa n es m fin
adj <- n meretu ures szomszedsagi lista letrehozasa
szinek <- hozzon létre nullakkal kitöltött n meretu vektort
minszinek <- 1-es meretu vektor letrehozasa

for i from 0 to m - 1:
  Read u and v from fin
  Add v - 1 to adj[u - 1]
  Add u - 1 to adj[v - 1]

start <- aktualis ido ezredmasodpercben
szinekSzama <- grafSzinezes()
end <- aktualis ido ezredmasodpercben
duration <- end - start

vertexszinek <- szinek masolata

for i from 0 to n - 1:
  Write "Az i + 1 csucs szinezett vertexszinek[i]" to fout

Write "A szukseges szinek szama: szinekSzama" to fout
Write "Vegrehajtsi ido: idotartam ezredmasodperc" to fout

bezar fin and fout

```

[Ant06][Ope23]

## 2.5. Íme az elkészített c++ program működésének vázlatos leírása:

1. A kód tartalmazza a szükséges könyvtárakat a bemenethez/kimenethez, a vektorkezeléshez, a rendezéshez, a fájlműveletekhez és az idő méréséhez.
4. Meghatároz egy konstans 'MAXN' értéket, amely a csúcsok maximális számát jelenti.
7. Deklarál egy "n" változót a csúcsok számának tárolására.
8. Adj szomszedsági listát deklarál a gráf reprezentálására, ahol minden csúcsnak van egy vektora a szomszédos csúcsokból.
9. Egy tömböt színnekdeklarál a csúcsokhoz rendelt színek tárolására.
10. Egy egész számot deklarál 'minSzinek'-ként, hogy nyomon kövesse az eddig talált színek minimális számát.
13. Meghatároz egy 'megFelelo'függvényt, amely ellenőrzi, hogy biztonságos-e a cszín hozzárendelése a 'vcsúchhoz a szomszédos csúcsok vizsgálatával.
20. Meghatározza a GrafSzinezesUtil'rekurzív függvényt, amely visszalépést hajt végre, hogy megtalálja a grafikon színezéséhez szükséges minimális számú színt.



21. Az alapeset: Ha az összes csúcsot kiszínezték, frissítse a 'minSzinek' értéket az aktuális érték minimumával és a szinekUsed' értékkel, majd térjen vissza.

24. Ismétlje meg a színeket 1-től szinekUsed + 1-ig, és próbáljon minden színt hozzárendelni a 'v' csúcshoz.

25. Ha biztonságos a "c" szín hozzárendelése a "v" csúcshoz, rendelje hozzá a színt, és rekurzívan hívja meg a "GrafSzinezesUtil"-t a következő csúcshoz, ahol ugyanannyi szín van felhasználva.

27. A rekurzív hívás után lépjen vissza a 'v' csúcs színének 0-ra való visszaállításával.

32. Meghatároz egy GrafSzinezes függvényt, amely a 'minSzinek'-t a lehető legnagyobb értékre inicializálja, és meghívja a GrafSzinezesUtil'-t a szükséges minimális számú szín megtalálásához.

36. A fő függvény kezdete.

37. Megnyit egy "GrafSzinezes.in" nevű bemeneti fájlt olvasásra.

38. Megnyit egy "GrafSzinezes.out" nevű kimeneti fájlt íráshoz.

41. Beolvassa a bemeneti fájlból az ncsúcsok számát és az 'm' élek számát.

44. Beolvassa az éleket abemeneti fájlból, és hozzáadja a szomszédsági listához.

47. Elkezdi mérni a végrehajtási időt.

49. Meghívja a GrafSzinezes' függvényt, és az eredményt a szinekSzama'-ban tárolja.

51. Leállítja a végrehajtási idő mérését.

54. Kinyomtatja az egyes csúcsok színeit mind a konzolba, mind a kimeneti fájlba.

59. Kinyomtatja a szükséges számú színt a konzolnak és a kimeneti fájlra is.

62. Kinyomtatja az eltelt időt a konzolra és a kimeneti fájlra is.

65. A program véget ér.

## 2.6. A kód összetettsége a következőképpen elemezhető:

- A "megFelelo" függvény időbonyolultsága  $O(\text{degree}(v))$ , mivel ellenőrzi a szomszédos csúcsok színeit, ahol a fok(v) egy "v" csúcs fokát jelöli. - A GrafSzinezeseSegedprogram függvény időbonyolítása  $O(\text{hasznaltSzinek}^n)$ , mivel minden lehetséges színkombinációt kipróbál minden csúcsnál, egészen a  $\text{hasznaltSzinek} + 1$ -ig. - A GrafSzinezes függvény egyszer meghívja a GrafSzinezeseSegedprogram-ot, tehát ugyanolyan bonyolultságú az idő szempontjából. - A teljes összetettség az n csúcsok számától és az élek számától függ. A domináns tényező a visszalépési folyamat a GrafSzinezeseSegedprogram függvényben, amelynek összetettsége  $O(\text{hasznaltSzinek}^n)$ . [Knu08]

## 2.7. A c++ program futtatása során a következő eredményeket kaptam

Az alábbi táblázat adatait a c++ programok futtatása során generáltattam.

Microsoft Excel táblázatkezelő használatával összesítettem az adatokat.

Zöld színnel emeltem ki a legjobb megoldások eredményeit.

Az adatok részletes értékelésére és összesítésére a mérések című fejezetben kerül sor.

[Ant06][CAA03]

**2.1. táblázat.** Általános gráf backtracking branch and bound statisztika táblázat

közelítő megoldás	Backtracking		BranchAndBound	
	Nr colors	Idő	Nr colors	Idő
1	4	42	3	0
2	3	41	3	0
3	4	45	4	0
4	5	97	5	0
5	6	515	6	0
6	4	44	4	0
7	4	92	4	0
8	4	42	4	0
9	4	67	4	0
10	6	1068	6	0
Átlag n=10	4.4	205.3	4.3	0
1	21	567	21	0
2	20	563	20	0
3	21	602	21	0
4	19	520	19	0
5	22	630	22	0
6	23	628	23	0
7	22	2580	22	0
8	23	666	23	0
9	22	613	22	0
10	23	625	23	0
Átlag n=100	21.6	799.4	21.6	0
1	127	234875	127	200
2	128	243072	128	134
3	128	231409	128	131
4	124	219030	124	127
5	124	225834	124	130
6	127	238315	127	136
7	125	220831	125	126
8	126	244642	126	127
9	124	222622	124	130
10	126	234478	126	133
Átlag n=1000	125.9	231510.8	125.9	137.4

Általános gráf esetén BranchAndBound sokkal gyorsabb, mint a backtracking. Az átlagos teljesítményüket nézve, mindkettő hasonló eredményeket ad, viszont az idő szempontjából a backtracking messze elmarad.

[Ope23]

**2.2. táblázat.** Hamiltoni gráf backtracking branch and bound statisztika táblázat

n	közelítő megoldás	Backtracking		BranchAndBound	
		Nr colors	Idő	Nr colors	Idő
10	Átlag n=100	10	429	10	0
100	Átlag n=1000	100	6063	100	4
1000	Átlag n=10000	1000	> 1 ora	1000	3100

Hamiltoni gráf esetén mindkét megoldás algoritmus megoldása helyes, viszont a branch and bound sokkal gyorsabb, mint a backtracking.

## 3. fejezet

# Bin packing feladat bemutatása

Tegyük fel, hogy korlátlan számú, külön - külön egy kapacitású tárolóhelyünk van és  $n$  darab  $s_1, s_2, \dots, s_n$  méretű tárggyal rendelkezünk, ahol  $0 < s_i \leq 1$ . Mi a legkisebb számú tárolóhely, amelybe a tárgyakat be lehet pakolni? A feladat alternatív megfogalmazása egy  $k$  egész számot ad meg, és azt kérdezi, hogy a tárgyak beférnek-e  $k$  tárolóba. [Sar78]

A tárolóba pakolás gyakorlati alkalmazásai közé tartozik az adatok számítógépes memóriákba csomagolása (...) és egy termék (például szövet vagy fűrészárú) nagy, szabványos méretű darabokból való kivágására vonatkozó megrendelések teljesítése.

Bin packing program bemeneti adatait egy adatgeneráló algoritmus segítségével készítettem el. Ezzel sok időt nyertem, mivel a bemeneti adatokat így nem volt szükséges egyenként begépelni, hanem a generáló program automatikusan kigenerálta azokat.

A Bin Packing feladat megoldására a következő pontos algoritmust készítettem el. [GKP98]

```
BinPackingAlgoritmus():
    Bemeneti fajl olvasasa "binpacking10.in" as fin
    Nyissa meg a kimeneti fajlt "binpacking10.out" as fout

    Olvas n from fin
    Kiir n

    Hozzon létre egy n meretu ures s vektort

    for i from 0 to n-1:
        olvas s[i] from fin
        kiir s[i]

    Rendezze a vektorokat csokkeno sorrendbe a Quicksort segitsegevel

    start = aktualis ido
    Call binPacking(s)
    end = aktualis ido

    idotartam = end - start
    Print "Vegrehajtsi ido: idotartam ezredmasodperc"
    Write "Vegrehajtsi ido: idotartam ezredmasodperc" to fout
```

```

    bezar fin and fout

Quicksort(arr, bal, jobb):
    if bal >= jobb:
        return

    pivot = arr[(bal + jobb) / 2]
    i = bal
    j = jobb

    while i <= j:
        while arr[i] > pivot:
            i++
        while arr[j] < pivot:
            j--
        if i <= j:
            swap arr[i] with arr[j]
            i++
            j--

    Quicksort(arr, bal, j)
    Quicksort(arr, i, jobb)

binPacking(s):
    n = size of vector s

    hozzunk létre egy üres vektort Bj vektorokból, amelyek mérete n
    Hozzon létre egy üres bj vektort, amelynek mérete n, kitöltve 0-val

    for t from 0 to n-1:
        j = 0

        while j < n and bj[j] + s[t] > 1:
            j++

        Bj[j].push_back(t)
        bj[j] += s[t]

    for j from 0 to n-1:
        if Bj[j] is üres:
            break

    Print "Bin j+1: "
    for t in Bj[j]:
        Print s[t]
        Write s[t] to fout

```

```
Print uj sor
Write uj sor to fout
```

```
Main():
    Call BinPackingAlgoritmus()
```

[CAA03]

Az elkészített pontos\_binpacking\_RandomSelect.cpp program bemutatása:

1. A kód tartalmazza a szükséges könyvtárakat a bemenethez/kimenethez, a vektorkezeléshez, a rendezéshez, a véletlenszám generálásához és az idő méréséhez.

5. Deklarál egy „random\_select” nevű függvényt, amely véletlenszerűen kiválaszt egy adott számú elemet egy lebegőpontos vektorból.

8. Létrehoz egy üres „random\_elemek” vektort a véletlenszerűen kiválasztott elemek tárolására.

9. Létrehoz egy vektort „fennmaradó\_indexek” indexekkel minden elemhez.

11. Iterál az elemvektor indexein, és minden indexet hozzárendel a megfelelő pozícióhoz a „fennmaradó\_indexek” mezőben.

14. Véletlenszerű magot generál az aktuális idő alapján a „std::chrono::system\_clock::now().time\_since\_epoch().count()” segítségével.

15. Megkeveri a „fennmaradó\_indexek” vektort a véletlen mag segítségével az indexek sorrendjének véletlenszerűvé tételéhez.

18. A „szám” ismétlésével kiválasztja a „szám” véletlenszerű elemet az „elemek” vektorból a kevert indexek alapján.

20. Lekéri az indexet a kevert indexekből.

21. Hozzáadja a megfelelő elemet az „elemek” vektorból a „random\_elemek” vektorhoz.

24. Visszaadja a véletlenszerűen kiválasztott elemeket tartalmazó „random\_elemek” vektort.

27. Meghatározza a „binPackingRandomSelect” nevű függvényt a tárolóedény-csomagolási algoritmus véletlenszerű kiválasztással történő végrehajtásához.

29. Lekéri az „elemek” vektor méretét.

30. Létrehoz egy üres „binSizes” vektort a táruk méretének tárolására.

31. Létrehoz egy „binIndexes” vektort indexekkel minden elemhez.

34. Iterál az elemvektor indexein, és minden indexet hozzárendel a megfelelő pozícióhoz a „binIndexes”-ben.

37. Véletlenszerű magot generál az aktuális idő alapján a „std::chrono::system\_clock::now().time\_since\_epoch().count()” segítségével.

38. Megkeveri a „binIndexes” vektort a véletlen mag segítségével az indexek sorrendjének véletlenszerűvé tételéhez.

41. Méri a kezdési időt a „high\_resolution\_clock” segítségével.

44. Iterál a kevert tárindexeken.

46. Lekéri az aktuális elemet az „elemek” vektorból a kevert tárindex alapján.

47. A „placed” logikai változót false értékre inicializálja. 50. Iterál a rekeszméret-vektoron, hogy találjon egy tárolót, amelybe belefér az aktuális elem.

52. Ha egy tálcán van elég hely az aktuális tétel számára, kivonja a tétel méretét a tároló méretéből, és az „elhelyezett” értéket igazra állítja.

56. Ha az elemet nem lehetett egy meglévő tálcába helyezni, a fennmaradó kapacitás hozzáadásával új tálcát hoz létre.

61. Kinyomtatja a konzolhoz használt tálcák számát.

64. Megnyitja a „binpacking20.out” kimeneti fájlt íráshoz.

65. Beírja a kimeneti fájlba a felhasznált rekeszek számát.

68. Az egyes tálcák méretét kinyomtatja a konzolra, és beírja a kimeneti fájlba.

74. Méri a befejezési időt a „high\_resolution\_clock” segítségével.

76. Kiszámítja az eltelt időt úgy, hogy kivonja a kezdési időt a befejezési időpontból.

77. Kiírja a végrehajtási időt a konzolra és beírja a kimeneti fájlba.

80. Bezárja a kimeneti fájlt.

83. Elkezdődik a „fő” függvény.

85. Deklarál egy „filename” karakterlánc-változót a bemeneti fájlnevével.

86. Egy vektor „elemek” deklarációját a bemeneti fájlból kiolvasott elemek tárolására.

89. Megnyitja olvasásra a „filename” által megadott bemeneti fájlt.

90. Beolvassa az „n” (elemek száma) értékét a bemeneti fájlból.

92. A „binkapacitas” értéket 1.0-ra állítja.

94. Átméretezi az „elemek” vektort, hogy illeszkedjen „n” elemhez.

95. Beolvassa az „n” elemet a bemeneti fájlból, és hozzárendeli őket az „elemek” vektorhoz.

100. Bezárja a bemeneti fájlt.

103. Meghívja a „binPackingRandomSelect” függvényt az „elemek” vektorral és a „binkapacitas” paraméterrel.

104. A program sikeres végrehajtása esetén 0-val tér vissza.

A kód összetettsége az „elemek” bemeneti vektor méretétől függ. A legjelentősebb része a tárolóedény-csomagolási algoritmus, amelynek időbonyolultsága  $O(n^2)$  az „n” elem és potenciálisan „n” ládaméret felett iteráló beágyazott hurkok miatt. A véletlen szelekciós rész bonyolultsága  $O(n)$ , mert egyszer megkeveri az indexvektort.

A Bin Packing feladat megoldására a következő közelítő algoritmust készítettem el.

