Szakdolgozat

Miskolci Egyetem

A szakdolgozat címe

Készítette:

Szakdolgozó Neve

Programtervező informatikus

Témavezető:

Témavezető neve

Miskolc, 2020

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

Szakdolgozat Feladat

Szakdolgozó Neve (N3P7UN) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: kulcsszavak, hasonlók

A szakdolgozat címe: A dolgozat címe

A feladat részletezése:

Ide kell a feladatkiírásban szereplő szöveget betenni.

(Kisebb tagolás lehet benne, hogy jól nézzen ki.)

Témavezető: Témavezető neve (beosztása)

A feladat kiadásának ideje:

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Szakdolgozó Neve**; Neptun-kód: `N3P7UN` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Szakdolgozat Címe* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

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

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

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

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

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

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

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. Bevezetés **1**

2. Koncepció **2**
2.1. A fejezet célja . . . 2
2.2. Tartalom és felépítés . . . 2
2.3. Amit csak említés szintjén érdemes szerepeltetni . . . 2
2.4. Lineáris egyenletrendszerek . . . 3
2.5. Hátizsák probléma . . . 3
2.5.1. Alkalmazása . . . 3
2.5.2. Fajtái . . . 4
2.5.3. Számítási bonyolultság . . . 4
2.5.4. Megoldások . . . 5
2.6. A közlekedési probléma megoldásan . . . 9

3. Tervezés **12**
3.1. Táblázatok . . . 12
3.2. Ábrák . . . 12
3.3. További környezetek . . . 12

4. Megvalósítás **14**

5. Tesztelés **18**

6. Összefoglalás **19**

Irodalomjegyzék **20**

1. fejezet

Bevezetés

A fejezet célja, hogy a feladatkiírásnál kicsit részletesebben bemutassa, hogy miről fog szólni a dolgozat. Érdemes azt részletezni benne, hogy milyen aktuális, érdekes és nehéz probléma megoldására vállalkozik a dolgozat.

Ez egy egy-két oldalas leírás. Nem kellenek bele külön szakaszok (section-ök). Az irodalmi háttérbe, a probléma részleteibe csak a következő fejezetben kell belemenni. Itt az olvasó kedvét kell meghozni a dolgozat többi részéhez.

2. fejezet

Koncepció

2.1. A fejezet célja

Ez a fejezet még nem a saját eredményekkel foglalkozik, hanem bemutatja, mi a problémakör, milyen módszerekkel, milyeneredményeket sikerült elérni eddig másoknak.

A hivatkozások jelentős része ehhez a fejezethez szokott kötődni. (Egy hivatkozás például így néz ki [1].) Itt lehet bemutatni a hasonló alkalmazásokat.

2.2. Tartalom és felépítés

A fejezet tartalma témától függően változhat. Az alábbiakat attól függően különböző arányban tartalmazhatják.

- Irodalomkutatás. Amennyiben a dolgozat egy módszer kidolgozására, kifejlesztésére irányul, akkor itt lehet részletesen végignézni (módszertani vagy időrendi bontásban), hogy az eddigiekben milyen eredmények születtek a témakörben.
- Technológia. Mivel jellemzően kutatásról vagy szoftverfejlesztésről van szó, ezért annak a jellemző elemeit, technikai részleteit itt kell bemutatni. Ez tehát egy módszeres bevezetés ahhoz, hogy ha valaki nem jártas a témakörben, akkor tudja, hogy a dolgozat milyen aktuálisan elérhető eredményeket, eszközöket használt fel.
- Piackutatás. Bizonyos témáknál új termék vagy szolgáltatás kifejlesztése a cél. Ekkor érdemes annak alaposan utánanézni, hogy aktuálisan milyen eszközök érhetők el a piacon. Ez szoftverek esetében a hasonló alkalmazások bemutatását, táblázatos formában történő összehasonlítását jelentheti. Szerepelhetnek képek és észrevételek a viszonyításként bemutatott alkalmazásokhoz.
- Követelmény specifikáció. Külön szakaszban érdemes részletesen kitérni az elkészítendő alkalmazással kapcsolatos követelményekre. Ehhez tartozhatnak forgatókönyvek (*scenario*-k). A szemléletesség kedvéért lehet hozzájuk képernyőkép vázlatokat is készíteni, vagy a használati eseteket más módon szemléltetni.

2.3. Amit csak említés szintjén érdemes szerepeltetni

Az olvasóról annyit feltételezhetünk, hogy programozásban valamilyen szinten járatos, és a matematikai alapfogalmakkal sem ebben a dolgozatban kell megismertetni. A spe-

ciális eszközök, programozási nyelvek, matematikai módszerekk és jelölések persze jó, hogy ha említésre kerülnek, de nem kell nagyon belemenni a közismertnek tekinthető dolgokba.

2.4. Lineáris egyenletrendszerek

A matematikában a lineáris egyenletrendszer (vagy lineáris rendszer) egy vagy több lineáris egyenlet gyűjteménye, amely ugyanazt a változóhalmazt tartalmazza. Például ez egy 3 ismeretlent tartalmazó rendszer:

2.5. Hátizsák probléma

A knapsack-probléma a kombinatorikus optimalizálás egyik problémája: A feladat a következő: Adott egy halmaznyi elem, amelyek mindegyikének van egy súlya és egy értéke, határozza meg, hogy hány darabot kell felvenni a gyűjteménybe úgy, hogy az összes súly kisebb vagy egyenlő legyen egy adott határértéknél, és az összes érték a lehető legnagyobb legyen. Nevét arról a problémáról kapta, amellyel az szembesül, akinek egy meghatározott méretű hátizsákot kell megtömnie a legértékesebb tárgyakkal. A probléma gyakran felmerül az erőforrás-elosztás során, amikor a döntéshozóknak nem osztható projektek vagy feladatok halmazából kell választaniuk egy rögzített költségvetés, illetve időkorlátozás mellett.

A hátizsákproblémát több mint egy évszázada tanulmányozzák, a korai munkák már 1897-ben megjelentek.[1] A "hátizsákprobléma" elnevezés Tobias Dantzig (1884-1956) matematikus korai munkáira vezethető vissza,[2] és arra a hétköznapi problémára utal, hogy a legértékesebb vagy leghasznosabb tárgyakat úgy kell becsomagolni, hogy a poggyász ne legyen túlterhelve.

