



MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Új eljárási módok a szimulált hűtési eljárás eredményének javítására

Csizmár Roland
I. éves mérnök informatikus hallgató

Konzulens:
Tóth Lajosné dr. Tuzson Ágnes (PhD)
egyetemi docens
Analízis Tanszék

Miskolc, 2012

Tartalomjegyzék

1. Probléma felvetése	3
2. Szimulált hűtés algoritmus	4
2.1. Jelölések	4
2.2. Matematikai modell	4
3. A kitűzött feladat megoldása	6
3.1. A szimulált hűtés kivitelezése	6
3.2. Javítás új sorrendek vizsgálatával	8
3.3. Javítás az entrópia csökkentésével	11
3.4. Futtatási tapasztalatok, a hatékonyság elemzése	17
4. Köszönetnyilvánítás	18

1. Probléma felvetése

A rúdszabási feladat során különböző hosszúságú rendeléseket kell a lehető leghatékonyabban levágni a rendelkezésre álló készletelemekből. A megoldás szempontjából a rendeléselemeknek csak a hosszúsága érdekes. Az optimális megoldás eléréséhez meg kell találnunk azokat a rendelésem-készletelem csoportokat, és ezek sorrendjét, amit alkalmazva a legkevesebb nyersanyag veszteséget okozzuk. Ezt az állapotot megtalálhatjuk, ha előállítjuk a rendeléselemek összes permutációját, és kiválasztjuk közülük a legkisebb maradékot adót. Erről az állapotról csak akkor tudhatjuk biztosan, hogy a legjobb, ha az összes többit is ismerjük. Ezen permutációk előállítása és vizsgálata azonban már nem túl nagy elemszám-nál is irreálisan időigényes. Például 20 különböző darabnál is $\sim 2,4 \times 10^{18}$ számú állapotot kellene megvizsgálni. Ez még akkor is több mint 70.000 évig tartana, ha egy újabb állapot előállítása és vizsgálata csupán $1\mu s$ -ig tartana. Ismétlődő, egyenlő hosszúságú elemeknél könnyen kizárhatóak azok a permutációk, amiket már vizsgáltunk, de a kombinatív eljárások még így is túl lassúak ahhoz, hogy a gyakorlatban hatékonyan alkalmazhatóak legyenek. Szükség van tehát olyan eljárásokra, amelyek más logikát követve képesek rövid idő alatt megtalálni az optimális, vagy közel optimális állapotokat.

Ilyen algoritmus a meta-heurisztikus *szimulált hűtés* [1] is, amit először Scott Kirkpatrick és Vlado Cerny írt le a 80-as évek elején. Az eljárás a statisztikus termodinamika által vizsgált folyamaton alapul, ahol azt vizsgálják, hogy az anyag miként viselkedik alacsony hőmérsékleten: folyadék marad, vagy megszilárdul és kristályszerkezet alakít ki, esetleg üveghez hasonló, amorf állapotba kerül. A kísérletek során ahhoz, hogy kristály keletkezzen, magas hőmérsékletről folyamatosan lassuló sebességgel kell csökkenteni a hőmérsékletet. Az így kialakított kristályokat az anyag optimális állapotának tekintjük. Minél gyorsabban hűtjük az anyagot, annál valószínűbb, hogy csak közel optimális szerkezetet vesz fel, kristályszerkezete több hibát tartalmaz [1].

A szimulált hűtési eljárás szerint a kombinatív optimalizálási feladat elemeit egy anyag atomjaihoz hasonlítjuk. Az elemek optimalizáltságát egy célfüggvénnyel vizsgáljuk, és az ezáltal meghatározott értékek minimumát keressük. A ciklusok során egy-egy elem cseréjével próbálunk javítani a minta állapotán. Ez nem mindig sikerül, ennek ellenére bizonyos eséllyel tovább haladhatunk a rosszabb állapotok felé, mert a későbbi cserék során ezekből kiindulva akár egy jobb állapothoz is eljuthatunk. Így elkerülhetjük azt is, hogy az algoritmus megrekedjen egy lokális minimum ponton. A nem javító állapotok elfogadásának esélye a rendszer "hőmérsékletétől" függ. Minél hidegebb a rendszer, annál kisebb eséllyel fogadhatjuk el a nem javító állapotokat, hiszen a modell szerint ekkorra már közel járunk az optimumhoz.

Az algoritmus sztochasztikus viselkedése miatt nem garantált a globális minimum elérése.

A későbbiekben olyan konfigurációk és eljárások kerülnek bemutatásra, amikkel a kezdeti algoritmushoz képest növelhető a legjobb állapot megtalálásának esélye.

2. Szimulált hűtés algoritmus

2.1. Jelölések

A S1D-CSP leírásához szükséges jelölések:

i	rendelés sorszáma ($i=1, \dots, n$)
s_i	a rendelés i -edik elemének hossza
j	készlet sorszáma ($j=1, \dots, m$)
d_j	a készlet j -edik elemének hossza
r_j	a készlet j -edik elemének szabásakor keletkező veszteség hossza
n_i	s_i hosszúságú elemek rendelésben szereplő száma
R	a teljes megrendelés kielégítése során keletkezett veszteségek összege
x_{ij}	s_i hosszúságú rendelés elemek száma, melyek a készlet d_j eleméből kerülnek leszabásra
y_j	logikai érték, igaz ha d_j darab már fel lett használva a készletből, egyébként hamis
α	"hűtési" állandó
T	"hőmérséklet", valószínűségi változó
T_{start}	kezdeti "hőmérséklet"
T_{stop}	leállító feltétel, a "hőmérséklet" alsó határa

2.2. Matematikai modell

A modellt M.H.M.A. Jahromi, R. Tavakkoli-Moghaddam, A. Rezapour-Ziba és E. Givaki 2011-ben megjelent dolgozata [2] alapján fogalmazzuk meg.

$$R = \sum_{j=1}^m r_j \rightarrow \min R \quad (1)$$

úgy, hogy

$$\sum_{j=1}^m x_{ij} = n_i, \forall i \quad (2)$$

és

$$\sum_{i=1}^n x_{ij} \times s_i + r_j = d_j \times y_j, \forall j \quad (3)$$

$$y_j \in 0, 1$$

$$x_{ij}, s_i, r_j, d_j \in \mathbb{Z}^+ (i = 1, \dots, n, j = 1, \dots, m).$$

A modell célja a teljes veszteség minimalizálása (1). A (2) feltétel biztosítja a rendelések leszabhatóságát. A d_j készletelemen a szabás során keletkező veszteséget a (3) egyenlet fejezi ki. A probléma NP nehéz, ennek következtében a számítási idő a rendelések számával exponenciálisan nő és az optimális megoldás megtalálása is egyre bonyolultabbá válik.