```
KozelitoBinPackingAlgoritmus():
```

```
    Bemeneti fajl olvasasa "binpacking10000_7.in" as fin
    Nyissa meg a kimeneti fajt "binpacking10000_7.out" as fout
    Read n from fin
    Print n
    Set binkapacitas = 1
    Print binkapacitas
    Hozzon létre ures vektorelemeket
    for i from 0 to n-1:
        Read size from fin
        Add size to elemek
    bezar fin
    Call KozelitoBinPackingAlgoritmus(elemek, binkapacitas)
    Close fout
```

```
quicksort(arr, low, high):
```

```
    if low < high:
        pivot = arr[high]
```

```

    i = low - 1

    for j from low to high - 1:
        if arr[j] > pivot:
            i++
            swap(arr[i], arr[j])

    swap(arr[i + 1], arr[high])
    partition = i + 1

    quicksort(arr, low, partition - 1)
    quicksort(arr, partition + 1, high)

kozelitoBinPacking(elemek, binkapacitas):
    n = elemek merete
    ures vektor létrehozasa binSizes

    Rendezze az elemeket csokkeno sorrendben a gyorsrendezes segitsegevel

    start = aktualis ido

    for i from 0 to n-1:
        aktualisElem = elemek[i]
        placed = false

        for j from 0 to size of binSizes - 1:
            if binSizes[j] >= aktualisElem:
                binSizes[j] -= aktualisElem
                placed = true
                break

        if not placed:
            binSizes.push_back(binkapacitas - aktualisElem)

    Print "Felhasznalt rekeszek szama: a tartalyok merete Meretek"

    kimeneti fajl megnyitasa "binpacking10000_7.out" as fout

    "felhasznalt szemetesek szama: szemetes meretek" to fout
    for i from 0 to size of binSizes - 1:
        Print "Bin i+1 size: binkapacitas - binSizes[i]"
        Write "Bin i+1 size: binkapacitas - binSizes[i]" to fout

    end = aktualis ido

    duration = end - start
    Print "Vegrehajtsi ido: idotartam ezredmasodperc"
    Write "Vegrehajtsi ido: idotartam ezredmasodperc" to fout

```



Close fout

Main():

Call ApproximateBinPackingAlgorithm()

[CAA03]

Az elkészített közelítő\_binpacking\_DFF.cpp program bemutatása:

1. A kód tartalmazza a szükséges könyvtárakat a bemenethez/kimenethez, a vektorkezeléshez, a rendezéshez, a matematikai függvényekhez, a fájlműveletekhez és az időméréséhez.

8. Definiál egy „gyorsrendezés” függvényt a lebegések „arr” vektorának csökkenő sorrendbe rendezéséhez a gyorsrendezési algoritmus segítségével.

28. Meghatároz egy „approximateBinPacking” függvényt, amely megvalósítja a hozzávetőleges tárolóedény-csomagolási algoritmust. Bemenetként egy vektor „elemeket” és egy „binkapacitas”-t vesz fel, és végrehajtja a hozzávetőleges tárolóedény-csomagolási folyamatot az algoritmus lépései szerint.

62. Elkezdődik a „fő” függvény.

64. Egy „n” egész változót deklarál az elemek számának tárolására.

65. A „binkapacitas” lebegő változót deklarálja az egyes táruk kapacitásának tárolására.

66. Létrehoz egy üres vektor „elemeket” az elemméretek tárolására.

68. Megnyitja olvasásra a „binpacking10000\_7.in” bemeneti fájlt.

69. Beolvassa az „n” értékét a bemeneti fájlból.

70. Kiírja az „n” értékét a konzolra.

71. A „binkapacitas” értéket 1-re állítja.

72. Kiírja a „binkapacitas” értékét a konzolra.

75. Beolvassa az egyes elemek „méretét” a bemeneti fájlból, és hozzáadja az „elemek” vektorhoz.

84. Bezárja a bemeneti fájlt.

87. Meghívja az „approximateBinPacking” függvényt az „elemek” vektorral és a „binkapacitas” bemenettel, hogy végrehajtsa a hozzávetőleges tárolóedény-csomagolási algoritmust.

93. Az „approximateBinPacking” függvény elindul.

95. A binSizes vektort inicializálja a tárolók méretének tárolására.

98. Meghívja a „quicksort” függvényt az „elemek” vektor csökkenő sorrendbe rendezéséhez.

103. Méri a kezdési időt a „high\_resolution\_clock” segítségével.

106. Iterál minden egyes elemet az „elemek” vektorban.

108. Lekéri az aktuális elemméretet.

109. A „placed” logikai változót false értékre inicializálja.

112. Megpróbálja elhelyezni az elemet egy meglévő tárolóban a „binSizes” vektor feletti iterációval.

115. Ha az elem elhelyezhető egy meglévő tálcába, frissíti a tároló méretét, és az „elhelyezett” értéket igazra állítja.

120. Ha az elemet nem lehetett egy meglévő tálcába helyezni, a fennmaradó kapacitás hozzáadásával új tárolót hoz létre.

128. Kiírja a konzolhoz használt tálcák számát.

131. Megnyitja a „binpacking10000\_7.out” kimeneti fájlt írásra.

133. A felhasznált rekeszek számát írja akimeneti fájlba.

136. Kinyomtatja az egyes tálcák méretét a konzolra, és beírja a kimeneti fájlba.

143. Méri a befejezési időt a „high\_resolution\_clock” segítségével.

146. Kiszámítja az eltelt időt úgy, hogy kivonja a kezdési időt a befejezési időpontból, és ezredmásodpercekre konvertálja.

147. Kiírja a végrehajtási időt a konzolra és beírja a kimeneti fájlba.

150. Bezárja a kimeneti fájlt.

154. A program véget ér.

A kód összetettsége a következőképpen elemezhető: - A „gyorsrendezés” függvény átlagos eseti időbonyolultsága  $O(n \log n)$ , ahol  $n$  a rendezett vektor mérete. A legrosszabb forgatókönyv szerint az időbonyolítása  $O(n^2)$ . - Az „approximateBinPacking” függvény az „elemek” vektor összes elemén áthalad, ami  $O(n)$  időbonyolultságot eredményez, ahol  $n$  a vektor mérete. - Összességében a kód időbonyolultságát a „quicksort” függvény uralja, ami  $O(n \log n)$  időbonyolultságot eredményez átlagos és legrosszabb esetben.

### 3.1. táblázat. pontos megoldás Bin packing statisztika táblázat

n	pontos megoldás	FF		RFF		IFF		DFF	
		Bin	Idő	Bin	Idő	Bin	Idő	Bin	Idő
10	7	7	4	7	0.0090639	8	4	7	3
15	11	11	6	12	0.0060736	12	5	11	5
20	12	13	6	13	0.0058952	14	6	12	6

### 3.2. táblázat. közelítő megoldás Bin packing statisztika táblázat

	FF		RFF		IFF		DFF	
	Bin	Idő	Bin	Idő	Bin	Idő	Bin	Idő
1	52	89	52	0.0689477	58	53	50	63
2	61	112	62	0.0887968	68	88	61	84
3	61	94	60	0.0803511	66	91	59	103
4	59	79	58	0.0803511	65	118	60	102
5	58	74	57	0.065474	65	99	58	91
6	55	93	56	0.0700975	62	105	55	79
7	59	102	59	0.0885393	65	106	58	92
8	60	99	60	0.1036	65	118	59	105
9	67	105	67	0.0865128	73	92	67	102
10	61	106	62	0.0897954	68	112	62	84
Átlag n=100	59.3	95.3	59.3	0.08224657	65.5	98.2	58.9	90.5
1	596	882	595	0.873757	666	1028	603	917
2	589	966	593	0.858663	662	1003	594	909
3	609	934	607	0.901153	676	1073	613	899
4	612	920	611	0.855988	677	1125	614	899
5	618	951	614	0.965832	685	1061	626	924
6	604	949	606	0.861831	672	1061	610	965
7	606	896	603	0.741969	669	1023	611	990
8	606	809	603	0.876714	672	967	608	905
9	591	935	590	0.935648	661	920	594	837
10	592	900	594	0.9173	663	1068	599	747
Átlag n=1000	602.3	914.2	601.6	0.8788855	670.3	1032.9	607.2	899.2
1	5950	8021	5953	8.62314	6665	10011	6034	8421
2	5931	7845	5920	7.79854	6644	9128	5990	8170
3	5970	7955	5973	8.02626	6692	9689	6055	8682
4	5918	7833	5913	8.3	6633	9144	5982	8142
5	5888	8485	5884	8.08707	6609	9573	5959	8341
6	5897	7627	5902	7.97438	6625	8982	5987	7424
7	5939	7043	5943	7.65755	6665	9957	6016	8300
8	5918	7401	5918	7.875	6651	9127	6002	8146
9	5894	8037	5889	7.71207	6626	9256	5971	8341
10	5866	7677	5871	9.20223	6584	9294	5928	8559
Átlag n=10000	5917.1	7792.4	5916.6	8.125624	6639.4	9416.1	5992.4	8252.6

Átlagértéket számolva a DFF a legjobb eredményt adta 100-as nagyságrendű ládaszám esetén. Átlagértéket számolva az RFF a legjobb eredményt adta 1000-es nagyságrendű ládaszám esetén, viszont még mindig a DFF a leggyorsabban ad eredményt. Átlagértéket számolva az RFF a legjobb eredményt adta 10000-es nagyságrendű ládaszám esetén. Fontos megjegyezni, hogy az FF is mindhárom kategória esetén az elméleti legjobb megoldás közeli eredményt generált a DFF futási idejéhez közelítő átlagos időn belül.

## 4. fejezet

# Hátizsák feladat bemutatása

Adott egy  $n$  darab,  $s_1, \dots, s_n$  méretű tárggyakból álló halmaz és egy  $C$  kapacitású hátizsák, ahol  $s_1, \dots, s_n$  és  $C$  pozitív egész számok. A kérdés: a tárgyak melyik részhalmaza tölti ki a legteljesebben a hátizsákot? (Annak rendje és módja szerint keressük meg azt a 0/1 X vektort, amely maximalizálja  $\sum_{j=1}^n s_i x_i$  azzal a feltétellel, hogy  $\sum_{j=1}^n s_i x_i \leq C$ ). A feladat alternatív megfogalmazása azt kérdezi, hogy van-e olyan részhalmaz, amely pontosan kitölti a zsákot.[Sar78]

A hátizsák pakolás feladatra a elkészített pontos algoritmus kódjának összetettsége a következőképpen elemezhető: - A „felosztás” függvény időbonyolultsága  $O(n)$ , ahol  $n$  a felosztott altömb mérete. - A „gyorsrendezés” függvény átlagos eseti időbonyolultsága  $O(n \log n)$ , ahol  $n$  a rendezett tömb mérete. A legrosszabb forgatókönyv szerint az időbonyolítása  $O(n^2)$ . - A „hatizsakDP” függvény dinamikus programozást használ egy két dimenziós DP tábla  $(n+1) \times (\text{kapacitás}+1)$  méretekkkel való kitöltésére. A beágyazott hurkok a DP-tábla összes elemén áthaladnak, ami  $O(n * \text{kapacitás})$  időbonyolultságot eredményez, ahol  $n$  az elemek száma, a kapacitás pedig a hátizsák kapacitása. - A „sortAndhatizsak” függvény meghívja a „quicksort” és a „hatizsakDP” függvényeket, amelyek mindegyike rendelkezik a fent leírt időbeli bonyolultsággal. Ezért ennek a függvénynek az időbeli összetettségét a „hatizsakDP” időbonyolultsága uralja, ami  $O(n * \text{kapacitás})$ . - A „main” függvény beolvassa a bemeneti adatokat, létrehozza az „elemek” vektort, és meghívja a „sortAndhatizsak” függvényt, amelynek időbeli összetettsége  $O(n * \text{kapacitás})$ . - Összességében a kód időbeli összetettsége  $O(n * \text{kapacitás})$ , ha a gyorsrendezési lépés nem uralja a végrehajtási időt. Ha a gyorsrendezési lépés válik dominánssá, az időbonyolultság  $O(n^2 * \text{kapacitás})$ .

A hátizsák pakolás feladatra a következő SarahBaase BruteForce algoritmus szerkesztettem:

```
struct Item:
    suly
    ertek

hatizsakNyersEro(elemek, kapacitas, elteltIdo):
    n = elemek merete

    start = aktualis ido
```

```

elemek kiválasztása = üres vector of elemek
maxSum = 0

for reszhalmaz from 0 to  $2^n - 1$ :
    sum = 0
    aktualisHalmaz = üres elemek vektora

    for i from 0 to n - 1:
        if (subset & (1 << i)) != 0:
            sum += elemek[i].suly
            Add elemek[i] to aktualisHalmaz

    if sum <= kapacitas and sum > maxSum:
        maxSum = sum
        kiválasztottElemek = aktualisHalmaz

end = aktualis idő
eltelt idő = end - start

return kiválasztottElemek

ElemekOlvasasaFajlból(filename, kapacitas):
    inputFile = filename fájl megnyitása
    elemek = üres vector of elemek

    if inputFile is open:
        Read kapacitas from inputFile

        miközben a súlyt és az értéket olvassa be az inputFile-ből:
            Adj hozzá súlyt és értéket tartalmazó elemet az elemekhez
        Close inputFile
    else:
        Print "Nem lehet megnyitni a bemeneti fájlt."

    return elemek

eredmenyKiiratasFajlba(filename, kiválasztottElemek, teljesSuly,
teljesertek, elteltIdo):
    outputFile = open file filename

    if outputFile is open:
        Write "elemek selected as a solution:" to outputFile

        for each item in kiválasztottElemek:
            Write "suly = item.suly, with érték = item.ertek" to outputFile

        Write "Sum suLy = teljesSuly" to outputFile
        Write "Sum érték = teljesertek" to outputFile
        Write "Eltelt idő (mikroszekundum): elteltIdo" to outputFile

```

```

        Close outputFile
    else:
        Print "Nem lehet megnyitni a kimeneti fajlt."