2.5.1. Alkalmazása

A Knapsack-problémák számos területen megjelennek a valós döntéshozatali folyamatokban, például a nyersanyagok legkevésbé pazarló darabolásának megtalálása, befektetések és portfóliók kiválasztása, eszközök kiválasztása eszközfedezetű értékpapírosításhoz, valamint kulcsok generálása a Merkle-Hellman és más Knapsack-kriptoszisztémákhoz.

A knapsack-algoritmusok egyik korai alkalmazása olyan tesztek szerkesztésében és pontozásában volt, amelyekben a tesztelők választhatnak, hogy milyen kérdésekre válaszolnak. Kis példák esetén meglehetősen egyszerű folyamat a tesztelőknek ilyen választási lehetőséget biztosítani. Ha például egy vizsga 12 kérdést tartalmaz, amelyek mindegyike 10 pontot ér, a vizsgázónak csak 10 kérdésre kell válaszolnia ahhoz, hogy elérje a maximálisan elérhető 100 pontot. A pontértékek heterogén eloszlásával rendelkező tesztek esetében azonban nehezebb választási lehetőséget biztosítani. Feuerman és Weiss egy olyan rendszert javasolt, amelyben a tanulók heterogén tesztet kapnak összesen 125 lehetséges pontszámmal. A tanulóknak az összes kérdésre a legjobb tudásuk szerint kell válaszolniuk. A feladatok lehetséges részhalmazai közül, amelyek összpontszáma 100 pontot tesz ki, egy knapsack-algoritmus meghatározná, hogy melyik részhalmaz adja az egyes tanulóknak a lehető legmagasabb pontszámot.

A Stony Brook University Algorithm Repository (Stony Brook Egyetem Algoritmustár) 1999-es vizsgálata kimutatta, hogy 75 algoritmikus probléma közül a knapsack-probléma a 19. legnépszerűbb és a harmadik legkeresettebb a szuffixfák és a kukacsomagolási probléma után.

2.5.2. Fajtái

A leggyakoribb megoldandó probléma a **0-1-es hátizsákprobléma**, amely az egyes tárgyfajták x_i példányszámát nullára vagy egyre korlátozza. Adott egy 1-től n-ig számozott n darabból álló halmaz, mindegyiknek van egy w_i súlya és egy v_i értéke, valamint egy W maximális súlykapacitás.

$$maximize \sum_{i=1}^{n} v_i x_i$$

$$subject to \sum_{i=1}^{n} w_i x_i \leq W \text{and} x_i \in \{0, 1\}.$$

Itt az x_i az i elemnek a hátizsákba felvenni kívánt példányainak számát jelöli. A feladat tehát az, hogy maximalizáljuk a hátizsákban lévő elemek értékeinek összegét úgy, hogy a súlyok összege kisebb vagy egyenlő legyen a hátizsák befogadóképességével.

A **korlátozott zsákprobléma** megszünteti azt a korlátozást, hogy minden tételből csak egy van, de az egyes tételek x_i példányainak számát egy maximális, nem negatív egész c értékre korlátozza:

$$maximize \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \geq 0, x_i \in \{0, 1, 2, \ldots, c\}.$$

A **korlátlan zsákprobléma** nem szab felső korlátot az egyes tárgyfajták példányszámára, és a fentiek szerint fogalmazható meg, azzal a különbséggel, hogy az x_i-re vonatkozó egyetlen korlátozás az, hogy az egy nemnegatív egész szám.

$$maximize \sum_{i=1}^{n} v_i x_i$$

$$subject to \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \geq 0, x_i \in \mathbb{Z}.$$

2.5.3. Számítási bonyolultság

A hátizsák-probléma a számítástechnika szempontjából több okból is érdekes: A hátizsák-probléma döntési problémaformája (lehet-e legalább V értéket elérni úgy, hogy a W súlyt ne lépjük túl?) NP-teljes, így nincs ismert algoritmus, amely minden esetben helyes és gyors (polinomiális idejű) lenne. Míg a döntési probléma NP-teljes, addig az optimalizálási probléma nem az, annak megoldása legalább olyan nehéz, mint a döntési probléma, és nincs olyan ismert polinomiális algoritmus, amely egy adott megoldásról meg tudná mondani, hogy az optimális-e (ami azt jelentené, hogy nincs nagyobb V-vel

rendelkező megoldás, így megoldva az NP-teljes döntési problémát). Létezik egy pszeudopolinomiális idejű algoritmus, amely dinamikus programozást használ. Létezik egy teljesen polinomiális idejű közelítési séma, amely az alább ismertetett pszeudopolinomiális idejű algoritmust használja alprogramként. Számos, a gyakorlatban előforduló eset, illetve bizonyos eloszlásokból származó "véletlen példány" ennek ellenére pontosan megoldható.

A "döntési" és az "optimalizálási" problémák között van egy olyan kapcsolat, hogy ha létezik egy polinomiális algoritmus, amely megoldja a "döntési" problémát, akkor az optimalizálási probléma maximális értékét polinomiális időben meg lehet találni úgy, hogy ezt az algoritmust iteratívan alkalmazzuk, miközben növeljük a k értékét. Másrészt, ha egy algoritmus polinomiális idő alatt találja meg az optimalizálási probléma optimális értékét, akkor a döntési probléma polinomiális idő alatt megoldható úgy, hogy az ezen algoritmus által kimeneti megoldás értékét összehasonlítjuk k értékével. Így a probléma mindkét változata hasonló nehézségű.

A kutatási irodalom egyik témája annak meghatározása, hogy milyenek a hátizsák-probléma "nehéz" példányai, vagy másképp nézve, milyen tulajdonságokkal rendelkeznek a gyakorlatban a példányok, amelyek a legrosszabb esetben NP-teljesnek tűnő viselkedésüknél fogva könnyebben megoldhatóvá tehetik őket. A cél ezen "nehéz" példányok megtalálása a nyilvános kulcsú kriptográfiai rendszerekben való felhasználásuk, például a Merkle-Hellman hátizsákos kriptorendszerben.