A megrendelések szokásosan (s_i, n_i) párosokkal adottak, ahol s_i egy leszabandó darab hossza, n_i pedig a belőle szükséges mennyiség. Az algoritmus szempontjából nem lenne hatékony így ábrázolni a megrendelés elemeit, mert hiába azonos

a hossza két elemnek, a megoldás jósága függ ezen elemek szabásmintában elfoglalt helyétől. Emiatt a rendelést egy $\sum n_i$ hosszúságú tömbbel ábrázoljuk. Az optimalizálás során felhasználható raktárkészlettel is hasonlóképpen járunk el.

Az algoritmus futása során egy szabásmintát a vele szomszédos, tőle csak kissé különböző szabásmintákkal hasonlítunk össze. Ezeket a szabásmintákat nevezzük szomszédos állapotoknak.

Induljunk ki egy, a feltételeket kielégítő, rendezetlen szabásmintából. Egy rákövetkező szabásmintát előállíthatunk úgy, hogy a rendelést tartalmazó tömb két elemét véletlenszerűen felcseréljük. Ezt a szabásmintát szomszédos állapotnak nevezzük, mely csak a megelőző állapottól függ. A cserénél érdemes azonban olyan rendeleselemeket választani, melyek szabáskor nem esnek azonos készletelemre. Szomszédos állapot generálásakor követelmény, hogy a következő állapotot előállító függvény alkalmas legyen az összes lehetséges állapot előállítására. Az eljárás során egy speciális célfüggvénnyel vizsgáljuk a szabásminták jóságát. Az algoritmus sorban halad a készletelemeken és próbálja feltölteni azokat a leszabandó elemekkel. Ha már nem fér el több rendelés a vizsgált készletelemen, akkor a vágások után megmaradt hosszt hozzáadja az eddigi maradékokhoz és a következő elemre lép. A következő elem kiválasztásakor figyelni kell arra is, hogy a kiválasztott rendeleselem leszabható legyen belőle. Ha ez nem teljesül, akkor tovább kell lépni a készleten.

A megkapott értékek alapján eldönthetjük, hogy az újonnan generált állapot javít-e a maradékokon. Ha nem javít, korlátozottan még akkor is elfogadhatjuk új kiinduló állapotként. Erre azért van szükség, mert idővel elérhető egy olyan lokális minimum pont, amiből már egyszerű cserével nem következik jobb állapot. A rossz, vagy nem javító megoldások elfogadásával ezekből a pontokból lehet kilépni. Ilyenkor az éppen vizsgált állapot ugyan romlik, de lehetséges, hogy ebből a korábbinál kevesebb veszteséget adó szabásmintához jutunk.

Az algoritmus egyik paramétere a maximális Markov-lánc hossz, ami meghatározza, hogy az adott hőmérsékleten legfeljebb hányszor engedélyezett nem javító állapot elfogadása. Nem javító állapot elfogadása esetén a Markov-lánc hosszát 1-gyel növeljük. Ha a Markov-lánc hossza eléri az engedélyezett maximumot, akkor a (4) szerinti új hőmérsékleti szintre lépünk, és a lánc hosszát lenullázzuk. Az algoritmus előrehaladásával csökken a rossz minták elfogadásának esélye. A csökkenés mértéke az α változótól függ, melynek értékét a szakirodalom szerint [2] 0.85 és 0.95 között célszerű megválasztani, és így a lassú hűlést leíró képlet

$$T_{k+1} = \alpha T_k. \quad (4)$$

Minél lassabb a hűtés folyamata, az algoritmus - nagy elemszámú rendelés esetén - annál nagyobb eséllyel ér el jobb megoldást. Kevesebb rendeleselem optimalizálásánál gyorsabb hűtés is lehet hasonlóan hatékony.

Mivel nem tudjuk, hogy az algoritmus mikor jár éppen a globális minimumnál, meg kell szabni, hogy a hőmérsékleti szinteken maximum hányszor kereshetünk új megoldást. A végtelen ciklus elkerülése érdekében bevezetünk egy ciklusváltozót. A ciklusváltozó 0 értékről indul. Minden esetben eggyel nő, amikor nem véletlenszerűen fogadunk el egy szabásmintát, és 0 értéket vesz fel, ha egy javító

állapot kerül elfogadásra. Ezzel maximalizálhatjuk a próbálkozások számát, amikor az algoritmus már kellően megközelítette vagy elérte a globális minimumot, és így a jobb szabásminta megtalálásának az esélye is lecsökken.

2.2.1. Megoldás keresése, a fő ciklus

Az algoritmus során lépésenként 3 szabásmintát és a hozzájuk tartozó célfüggvényértéket vizsgáljuk. Ezek

- az eddig elért legjobb állapot,
- az aktuálisan lokális minimumot mutató állapot,
- és az újonnan generált (szomszédos) szabásminta.