Main():
    elemek = elemek olvasasa fajlbol("hatizsak100_9.in", kapacitas)

    if kapacitas <= 0:
        Print "Invalid kapacitas ertek in the input file."
        Lepjen ki a programbol hibakoddal 1

    if elemek is ures:
        Print "Nincsenek elemek found in the input file."
        Exit program with error code 1

    elteltIdo = 0
    kivalasztottelemek = hatizsakNyersEro(elemek, kapacitas, elteltIdo)

    teljessuly = 0
    teljesertek = 0

    for each item in kivalasztottelemek:
        teljessuly += item.suly
        teljesertek += item.ertek

    eredmenyKiiratasaFajlba("hatizsak100_9_brute_force.out",
        kivalasztottelemek, teljessuly, teljesertek, elteltIdo)

    Lepjen ki a programbol sikerkoddal 0

```

#### [CAA03]

Az elkészített SarahBaase\_BruteForce\_hatizsak.cpp program bemutatása:

1. A kód tartalmazza a szükséges könyvtárakat a bemenethez/kimenethez, a vektor-kezeléshez, az algoritmusokhoz és az idő méréséhez.
4. A kód az „std” névteret használja a kényelem érdekében.
7. Meghatároz egy „Cikk” nevű „szerkezetet”, amely egy súllyal és értékkel rendelkező elemet jelöl.
9. Meghatározza a „hatizsakBruteForce” nevű függvényt, amely a hátizsák- problémát brute force megközelítéssel oldja meg.
13. Lekéri az elemek számát az „elemek” bemeneti vektorból.
15. Az „std::chrono::steady\_clock” segítségével méri a kezdési időt.
17. Létrehoz egy üres „kivalasztottelemek” vektort a kiválasztott elemek tárolására.
18. A „maxSum” értéket 0-ra inicializálja a súlyok maximális összegének tárolásához.
21. Bitenkénti műveletek segítségével iterálja az elemek összes lehetséges részhalma-  
zatát.

24. Az „összeg” értéket 0-ra inicializálja, hogy kiszámítsa az aktuális részhalmaz súlyainak összegét.
25. Létrehoz egy üres „aktualisHalmaz” vektort az aktuális részhalmaz elemeinek tárolásához.
28. Bitenkénti ÉS művelettel ellenőrzi, hogy az „i”-edik elem benne van-e az aktuális részhalmazban.
30. Ha az „i”-edik elem szerepel, hozzáadja a súlyát az „összeghez”, és hozzáadja az elemet a „aktualisHalmaz”-hez.
34. Ellenőrzi, hogy az aktuális részhalmaz súlya kisebb-e vagy egyenlő-e a kapacitással és nagyobb-e az aktuális maximális összegnél.
36. Frissíti a „maxSum” értéket az aktuális összeggel, és hozzárendeli a „aktualisHalmaz” értéket a „kiválasztottelemek”-hez.
43. A befejezési időt az „std::chrono::steady\_clock” segítségével méri.
44. Kiszámítja az eltelt időt mikroszekundumban úgy, hogy kivonja a kezdési időt a befejezés időpontjából.
47. Visszaadja a kiválasztott elemeket tartalmazó „kiválasztottelemek” vektort.
51. Meghatározza a „readelemekFromFile” nevű függvényt, amely kiolvassa az elemeket és a kapacitást egy bemeneti fájlból, és visszaadja azokat az „Elem” vektoraként.
52. Megnyitja a „fájlnév” által megadott bemeneti fájlt.
53. Létrehoz egy üres vektor „elemeket” az elemek tárolására.
56. Beolvassa a kapacitást a bemeneti fájlból.
57. Deklarálja a „suly” és „ertek” változókat az egyes tételek súlyának és értékének olvasásához.
58. Beolvassa a súly- és értékpárokat a bemeneti fájlból, és hozzáadja őket az „elemek” vektorhoz, mint „elem” struktúra. 62. Bezárja a bemeneti fájlt. [SK11] 66. Az olvasott elemeket tartalmazó „elemek” vektort adja vissza.
68. Meghatározza a „writeOutputToFile” nevű függvényt, amely a kiválasztott elemeket, a teljes súlyt, a teljes értéket és az eltelt időt egy kimeneti fájlba írja.
69. Megnyitja a „fájlnév” által megadott kimeneti fájlt.
72. A kiválasztott elemeket a kimeneti fájlba írja.
76. Beírja a teljes súlyt és a teljes értéket a kimeneti fájlba.
79. A kimeneti fájlba írja az eltelt időt mikroszekundumban.
83. Bezárja a kimeneti fájlt.
87. Elkezdődik a „fő” függvény.
89. A „kapacitás” egész változót deklarálja.
90. Beolvassa az elemeket és a kapacitást a „hatizsak100\_9.in” bemeneti fájlból, és hozzárendeli őket az „elemek” vektorhoz.
93. Ellenőrzi, hogy a kapacitás értéke érvénytelen-e (0-nál kisebb vagy egyenlő).
94. Hibaüzenetet nyomtat a „cerr”-re, és 1-et ad vissza, jelezve a hibát.
97. Ellenőrzi, hogy nem található-e elem a bemeneti fájlban.
98. Hibaüzenetet nyomtat a „cerr”-nek, és 1-et ad vissza, jelezve a hibát.
101. Az eltelt idő eltárolásához kettős változót deklarál, az ‘elteltIdo’.
103. Meghívja a „hatizsakBruteForce” függvényt az „elemek” vektorral, a „kapacitás” és az „eltelt idő” argumentumokkal, és az eredményt a „kiválasztottelemek”-hez rendeli.
106. Inicializálja a „teljessuly” és a „teljesertek” változókat a kiválasztott cikkek összsúlya és összértéke.

107. Iterál a „kiválasztottelemek” vektoron, és hozzáadja az egyes elemek súlyát és értékét a megfelelő változókhoz.

112. Meghívja a „writeOutputToFile” függvényt, hogy a kiválasztott elemeket, a teljes súlyt, a teljes értéket és az eltelt időt a „hatizsak100\_9\_brute\_force.out” kimeneti fájlba írja.

115. A program sikeres végrehajtása után 0-val tér vissza. [K04] A kód összetettsége az elemek számától függ, amelyet „n” jelöl. A brute force algoritmus az elemek összes lehetséges részhalmazát generálja, ami  $O(2^n)$  időbonyolultságot eredményez. Ennek az az oka, hogy minden elem esetében két lehetőség van: vagy szerepel, vagy kizár egy részhalmazból. Ezért a részhalmazok száma  $2^n$ . Ezen túlmenően a kód minden részhalmazon iterál, és ellenőrzi annak súlyát és értékét, ami  $O(n * 2^n)$  bonyolultságú beágyazott hurkokat eredményez. A fájlok olvasásának és írásának időbonyolultsága általában  $O(n)$ , ahol n a fájl mérete.

A hátizsak pakolás feladatra a következő dinamikus programozással megoldott algoritmust szerkesztettem:

```
struct Item:
    suly
    ertek

hatizsak100_9Approximation(elemek, kapacitas, elteltIdo):
    n = size of elemek
    dp = 2D egesz szamok vektora meretekkel (n + 1) x (kapacitas + 1)

    start = aktualis ido

    for i from 1 to n:
        for j from 1 to kapacitas:
            if elemek[i - 1].suly <= j:
                dp[i][j] = maximum of (dp[i - 1][j], elemek[i - 1].ertek + dp[i - 1][j - elemek[i - 1].suly])
            else:
                dp[i][j] = dp[i - 1][j]

    kiválasztottelemek = ures vector of elemek
    i = n
    j = kapacitas

    while i > 0 and j > 0:
        if dp[i][j] is not equal to dp[i - 1][j]:
            Add elemek[i - 1] to kiválasztottelemek
            j -= elemek[i - 1].suly
        i--

    Reverse kiválasztottelemek

    end = aktualis ido
    elteltIdo = end - start
```



```

    return kivalasztottelemek

readelemekFromFile(filename, kapacitas):
    inputFile = open file filename
    elemek = empty vector of elemek

    if inputFile is open:
        Read kapacitas from inputFile

        while reading suly and ertek from inputFile:
            Add Item with suly and ertek to elemek

        Close inputFile
    else:
        Print "Unable to open the input file."

    return elemek

eredmenyKiiratasFileba(filename, kivalasztottelemek, teljessuly,
teljesertek, elteltIdo):
    outputFile = open file filename

    if outputFile is open:
        Write "elemek kivalasztva, mint megoldas:" to outputFile

        for each item in kivalasztottelemek:
            Write "suly = item.suly, with ertek = item.ertek" to outputFile

        Write "Sum suly = teljessuly" to outputFile
        Write "Sum ertek = teljesertek" to outputFile
        Write "Eltelt ido (mikroszekundum): elteltIdo" to outputFile

        Close outputFile
    else:
        Print "Nem lehet megnyitni a kimeneti fajlt."

Main():
    elemek = readelemekFromFile("hatizsak100_9.in", kapacitas)

    ha kapacitas <= 0:
        Print "ervenytelen kapacitas ertek a bemeneti fajlban."
        Lepjen ki a programbol 1 hibakoddal

    ha elemek ures:
        Print "Nem talalhato elemek a bemeneti fajlban."

```

Lepjen ki a programbol 1 hibakoddal

```
elteltIdo = 0
kivalasztottelemek = hatizsak100_9Approximation(elemekek, kapacitas,
    elteltIdo)

teljessuly = 0
teljesertek = 0

for each item in kivalasztottelemek:
    teljessuly += item.suly
    teljesertek += item.ertek

writeOutputToFile("hatizsak100_9.out", kivalasztottelemek, teljessuly,
    teljesertek, elteltIdo)