Figyelemre méltó továbbá az a tény, hogy a knapsack-probléma nehézsége függ a bemenet formájától. Ha a súlyokat és a nyereséget egész számokként adjuk meg, akkor gyengén NP-teljes, míg ha a súlyokat és a nyereséget racionális számokként adjuk meg, akkor erősen NP-teljes. Racionális súlyok és nyereségek esetén azonban még mindig van egy teljesen polinomidejű közelítő séma.

2.5.4. Megoldások

A hátizsák-problémák megoldására számos algoritmus áll rendelkezésre, amelyek a dinamikus programozási megközelítésen, az ág és korlát megközelítésen vagy a két megközelítés hibridizációján alapulnak.

Dinamikus programozási algoritmus

A következő dinamikus programozási megoldás a 0-1 knapsack problémára pszeudopolinomiális idő alatt fut. Tegyük fel, hogy $W w_1$, $w_2, \ldots, w_n, W$ szigorúan pozitív egészek.

Definiáljuk $m[i, w]$-nek azt a maximális értéket, amely w-nél kisebb vagy azzal egyenlő súllyal érhető el i elemig (első i elem).

Az $m[i, w]$-t rekurzívan a következőképpen határozhatjuk meg:

- $m[0, w] = 0$
- $m[i, w] = m[i-1, w]$ ha $w_i > w$ (az új elem meghaladja a jelenlegi súlyhatárt)
- $m[i, w] = max(m[i-1, w], m[i-1, w-w_i] + v_i)$ ha $w_i \leq w$.

A megoldás ezután $m[n, W]$ kiszámításával található meg. Ahhoz, hogy ezt hatékonyan elvégezhessük, használhatunk egy táblázatot a korábbi számítások tárolására.

Az alábbiakban a dinamikus program pszeudokódja következik:

```
// Bemenet:
// Értékek (v -tömben tároljuk)
// Tömegek (w -tömben tároljuk)
// Elemek száma (n)
// Hátizsák kapacitása (W)

array m[0..n, 0..W];
for j from 0 to W do:
m[0, j] := 0
for i from 1 to n do:
m[i, 0] := 0

for i from 1 to n do:
for j from 0 to W do:
if w[i] > j then:
m[i, j] := m[i-1, j]
else:
m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

Ez a megoldás tehát $O(nW)$ idő alatt és $O(nW)$ tárterülettel fut le. (Ha csak az $m[n, W]$ értékre van szükségünk, akkor a kódot úgy módosíthatjuk, hogy a szükséges memória mennyisége $O(W)$ legyen, amely az "m" tömb legutóbbi két sorát tárolja.)

Ha azonban egy-két lépéssel továbbmegyünk, akkor tudnunk kell, hogy a módszer $O(nW)$ és $O(2^n)$ közötti idő alatt fut le. Tudjuk, hogy nincs szükség az összes súly kiszámítására, ha az elemek száma és maguk a választott elemek száma fix. Vagyis a fenti program a szükségesnél többet számol, mert a súlyok állandóan 0-ról W-re változnak. Ebből a szempontból úgy programozhatjuk ezt a módszert, hogy rekurzívan fusson le.

```
// Bemenet:
// Értékek (v -tömben tároljuk)
// Tömegek (w -tömben tároljuk)
// Elemek száma (n)
// Hátizsák kapacitása (W)

Változó definiálás[n, W]

Inicializálás[i, j] = -1

Definiálás m:=(i,j)
{
if i == 0 or j <= 0 then:
value[i, j] = 0
return

if (value[i-1, j] == -1) then:
value[i-1, j] = m(i-1, j)

if w[i] > j then:
```

```
value[i, j] = value[i-1, j]
else:
if (value[i-1, j-w[i]] == -1) then:
value[i-1, j-w[i]] = m(i-1, j-w[i])
value[i, j] = max(value[i-1,j], value[i-1, j-w[i]] + v[i])
}
Run m(n, W)
```

Például 10 különböző tétel van, és a súlyhatár 67. Tehát,

Ha a fenti módszerrel kiszámítjuk az $m(10, 67)m(10, 67)m(10, 67)$, akkor ezt kapjuk, kizárva azokat a hívásokat, amelyek $m(i, j) = 0m(i, j) = 0m(i, j) = 0$:

Emellett megszakíthatjuk a rekurziót és átalakíthatjuk fává. Ezután levághatunk néhány levelet, és párhuzamos számítást használhatunk a módszer futásának felgyorsítására.

Ahhoz, hogy az elemek tényleges részhalmazát találjuk meg, és ne csak az összértéküket, a fenti függvény futtatása után futtathatjuk ezt:

```
/**
* Returns the indices of the items of the optimal knapsack.
* i: We can include items 1 through i in the knapsack
* j: maximum weight of the knapsack
*/
function knapsack(i: int, j: int): Set<int> {
if i == 0 then:
return {}
if m[i, j] > m[i-1, j] then:
return {i} \cup knapsack(i-1, j-w[i])
else:
return knapsack(i-1, j)
}