Induláskor a három állapot megegyezik. A ciklus minden futásakor új szabásmintát készítünk és megvizsgáljuk, hogy jelent-e javulást a jelen állapotban minimumnak tekintett mintához képest. Ha igen, akkor elfogadjuk mint lokális minimum értéket, és tovább vizsgáljuk, hogy az eddig globális minimumnak tartott állapothoz képest is jobb-e. Ha ehhez képest is javít a rendszeren, akkor elfogadjuk az eddig ismert leghatékonyabb szabásmintának, és a ciklusváltozót nullázzuk. Amennyiben egyik feltétel sem teljesül a szabásmintára, a Metropolis-Hastings [1], [2] algoritmusnak megfelelően egy arányszámot állapítunk meg, ami meghatározza, hogy mekkora eséllyel fogadhatjuk mégis el

$$\lambda = e^{\frac{-\Delta}{T}}, ahol -\Delta = R_{uj} - R_{lokmin}. \quad (5)$$

Az (5) kifejezésből adódóan az elfogadás esélye a szabásminták célfüggvényértékének különbségétől és a jelenlegi hőmérséklettől függ.

3. A kitűzött feladat megoldása

3.1. A szimulált hűtés kivitelezése

3.1.1. Szomszéd generálás

Szomszédos állapotok generálásakor a kívánt hatást elérhetjük csupán 1 elem áthelyezésével is. Ilyenkor kiválasztunk egy rendeleselemet, majd áthelyezzük egy véletlenszerű pozícióba. Ezzel a módszerrel maximum két lépéssel elérhetőek azok a szabásminták, amiket 2 elem cseréjével állítunk elő, de már az első csere után is megfelelő mintát kapunk.

Az összeadás kommutatív tulajdonságából adódik, hogy az azonos készletelemre eső rendeleselemeket nem érdemes cserélni, mert az nem változtat a szabásmintán. Gyakorta kell sok azonos méretű rendeleselemmel dolgozni, ezért figyelni kell azt is, hogy a cserélni kívánt elemek különböző hosszúságúak legyenek.

3.1.2. Leállító feltétel

Az algoritmus indításakor a kezdeti szabásminta elkészülte után kiszámolható egy abszolút minimum érték a felhasznált készletelemek számára. Ha összeadjuk a készletelemek hosszát, akkor egy olyan szabásmintát feltételezünk, amin a készletelemekre eső maradék 0. A kezdeti szabásmintához szükséges készletelemekből addig vehetünk el egyet, mindig az utolsót, amíg azok összesített hossza nagyobb, vagy egyenlő, mint a készletelemek hosszának maradék nélküli összege. Az így kapott értéktől függ hogy a fő ciklust leállítjuk-e vagy sem. Ha ugyanis elértünk egy olyan szabásmintát, ami minimális számú készletelemet használ, akkor már nem szükséges tovább keresni.

3.1.3. Kezdeti hőmérséklet

A kezdeti hőmérsékletet érdemes úgy megválasztani, hogy az közel legyen a készletelemek átlagos hosszához. Mivel az egyes cserék során a javulás vagy romlás általában egy készletelemnyi, ezért az (5) képletben a kitevő számlálójának értéke csak ettől a hosszától függ. Célunk, hogy a hűtési folyamat elején minél többféle állapotot megvizsgáljunk, ezért érdemes olyan értéket választani, amivel kezdetben nagy elfogadási arányt érhetünk el. Ez akkor lehetséges, ha a kezdeti hőmérséklet a lehetséges célérték-függvény változások számszorosa. Ha egyenlő hosszúságú készletelemekkel dolgozunk, akkor a romlás mértéke megegyezik a készletelem hosszával. Ha például a kezdeti hőmérsékletet az átlag hosszúság kétszeresének választjuk, akkor

$$\lambda = e^{\frac{-x}{2x}} = e^{\frac{-1}{2}} \approx 0,60 \quad (6)$$

adódik, tehát 60% körüli eséllyel fogadhatunk el nem javító szabásmintát.

A leállító feltétel, T_{stop} is ennek megfelelően változik, azért hogy nagyságrendileg kövesse a kezdeti hőmérsékletet. Már a T_{start} hetedénél is olyan kis értékeket kapunk, hogy az elfogadás esélye elenyésző. Minél alacsonyabb a hőmérséklet, annál kisebb az esélye a nem javító szabásminta elfogadásának.

$$\lambda = e^{\frac{-7x}{x}} \Rightarrow e^{-7} * 100 \approx 0,1\%. \quad (7)$$

3.1.4. Markov-lánc hossz és a ciklusváltozó

A fejlesztés közben futtatott tesztek alapján a Markov-lánc maximális hosszát, és a ciklusváltozó felső határát a készlet méretétől függően érdemes megválasztani. A Markov-lánc esetében hatékony, ha a maximális értéke a megrendelések darabszámának kis egész számú gyöke. A ciklusváltozó korlátjának érdemes a darabszámhoz közeli, esetleg annak valamilyen kis szorzóval növelt vagy csökkentett mennyiségét felvenni.

3.2. Javítás új sorrendek vizsgálatával

A hűtési algoritmus leállásakor nem biztos, hogy a maradék globális minimumát biztosító szabásmintát kapjuk meg, ezért további javításra lehet szükség.