Lepjen ki a programbol sikerkoddal 0
```

[K08] [K04]

A hátizsák pakolás dinamikus programozással algoritmus működésének leírása:

1. A kód tartalmazza a szükséges könyvtárakat a bemenethez/kimenethez, a vektorkezeléshez, az algoritmusokhoz és az idő méréséhez.

4. A kód az „std” névteret használja a kényelem érdekében.

7. Meghatároz egy „Cikk” nevű „szerkezetet”, amely egy súllyal és értékkel rendelkező elemet jelöl.

9. Meghatározza a „hatizsakApproximation” nevű függvényt, amely dinamikus programozással oldja meg a hátizsák problémáját. [K04] 13. Lekéri az elemek számát az „elemek” bemeneti vektorból.

14. Létrehoz egy „dp” két dimenziós vektort, amelynek mérete  $(n+1) \times (kapacitas+1)$ , és nullákkal inicializálja.

17. Az „std::chrono::steady\_clock” segítségével méri a kezdési időt.

20. Iterál minden elemet 1-től „n”-ig.

21. Minden tételnél az egyes kapacitásokon át iterál 1-től „kapacitás”-ig.

23. Ellenőrzi, hogy az aktuális cikk súlya kisebb vagy egyenlő-e az aktuális kapacitással.

25. Frissíti a dinamikus programozási táblát az aktuális elem kizárása ( $dp[i-1][j]$ ) és az aktuális elem ( $elemek[i-1].ertek + dp[i-1][j - tetelek[i-1].suly]$ ) közötti maximális érték kiválasztásával).

29. Ha az aktuális elem súlya nagyobb, mint az aktuális kapacitás, a „dp[i][j]” értéket  $dp[i-1][j]$  értékre állítja (kivéve az aktuális elemet).

34. Létrehoz egy üres „kivalasztottelemek” vektort a kiválasztott elemek tárolására.

35. Az „i” és „j” változókat a dinamikus programozási táblázat utolsó sorába és oszlopába inicializálja. [K04] 36. Iterál, amíg az „i” nagyobb, mint 0, és a „j” nagyobb, mint 0.

38. Ellenőrzi, hogy a dinamikus programozási táblázat aktuális cellájának értéke eltér-e a felette lévő cella értékétől ( $dp[i][j] \neq dp[i-1][j]$ ).

40. Hozzáadja az aktuális elemet a „kiválasztottelemek” vektorhoz, és kivonja a súlyát a „j”-ből.

42. Csökkentse az „i” értéket az előző sorba lépéshez.

46. Megfordítja az elemek sorrendjét a „kiválasztottelemek” vektorban az eredeti sorrend visszaállításához.

51. A befejezési időt az „std::chrono::steady\_clock” segítségével méri.

52. Kiszámítja az eltelt időt mikromásodpercben úgy, hogy kivonja a kezdési időt a befejezés időpontjából.

55. Visszaadja a kiválasztott elemeket tartalmazó „kiválasztottelemek” vektort.

59. Meghatározza a „readelemekFromFile” nevű függvényt, amely kiolvassa az elemeket és a kapacitást egy bemeneti fájlból, és visszaadja azokat az „Elem” vektoraként.

60. Megnyitja a „fájlnev” által megadott bemeneti fájlt.

61. Létrehoz egy üres vektor „elemeket” az elemek tárolására.

64. Beolvassa a kapacitást a bemeneti fájlból.

65. Deklarálja a „suly” és „ertek” változókat az egyes tételek súlyának és értékének olvasásához.

66. Beolvassa a súly- és értékpárokat a bemeneti fájlból, és hozzáadja őket az „elemek” vektorhoz, mint „elem” struktúra.[\[Iva19\]](#)

70. Bezárja a bemeneti fájlt.

74. Az olvasott elemeket tartalmazó „elemek” vektort adja vissza.

76. Meghatározza a „writeOutputToFile” nevű függvényt, amely a kiválasztott elemeket, a teljes súlyt, a teljes értéket és az eltelt időt egy kimeneti fájlba írja.

77. Megnyitja a „fájlnev” által megadott kimeneti fájlt.

80. A kiválasztott elemeket a kimeneti fájlba írja.

84. Beírja a teljes súlyt és a teljes értéket a kimeneti fájlba.

87. Beírja a kimeneti fájlba az eltelt időt mikroszekundumban.

91. Bezárja a kimeneti fájlt.

95. Elkezdődik a „fő” függvény.

97. A „kapacitás” egész változót deklarálja.

98. Beolvassa az elemeket és a kapacitást a „hatizsak10\_9.in” fájlból, és az „elemek” vektorban tárolja őket.

101. Ellenőrzi, hogy a kapacitás kisebb-e vagy egyenlő-e 0-val, és hibaüzenetet jelenít meg, ha igaz.

104. Ellenőrzi, hogy nem található-e elem a bemeneti fájlban, és ha igaz, hibaüzenetet jelenít meg.

108. Az algoritmus eltelt idejét tárolja egy double „elteltIdo” változóba.

109. Meghívja a „hatizsak100\_9Approximation” függvényt a hátizsák probléma megoldásához, és a kiválasztott elemeket a „kiválasztottelemek” vektorban tárolja.

112. A „teljessuly” és a „teljesertek” változókat 0-ra inicializálja.

113. Iterál minden egyes elemet a „kiválasztottelemek” vektorban, és kiszámítja a teljes súlyt és a teljes értéket.

117. Meghívja a „writeOutputToFile” függvényt, hogy a kiválasztott elemeket, a teljes súlyt, az összértéket és az eltelt időt a „hatizsak100\_9.out” kimeneti fájlba írja.

120. A program sikeres végrehajtása esetén 0-val tér vissza.[\[K08\]](#)

Ennek a kódnak a bonyolultsága az „n” betűvel jelölt elemek számától és a kapacitástól függ. A dinamikus programozási algoritmus „n” elem és „kapacitás” érték fe-

lett iterál, ami  $O(n * kapacitas)$  időbonyolultságot eredményez. A tér összetettsége szintén  $O(n * kapacitas)$ , mivel a dinamikus programozási táblázat  $(n + 1) \times (kapacitas + 1)$  méretű. [K04]

**4.1. táblázat.** Hátizsák feladat Brute Force és Dinamikus programozási módszerrel megoldva statisztika táblázat

közelítő megoldás	BruteForce			Dinamikus		
	weight	value	idő	weight	value	idő
1	47	58	169	47	169	717
2	96	61	164	100	254	1600
3	56	67	165	100	247	1449
4	99	70	167	100	246	1466
5	82	100	182	100	263	1454
6	99	63	190	100	237	1545
7	93	46	164	100	236	1720
8	89	24	217	100	252	1654
9	51	61	166	100	275	1460
10	80	61	166	100	244	1550
Átlag n=100	79.2	61.1	175	94.7	242.3	1461.5
1	47	58	169	1000	2523	151786
2	96	61	164	1000	2593	154409
3	56	67	165	1000	2519	162422
4	99	70	167	1000	2481	157283
5	82	100	182	1000	2486	158218
6	99	63	190	1000	2516	151547
7	93	46	164	1000	2493	152242
8	89	24	217	1000	2606	153122
9	51	61	166	1000	2427	154379
10	80	61	166	1000	2461	156068
Átlag n=1000	79.2	61.1	175	1000	2510.5	155147.6
1	975	906	16381	10000	25351	14935
2	934	833	18077	10000	24839	14820
3	1000	841	16961	10000	25029	14802
4	999	980	18439	10000	24867	14759
5	976	866	16499	10000	25278	14715
6	984	958	23891	10000	25044	14697
7	998	885	17084	10000	25283	14858
8	1000	1083	17051	10000	25397	14712
9	941	854	17171	10000	24749	14778
10	994	1073	16722	10000	25448	14724
Átlag n=10000	980.1	927.9	17827.6	10000	25128.5	14780

[K08]

Átlagértéket számolva a Dinamikus programozási módszer töltötte meg a legjobban a hátizsákot és adta a legnagyobb értéket száz, ezer és tízezer tárgy esetében. Gyors

megoldást viszont a BruteForce módszer ad, viszont nem tölti meg teljesen a hátizsákot és kis értékeket generál.

## 5. fejezet

### A P osztály

A problémák P osztályához ésszerűen hatékony algoritmusok tartoznak.[\[CLR<sup>+</sup>03\]](#)

Egy feladatra akkor mondjuk, hogy polinomiálisan korlátos, ha a legrosszabb esetben a komplexitása a bemeneti méret polinomiális függvényével korlátozott, ami azt jelent, ha van olyan  $p$  polinom, hogy minden  $n$  méretű bemenet esetén az algoritmus legfeljebb  $p(n)$  lépés után véget ér. Egy problémát polinomiálisan korláatosnak nevezünk, ha létezik rá polinomiálisan korlátos algoritmus.[\[Sar78\]](#)

P a problémák azon osztálya, amelyek polinomiálisan korlátozottak.[\[Sar78\]](#)

Míg a P meghatározása túl tág ahhoz, hogy szempontot adjon a kis időigényű feladatokra, addig a P-ben való nem szereplés a túl sok időt igénylő feladatokra hasznos szempontot ad.[\[Sar78\]](#)

Egy összetett feladat megoldására vonatkozó algoritmus, több egyszerű feladatra írt algoritmus kombinálásával kapható. Az egyszerű feladatokra írt algoritmusok dolgozhatnak ugyanazon a bemeneti adatokon, vagy ezek egy része dolgozhat más algoritmusok kimeneti adatain, vagy köztes adatain. Az új, összetett algoritmus bonyolultságát összeadással, szorzással és kombinációval lehet korlátozni. Bármely algoritmus, amely több polinomkorlátos algoritmusból különböző módon épül fel, szintén polinomkorlátos lesz.[\[Sar78\]](#)

Sarah Baase könyvében a következő három okot nevezi meg, hogy miért érdemes polinomiális időbeli korlát létezését használni kritériumként:[\[Sar78\]](#)

1. Ha egy probléma nem tartozik a P-be, akkor az rendkívül költséges és a gyakorlatban valószínűleg megoldhatatlan.
2. Egy összetett problémára vonatkozó algoritmus több egyszerű problémára vonatkozó algoritmus kombinálásával kapható. Bármely algoritmus, amely több polinomkorlátos algoritmusból különböző módon épül fel, szintén polinomkorlátos lesz.
3. Az összes reális modell esetében, ha egy probléma polinomiálisan korlátos az egyik modellben, akkor a többi modellben is polinomiálisan korlátos. Ezért a P osztály invariáns (állandó, változatlan) a gyakran használt formális számítási modellek nagy halmazára.

## 6. fejezet

# Az NP osztály

Az NP az igen/nem problémák azon osztálya, amelynél egy adott bemenetre adott megoldási javaslatról gyorsan eldönthető, hogy valódi megoldás-e, kielégíti-e a probléma összes követelményét. [Sar78]

A javasolt megoldás leírható valamilyen véges halmazból származó szimbólum sorral. Az NP formális definíciójához szükséges, hogy gyorsan ellenőrizni tudjuk, hogy a karakterláncnak van-e értelme a javasolt megoldás leírásaként, valamint megfelel-e a feladat követelményeinek.

Tegyük fel, hogy kiválasztottunk egy adott szimbolumkészletet, vannak szabályaink a gráfok, halmazok, függvények leírására e szimbólumok segítségével. Az NP-t úgy definiáljuk mint a problémák azon osztályát, amelyekre olyan  $a$  és  $b$  polinomokléteznek, hogy

1. Létezik olyan algoritmus, amely legfeljebb  $a(s + n)$  lépésben meghatározza, hogy egy adott méretű szimbólumsor egy adott  $n$  méretű bemenetre ad-e megoldást, 2. Ha van egyáltalán megoldás egy  $n$  méretű bemenetre, akkor van egy legfeljebb  $b(n)$  méretű megoldás.

Egy  $n$  méretű bemenetre egy NP problémára akkor adhatunk megfelelő választ (igen vagy nem), ha minden legfeljebb  $b(n)$  hosszúságú karakterláncot megvizsgálunk. Mindegyik ellenőrzéséhez legfeljebb  $a(b(n) + n)$  lépés szükséges.

A legegyszerűbb esetben az optimalizáció egy valós függvény maximumának vagy minimumának meghatározását jelenti. A feladat egy egyszerű (de nem optimális) megoldása az, hogy az algoritmus végigpróbálgatja a megengedett halmaz elemeit, mindegyikhez kiszámítva a függvény értékeit. Általánosabban, az optimalizáció egy adott megengedett tartományon keresi egy függvény legjobb értékét, ahol mind a megengedett tartomány, mind a függvény sok különböző típusba tartozhat. [CLR<sup>+</sup>03]

### 6.1. NP-Teljes problémák

NP-teljesnek nevezzük azokat a problémákat, amelyek NP-ben a legnehezebbek abban az értelemben, hogy ha egy NP teljes problémára lenne egy polinomiális korlátos algoritmus, akkor minden egyes problémára lenne egy polinomiális korlátos algoritmus. Formálisabban, egy P probléma a NP-ben NP-teljes, ha adott egy  $A_P$  algoritmus a P-re, találunk egy olyan algoritmust az NP bármely más problémájára, amely polinomiális korlátot ad az általa végrehajtott utasítások számának, beleértve az  $A_P$ -hoz tartozó al-

programok vagy eljárások hívását, ha van ilyen, de nem beleértve az  $A_P$  által végzett munkát.

## 6.2. CNF kielégíthetőség

Logikai műveletek alatt az egyértelműen igaz vagy hamis (jelentő mondatok) kijelentések ítéletein definiált műveleteket értünk, amelyek alapján az ítéletekből újabb, összetett ítéleteket alkothatunk. Az így alkotott összetett ítéletek igazságértéke pedig egyértelműen meghatározható a kiindulási ítéletek igazságértékeiből. [Sar78]

Logikai művelet: logikai függvény, igazságfüggvény, logikai operátorok.

A formális nyelv a matematika, a logika és az informatika számára egy véges ábécéből (jelentése: tetszőleges, azonban meghatározott jelek halmaza; általában véges halmaz) generálható, véges hosszúságú szavak (például karakterek sorozata, jelsorozat) halmaza, amelyekkel a formális nyelvek elmélete foglalkozik. Legyen  $A = \{ a_1, a_2, \dots, a_n \}$  véges halmaz, amit ábécének nevezünk. Készítsünk  $A$  elemeiből véges sorozatokat minden lehetséges módon.

A legáltalánosabban használt logikai műveletek: a negáció, a konjunkció, a diszjunkció, az implikáció és az ekvivalencia.

A negáció egy állítás igazságértékét az ellenkezőjére váltja.

Konjunkció (logikai és) alatt egy olyan két változós logikai műveletet értünk, ami akkor és csak akkor igaz, ha mind a két változó logikai értéke igaz.

Diszjunkció (logikai vagy) alatt egy olyan két változós logikai műveletet értünk, ami akkor és csak akkor hamis, ha mind a két változó logikai értéke hamis.

Az implikáció, kondicionálás, vagy szubjunkció logikai művelet, használjuk a matematikai logikában, informatikában. Két állítást kapcsol össze, és jelentése a ha, akkor nyelvi kifejezéshez áll közel. Példa: ha szárazság van, akkor a föld összeroppedezik.

Ekvivalencia (bikondicionális): az akkor és csak akkor kifejezés egy természetes nyelvi, logikai természetű viszony (reláció) elnevezése a logikai grammatikában. Két tagmondat felhasználásával olyan összetett mondatot képezzünk, hogy mindkettő ugyanazon körülmények között tekinthető igaznak és hamisnak. Például: „A lakás bérbeadása akkor és csak akkor tekinthető törvényesnek, ha formailag megfelelő szerződés szól róla.” „Egy polinomnak az a szám akkor és csak akkor gyöke, ha a polinomfüggvénynek zérushelye.”

A konjunktív normálforma (KNF) a matematikai logika egy területén, a nulladrendű logikán belül definiálható fogalom, egy logikai műveletet leíró olyan ítéletloikai formulát jelent, mely a művelet változóinak vagy negáltjainak diszjunkcióinak konjunkciója.

A konjunktív normálformák olyan nulladrendű logikai formulák, melyekben csak változók,  $\neg$  negáció,  $\wedge$  konjunkció,  $\vee$  diszjunkció fordulnak elő.

Matematikai definíció: ha a logikai művelet az  $X_1, X_2, \dots, X_n$  változókon van értelmezve, akkor egy konjunktív normálformája lehet például:  $(\neg X_1 \vee X_2) \wedge (\neg X_2 \vee X_3) \wedge \dots \wedge (\neg X_{n-1} \vee X_n)$

A CNF kielégíthetőségi probléma annak meghatározása, hogy létezik-e igazság hozzárendelés, van-e mód arra, hogy a kifejezésben szereplő változókhoz az IGAZ és a HAMIS értékeket hozzárendeljük úgy, hogy a kifejezés értéke IGAZ legyen.

Ez a probléma a számítógépes tételbizonyításban is alkalmazható.



## 7. fejezet

# Közelítő algoritmusok

Az olyan algoritmust, amely az optimálishoz közeli megoldást ad közelítő algoritmusnak hívjuk.

$k(n)$  egy közelítő algoritmusnak az adott feladatra vonatkozó hibakorlát - függvénye, ha a közelítő algoritmus által előállított  $C$  költsége - minden  $m$  méretű bemenetre - az optimális megoldás  $C^*$  költségének legfeljebb  $k(n)$  szerese és legfeljebb  $k(n)$  - ed része  $\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq k(n)$

Ha az algoritmus biztosítja a  $k(n)$  hibakorlát függvény betartását,  $k(n)$  közelítő algoritmusnak nevezzük.

Maximalizálási feladatnál  $0 < C \leq C^*$  és a  $\frac{C^*}{C} > 1$  hányados megadja azt a tényezőt, ahányszor az optimális megoldás költsége nagyobb, mint a közelítő megoldás költsége.[Sar78]

Minimalizálási feladatnál  $0 \leq C^* < C$  és a  $\frac{C}{C^*} < 1$  hányados megadja azt a tényezőt, ahányszor a közelítő megoldás költsége nagyobb, mint az optimális megoldás költsége.[Sar78]

Ha a közelítő algoritmus hibakorlátja egy optimális megoldást állít elő. Ha a közelítő algoritmusnak nagy hibakorlátot adunk, akkor az optimálisnál sokkal rosszabb megoldást is eredményezhet.

NP teljes feladatok egy részére ismertek olyan polinomiális futási idejű algoritmusok, amelyek a futási idő növelése árán egyre kisebb hibakorlátot biztosítanak. Ebből következik, hogy szoros összefüggés van a futási idő és a közelítés minősége között.[Sar78]

### 7.1. Minimális lefedő csúcshalmaz

Minimális lefedő csúcshalmaz feladat egy irányítatlan gráfban megtalálni a minimális lefedést, amit optimális lefedésnek hívunk.

alsó korlát  $|C^*| \geq |A|$  és felső korlát  $|C| = 2 |A| \}$   $|C| = 2 |A| \leq 2 |C^*|$

A közelítő lefedés olyan megoldást talált, melynek mérete legfeljebb a duplája az optimális lefedés méretének.[Ant07]

A maximális párosítás, mivel ez az eset a legnagyobb, így egyetlen más párosításnak sem lehet valódi részhalmaza.

## 7.2. Az utazóügynök feladat

Az utazóügynök feladat egy teljes irányítatlan gráfban, minden élhez költséget rendelünk és meg kell találni egy minimális költségű Hamilton kört.

