Alkalmazott Matematikai Lapok **27** (2010), 123-134.

EGY ÚJ FELADAT: LÁDAFEDÉS SZÁLLÍTÁSSAL ÉS ENNEK MEGOLDÁSA ALGORITMUSOK EVOLÚCIÓJÁVAL

BENKŐ ATTILA, DÓSA GYÖRGY

A cikkben egy új feladatot definiálunk, amelyet „*Ládafedés szállítással*"-nak nevezünk. A feladat egyik jellegzetessége, hogy nem csak jó, hanem „*jó és gyors*" pakolást, vagy fedést keresünk. Néhány algoritmust adunk meg ilyen típusú (nagyon nehéz) feladatok megoldására, és egy új módszert is bemutatunk, amelyet „*Algoritmusok evolúciójá*"-nak nevezünk. Ezen a következőt értjük: Definiálunk egy algoritmuscsaládot, amely képes megoldani a feladatot, utána egy szomszédsági struktúrát ezen algoritmusok között, majd egy metaheurisztikát használunk (ebben a cikkben szimulált hűtést) a legmegfelelőbb (legjobb megoldást adó) algoritmus kiválasztására. Számítógépes tesztek segítségével demonstráljuk a módszer hatékonyságát.

1. Bevezető

Egy új problémával foglalkozunk, amit „*Ládafedés szállítással*"-nak nevezünk. Ezen típusú feladatok esetében tárgyakat pakolunk ládákba, majd a lezárt ládákat (ahova már nem akarunk további tárgyakat pakolni) elszállítjuk. A feladatot a [8] cikkben definiáltuk, és megadtuk annak néhány lehetséges változatát. Itt most a lehetséges változatok közül csak egyet vizsgálunk. Ezt a későbbiekben pontosan definiáljuk, elöljáróban csak annyit, hogy a fedett (és elszállított) ládákért pénzt kapunk, és ezt a profitot maximalizáljuk.

A tisztán ládapakolási feladat (BP) esetén adott a tárgyak mérete, a tárgyakat be kell pakolni minimális számú ládába, úgy, hogy a ládákba pakolt tárgyak összmérete nem haladhatja meg a láda kapacitását, amit 1-nek szokásos választani. A ládafedési feladat (BC) esetén a ládát fedettnek tekintjük, ha a ládába pakolt tárgyak összmérete nem kisebb, mint a láda kapacitása (megint általában 1), és annyi ládát akarunk fedni, amennyit csak lehetséges. Közismert tény [3], hogy mindkét probléma megoldása NP-nehéz. Ez azt jelenti, hogy az optimális megoldás megtalálásához általában exponenciálisan sok lépés szükséges. Sok esetben nem áll rendelkezésünkre ennyi idő, viszont megelégszünk a feladatnak elég jó, közel-optimális megoldásával is.

Amikor a tárgyak egyesével jönnek, és amint megjelenik a következő tárgy, azonnal kénytelenek vagyunk azt valamely ládába pakolni, vagyis az online esetben, általában nem lehet olyan jó algoritmust konstruálni, mint az offline esetben, amikor előre ismerjük a tárgyakat.

Vegyük például a BC feladatot megoldó Duál Next Fit online algoritmust. Ez egyszerre csak egyetlen nyitott ládát használ, a következő tárgyat mindig az éppen nyitott ládába helyezi, amíg a ládába helyezett tárgyak összmérete kisebb mint a láda mérete. Amint megtelt a láda, lezárja a ládát, és egy új ládát nyit meg. Ennél az algoritmusnál a felhasznált ládák száma (a legrosszabb esetben) kétszer több is lehet, mint az offline optimális megoldás esetén.

Ebben a cikkben az új feladatnak csak az online esetével foglalkozunk, vagyis itt is feltesszük majd, hogy a tárgyak egyesével érkeznek. A tárgyakkal ládákat akarunk fedni (vagyis telepakolni), de a célfüggvényünk most nemcsak a fedés *jóságától*, de a fedési eljárás *gyorsaságától* is függ. Tehát, egyszerre „*jó és gyors*" fedést keresünk.