A következőkben két, külön-külön vagy akár kombináltan is alkalmazható módszer mutatok be a hűtési algoritmus által előállított megoldás javítására. A hűtési algoritmus leállásakor ismert egy olyan állapot, aminek elemeiről tudjuk, melyik készletelemre esnek, és ezen az elemen keletkező maradék is ismert. Ezt kihasználva a szabásmintát úgy rendezzük át, hogy az egyazon készletelemre eső rendeleselemek növekvő sorrendben kövessék egymást. Ezután, a szabásminta elejéről indulva, olyan elemeket keresünk a rákövetkező elemek közt, amelyek hossza nagyobb, mint a vizsgált elem, de kisebb vagy egyenlő, mint a vizsgált elemnek a készletelemen keletkező maradékkal megnövelt hossza

$$s_i < s_{i+offset} \leq s_i + r_j, (s_i \in d_j, offset \geq 0). \quad (8)$$

Ha találunk ilyen elemet, akkor kicseréljük a vizsgált elemmel úgy, hogy közben a készlet többi elemének a szabásmintája nem változik. Ehhez a szabásmintát már nem elég listaként tárolni, mert szükséges az egy készletelemre eső szabásminták megőrzése. Az így végrehajtott cserék miatt a szabásminta elején egyre csökkenni fognak a maradékok, a vége felé pedig nőni. A rendezés után lehetséges, hogy az utolsó készletelemeken összegyűlt maradékok kitesznek egy vagy több teljes készletelemet, így ezeket már nem szükséges hozzáadni az R teljes veszteséghez. A szabásminta elején a maradékok általában rövidebbek annál, hogy egy új rendelésben felhasználhatók legyenek, tehát az itt keletkező maradékok ténylegesen veszteségnek tekinthetők; míg a szabásminta végén leeső darabok nagyobb hosszúságuk miatt egy új rendelésben várhatóan felhasználhatók, azaz összhosszúságuk csökkentheti a későbbi teljes üzemi veszteséget.

3.2.1. Javító ciklus adatszerkezetei

Bevezetjük a minta megnevezést, mely alatt az egy készletelemre eső "szabásmintát" értjük. Szabásminta alatt továbbra is a teljes rendelést kielégítő szabásmintát értjük. A javításokhoz rögzítjük az egyes készletelemeken keletkező mintákat, a cserék során csak az érintett készletelemeken változtatunk. Negatív maradékot nem engedünk meg.

Pattern
+ pieces: list<int> + stockLength: int = 0 + remainder: int = INFINITY + entrophy: float = INFINITY + quality: float = 0 + lock: bool = false
+ Pattern() + add(orderLength: int) + move(piecePointer: list<int>::iterator, toPattern: Pattern*) + equals(otherPattern: Pattern*): bool - del(piecePointer: list<int>::iterator) - del(int)

Solution
- orderCuts: vector<int> - stock: vector<int> - solution: list<Pattern> - remainder: int - firstPattern: list<Pattern> iterator - entropia: float - entropiaMax: float - minOrderLength: int - minStockLength: int - limit: int - bestPatterns: list<Patterns> - counts: map<int, float>
+ Solution(inCuts: vector<int>, inStock: vector<int>) + sort() - analyze() - fillPattern(pivot: list<Pattern> iterator) - growPattern(pivot: list<Pattern> iterator) - getOptimalPatterns() - entropySort() - getRemainders()

A fent vázolt objektumokban tároljuk a szimulált hűtés után kapott állapotot, majd ezeken végezzük a javítást is. A *Solution* osztály konstruktora (1. Alg.) paraméternek az eddig ismert legjobb állapotot és a készletelemek listáját várja. Ezek alapján *Pattern* objektumokat képez úgy, hogy sorban halad a rendelés-elemeken és addig próbál újabb elemet hozzáadni az utolsó mintához, amíg az meg nem telik. Új *Pattern* objektum létrehozásakor annak konstruktora inicializálja a felhasznált készletelem hosszát (*stockLength*) és az azon keletkező veszteséget (*remainder*). Kezdetben ezek az értékek megegyeznek a következő, még nem használt készletelem hosszával. Új rendeléselem felvételekor a *Pattern*

Algorithm 1 Solution::Solution(*inCuts*, *inStock*)

```
1: orderCuts  $\leftarrow$  inCuts
2: stock  $\leftarrow$  inStock
3: stockIndex  $\leftarrow$  0
4: for cut in orderCuts do
5:   if solution.size = 0 or solution.last.remainder < cut then
6:     solution.push(next stock element)
7:   end if
8:   solution.last.add(cut)
9: end for
```

objektum *add* metódusa hozzáadja az eddig felhasznált elemek listájához (*pieces*) az új elem hosszát, miközben ugyanennyivel csökkenti a minta veszteségét is. A felhasznált elemek listája mindig rendezett kell legyen a későbbi keresések gyorsítása érdekében, ezért az új elemeket mindig a nekik megfelelő helyre, a náluk kisebbek után szúrjuk be.

3.2.2. Rendezés

Az elkészült Solution objektumon ezután a 3.2 pontnak megfelelő rendezést hajtunk végre. A cserék megtalálásához szükségünk van egy függvényre, ez a Pattern objektum *search* metódusa. A függvény paramétere a keresett elem maximális hossza. Ha talál cserére alkalmas elemet, visszatér annak mutatójával, egyébként egy hibát jelző értékkel, amit a futás során később értékelünk ki. Rendezéskor kétféleképpen tudunk javítani az eddigi állapoton. Elsősorban olyan elemeket keresünk a vizsgált szabásmintához, amik nem nagyobbak a minta maradékánál. Így egy gyors áthelyezéssel csökkenthetjük a vizsgált minta maradékát.

Algorithm 2 Solution::fillPattern(*pivot*:pointer to Pattern object)

```
1: if pivot.remainder > minOrderLength then
2:   iterator  $\leftarrow$  pivot.next
3:   while iterator  $\neq$  solution.end do
4:     if iterator.size = 0 then
5:       continue
6:     end if
7:     csere  $\leftarrow$  iterator.search(parameter : pivot.remainder)
8:     if valid(parameter : csere) then
9:       iterator.move(parameters : csere, pivot)
10:      iterator  $\leftarrow$  iterator.previous
11:     end if
12:     iterator  $\leftarrow$  iterator.next
13:   end while
14: end if
```

Keresés előtt érdemes ellenőrizni, hogy a maradék kisebb-e, mint a rendeléselemek legkisebbike. Ha kisebb, akkor nem szükséges folytatni, mert nem létezik olyan

elem, amit hozzá tudnánk adni a mintához. A cseréket a Pattern objektum *move* metódusával hajtjuk végre, ami az első paraméterben kapott rendeleselemet áthelyezi a második paraméterben kapott szabásmintába. Az algoritmus futása során lehetséges, hogy egy szabásminta összes elemét át tudjuk helyezni más mintákba. Az így kiürült szabásmintákat a vizsgálatból ki kell hagyni, nem kell megpróbálni újra feltölteni őket. Ezek a minták teljes készlethosszukkal csökkentik a rendszer veszteségét.