```
struct City {
    int x, y;

    double tavolsagTo(const City& other) const {
        int dx = x - other.x;
        int dy = y - other.y;
        return sqrt(dx * dx + dy * dy);
    }
};

double tavolsagKiszamitasa(const vector<City>& path) {
    double teljesTavolsag = 0.0;
    for (size_t i = 0; i < path.size() - 1; ++i) {
        teljesTavolsag += path[i].tavolsagTo(path[i + 1]);
    }
    return teljesTavolsag;
}

vector<City> megtalalniAlegrovidebbUtat(const vector<City>& cities) {
    vector<City> legrovidebbUt;
    double legrovidebbTavolsag = infinity;

    vector<size_t> indices(cities.size());
    iota(indices.begin(), indices.end(), 0);

    do {
        vector<City> path;
        for (size_t i = 0; i < cities.size(); ++i) {
            path.push_back(cities[indices[i]]);
        }

        double tavolsag = tavolsagKiszamitasa(path);
        if (tavolsag < legrovidebbTavolsag) {
            legrovidebbTavolsag = tavolsag;
            legrovidebbUt = move(path);
        }
    } while (next_permutation(indices.begin(), indices.end()));

    return legrovidebbUt;
}

int main() {
    ifstream inputFile("tsp.in");
    if (!inputFile) {
```

```

        print "Failed to open input file."
        return 1;
    }

    int numCities;
    inputFile >> numCities;

    vector<City> cities(numCities);
    for (int i = 0; i < numCities; ++i) {
        inputFile >> cities[i].x >> cities[i].y;
    }

    inputFile.close();

    auto startTime = get_current_time();

    vector<City> legrovidebbUt = megtalalniAlegrovidebbUtat(cities);

    auto endTime = get_current_time();
    auto elapsedTime = calculate_elapsed_time(startTime, endTime);

    ofstream outputFile("tsp.out");
    if (!outputFile) {
        print "Nem sikerult megnyitni a kimeneti fajlt."
        return 1;
    }

    for (const City& city : legrovidebbUt) {
        outputFile << "(" << city.x << ", " << city.y << ")" << endl;
    }

    outputFile << "Total tavolsag: " << tavolsagKiszamitasa(legrovidebbUt) <<
        endl;
    outputFile << "Eltelt ido (mikroszekundum): " << elapsedTime << endl;

    outputFile.close();

    return 0;
}

```

[?]

### 7.3. Az utazóügynök feladat háromszög - egyenlőtlenséggel

Első lépésben meghatározunk egy minimális feszítőfát.

Második lépésként előállítunk egy körutat, figyelve, hogy a súlyok kielégítsék a háromszög egyenlőtlenséget az első lépésben készített minimális feszítőfából.

Az utazó ügynök feladatra a következő Christofides algoritmust szerkesztettem:

```

fuggveny calculate_tavolsag(p1, p2):
    dx = p1.x - p2.x
    dy = p1.y - p2.y
    return sqrt(dx * dx + dy * dy)

class Graph:
    fuggveny __init__(vertices):
        self.V = vertices
        self.points = []
        self.tavolsags = 2D array of size V x V

    fuggveny add_point(p):
        add p to self.points

    fuggveny calculate_tavolsags():
        for i = 0 to V-1:
            for j = i + 1 to V-1:
                tavolsag = calculate_tavolsag(points[i], points[j])
                set tavolsags[i][j] = tavolsag
                set tavolsags[j][i] = tavolsag

fuggveny print_tura(tura):
    print "tura: ",
    for i in tura:
        print i,
    print

fuggveny kiszamitja_tura_hosszat(tura, graph):
    length = 0.0
    n = graph.V
    for i = 0 to n-1:
        u = tura[i]
        v = tura[(i + 1) mod n]
        length = length + graph.tavolsags[u][v]
    return length

fuggveny minimum_megfeleles_keresese(graph, suly_matrix):
    n = graph.V
    visited = array of size n, initialized with False
    hozzaallo = array of size n, initialized with -1

    for u = 0 to n-1:
        if visited[u]:
            continue

```

```

        min_suly = infinity
        min_vertex = -1
        for v = 0 to n-1:
            if u == v or visited[v]:
                continue
            if suly_matrix[u][v] < min_suly:
                min_suly = suly_matrix[u][v]
                min_vertex = v
        hozzaIllo[u] = min_vertex
        hozzaIllo[min_vertex] = u
        visited[u] = True
        visited[min_vertex] = True

    return hozzaIllo

fuggveny eulerian_tura_megtalalasa(v, graph, suly_matrix, tura):
    n = graph.V
    for u = 0 to n-1:
        if suly_matrix[v][u] > 0:
            suly_matrix[v][u] = 0
            suly_matrix[u][v] = 0
            eulerian_tura_megtalalasa(u, graph, suly_matrix, tura)
    tura.append(v)

fuggveny hamiltonian_tura_tura_megtalalasa(graph, hozzaIllo, eulerian_tura):
    n = graph.V
    visited = array of size n, initialized with False
    tura = empty array
    for v in eulerian_tura:
        if not visited[v]:
            tura.append(v)
            visited[v] = True
    for v in hozzaIllo:
        if not visited[v]:
            tura.append(v)
            visited[v] = True
    return tura

fuggveny christofides(graph):
    n = graph.V

    # Step 1: Szamitsa ki a minimalis feszitofat
    suly_matrix = graph.tavolsags
    parent = array of size n, initialized with -1
    key = array of size n, initialized with infinity
    in_mst = array of size n, initialized with False

```

```

key[0] = 0
for count = 0 to n-2:
    min_suly = infinity
    min_vertex = -1

    for v = 0 to n-1:
        if not in_mst[v] and key[v] < min_suly:
            min_suly = key[v]
            min_vertex = v

    in_mst[min_vertex] = True

    for v = 0 to n-1 Ime a pszeudokod folytatasa for the Christofides
        algoritmus:
            if not in_mst[v] and suly_matrix[min_vertex][v] < key[v]:
                parent[v] = min_vertex
                key[v] = suly_matrix[min_vertex][v]

# Step 2: Letrehozás a minimum-suly perfect hozzáIllo
hozzaIllo = minimum_megfeleles_keresese(graph, suly_matrix)

# Step 3: Letrehozás an Eulerian tura
eulerian_tura = []
eulerian_tura_megtalalasa(0, graph, suly_matrix, eulerian_tura)

# Step 4: Letrehozás a Hamiltonian tura
tura = hamiltonian_tura_tura_megtalalasa(graph, hozzáIllo, eulerian_tura)

return tura
Example usage
graph = Graph(num_vertices)