knapsack(n, W)
```

Meet-in-the-middle

Egy másik algoritmus a 0-1 hátizsákra, amelyet 1974-ben fedeztek fel és néha "meet-in-the-middle"-nek neveznek a kriptográfiában használt hasonló nevű algoritmussal való párhuzamosság miatt. Exponenciális a különböző elemek számában, de előnyösebb lehet a DP algoritmusnál, ha W nagy az n-hez képest. Különösen, ha a w_i nemnegatív, és nem egész szám, akkor a dinamikus programozási algoritmust még mindig használhatjuk skálázással és kerekítéssel (azaz fixpontos aritmetikával), de ha a probléma d tört számjegy pontosságot igényel a helyes válaszhoz, akkor W-t 10^d, és a DP algoritmusnak szüksége lesz $O(W10^d)$ tárterületre és $O(nW10^d)$ időre.

Az algoritmus $O(2n/2)O(2^{n/2})O(2^{n/2})$ hely, és a 3. lépés hatékony megvalósításai (például a B részhalmazok súly szerinti rendezése, a B azon részhalmazainak elvetése, amelyek súlya nagyobb, mint a B más, nagyobb vagy egyenlő értékű részhalmazainak súlya, és bináris keresés a legjobb egyezés megtalálásához) $O(n2n/2)O(n2^{n/2})$ futási időt eredményeznek. $O(n2^{n/2})$. A kriptográfiában a meet in the middle támadáshoz hasonlóan ez is javítja az $O(n2n)O(n2^n)$ futási időt. $O(n2^n)$ futási idejét a naiv nyers

erő megközelítésnek (az $\{1...n\}$ $\{1...n\}\{1...n\}$ összes részhalmazának vizsgálata), de ez nem állandó, hanem exponenciális helyigényű (lásd még baby-step giant-step).

Megközelítő algoritmusok

Mint a legtöbb NP-teljes probléma esetében, elegendő lehet működőképes megoldásokat találni, még akkor is, ha azok nem optimálisak. Előnyös azonban, ha a közelítéssel együtt jár a talált megoldás értéke és az optimális megoldás értéke közötti különbség garantálása.

Mint sok hasznos, de számításigényes algoritmus esetében, a megoldást közelítő algoritmusok létrehozására és elemzésére is jelentős kutatások folytak. A Knapsack-probléma, bár NP-nehez, egyike azon algoritmusok gyűjteményének, amelyek még mindig közelíthetők bármilyen meghatározott mértékben. Ez azt jelenti, hogy a problémának van polinomiális idejű közelítési sémája. Pontosabban a zsákproblémának van egy teljesen polinomiális idejű közelítési sémája (FPTAS)[19].

Mohó közelítő algoritmus George Dantzig egy mohó közelítő algoritmust javasolt a korlátlan zsákprobléma megoldására.[20] Az ő változata a súlyegységre jutó érték csökkenő sorrendjében rendezi a tételeket, $v1/w1 \geq \cdots \geq vn/wnv_1/w_1 \geq \cdots \geq v_n/w_n$.

$v_1/w_1 \geq \cdots \geq v_n/w_n$. Ezután folytatja a zsákba helyezésüket, kezdve az első fajta elem minél több példányával, amíg a zsákban már nincs hely többnek. Feltéve, hogy az egyes tárgyfajtákból korlátlan mennyiség áll rendelkezésre, hammm a zsákba beférő tárgyak maximális értéke, akkor a mohó algoritmus garantáltan eléri legalább az $m/2m/2m/2$ értéket.

A korlátos probléma esetén, ahol az egyes tárgyfajták kínálata korlátozott, a fenti algoritmus messze nem lehet optimális. Mindazonáltal egy egyszerű módosítással ezt az esetet is megoldhatjuk: Az egyszerűség kedvéért tegyük fel, hogy minden tétel egyenként elfér a zsákban

($wi \leq Ww_i \leq Ww_i \leq W$ minden ii i-re).

Konstruáljunk egy megoldást $S1S_1S_1$ az elemek mohó pakolásával, ameddig csak lehet, azaz $S1 = 1, ..., kS_1 = \{1, \ldots, k\}S_1 = \{1, \ldots, k\}$ ahol $k = max1 \leq k' \leq n \sum i = 1k'wi \leq Wk = \max_{1 \leq k' \leq n} \sum_{i=1}^{k'} w_i \leq Wk = \max_{1 \leq k' \leq n} \sum_{i=1}^{k'} w_i \leq W$. Konstruáljunk továbbá egy második megoldást $S_2 = k + 1S_2 = \{k + 1\}\, S_2 = \{k + 1\}$, amely tartalmazza az első nem megfelelő elemet. Mivel $S1\cup \leq 2S_1 \cup S_2S_1 \cup S_2$ felső korlátot ad a probléma LP relaxációjához, az egyik halmaznak legalább $m/2m/2m/2$ értékűnek kell lennie; így az $S1S_1$ közül bármelyiket is adjuk vissza. S_1 és $S2S_2S_2$ jobb értékkel rendelkezik, hogy $1/21/21/2$ közelítést kapjunk.

Teljesen polinomiális idejű közelítési rendszer

A teljesen polinomiális idejű közelítési séma (FPTAS) a Knapsack-problémára kihasználja azt a tényt, hogy a problémának azért nincs ismert polinomiális idejű megoldása, mert a tételekhez kapcsolódó nyereségek nem korlátozottak. Ha a nyereségértékek néhány legkisebb értékű számjegyét lekerekítjük, akkor azok polinommal és $1/\varepsilon$-vel lesznek korlátosak, ahol ε a megoldás helyességére vonatkozó korlát. Ez a korlátozás akkor azt jelenti, hogy egy algoritmus polinomiális idő alatt találhat olyan megoldást, amely az optimális megoldás $(1 - \varepsilon)$ tényezőjén belül helyes[19].

Tétel: Az $S'S'$ halmaz S' a fenti algoritmussal kiszámított S' teljesíti a $profit(S') \leq (1 - \varepsilon)\ profit(S*)$profit $(S') \geq (1 - \varepsilon) \cdot$ profit (S^*)profit$(S') \geq (1 - \varepsilon) \cdot$ profit(S^*), ahol $S * S^*S^*$ egy optimális megoldás.