Miután végigértünk a mintákon, és már nem lehetséges további javítás ezzel a módszerrel, még van lehetőségünk olyan cserékre, amik csökkenthetik a rendszer elején a maradékokat. A szabásminták rendeleselemeihez olyan elemeket keresünk a későbbi objektumokban, melyeknek a hossza nem nagyobb, mint a vizsgált rendeleselem és a hozzá tartozó minta maradéka. A *valid(swap)* feltétel akkor

Algorithm 3 Solution::growPattern(pivot:pointer to Pattern object)

```

1: searchPattern  $\leftarrow$  pivot.next
2: searchPiece  $\leftarrow$  pivot.pieces.first
3: while searchPiece  $\neq$  pivot.pieces.last do
4:   while searchPattern  $\neq$  solution.end do
5:     if searchPattern.size = 0 then
6:       continue
7:     end if
8:     swap  $\leftarrow$  searchPattern.search(parameter : pivot.remainder + searchPiece)
9:     if valid(parameter : swap) then
10:      pivot.move(parameters : searchPiece, searchPattern)
11:      searchPattern.move(parameters : swap, pivot)
12:     else
13:       searchPattern  $\leftarrow$  searchPattern.next
14:     end if
15:   end while
16:   searchPiece  $\leftarrow$  searchPiece.next
17: end while

```

lesz igaz, ha az elemek anélkül felcserélhetők, hogy valamelyik készletelemen negatív maradék keletkezne. A maradékokat tekintve ez a fajta rendezés hasonlatos a buborékrendezéshez, mert az egyes cserékkel a kisebb maradékú minták a lista eleje felé, a nagyobb maradékú minták pedig a lista vége felé indulnak el.

3.3. Javítás az entrópia csökkentésével

3.3.1. Osztályozás

Lehetséges, hogy a fent ismertetett rendezések után kapott állapot tovább javítható. Javulás csak akkor következik be, ha sikerül kiüríteni egy teljes készletelemre vonatkozó szabásmintát. Ha ez nem történik meg, akkor a maradékok összege se változik. A megoldás optimális voltáról csak bizonyos esetekben tudunk meggyőződni. Például biztosan tudjuk, hogy a felhasznált készletelemek

hossza nem lehet kisebb, mint a rendeleselemek hossza. Ha a felhasznált készlet-elemek számát eggyel csökkentjük és az ezután kiszámolt teljes hossz kisebb lesz a szabásminta hosszánál, akkor már nem lehetséges tovább optimalizálni úgy, hogy R -t csökkentsük. Azonban még ilyenkor se biztos, hogy nem érdemes tovább folytatni a keresést. Előnyös állapot az, amikor a nagyobb maradékok a lehető legkevesebb készletelemre csoportosulnak, mert az így kapott minták maradékát a későbbi megrendelések során is felhasználhatjuk. Ahhoz, hogy a rendszert tovább javíthassuk, találni kell egy olyan mérőszámot, ami megmutatja, hogy egy szabásminta mennyire jó, avagy mekkora eséllyel optimális. A termodinamikusan analógiánál maradva ismert, hogy egy rendszer hőmérsékletének növekedésével annak entrópiája, rendezetlenségének mértéke is nő. Ez a szabály alkalmazható a szabásmintára is. A megrendelésekben általában több azonos hosszúságú elem szerepel. Ha találunk olyan mintát, amiben minél többféle elem felhasználásával, minél kevesebb ismétlődéssel elérhető a minimális maradék, az a többi ugyanilyen elemre is felhasználható. Ezért, a legkisebb maradékot eredményező szabásminta várhatóan csak kis számú, jól megválasztott minták ismétlődésével érhető el.

Ahhoz, hogy az egyes készletelemekre eső mintákat vizsgálni tudjuk, ismernünk kell a teljes rendszer entrópiáját, ezért az algoritmus elején egy előzetes elemzést végzünk. Az elemzés során megállapítjuk, hogy milyen relatív gyakorisággal jelenik meg egy adott hosszúságú megrendeleselem az egyes készletelemeken. Kiszámítjuk azt is, hogy a megrendelés mennyi információtartalmat hordoz, mekkora az entrópiája. Az így megkapott érték a minta maximális entrópiája. Egy rendelés halmaz rendezettsége tehát megállapítható annak Shannon-entrópiájával. Feltételezzük, hogy a szimulált hűtés során a készletelemeken legalább egy olyan szabásmintát sikerült találni, aminek a maradéka elfogadhatóan kicsi. A továbbiakban ez alapján próbájuk meg javítani a teljes szabásmintát. Először "félretesszük" azokat a készletelemeket, melyekre eső szabásminták maradéka kisebb, mint az egymástól legkevésbé különböző, de nem azonos két rendeleselem hosszának különbsége. A "félretett" mintákat referencia-mintának nevezzük. A "maradék" készleten, melyet a továbbiakban szűkített készletnek nevezünk, tulajdonképpen egy szűkített megrendelést - a kiinduló mintánál hatékonyabban - szándékozunk kielégíteni.

A különböző minták egymáshoz viszonyított "jóságát" az entrópiájuk különbségével jellemezzük.