Add points to the graph using graph.add_point(p)
graph.calculate_tavolsags()
tura = christofides(graph)
print_tura(tura)

```

[CAA03]

Az utazó ügynök feladatra Christofides algoritmus program bemutatása:[K83]

A kód tartalmazza a szükséges könyvtárakat a bemenethez/kimenethez, a vektorkezeléshez, a rendezéshez, a fájlműveletekhez, a matematikai függvényekhez és a numerikus korlátokhoz. Egy állandó INF-et határoz meg, amely a végtelent reprezentálja az int adattípus maximális értékével. Meghatároz egy Pont nevű struktúrát, amely egy két dimenziós pontot ábrázol x és y koordinátákkal. Definiál egy számítási távolság függvényt, amely két p1 és p2 pont euklideszi távolságát számítja ki. Meghatároz egy Graph nevű struktúrát egy V csúcsú gráf reprezentálására. Vektorokat tartalmaz a pontok, a pontok közötti távolságok és a grafikonnal kapcsolatos egyéb információk tárolására. AdPoint

függvényt határoz meg a Graph struktúrában, hogy pontot adjon a gráf pontvektorához. A Graph struktúrában meghatároz egy számítási távolságok függvényt az összes pontpár közötti távolság kiszámításához és a távolságvektorban való tárolásához. Meghatároz egy printtura függvényt, amely kiírja a körút csúcsait. Definiál egy kalkulációturaLength függvényt, amely a grafikon csúcsai közötti távolságok alapján kiszámítja a körút teljes hosszát. Meghatároz egy minimalisIllesztésKeresése függvényt, amely egy súlymátrix segítségével megtalálja a minimális súlyú illeszkedést a gráf minden csúcsához, és az illesztési információt az illesztési vektorban tárolja. Meghatároz egy eulerianUtvonalKereses függvényt, amely egy súlymátrix segítségével megtalálja a gráf v csúcsától kiinduló Euler-körutat, és eltárolja a körutat a körútvektorban. Meghatároz egy findHamilton-túra függvényt, amely az Euleri-körutat a minimális súlyú illesztéssel kombinálja, hogy egy Hamilton-túrát képezzen a grafikonon. Egy christofides függvényt határoz meg, amely a Christofides algoritmus segítségével megoldja az utazó kereskedő problémát (TSP). Több lépést is végrehajt az optimális túra megtalálásához. Megnyit egy "tsp.in" nevű bemeneti fájlt olvasásra. Megnyit egy "tsp.out" nevű kimeneti fájlt íráshoz. Beolvassa az n csúcsok számát a bemeneti fájlból. Egy n csúcsú gráfobjektum gráfot hoz létre. Beolvassa az egyes pontok koordinátáit a bemeneti fájlból, és hozzáadja a gráfhoz. Kiszámítja a távolságot a grafikonon szereplő összes pontpár között. Elkezdi mérni a végrehajtási időt. Meghívja a christofides függvényt, hogy megoldja a TSP-t és megkapja az optimális körutat. Leállítja a végrehajtási idő mérését. Kiszámítja az eltelt időt mikroszekundumban. Kinyomtatja a körutat a konzolra. Kiszámítja a túra hosszát. A körút hosszát és az eltelt időt megjeleníti a konzolon.

A kód összetettsége a következőképpen elemezhető: A CalculatedTavolsag függvény időbonyolultsága  $O(1)$ , mivel meghatározott számú műveletet hajt végre. A CalculatedTavolsags függvény időbonyolultsága  $O(n^2)$ , mivel kiszámítja a távolságot a gráf összes pontpárja között. A minimalisIllesztésKeresese függvény időbonyolultsága  $O(n^2)$ , mivel minden csúcspáron iterál, hogy megtalálja a minimális súlyú illesztést. A eulerianUtvonalKereses függvény időbonyolultsága  $O(n^2)$ , mivel bejárhatja a gráf összes élét. A hamiltonianUtvonalKereses függvény időbonyolultsága  $O(n)$ , mivel a körút és az egyeztetés csúcsai felett iterál. A christofides függvény időbonyolultsága  $O(n^3)$ , mivel több lépést hajt végre beágyazott hurkokkal, de a domináns tényező a minimális feszítőfa számítás. A kód általános összetettsége az n bemeneti mérettől függ. A domináns tényező a christofides függvény, amelynek összetettsége  $O(n^3)$ . Más funkciók és műveletek kisebb hatással vannak az általános komplexitásra.

**7.1. táblázat.** Utazó ügynök feladat Brute Force és Christofides programozási módszerrel megoldva statisztika táblázat

közelítő megoldás / példák száma	TSP BruteForce		Christofides	
	útvonal hossza	idő	útvonal hossza	idő
1	22.35485233	16672363	58.41858134	0
2	28.83245264	16562968	59.98544312	0
3	19.51166563	16453570	48.61561537	0
4	22.60181429	16719725	74.1343885	0
5	26.26528796	16861337	54.02700924	0
6	28.39834564	16812997	59.70635132	0
7	24.62704349	16859849	70.68855066	0
8	27.77829856	16687964	59.16312296	0
9	23.58594637	16609818	46.75087839	0
10	28.01773923	17109818	49.67136908	0
Átlag n=100	25.19734461	16735040.9	58.116131	0

Átlagértéket számolva a Christofides programozási módszer adta meg gyorsabban az útvonalat 10 város esetében.



## 8. fejezet

# A rendszer specifikációja

### 8.1. Felhasználói követelmények

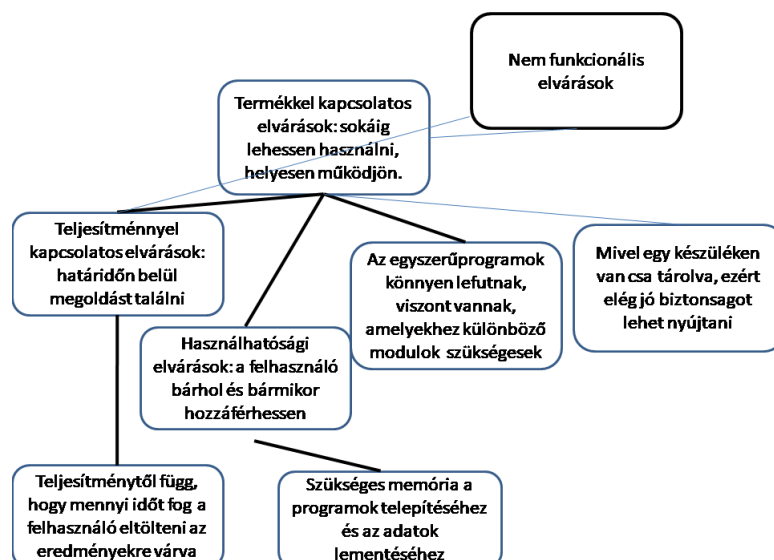
A felhasználó meg akarja tudni, hogy mik azok az NP-teljes feladatok és szeretne egy-egy példát látni rájuk.

A felhasználó el akarja készíteni a teszteket, hogy meggyőződjön, hogy valóban hatékonyak az ajánlott algoritmusok.

### 8.2. Rendszerkövetelmények

Windows alapú operációs rendszer, amelyen futnak a CodeBlocks, Visual Studio Code, python programok.

### 8.3. Nem funkcionális elvárások



8.1. ábra. Nem funkcionális elvarasok

## 9. fejezet

# Tervezés

Logikai nézet szempontjából az elkészített programok eléggé egyszerűnek és célrátórónak tűnhetnek, mivel ezek célirányosan az NP-teljes feladatok megoldására összpontosítanak.

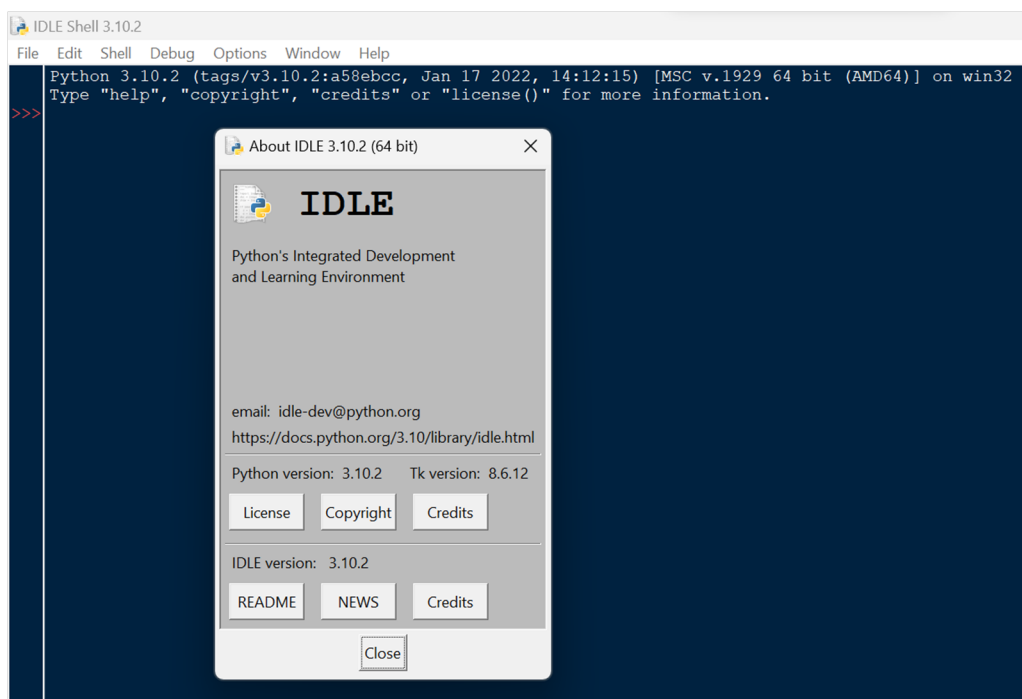
Folyamat nézet szempontjából a rendszer beolvassa fájlból az adatokat, feldolgozza, valamilyen kevésbé hatékony, vagy hatékony módszerrel (a felhasználó döntésétől függ, hogy melyik módszerrel fog neki megoldani a feladatot). Majd kiírja az eredményt, ami lehet jó, vagy lehet kevésbé jó.

## 10. fejezet

# Kivitelezés

A megtervezett alkotás értékelése és kritikai elemzése.

A projekt megvalósítása során sok időbe telt, amíg megértettem, hogy az egész disszertációs dolgozat lényege, célja. Ez motivált arra, hogy egy olyan barátságos felhasználói felületet készítsek (User Interface), amely segítségével bárki könnyen felfedezheti, megtapasztalhatja az NP-teljes feladatok lényegét. Az egyes módszereket vizuálisan, a képernyőre kirajzolva jeleníti meg ez az egyszerű kis alkalmazás. Nincs nagy tárhely igénye. Bármely olyan készüléken fut, ahova fel van telepítve a python 3.10-es változata. Fontos megjegyezni, hogy a python 3,9-es verzióján már nem fut ez az alkalmazás. Sokáig kerestem az okát, hogy tudjam minél pontosabban meghatározni a probléma forrását, viszont sajnos egyetlen válasz, amit találtam az a különböző python verziók közötti különbségekben található meg.



10.1. ábra. Python IDLE 3.10.2

[TIS04]

A fenti ábra mutatja be azt a fejlesztői környezetet, ahol a jelenlegi dolgozathoz tartozó alkalmazás elkészült.

A következő modulokat szükséges a Python program mellé telepíteni azért, hogy megfelelően működjön a programunk.

`tkinter`— Python interfész a Tcl/Tk-hoz A `tkinter` csomag („Tk interfész”) a szabvány Python interfész a Tcl/Tk GUI eszközkészlethez. Mind a Tk, mind `tkinter` a legtöbb Unix platformon elérhető, beleértve a macOS-t, valamint a Windows rendszereken is.

`time` A `time` Python modulja funkciókat biztosít az időhöz kapcsolódó feladatok kezeléséhez. Az időhöz kapcsolódó feladatok közé tartozik, az aktuális idő leolvasása, formázási idő, meghatározott számú másodpercig alszik és így tovább.

`sys` Ez a modul hozzáférést biztosít néhány, az értelmező által használt vagy karbantartott változóhoz, valamint olyan funkciókhoz, amelyek erősen kölcsönhatásba lépnek az értelmezővel. Mindig elérhető.

`subprocess` A `subprocess` modul lehetővé teszi új folyamatok létrehozását, a bemeneti/kimeneti/hibacsövekhez való csatlakozást és a visszatérési kódok beszerzését.

Fontos megjegyezni, hogy a programok hibaelhárításában nagy segítségemre volt a chatGPT, aki folyamatosan segített megkeresni a hibákat és működőképessé tenni a megszerkesztett programot.