2.6. A közlekedési probléma megoldásan

A termékeknek a termelési és raktározási helyekről a keresleti központokba történő szállításával járó költségek minimalizálása a termékforgalmazással foglalkozó vállalatok számára a nyereségesség fenntartásának lényeges része. Mivel a szállítási költségek általában nem ellenőrizhetők, a teljes költség minimalizálása a legjobb termék útvonalválasztási döntések meghozatalát igényli. Ezt az alapvető problémát először az 1940-es évek elején fogalmazták meg lineáris programozási problémaként, és a köznyelvben szállítási problémaként ismert.

Mi a közlekedési probléma?

A szállítási probléma egyfajta lineáris programozási probléma, amelynek célja, hogy minimalizálja egy termék M forrásból N célállomásra történő elosztásának költségét.

A közlekedési problémát számos területről származó példával lehet leírni. Az egyik alkalmazás a csapatok hatékony mozgatásának problémája a bázisokról a harctéri helyszínekre. Egy másik az ügynökök vagy munkások optimális hozzárendelése különböző munkakörökhöz vagy pozíciókhoz. Messze a leggyakoribb alkalmazás az áruk szállítása több gyárból több raktárhelyszínre, vagy a raktárakból a kirakatokba.

Szállítási probléma példa

Vegyük a következő példát:

XYZ Inc. két gyárral rendelkezik az ország különböző pontjain, ahol kütyüket gyártanak. Értékesítési partnerüknek három központi raktára van, ahonnan ezeket a widgeteket szállítják a különböző ügyfeleiknek. A gyárak hetente egy-egy adott számú widgetet tudnak előállítani, és az egyes raktárak várható kereslete is ismert. Minden gyárból minden raktárba szállítási költséget kell fizetni. Melyik gyárnak hány widgetet kell gyártania és melyik raktárba szállítania, hogy az egyes helyszíneken minimális költséggel kielégítse a keresletet?

Ez a problémafeltevés egy tipikus szállítási probléma minden összetevőjét tartalmazza. A források és a célállomások általánosak - lehetnek fakitermelő helyek és fűrészüzemek, gyárak és raktárak, raktárak és raktárak, bázisok és csataterek stb.

Minden esetben van valamilyen kereslet vagy szükséglet D minden egyes N helyen, valamilyen S kínálat minden egyes M helyen, és egy egységnek egy adott M helyről egy adott N helyre történő szállításához (vagy felhasználásához) kapcsolódó c költség. Összesen M x N

ilyen költség.

A költség, c

lehet egy olyan számítás, amely olyan tényezőket foglal magában, mint az idő, a távolság, az anyagköltségek stb., de lehet bármilyen, a probléma szempontjából releváns mennyiség is.

Miután a készletek, az igények és a költségek ismertek, a probléma az, hogy meghatározzuk az egységek számát, x , amelyet le kell gyártani és el kell küldeni az M ellátóközpontok mindegyikéből az N

keresleti helyekre.

A teljes költség az összes egyedi költség összege, szorozva az egyes ellátóközpontokból az egyes keresleti központokba gyártandó és onnan elszállítandó egyedi egységekkel. Ha ezt optimalizálási problémaként fogalmazzuk meg, a cél a teljes költség minimalizálása:

$$\sum_{i=1}^{M}\sum_{j=1}^{N} c_{ij}x_{ij}$$

Ezzel párhuzamosan van néhány szabály (megkötés), amelyet teljesíteni kell:

A szállított egységek számának kisebbnek vagy egyenlőnek kell lennie a teljes kínálattal. A szállított darabszámnak meg kell egyeznie, vagy meg kell felelnie az egyes helyszíneken jelentkező keresletnek. A szállítandó egységek számának nagyobbnak vagy egyenlőnek kell lennie nullánál (nincs negatív érték).

Kiegyensúlyozott szállítási probléma esetén, amikor a teljes kereslet megegyezik a teljes kínálattal, a kényszerek a következő matematikai ábrázolással rendelkeznek:

$$\sum_{j=1}^{N} x_{ij} = S_i, i = 1, \ldots, M$$

$$\sum_{i=1}^{M} x_{ij} = D_j, j = 1, \ldots, N$$

$$x_{ij} \geqslant 0, i = 1\ldots, M, j = 1\ldots N$$

Megjegyzendő, hogy előfordulhat, hogy a túlkínálat vagy a túlkereslet kiegyensúlyozatlan problémához vezet. Ezt az esetet az alábbiakban tárgyaljuk, és hasonlóképpen megoldható dummy változókkal és esetleg a kielégítetlen keresletért járó büntetésekkel vagy a többletkínálat tárolási költségeivel.

A teljes költségfüggvény a három megszorítással együtt egy jól formált lineáris programozási (LP) optimalizálási problémát határoz meg lineáris megszorításokkal. Ez a probléma kifejezetten a kiegyensúlyozott szállítási probléma néven ismert.

A közlekedési probléma megoldása

Északnyugati sarok szabály

Meghatározás: Az északnyugati sarokszabály egy olyan módszer, amelyet a szállítási probléma kezdeti megvalósítható megoldásának kiszámítására alkalmaznak. Az északnyugati sarok elnevezést azért kapta ez a módszer, mert az alapváltozókat a bal szélső sarokból választjuk ki.