Az egyes minták entrópiáját a rájuk eső rendeleselemek relatív gyakoriságából számítjuk ki. Legyen C az egy készletelemre eső rendeleselemek száma, és $\vec{c}$ vektor, ami az azonos hosszúságú elemek darabszámát tárolja. A mintából $p_i = \frac{c_i}{C}$ valószínűséggel választható ki bármelyik elem. Az egyes minták entrópiája a Shannon-féle képlet szerint

$$\sum_i -p_i \times \log_2 p_i. \quad (9)$$

Az egyes készletelemekre eső minták entrópiájának ismeretében kiszámíthatjuk a szűkített készlet elemei és a hozzájuk entrópia szempontjából legközelebb eső referencia-minták entrópiájának különbségét. Ezt a különbséget minden szűkített készletbeli elemnél nyilvántartjuk, mint minősítő értéket. A minősítő érték megmutatja, hogy egy minta mennyire különbözik a hozzá leginkább hasonlító

referencia-mintától, azaz milyen bonyolult lenne átalakítani olyanná.

Ezután a szűkített készlet elemeinek listáját átalakítjuk, elsődlegesen ezen minősítő érték szerint csökkenő sorrendbe, egyenlőség esetén pedig maradék szerint csökkenő sorrendbe rendezve. Rendezés közben a referencia-elemeket a lista elejére gyűjtjük. Megkeressük az első olyan elemet, ami még nem eléggé optimalizált. Ennek az elemnek lesz a legnagyobb minősítő-értéke, tehát a referencia-elemek bármelyikére ez hasonlít a legkevésbé, és a hozzá tartozó maradék is maximális. Vesszük a minta legnagyobb elemét, és a későbbi -jobban rendezett, kisebb maradékú - minták elemei közül keresünk egy rendeleselemet, amivel kicserélhetjük. A megfelelő elem úgy cserélhető, hogy

- a csere során egyik mintán se keletkezik negatív maradék,
- és a csere elem kisebb, mint a cserélendő elem.

Ezzel lehetőséget biztosítunk olyan cserére is, ami eddig nem volt lehetséges, hogy egy későbbi mintából olyan elemet vegyünk ki, ami kisebb mint a vizsgált mintából kiválasztott rendeleselem. A csere azon a mintán történik, ami leghibásabb, annak javítása jó esetben a teljes szabásminta javítását eredményezi. A vizsgált minta maradéka ugyan nő, de ebben az esetben ez nem számít, mert közben lehet, hogy változtatunk a rendezetlenségén, és így jobban fog hasonlítani valamelyik referencia-mintához. Az entrópia alapú rendezés biztosítja, hogy az így kiválasztott elemmel a lehető legjobb cserét hajtsuk végre.

Az eddig ismerttetett cserélő és rendező eljárásokból ciklust képezve kapjuk meg a javító algoritmust. A ciklus során először olyan mintákat keresünk, amelyekbe egy elem áthelyezésével a maradék helyére lehet másik rendeleselemeket tenni. Ezután azok a cserék következnek, amikhez két elem szükséges, majd az így kapott stabil állapotot vizsgáljuk meg azt entrópielemző függvénnyel, és újból rendezzük. A minősítő érték szerint rendezett listán végrehajtjuk a fentebb leírt cserét, ezzel felborítva a sorrendet, és megfelelő alapot nyújtva a későbbi átalakításoknak. Ezután a ciklus elejére ugrunk. A ciklus addig tart, amíg el nem érünk egy olyan állapotot, ami sorozatosan nem vezet jobb szabásmintához, a cserék után is ugyanaz a szabásminta jön létre.

3.3.2. Megvalósítás

Az algoritmus teljes felírásához szükséges eljárások pszeudokódjaihoz az alábbi megjegyzéseket fűzöm:

- A 4. algoritmus 2-9 soraiban megkeressük a legkisebb maradékkal rendelkező mintát és az összes mintán feloldjuk a zárolást. Lehetséges, hogy a cserék során kialakul az eddig ismert legjobb mintánál jobb minta, ezért szükséges újra és újra megvizsgálni és zárolni a legjobbakat.
- A 13-20 sorokban megkeressük és zároljuk a legjobb szabásmintákat, így jelölve ki azokat referencia-mintának.

Algorithm 4 Solution::getOptimalPatterns

```
1: searchPattern  $\leftarrow$  solution.begin
2: best  $\leftarrow$  searchPattern
3: while searchPattern  $\neq$  solution.end do
4:   compute searchPattern's entropy
5:   searchPattern.lock  $\leftarrow$  false
6:   if searchPattern.remainder < best.remainder then
7:     best  $\leftarrow$  searchPattern
8:   end if
9: end while
10: clear the bestPatterns list
11: push best to bestPatterns
12: searchPattern  $\leftarrow$  solution.begin
13: while searchPattern  $\neq$  solution.end do
14:   if  $|best.remainder - searchPattern.remainder| < limit$  then
15:     if searchPattern  $\notin$  bestPatterns then
16:       push searchPattern to bestPatterns
17:     end if
18:   end if
19:   searchPattern  $\leftarrow$  searchPattern.next
20: end while
```

Algorithm 5 Solution::entropySort

```
1: searchPattern  $\leftarrow$  solution.begin
2: while searchPattern  $\neq$  solution.end do
3:   ptrBestPattern  $\leftarrow$  bestPatterns.begin
4:   candidate  $\leftarrow$  ptrBestPattern
5:   while ptrBestPattern  $\neq$  bestPatterns.end do
6:     bestEntropyDif  $\leftarrow |ptrBestPattern.entropy - searchPattern.entropy|$ 
7:     candidateEntropyDif  $\leftarrow |candidate.entropy - searchPattern.entropy|$ 
8:     if bestEntropyDif < candidateEntropyDif then
9:       candidate  $\leftarrow$  ptrBestPattern
10:    end if
11:    ptrBestPattern  $\leftarrow$  ptrBestPattern.next
12:  end while
13: searchPattern.quality  $\leftarrow |candidate.entropy - searchPattern.entropy|$ 
14: if searchPattern.quality = 0 then
15:   if  $|searchPattern.remainder - bestPatterns.begin.remainder| < limit$  then
16:     searchPattern.lock  $\leftarrow$  true
17:   end if
18: end if
19: remove empty patterns
20: sort patterns according to sort_by_entropy
21: end while
```

- Az 5. algoritmus 4-12 soraiban megkeressük a vizsgált sorhoz entrópia szempontjából legközelebb eső referencia-mintát. A keresés szempontjából mindegy, hogy a két minta közti különbség pozitív vagy negatív.
- Ezután megvizsgáljuk, hogy a minta elég kis maradékkal rendelkezik-e ahhoz, hogy elfogadhasuk referencia-mintának; ha igen, akkor zároljuk a következő rendezésekre.
- Ha menetközben egy minta kiürült – már nem tartozik hozzá egy rendelés-elem se – akkor töröljük. Végül rendezzük a mintákat entrópia szerint a 6. algoritmus által leírt függvény alapján.