Minden komponensre egy alfejezetet kell szánni, amelyben részletesen leírjuk a komponens belső működését, pl. osztálydiagramok segítségével. Ha komponens más komponensekkel is kollaborál (igénybe veszi egy másik komponens publikus interface-ét vagy egy külső libraryt, függőséget/dependenciát használ), akkor ezt egy aktivitás-, vagy szekvencia diagrammal kell szemléltetni: pl. ki indítja a kommunikációt, mennyit vár, stb.

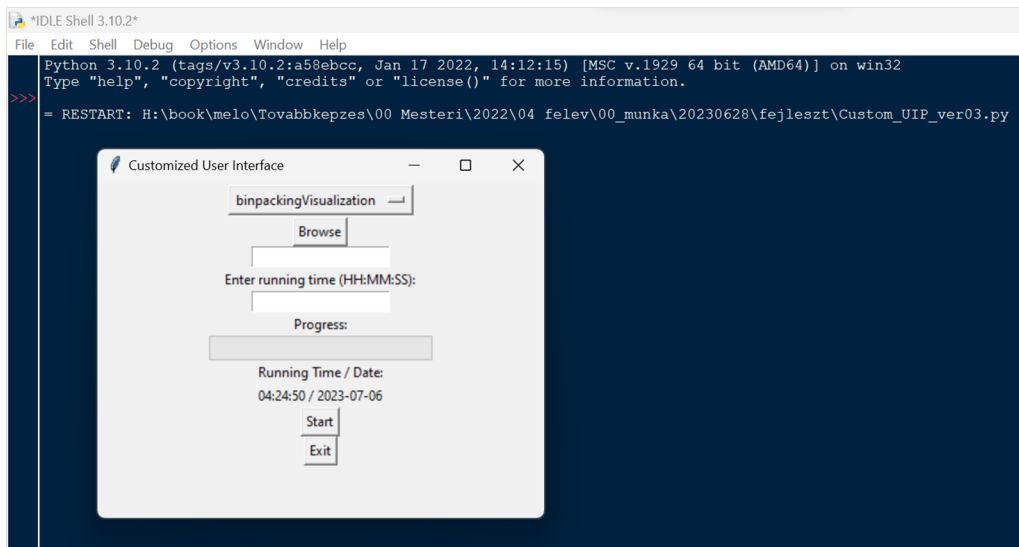
Ezt a programot elsőrörban vizualizációs céllal készítettem. Azt tapasztaltam, hogy képek segítségével sokkal könnyebb megérteni az új, elvont dolgokat. Főleg olyan esetben nagyon hasznosak, ha valami újjal kell találkozni és ha látod, akkor egyből megvilágosodik, hogy jajj tényleg, erről van szó és azután már sokkal könnyebben megérted a működését a dolognak.

## 10.1. Bin packing vizuális ábrázolása

Ehhez a komponenshez szükséges további modulokat telepíteni és használni. A következő modulokat szükséges a Python program mellé telepíteni azért, hogy megfelelően működjön a programunk.

`matplotlib` A `Matplotlib` a Python programozási nyelv és annak `NumPy` numerikus matematikai kiterjesztésének ábrázoló könyvtára. Objektum-orientált API-t biztosít a telkek alkalmazásokba való beágyazásához olyan általános célú grafikus felhasználói felület eszközkészletek használatával, mint a `Tkinter`, `wxPython`, `Qt` vagy `GTK`.

`pyplot` `matplotlib.pyplot` olyan függvények gyűjteménye, amelyek a `matplotlib`-et a `MATLAB`-hoz hasonló működésre készítetik. Mindegyik `pyplot` függvény módosít egy ábrát: pl. létrehoz egy ábrát, hoz létre egy ábrázolási területet az ábrán, kirajzol néhány vonalat egy nyomtatási területen, díszíti a telket címkékkel stb.

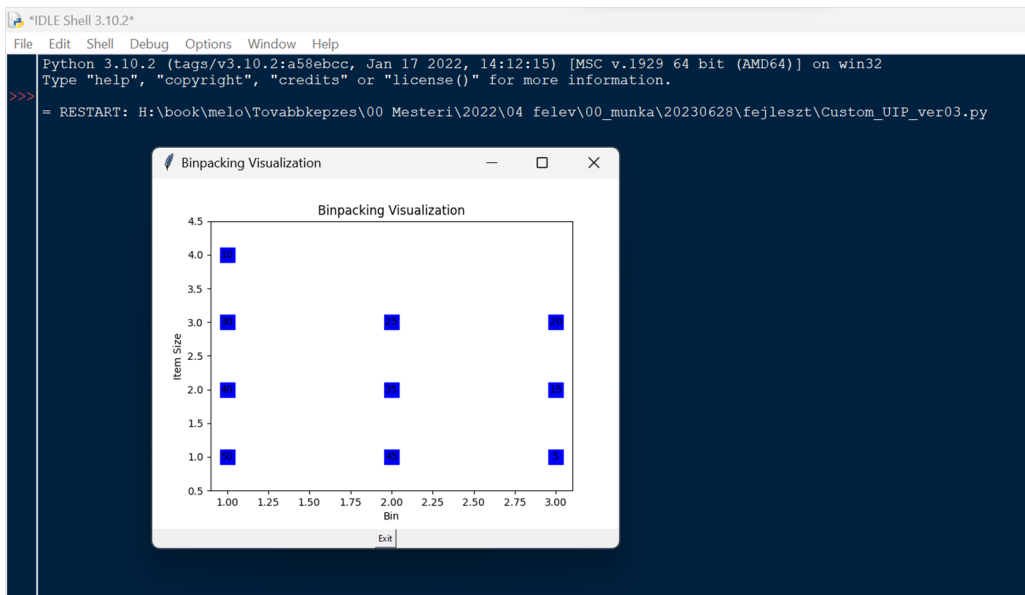


**10.2. ábra.** Python IDLE 3.10.2

A fenti ábrán látható User Interface fogad minket a program indítása után. A programot több módon is elindíthatjuk. Beléphetünk a Python IDLE-be, ahonnan a File - Open utasítással megjelenik egy párbeszédablak, amely segítségével megkereshetjük, hogy melyik mappába mentettük le az elkészített alkalmazásunkat és megnyithatjuk. Miután betöltődik, ajánlott leellenőrizni, hogy minden rendben, nincsenek túl nagy bemeneti adatok megadva. A program futtatása az F5-ös gombbal, vagy a főmenüből, a Run - Run module gombra kattintva indul.

A képen látható párbeszédablak jelenik meg, ahol kiválaszthatjuk, hogy mely algoritmust szeretnénk tesztelni. Közben megfigyelhetjük, hogy a párbeszédablak alsó részén megjelenik a dátum és a pontos idő egyaránt.

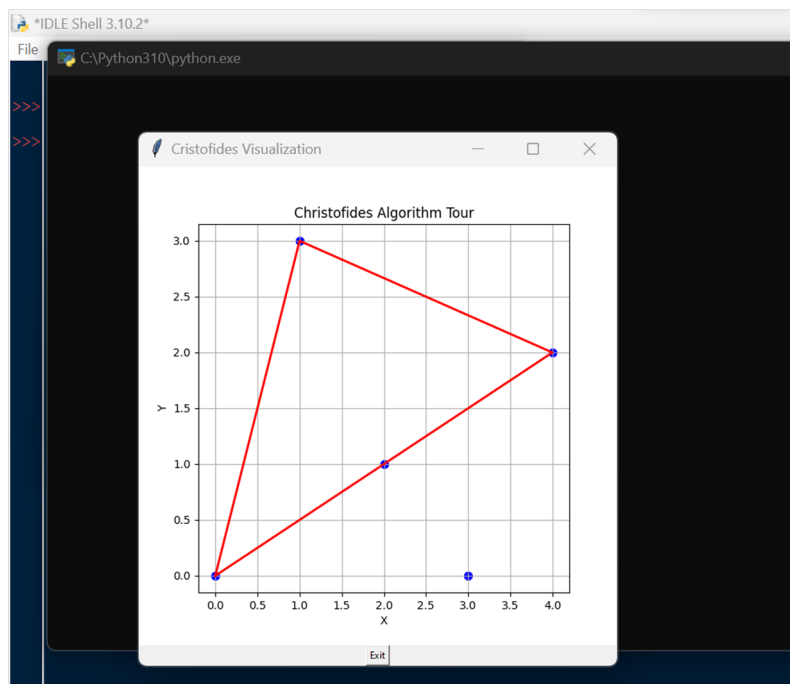
Miután kiválasztottuk a megfelelő algoritmus nevét, a Start gombra kattintva tudjuk kipróbálni annak működését. Néhány másodperc után kirajzolódik eléünk a Bin Packing feladat vizuális ábrázolása.



10.3. ábra. Bin Packing feladat vizuális ábrázolása

## 10.2. Christofides algoritmus vizuális ábrázolása

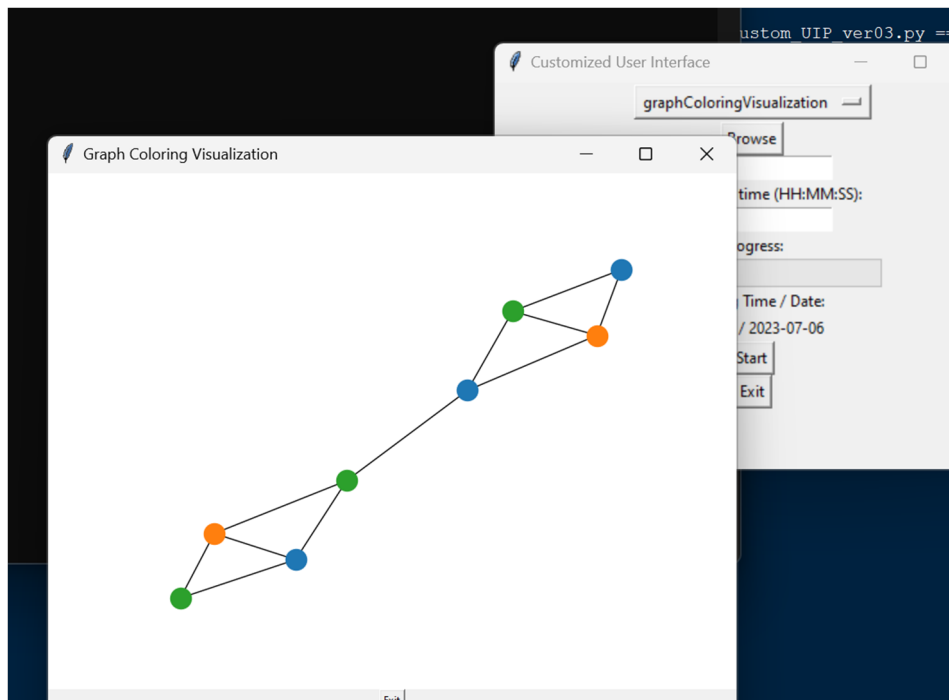
Hasonló módon működik a Christofides algoritmus vizuális ábrázolása.



10.4. ábra. Christofides algoritmus vizuális ábrázolása

## 10.3. Gráf színezés vizuális ábrázolása

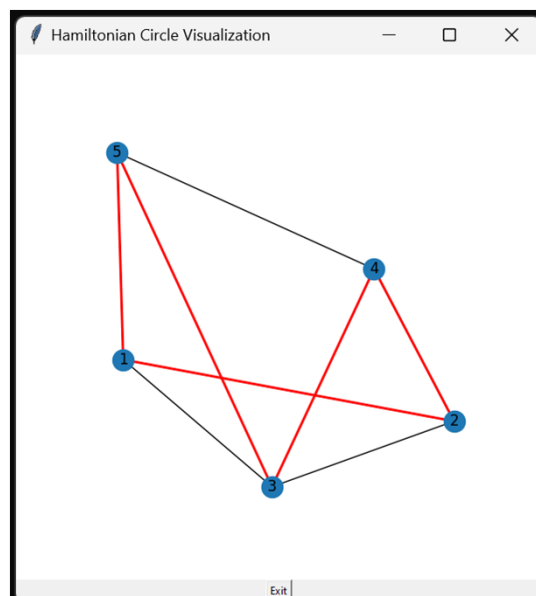
A következő minta példa a gráf színezés vizuális ábrázolása,



10.5. ábra. gráf színezés vizuális ábrázolása

Szerintem ennél szemléletesebben nem lehet bemutatni ezen Np-teljes feladatok működését, használati módját.

#### 10.4. Hamiltoni körnek a vizuális ábrázolása



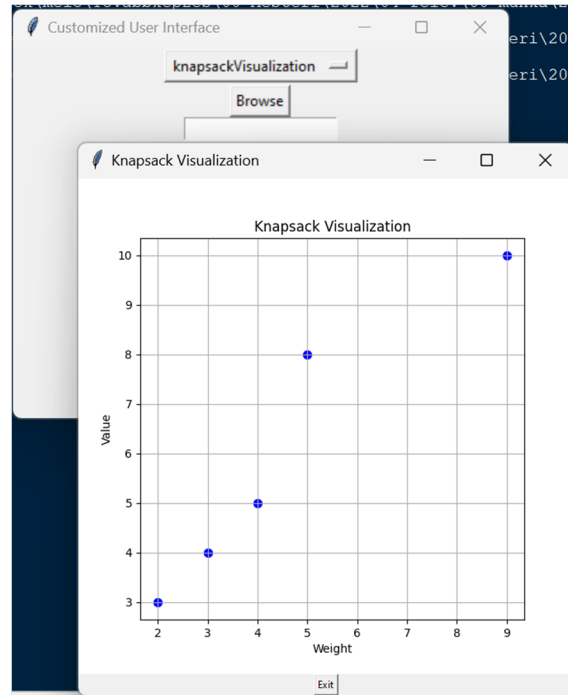
10.6. ábra. Hamiltoni körnek a vizuális ábrázolása

[LS15]

A következő ábra, ami elénk tárul az a Hamiltoni körnek a vizuális ábrázolása. Így játszva, szemléletesen meg lehet tanítani, akár egy kisiskolásnak is ezeket a dolgokat.

## 10.5. Hátizsák feladat vizuális ábrázolása

A hátizsák feladatnak a vizuális ábrázolása sajnos nem annyira látványos, de azért működik.

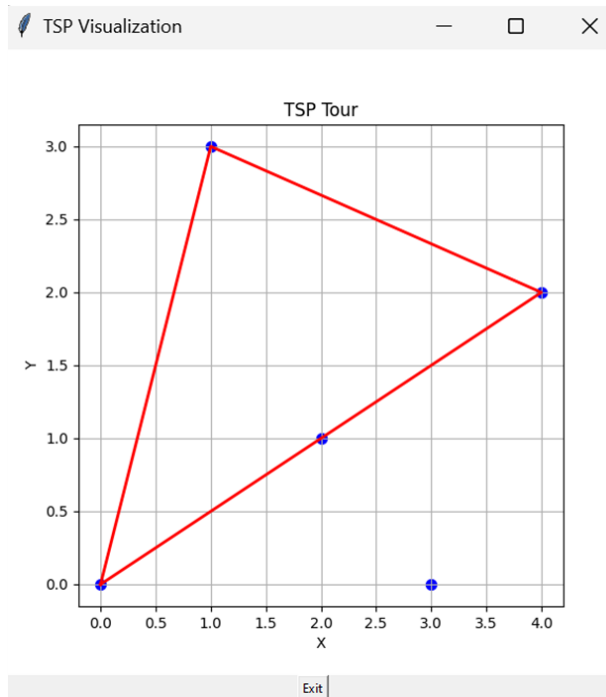


10.7. ábra. Hátizsák feladat vizuális ábrázolása

## 10.6. TSP körút vizuális ábrázolása

A következő ábra, amit megtekinthetünk az a TSP körút vizuális ábrázolása.



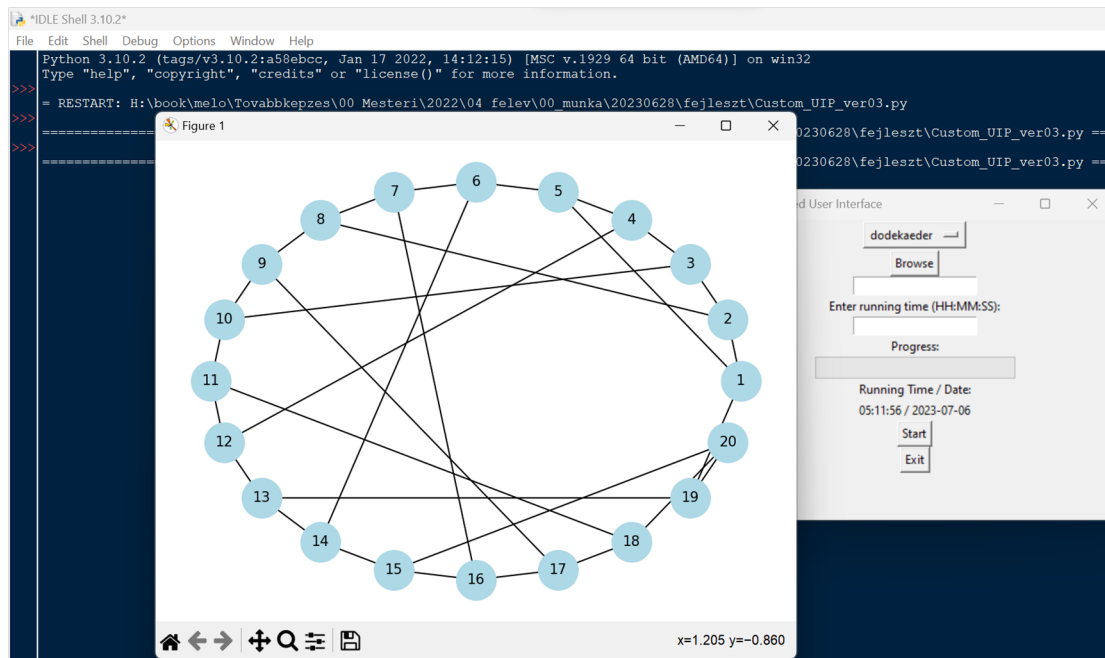


**10.8. ábra.** TSP körút vizuális ábrázolása

Egy másik fontos összetevő, amiről még nem beszéltünk a NumPy modul. NumPy a Python programozási nyelv könyvtára, amely támogatja a nagy, többdimenziós tömböket és mátrixokat, valamint a magas szintű matematikai függvények nagy gyűjteményét az ezeken a tömbökön való működéshez. Erre is nagy szükség van a programunk működéséhez.

## 10.7. Dodekaéder vizuális ábrázolása

A következő érdekesség, amit megtekinthetünk az a Dodekaéder vizuális ábrázolása, melyet 12 szabályos ötszög alkot.



10.9. ábra. Dodekaéder vizuális ábrázolása

Szerintem, ha ilyen szép, igényes és modern eszközök használatával ezeket a fogalmakat bárkinek játszva el lehet magyarázni és meg lehet tanítani ezek használatát.

Algoritmus neve	Alsó határ	Felső határ	Megjegyzés
01_binpacking_kozelito_dff	$\Omega(n \log n)$	$O(n^2)$	
01_binpacking_kozelito_randomSelect	$\Omega(n)$	$O(n^2)$	
01_binpacking_pontos_randomSelect	$\Omega(n)$	$O(n^2)$	
01_binpacking_rogzitettPermutaciosMinta	$\Omega(\text{elemekSzama})$	$O(\text{elemekSzama} * \text{Maximalis\_Bins})$	
01_binpacking_rogzitettPermutaciosMinta_fileMuveletekkel	Az alsó határ akkor lép fel, ha ezek az értékek a minimumon vannak (pl. "permSzam = 1", "n = 1", "binsSam = 1")	a felső határ pedig akkor, amikor ezek az értékek a maximumon vannak (pl. "permSzam = Maximalis_Elemszam!", "n = Maximalis_Elemszam", "binsSzam = Maximalis_Bins")	Megjegyzés: A faktoriális művelet (!) a 'Maximalis_Elemszam!' -ban a permutációk számát jelzi, így számítása nagy értékek esetén számításigényessé válhat.
02_graphColoring_backtracking	exponenciális	exponenciális	
02_graphColoring_branchAndBound	exponenciális	exponenciális	
03_DinamikusProgramozással_Knapsack	$\Omega(n * \text{kapacitás})$	$O(n * \text{kapacitás})$	
03_SarahBaase_BruteForce_Knapsack	$\Omega(2^n)$	$O(2^n)$	exponenciális jellegű
04_TSP_BruteForce	$\Omega(n!)$	$O(n!)$	
04_TSP_Christofides	$\Omega(n^2)$	$O(n^2)$	

10.10. ábra. az elkészített algoritmusok bonyolultsága

# 11. fejezet

## Mérések

A Bin Packing, láda pakolással kapcsolatosan, négy különböző mérést végeztünk el.

FF fast forward, ami sorba vette a bemeneti fájlokban lévő adatokat és azokat próbálta minél kevesebb ládába bepakolni. [KA]

Az RFF random first fit, véletlenszerű első illesztés, ami véletlenszerűen választott ki egy-egy ládát és így próbálta minél kevesebb ládába betenni a csomagokat.

A DFF decreasing first fit, csökkenő első illesztés, ami előbb csökkenő sorrendbe rendezte a csomagokat és azután próbálta a csomagokat a ládába bepakolni.

Az IFF increasing first fit, növekvő első illesztés, ez a módszer előbb növekvő sorrendbe helyezte a csomagokat és azután próbálta minél kevesebb ládába bepakolni azokat.

### 11.1. Bin Packing, láda pakolással kapcsolatosan, n=100 elem, közelítő megoldás esetén

- Futási idők **átlaga**: 71.0206 sec
- Futási idők **minimuma**: 0.0822 sec
- Futási idők **maximuma**: 98.2000 sec

Átlagértéket számolva a DFF a legjobb eredményt adta 100-as nagyságrendű ládaszám esetén.

### 11.2. Bin Packing, láda pakolással kapcsolatosan, n=1000 elem, közelítő megoldás esetén

- Futási idők **átlaga**: 711.7947 sec
- Futási idők **minimuma**: 0.8789 sec
- Futási idők **maximuma**: 1032.9000 sec

Átlagértéket számolva az RFF a legjobb eredményt adta 1000-es nagyságrendű ládaszám esetén, viszont még mindig a DFF a leggyorsabban ad eredményt.

### 11.3. Bin Packing, láda pakolással kapcsolatosan, $n=10000$ elem, közelítő megoldás esetén

- Futási idők **átlaga**: 6367.3064 sec
- Futási idők **minimuma**: 8.1256 sec
- Futási idők **maximuma**: 9416.1000 sec

Átlagértéket számolva az RFF a legjobb eredményt adta 10000-es nagyságrendű ládaszám esetén.