Az északnyugati sarok fogalma jól megérthető az alábbiakban megadott szállítási problémán keresztül:

A táblázatban három A, B és C forrás van megadva, amelyek termelési kapacitása 50 egység, 40 egység, illetve 60 egység termék. A három kiskereskedő D, E és F keresletét minden nap legalább 20 egység, 95 egység és 35 egység termékkel kell kielégíteni. A szállítási költségek szintén szerepelnek a mátrixban.

A szállítási feladat megoldásának előfeltétele, hogy a keresletnek meg kell egyeznie a kínálattal. Ha a kereslet nagyobb, mint a kínálat, akkor a táblázatba dummy-eredetet kell beilleszteni. A fiktív származási hely kínálata megegyezik a teljes kínálat és a teljes kereslet különbségével. A fiktív eredethez kapcsolódó költség nulla lesz.

Hasonlóképpen, ha a kínálat nagyobb, mint a kereslet, akkor létre kell hozni egy fiktív forrást, amelynek kereslete megegyezik a kínálat és a kereslet különbségével. A fiktív forráshoz kapcsolódó költség ismét nulla lesz.

Ha a kereslet és a kínálat megegyezik, a következő eljárást kell követni:

Válassza ki a mátrix északnyugati vagy bal szélső sarkát, és a keresleti és kínálati korlátok között a lehető legtöbb egységet rendelje az AD cellához. Például 20 egységet

rendelünk az első cellához, amely kielégíti a D célállomás keresletét, miközben a kínálat többletet mutat. Most mozogjon vízszintesen, és rendeljen 30 egységet az AE cellához. Mivel 30 egység áll rendelkezésre az A forrással, a kínálat teljesen telítetté válik. Most mozogjunk függőlegesen a mátrixban, és rendeljünk 40 egységet a BE cellához. A B forrás kínálata szintén teljesen telítetté válik. Ismét lépjünk függőlegesen, és rendeljünk 25 egységet a CE cellához, az E célállomás kereslete teljesül. Mozogjunk vízszintesen a mátrixban, és rendeljünk 35 egységet a CF cellához, mind a forrás, mind a célállomás kereslete és kínálata telítődik. Most már kiszámítható a teljes költség.

A teljes költség kiszámítható az egyes cellákhoz rendelt egységek és az érintett szállítási költség szorzataként. Ezért,

Teljes költség = 20*5+ 30*8+ 40*6+ 25*9+ 35*6 = 1015 rúpia.

3. fejezet

Tervezés

Itt kezdődik a dolgozat lényegi része, úgy értve, hogy a saját munka bemutatása. Jellemzően ebben szerepelni szoktak blokkdiagramok, a program struktúrájával foglalkozó leírások. Ehhez célszerű UML ábrákat (például osztály- és szekvenciadiagramokat) használni.

Amennyiben a dolgozat inkább kutatás jellegű, úgy itt lehet konkretizálni a kutatási módszertant, a kutatás tervezett lépéseit, az indoklást, hogy mit, miért és miért pont úgy érdemes csinálni, ahogyan az a későbbiekben majd részletezésre kerül.

Ebben a fejezetben az implementáció nem kell, hogy túl nagy szerepet kapjon. Ez még csak a tervezési fázis. (Nyilván ha olyan a téma, hogy magának az implementációnak a módjával foglalkozik, adott formális nyelvet mutat be, úgy a kódpéldákat már innen sem lehet kihagyni.)

3.1. Táblázatok

Táblázatokhoz a `table` környezetet ajánlott használni. Erre egy minta a 3.1. táblázat. A hivatkozáshoz az egyedi `label` értéke konvenció szerint `tab:` prefixszel kezdődik.

3.1. táblázat. Minta táblázat. A táblázat felirata a táblázat felett kell legyen!

a	b	c
1	2	3
4	5	6

3.2. Ábrák

Ábrákat a `figure` környezettel lehet használni. A használatára egy példa a 3.1. ábrán látható. Az `includegraphics` parancsba Az ábrák felirata az ábra alatt kell legyen. Az ábrák hivatkozásához használt nevet konvenció szerint `fig:`-el célszerű kezdeni.

3.3. További környezetek

A matematikai témájú dolgozatokban szükség lehet tételek és bizonyításaik megadására. Ehhez szintén vannak készen elérhető környezetek.

3.1. ábra. A Miskolci Egyetem címere.

3.1. definíció. Ez egy definíció

3.2. lemma. *Ez egy lemma*

3.3. tétel. *Ez egy tétel*

Bizonyítás. Ez egy bizonyítás □

3.4. következmény. *Ez egy tétel*

3.5. megjegyzés. Ez egy megjegyzés

3.6. példa. Ez egy példa

4. fejezet

Megvalósítás

Ez a fejezet mutatja be a megvalósítás lépéseit. Itt lehet az esetlegesen előforduló technikai nehézségeket említeni. Be lehet már mutatni a program elkészült részeit.

Meg lehet mutatni az elkészített programkód érdekesebb részeit. (Az érdekesebb részek bemutatására kellene szorítkozni. Többségében a szöveges leírásnak kellene benne lennie. Abból lehet kiindulni, hogy a forráskód a dolgozathoz elérhető, azt nem kell magába a dolgozatba bemásolni, elegendő csak behivatkozni.)

A dolgozatban szereplő forráskódrészletekhez külön vannak programnyelvenként stílusok. Python esetében például így néz ki egy formázott kódrészlet.