Algorithm 6 *sort_by_entropy(p1, p2)*

Require: *p1, p2* Pattern type objects

Ensure: boolean variable, describing *p1 - p2* relation

```

1: if p1.lock + p2.lock = true then
2:   return p1.lock > p2.lock
3: else if p1.quality ≠ p2.quality then
4:   return p1.quality > p2.quality
5: else
6:   return p1.remainder > p2.remainder
7: end if

```

- A 6. algoritmus először megvizsgálja, hogy a minták közül az egyik zárolt-e. Ha igen, és a másik minta nincs zárolva, akkor a zároltat helyezi előbbre. Amennyiben egyik se zárolt, a minősítő-értékük dönt. Ha ez alapján se lehet dönten, akkor a nagyobb maradékú kerül előbbre.

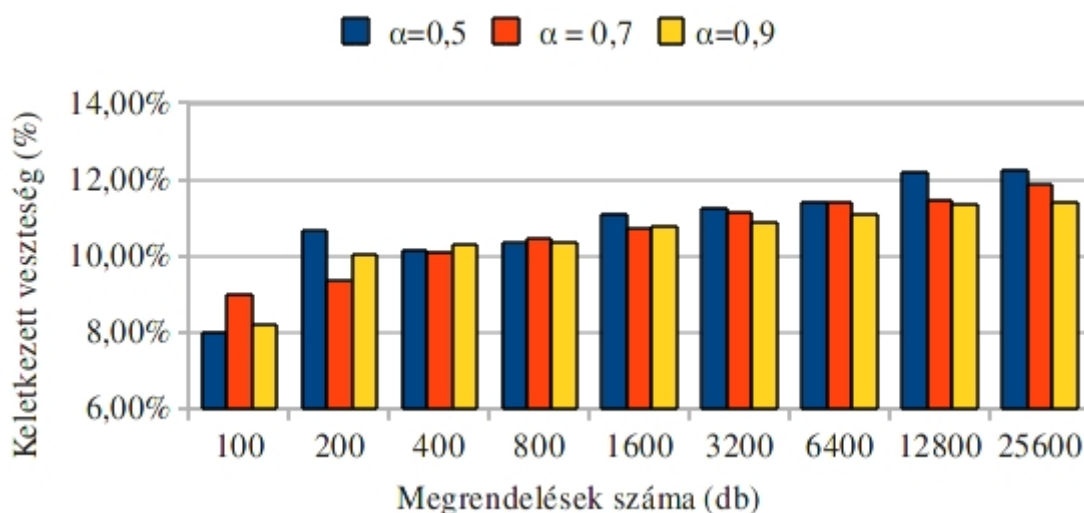
3.3.3. A teljes javító algoritmus

Algorithm 7 sort

```
1:  $minSumEntropy \leftarrow \infty$ 
2:  $repeat \leftarrow remainder$ 
3: while  $repeat > 1$  and  $remainder \geq minStockLength$  do
4:   if  $minSumEntropy \leq entropia$  then
5:      $repeat = repeat * 0.1$ 
6:   else
7:      $repeat = remainder$ 
8:      $minSumEntropy \leftarrow entropia$ 
9:   end if
10:   $searchPattern \leftarrow firstPattern$ 
11:  for all  $searchPattern \in solution$  and  $searchPattern.lock \neq true$  do
12:    call  $fillPattern$  with  $searchPattern$ 
13:    call  $growPattern$  with  $searchPattern$ 
14:  end for
15:  call  $getOptimalPatterns$ 
16:  call  $entropySort$ 
17:   $firstPattern \leftarrow$  first not locked pattern
18:  if  $firstPattern \neq solution.last$  then
19:     $searchPattern \leftarrow firstPattern.next$ 
20:    while  $searchPattern \neq solution$  do
21:       $swap \leftarrow$  call  $searchPattern.searchStrict$  with  $firstPattern.pieces.last$  and  $firstPattern.remainder$ 
22:      if  $valid(swap)$  then
23:        move  $swap$  to  $firstPattern$ 
24:        move  $firstPattern.pieces.last$  to  $searchPattern$ 
25:        break
26:      end if
27:       $searchPattern \leftarrow searchPattern.next$ 
28:    end while
29:  end if
30:  compute  $remainder$  again
31: end while
```