A first fit algoritmus előnye, hogy nagyon gyorsan talál egy megoldást.

Hátránya, hogy nem minden esetben talál minden elemnek egy helyet, még akkor sem, ha ez lehetséges lenne.

Ezért jobb a best fit algoritmus.

### 11.4. Gráfszínezés backtracking és branch and bound módszerekkel, $n=10$ elem esetén

- Futási idők **átlaga**: 102.65 sec
- Futási idők **minimuma**: 0.00 sec
- Futási idők **maximuma**: 1068.00 sec

### 11.5. Gráfszínezés backtracking és branch and bound módszerekkel, $n=100$ elem esetén

- Futási idők **átlaga**: 399.70 sec
- Futási idők **minimuma**: 0.00 sec
- Futási idők **maximuma**: 2580.00 sec

### 11.6. Gráfszínezés backtracking és branch and bound módszerekkel, $n=1000$ elem esetén

- Futási idők **átlaga**: 115824.10 sec
- Futási idők **minimuma**: 126.00 sec
- Futási idők **maximuma**: 244642.00 sec

Általános gráf esetén branch and bound módszer sokkal gyorsabb, mint a backtracking módszer. Az átlagos teljesítményüket nézve, mindkettő hasonló eredményeket ad, viszont az idő szempontjából a backtracking messze elmarad.

### 11.7. Bruteforce és dinamikus módszerrel hátizsák feladat megoldása, n=100 elem esetén

- Futási idők **átlaga**: 818.25 sec
- Futási idők **minimuma**: 164.00 sec
- Futási idők **maximuma**: 1720.00 sec

### 11.8. Bruteforce és dinamikus módszerrel hátizsák feladat megoldása, n=1000 elem esetén

- Futási idők **átlaga**: 77661.30 sec
- Futási idők **minimuma**: 164.00 sec
- Futási idők **maximuma**: 162422.00 sec

### 11.9. Bruteforce és dinamikus módszerrel hátizsák feladat megoldása, n=10000 elem esetén

- Futási idők **átlaga**: 16303.80 sec
- Futási idők **minimuma**: 14697.00 sec
- Futási idők **maximuma**: 23891.00 sec

Átlagértéket számolva a Dinamikus programozási módszer töltötte meg a legjobban a hátizsákot és adta a legnagyobb értéket 100 tárgy, 1000 tárgy, 10000 tárgy esetében egyaránt. Gyors megoldást viszont a bruteforce módszer ad, viszont nem tölti meg teljesen a hátizsákot és kis értékeket generál.

### 11.10. TSP Bruteforce és Christofides megoldásának összehasonlítása, n=10 elem esetén

- Futási idők **átlaga**: 58.1161 sec
- Futási idők **minimuma**: 1.0000 sec
- Futási idők **maximuma**: 17109818.00 sec

Átlagértéket számolva a Christofides programozási módszer adta meg gyorsabban az útvonalat 10 város esetében.

### 11.11. Szoftverek összehasonlítása

Csak akkor tudjuk összehasonlítani a mérési eredményeket, ha ugyanazon az eszközön dolgozunk. Az én laptopom hardver beállításai a következők:

- Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz (8 CPUs), 1.8GHz

# Összefoglaló

<https://github.com/mesteri15micro/disszertaciosDolgozat.git>

## 11.12. Következtetések

Dolgozatunkban a Közelítő algoritmusokkal foglalkoztunk, NP-teljes feladatok megoldására használtuk, különböző programozási módszerekkel valósítottuk meg. Először az elméleti megalapozással kezdtük a dolgozat megírását, majd megvizsgáltunk és elkészítettünk hét algoritmust a Bin Packing feladat megoldására, hátizsák feladat megoldására, gráf színezés elkészítésére, valamint az utazóügynök feladat megoldására. Ezekre generáltunk bemeneti adatokat, amelyekkel készítettünk tesztek. Ezekből átlagokat számoltunk és megmagyaráztuk a kapott eredményeket. Ráadásul, hogy könnyebben meg lehessen érteni ezeket készítettünk két felhasználói felületet, amelyekbe beépítettük az egyes feladattípusok szemléltető programjait, ami gyakorlatilag bárki számára könnyebben érthetővé és világossá teszi, hogy miről is szólnak ezek az NP-teljes feladatok. Az eredményeket a mérések részénél bemutattuk. Ez alapján megindokoltuk, hogy melyik módszerek voltak a legjobbak az egyes feladatok tanulmányozása során, így most már tudjuk, hogy melyik algoritmusokkal érdemes tovább foglalkozni.

Továbbfejleszteni kellene ezeket a megoldási módszereket. Ehhez az szükséges, hogy hosszabb ideig, alaposabban kell tanulmányozni őket. Továbbá érdemes lenne olyan nyílt forráskódú és szabadon felhasználható szoftvereket kifejleszteni, amelyekkel alaposabban lehet tanulmányozni ezeket a Közelítő algoritmusokat, amelyekkel az NP-teljes feladatok megoldása könnyebbé válna.

# Köszönetnyilvánítás

Mindenekelőtt köszönettel tartozom témavezetőmnek, Dr. Kása Zoltán, egyetemi tanáromnak, aki megtisztelt azzal, hogy elfogadta felkérésem, és építő kritikával támogatott a disszertációs dolgozat elkészítésében.

Külön köszönet illeti Dr. Kupán A. Pál egyetemi docens, aki egyetemi tanulmányaim folyamán mindig türelemmel és megértéssel segítőkészen támogatta munkámat. Nagy segítségemre volt a disszertációs dolgozat elkészítésében.

Külön köszönettel tartozom dr. Kitlei Róbert László egyetemi tanáromnak, aki építő kritikával, útmutatással támogatott a disszertációs dolgozat elkészítésében.

# Ábrák jegyzéke

8.1. Nem funkcionális elvarasok . . . . .	49
10.1. Python IDLE 3.10.2 . . . . .	51
10.2. Python IDLE 3.10.2 . . . . .	53
10.3. Bin Packing feladat vizuális ábrázolása . . . . .	54
10.4. Christofides algoritmus vizuális ábrázolása . . . . .	54
10.5. gráf színezés vizuális ábrázolása . . . . .	55
10.6. Hamiltoni körnek a vizuális ábrázolása . . . . .	55
10.7. Hátizsák feladat vizuális ábrázolása . . . . .	56
10.8. TSP körút vizuális ábrázolása . . . . .	57
10.9. Dodekaéder vizuális ábrázolása . . . . .	58
10.10 az elkészített algoritmusok bonyolultsága . . . . .	58



# Táblázatok jegyzéke

2.1.	Általános gráf backtracking branch and bound statisztika táblázat . . . .	18
2.2.	Hamiltoni gráf backtracking branch and bound statisztika táblázat . . . .	19
3.1.	pontos megoldás Bin packing statisztika táblázat . . . . .	26
3.2.	közelítő megoldás Bin packing statisztika táblázat . . . . .	27
4.1.	Hátizsák feladat Brute Force és Dinamikus programozási módszerrel megoldva statisztika táblázat . . . . .	36
7.1.	Utazó ügynök feladat Brute Force és Christofides programozási módszerrel megoldva statisztika táblázat . . . . .	48

# Irodalomjegyzék

- [Ant06] Margit Antal. *Fejlett Programozási Technikák*. Cluj Napoca Scientia, 2006.
- [Ant07] Margit Antal. *Toward a Simple Phoneme Based Speech Recognition System*, volume 52. 2007.
- [CAA03] Thomas H. Cormen, Iványi Antal, and Benczúr András. *Új algoritmusok*. Scholar, 2003.
- [CLR<sup>+</sup>03] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (fordítók: Ivanyi Antal, and Benczur Andras). *Új algoritmusok*. Scholar Informatika, Budapest, 2nd edition, 2003.
- [GKP98] Ronald Lewis Graham, Donald Ervin Knuth, and Oren Patashnik. *Konkrét Matematika: [A Számítástudomány Alapja]: [Tankönyv]*. Muszaki K, 1998.
- [Iva19] Cosmina Ivan. *Calculul Paralel și Distribuít*. Editura UTPRESS Cluj-Napoca, 2019.
- [K83] Zoltán Kása. *Ismerkedés az Informatikával*. Dacia Könyvkiadó, 1983.
- [K03] Zoltán Kása. *Combinatorica cu Aplicatii*. Presa Universitara Clujeana, 2003.
- [K04] Zoltán Káta. *Programozás C Nyelven, Jegyzet*. Cluj-Napoca: Scientia, 2004.
- [K08] Zoltán Káta. *Dynamic Programming as Optimal Path Problem in Weighted Digraphs*, volume 24. 2008.
- [KA] Zoltán Kása and Bege Antal. *Matematică Discretă / Sapiientia EMTE Marosvásárhelyi Kar Könyvtára*.
- [Knu08] Donald E. Knuth. *Fák előállítása; Kombinatorikus előállítások története*. 4/4. kot. 2008.
- [Knu11] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 1st edition, 2011.
- [LS15] László Lovász and Balázs Szegedy. *The Automorphism Group of a Graphon*, volume 421. 2015.
- [Ope23] OpenAI. Gpt-3.5 (chatgpt) code explanations for guide code explanations, 2023.

- [Sar78] Baase Sara. *Computer Algorithms: Introduction to Design and Analysis*. San Diego State University, Addison-Wesley Publishing Company, 1st edition, 1978.
- [SK11] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.
- [TIS04] Andrew S. Tanenbaum, Ketler Ivan, and Maarten Van Steen. *Elosztott rendszerek: alapelvek es paradigmak*. Panem, 2004.