```
import sys

if __name__ == '__main__':
    pass
```

A stílusfájlok a `styles` jegyzékben találhatók. A stílusok között szerepel még C++, Java és Rust stílusfájl. Ezek használatához a `dolgozat.tex` fájl elején `usepackage` paranccsal hozzá kell adni a stílust, majd a stílusfájl nevével megegyező környezetet lehet használni. További példaként C++ forráskód esetében ez így szerepel.

```
#include <iostream>

class Sample : public Object
{
    // An empty class definition
}
```

Stílusfájlokból elegendő csak annyit meghagyni, amennyire a dolgozatban szükség van. Más, C szintaktikájú nyelvekhez (mint például a JavaScript és C#) a Java vagy C++ stílusfájlok átszerkesztésére van szükség. (Elegendő lehet csak a fájlnevet átírni, és a fájlban a környezet nevét.)

Nyers adatok, parancssori kimenetek megjelenítéséhez a `verbatim` környezetet lehet használni.

```
$ some commands with arguments
1 2 3 4 5
$ _
```

A kutatás jellegű témáknál ez a fejezet gyakorlatilag kimaradhat. Helyette inkább a fő vizsgálati módszerek, kutatási irányok kaphatnak külön-külön fejezeteket. A program úgy dolgozik, mint ahogy a felhasználó tenné papíron. Kiszámolja a kapott értékek es tömegek arányát, és ez alapján felállít egy új sorrendet: a legnagyobb aránypár kerül az elsőhelyre és a legkisebb a végére. Ezután megnézni, hogy a legnagyobb aránnyal rendelkező belefér e a zsákba. Ha igen, akkor 1 -es kerül az első (nulladik) indexre, és nézi a következő elemet. Ha egy olyan elemet találunk, ami már nem fér bele, akkor egy tört kerül megfelelő indexre, hogy a tömegének hanyadrésze fér bele a zsákba. A maradék elemeket kinullázza. A Z érték is megfelelően kiszámításra kerül. Ezután végig megyünk a fán. Az összes helyre, ahova tört került, megnézzük 1 és 0 érték fixálásával és a szülők már fixált értékeinek öröklésével a Z értéket, hogy megtaláljuk azt a kombinációt, amikor a legtöbb elem ténylegesen belefér a zsákba. A program a kapott érték és tömeg értékeket külön 1-1 tömben tárolja. Az aranyTomb függvény segítségével kerül feltöltésre a harmadik tömb, ahova a kiszámolja a paraméterként kapott tömeg és érék tömbökből az elemek arányát. Ezután következik az arányok szerinti rendezés a rendezés metódus segítségével. Paraméterként megkapja mindhárom tömböt. A Java Arrays.sort() metódusával rendezem növekvő sorrendbe az aránytömb értékeit, majd egy ciklus és egy segédváltozó segítésével megfordítom az értékek sorrendjét, ezzel csökkenő értékbe rendezem őket. Ezáltál az arányok tömbje már megfelelő, de az értékek és a hozzájuk tartozó tömegek továbbra is rendezetlenek. Ezért két darab for ciklus segítésével össze hasonlítom az eredeteti arány tömböt, és a rendezetett, hogy megtaláljam, melyik elem melyik indexről melyik indexre került. Ha megvan, akkor a tömeg és értékek tömb egyes elemeinek indexét is hasonlóan mozgatom. A rendezés után jön a fa egyes leveleinek kiszámítása az eredmeny metódus segítségével. Ez a metódus paraméterként a már rendezett tömeg és érték tömböt, a hátizsak méretét és egy láncolt listát kap. Emellett a metóduson belül szükség volt egy újabb tömbre, amiben azt tárólom, hogy a levél mely már fixelt elemeket örökli, és azok helyén nullával vagy eggyel számoljon. Ha az adott elem szabad, akkor -1 az értéke, ha fix nullánál 0, egynél pedig 1. Kezdetben minden értéke -1. Mérete megegyezik a kapott elemek számával. A listához hozzá adok a kapott elemek számával megegyező elemet és plusz mégegyet, ahova Z értéke kerül. Ezekután elkezdtem egy ciklus segítségével vizsgálni, hogy belefér e az aktuális elem a hátizsákba, azaz a hátizsák szabad mérete nagyobb vagy egyenlő e az aktuális elem tömegével. Ha igen, akkor az eredmény lista ciklusváltozó aktuális értékével megegyező indexre egy 1 -es érték kerül, a hátizsák szabad értékét csökkentem az aktuális elem tömegével, és z értékét növelem az akutális elem értékével. A Z értékét végül a lista végére, a tömeg tömb hosszúságával megegyező indexű elem helyére írom. Amennyiben az aktuális elem már nem fér bele teljes egészében a hátizsákba, akkor meghivjuk a nemFerBele metódust. Ebben a metódusban a ciklusváltozó aktuális értékével megegyező indexre a hátizsák szabad területe és az aktuális elem tömegének aránya kerül. A Z értékhez pedig hozzá adjuk ennek az aránynak és elem értékének szorzatát. Emellett a fixek tömb i-edik elemét 0 -ra állítjuk. A következő lépésben a tortHelyereNulla metódust hívjuk meg. Paraméterként a tömeg, érték és fixek tömböt, a hátizsák méretét, az eredménylistát, és a ciklus változó aktuális értékét kapja meg. Első lépésben bővíti az eredmény listának elemeinek a számát eggyel többel, mint a tömbök mérete. A hanyadik globális változó méretét növeljük eggyel, jelezve, hogy már egy szinttel méllyebben vagyunk a fában. Egy ciklus segítségével végig megyünk a fixek tömbön. Ha az aktuális érték 1, azaz az egyik szülőtől azt örökölte, hogy annak az elemnek fixen be kell kerülnie a hátizsákba, akkor bele is kerül: csökken a hátizsák

mérete, a Z értéke pedig a bekerült elem értékével növekszik. Az eredmény lista végére bekerül a Z értéke. Az eredmény lista hanyadik nevű változóval megszorzott ciklus változó értékének megfelelő helyére egy 1 -es kerül. Ha a fixek tömb értéke 0, akkor a hanyadik nevű változóval megszorzott ciklus változó értékének megfelelő helyére 0 kerül. Ez esetben sem a Z értéke, sem a hátizsák szabad mérete nem változik. Az eredmény metódushoz hasonlóan elkezdi az eredmény listát feltölteni egy plusz felétellel: csak abban az esetben számol, ha fixek tömb ciklus változó szerinti értéke -1, azaz egy nem fix elemről van szó. Ha olyan elemhez érkeztünk, ami már nem fér teljes egészében bele a hátizsákba, akkor ismét a nemFerBele metódust hívjuk meg. Most is eljutunk a tortHelyereNulla metódus meghívásáig, és egyre mélyebbre megyünk a fában, a törtek helyére mindig 0 értéket fixálva. Amint végig értünk, lépkedünk vissza egy-egy szintet, és a fixek tömb i-edik elemét 1 -re állítjuk, és a tortHelyerEgy metódust hívjuk meg. Paraméterei ugyanazok, mint a tortHelyereNulla metódusnak. Az első lépés itt is az eredmény lista elemeinek bővítése annyi elemmel, ahány elem volt a bemenet, és még egy, a Z értékének, a hanyadik változót növeljük, és hátizsák eredeti méretét elmentjük. Ezután a fixek több értékei alapján csökkentjük a hátizsák szabad méretét, számoljuk a Z méretét, és az indexeket állítjuk 0-ra vagy 1 -re. Következőnek megvizsgáljuk, hogy a hátizsák nagyobb e, mint nulla. Ugyanis előfordulhat, hogy önmagukban a fixen zsákba került elemek sem férnek valójában bele a korlátba. Ha negatív lett a hátizsák szabad mérete nem megyünk tovább, töröljük a listából a metódus elején hozzáadott sorokat, és a hanyadik változó értékét csökkentjük eggyel, és a fixek tört tortHelyerEgy metódus meghívásakor paraméterként átadott i ciklusváltozó értékének megfelelő idexű elemét -1 -re állítjük, és visszatérünk korábbi metódus hívás helyére: a