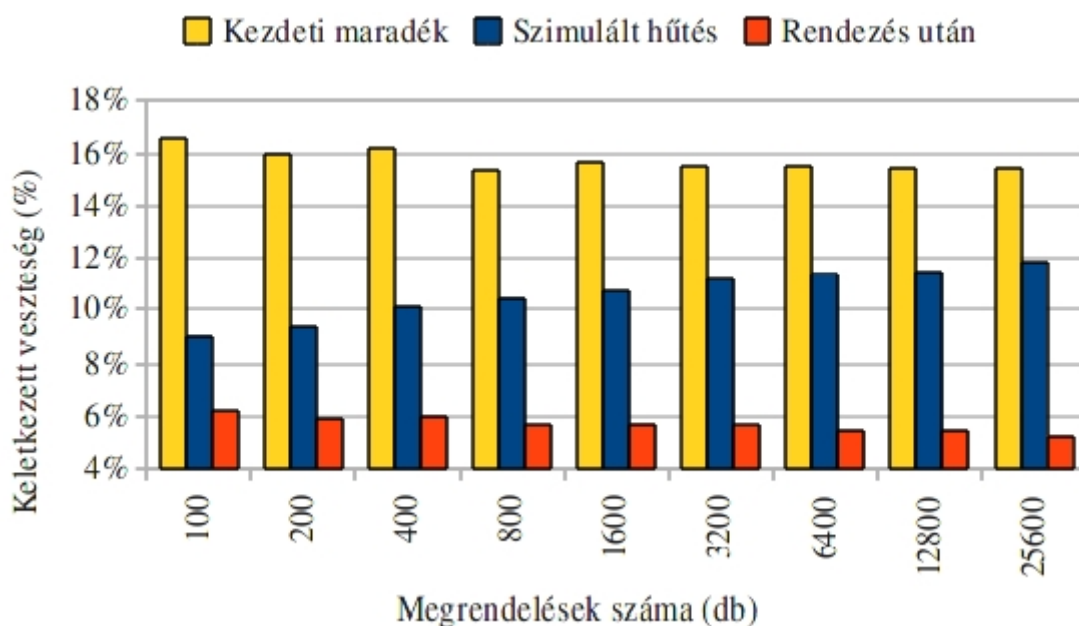
3.4. Futtatási tapasztalatok, a hatékonyság elemzése

Az elkészült program egy 1800 Mhz-es processzorral és 875Mb memóriával rendelkező számítógépen került tesztelésre. Az algoritmus minden esetben 9 különböző méretű megrendelésen futott le. A rendelések alapvetően az elemek számában tértek el. A rendelések elemeinek számát a $2^n \times 100$ sorozat elemei adják. A futáseredményeket az alábbi grafikonokon szemléltetem.



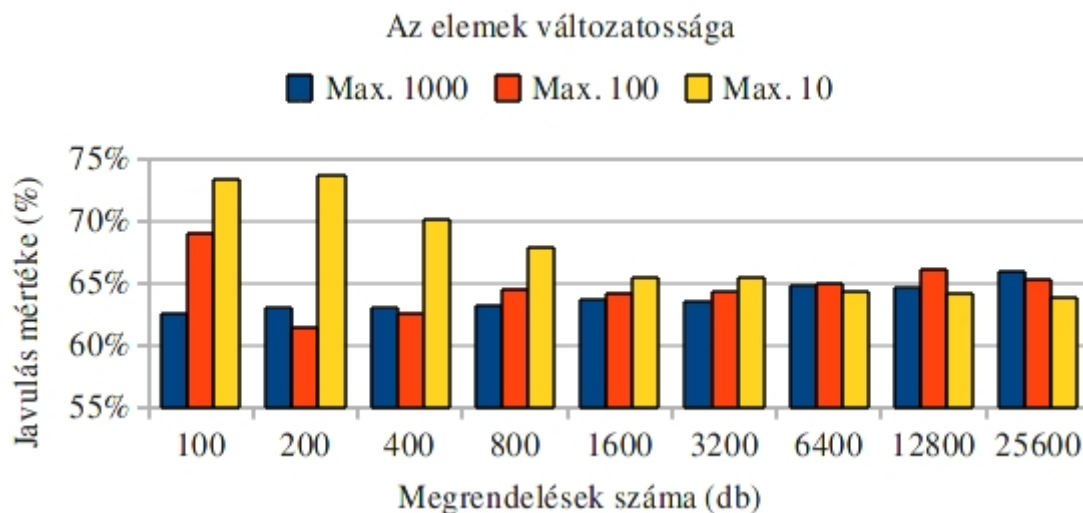
1. ábra.

A szimulált hűtés hatékonysága különböző hűtési állandók mellett



2. ábra.

Az eljárások összehasonlítása a maradék szempontjából



3. ábra.

A javító eljárások hatékonysága a szimulált hűtéshez viszonyítva

A 3. ábra a javító eljárások eredményességét mutatja, ha inputként a szimulált hűtési eljárás outputját vesszük. Az elemek változatosságán a különböző hosszúságú rendeléselemek számát értjük.

Összegzésként az alábbiak állapíthatók meg:

- Kis elemszámú megrendelés esetén a hűtési állandó megválasztása kevésbé befolyásolja az eredményt. Nagyobb elemszám esetén a hűtés lassításával javuló eredmények adódnak (1. ábra).
- A szimulált hűtés ($\alpha = 0,7$) eredménye a bevezetett javító eljárásokkal minden esetben lényegesen javult: kis elemszám esetén kb. 50%-kal, míg nagy elemszámok esetén 60-70%-kal csökkent a veszteség aránya (2. ábra).
- Kis elemszám esetén az elemek változatossága nagy hatással van a javító eljárások szimulált hűtéshez viszonyított hatékonyságára. Nagy elemszám esetén ez a hatás úgy tűnik, elenyészik.

4. Köszönetnyilvánítás

A bemutatott TDK dolgozat a TÁMOP-4.2.1.B-10/2/KONV-2010-0001 jelű projekt részeként az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Továbbá köszönet jár Tóth Lajosné Dr. Tuzson Ágnesnek a dolgozat elkészítése közben nyújtott segítségéért.

Irodalomjegyzék

- [1] S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi **Optimization by Simulated Annealing**, SCIENCE, 1983. Vol. 220. , pp. 671-681.
- [2] M.H.M.A. Jahromi , R. Tavakkoli-Moghaddam , E. Givaki , A. Rezapour-Ziba, **A SIMULATED ANNEALING APPROACH FOR A STANDARD ONE-DIMENSIONAL CUTTING STOCK PROBLEM** INTERNATIONAL JOURNAL Of ACADEMIC RESEARCH, 2011. Vol. 3., Part II., pp. 353-358.