nemFerBele metódusba, ami ezután a return utasítással vissza adja a vezérlést a tortHelyereNulla metódusnak. Mivel lefutott mindkét metódushívás, a tört helyére be próbáltuk a nulla és egy értéket is, ezért ez is visszatér az előző hívásba, egy másik nemFerBele metódusba. Itt a fixek tömb i-edik eleme 1 -et kap, és meghívja a tortHelyerEgy metódust. Ez addig folytatódik, amíg az összes lehetőségen végig nem megy, de listába csak azok a elemek kerülnek, ahol minden fix elem belefért a zsákba. A listában minden "blokk" annyi elemből áll, ahány elemű volt az input, és még egy elem, a Z értéke. Optimális megoldásnak azt a lehetőséget keressük, amikor egy elem vagy teljes egészében bekerült, vagy egyáltalán nem, és a Z értéke a lehető legmagasabb. Ennek az optimalis megoldásnak megtalalásához végig kell mennünk a lista minden Z -edik elemén, és megkeresni közülök a legnagyobbat, ahol az előtte álló bevitelielemszámnyi elem mindegyik egész szám. Ezt a maximumKiiras fügvény végzi. Kettő darab egymásbaágyazott ciklus dolgolzik: a külső a minden Z -edik elemet választja ki, tehát a ciklusváltozó bemenetielemszám + 1 értékkel növekszik minden iterációban. A belső ciklus pedig a Z értékekhez képest lépked egyessével az előző Z értékig, és nézi, hogy egész csupa egész szám található e a listában. Amennyiben igen, beállítjuk maxiumumnak. A függvény ezzel a maximum értékkel tér vissza.

go(); convertToArray(); aranyTomb(); rendezes(); tombMasolas(); eredmeny(); tortHelyereNulla(); torteHelyereEgy(); listaKiiras(); maximumKiiras();

Szallitasifeladata Ennél a feladatrész leprogramozásánál is papíron történő megoldás követését tűztem ki célul. A program a kezdeti megoldást tudja kiszámolni az északnyugat sarok módszerrel, lépésenként követhető módon. A bemenet egy tetszőleges nagyságú mátrix, de fontos, hogy csak kiegyensúlyozott szállítási feladatot tud megoldani. A számítást a kezdetiMátrix metódus végzi. Egy while ciklussal kezdünk azzal a feltétellel, hogy addig

ismétlünk, amíg a két darab ciklus változó nem érik el a mátrix utolsó sorának illetve oszlopának indexét. Az oszlopVagySor függvény segítségével döntönem el hogy a két ciklus változó által kiválasztott elem sorának vagy oszlopának utolsó eleme a kisebb. Boolean típusú függvény, true értékkel tér vissza, ha a sor utolsó eleme a kisebb, és false -t, ha az oszlop utolsó eleme a kisebb. Amennyiben a két érték megegyezik, akkor is true értékkel tér vissza. Ha a sor végén található elem a kisebb, akkor az ott találhatót egy változóba elmentjük, a két ciklusváltozóval kijelölt mátrix elem helyére beírjuk ezt az értéket, és egy for ciklus segítségével kinullázza a sort. Ezekután beállítom a ciklusváltozókat. Fontos, hogy abban a sorban, amit már kinulláztunk, azzal a program már ne foglalkozzon. Ezt a funkciót tölti be az i és j ciklusváltozó mellett bevezetett kezdoI és kezdoJ változó, amik segítségével az i és j váltózok abból megnövelt indexről indulnak, követve a már kinullázott sorokat, oszlopokat. Ha a oszlop végén található elem a kisebb, akkor a folyamat hasonló az előzőekhez, annyi különbséggel, hogy az oszlop lesz kinullázva. A program kiírja az összes lépést, szépen végig követhető, hogy melyik lépsében melyik fogyasztót elégítettük ki, vagy mely termelőtől szállítottuk el az összes anyagot. A futás végén kalkulál egy kezdetleges megoldást. Feladattól függően elképzelhető, hogy ez már az optimális megoldás, de ezt a program jelenleg nem képes ellenőrizni, továbbfejlesztés szükséges.

matrixKiiras(); go(); kezdetiMegoldas(); oszlopVagySor(); megoldas();

———————————————————————————————- Linearis egyenlettrendszerek megoldása pivotálássa

A programban tetszőleges nagyságú egyenletrendszereket tudunk megoldni pivotálással. A feladatot mátrix alakban kell feltöltenünk. Először kiválasztjuk, az ismeretlenek és az egyenletek számát, és a generálás gombra kattintva a kiválasztott nagyságú mátrixot kapunk egy plusz, B oszloppal. A táblázatot feltöltjük az értékéekkel, majd kattintással kiválasztunk egy pivot elemet, és go -gomra kattintva pivotál. Kapunk hozzáfűzve az előzőhöz egy újabb táblázatot, a számolt eredményekkel. Itt ki tudunk választani egy újabb pivotelemet a folytatáshoz. A go gomb hatására egy-egy változóba elmentem a kiválasztott pivotelem koordinátáit, és egy tömbbe elmentem, hogy hanyadik sorból lett kiválasztva az első iterációban a pivot elem. Ez a végeredmény szempontjából lesz fontos, így tudom beazonosítani, hogy a végén a B oszlopban megmaradt értékek mely ismeretlenekhez tartoznak. Két másik boolean típusú tömb segítségével követem, hogy melyik sorból és melyik oszlopból választottam pivotelemet, melyiket vittem be a bázisba, honnan nem választhatok többet elemet. A program figyel erre, amennyiben egy olyan sorból vagy oszlopból választottunk pivotelemet, amit már korábban is választottunk, a go gomb megnyomására a program nem számol tovább. A pivotálást egy pivotalas nevű függvény végzi. Először másolatot készít a bemenetként kapott kétdimenziós tömbről. Ezután a pivotelem sorát elosztja a pivotelemmel, és az oszlopát a pivotelem mínusz egyszeresével. A többi elemet a téglalapszabály alapján számolja: Végül pedig a pivotelem helyére a pivotelem reciprokát írja. Minden pivotálás végén ellenőrizzük, hogy lehetséges e még pivot elemet választani. Ha már nem, akkor vége a feladatnak, és lefut a megallpitas függvény, és egy szövegdobozba írja a megoldást, amik a következők lehetnek: Nincs megoldás. Egy megoldás van, és ki is írja az X értékeket. Végtelen sok megoldás van.

5. fejezet

Tesztelés

A fejezetben be kell mutatni, hogy az elkészült alkalmazás hogyan használható. (Az, hogy hogyan kell, hogy működjön, és hogy hogy lett elkészítve, az előző fejezetekben már megtörtént.)

Jellemzően az alábbi dolgok kerülhetnek ide.

- Tesztfuttatások. Le lehet írni a futási időket, memória és tárigényt.
- Felhasználói kézikönyv jellegű leírás. Kifejezetten a végfelhasználó szempontjából lehet azt bemutatni, hogy mit hogy lehet majd használni.
- Kutatás kapcsán ide főként táblázatok, görbék és egyéb részletes összesítések kerülhetnek.

6. fejezet

Összefoglalás

Hasonló szerepe van, mint a bevezetésnek. Itt már múltidőben lehet beszélni. A szerző saját meglátása szerint kell összegezni és értékelni a dolgozat fontosabb eredményeit. Meg lehet benne említeni, hogy mi az ami jobban, mi az ami kevésbé jobban sikerült a tervezettnél. El lehet benne mondani, hogy milyen további tervek, fejlesztési lehetőségek vannak még a témával kapcsolatban.

Irodalomjegyzék

[1] James H Coombs, Allen H Renear, and Steven J DeRose. Markup systems and the future of scholarly text processing. *Communications of the ACM*, 30(11):933–947, 1987.

CD Használati útmutató

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

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

A CD lemezre mindenképpen rá kell tenni

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