Számos módon lehet ilyen problémát definiálni: Egyik lehetőség lenne egy K méretű puffer használata, (lásd, ütemezési feladat esetén [1, 2]), ahol a puffer mérete, vagy a pufferben tárolt tárgyak száma növeli a célfüggvényt: bünteti a várakozást. Ebben a cikkben egy másik módot választunk: a célfüggvény „büntetése" a megnyitott ládák száma alapján történik, hiszen egyszerre több megnyitott láda kezelése több időt igényel.

Létezik néhány publikáció az „Ütemezés szállítással" témakörében, (lásd: [4], és az ebben hivatkozott cikkeket), de a ládapakolás vagy ládafedés szállítással kombinált változatát korábban még nem vizsgálták.

A cikk második fejezetében megadjuk a vizsgált feladat pontos definícióját, továbbá megvizsgáljuk néhány jellegzetes tulajdonságát.

A harmadik fejezetben bemutatunk néhány természetesen adódó algoritmust, a negyedik fejezetben definiáljuk az algoritmusok evolúcióját (EOA), és ennek hatékonyságát demonstráljuk. Néhány kiegészítő megjegyzéssel zárul a dolgozatunk.

2. A probléma definíciója

Számos lehetőség van, hogy definiáljuk ezt a problémát: „Ládafedés szállítással", most ezekből csak egy lehetőséget választunk, ebben a cikkben csak ezzel foglalkozunk. A többi érdekes, ill. fontos változat vizsgálata további kutatás tárgya marad. A problémánk pontos megfogalmazása a következő:

Egyenként érkeznek a tárgyak, az i-dik tárgy mérete legyen $p_i > 0$. Adott továbbá egy $K > 0$ egész szám, egyszerre legfeljebb K számú láda lehet nyitva. A ládák kapacitása 1. Amint megérkezik egy tárgy, rögtön be kell hogy pakoljuk egy ládába. A tárgyat tehetjük egy már megnyitott ládába, vagy nyithatunk egy új ládát és a tárgyat ebbe az újonnan nyitott ládába is pakolhatjuk, de amint mondtuk, egyszerre legfeljebb csak K számú láda lehet nyitva. Amint megtelik egy láda (a láda megtelt, vagy más szóval fedett, ha a ládába pakolt tárgyak méretének összege legalább 1), azt azonnal elszállítjuk. A célfüggvény egy $G : \{1, \ldots, K\} \to R$ haszon-függvénnyel van megadva a következőképpen: ha éppen k a nyitott ládák száma (ahol $1 \leq k \leq K$) amikor egy láda megtelik, $G(k)$ profitot kapunk a láda

fedéséért. A G haszon-függvényről feltesszük, hogy pozitív, monoton nem növekvő függvény. Ebben a megközelítésben azt modellezzük, hogy több nyitott láda kezeléséhez több időre van szükségünk, hogy eldöntsük, hova pakoljuk az aktuális tárgyat. A cél a teljes haszon maximalizálása. (Ha végül marad fedetlen láda, azért nem jár pénz.) A tárgyak számát n-el jelöljük, de ezt előre nem ismerjük, csak amikor kiderül, hogy nem érkezik több tárgy.

Valós-életbeli alkalmazáshoz álljon itt a következő példa: egy kisebb konzervgyárban kézzel pakolják a gyümölcsöt dobozokba, a gyümölcs egy ablakon keresztül érkezik, egy rakodómunkás van az ablak túloldalán, aki ezt a feladatot végzi. Minden ládába legalább s összsúlyú gyümölcsöt kell pakolni, ennél többet szabad, de nyilván nem érdemes sokkal többet pakolni, mint amennyit muszáj.

Természetes azt feltételezni, hogy ez az ember nem tud túl sok nyitott ládát egyszerre kezelni, továbbá minél több nyitott ládával dolgozik, ez annál több időt igényel tőle, és a feladat egyre bonyolultabb lesz számára. Tehát a következő észrevételeket tehetjük:

Egyrészt, igyekeznie kell minél jobb fedést generálnia (vagyis mindegyik láda legyen telepakolva, de ne túlságosan: a ládába pakolt gyümölcsök összsúlya legyen legalább s, de azt ne nagyon haladja meg), ami nem könnyű feladat, minél kevesebb nyitott láda van, annál nehezebb. Másrészt mivel a gyorsaság is számít, érdeke, hogy egyszerre lehetőleg kevés számú láda legyen nyitva, vagyis kevés lehetőség közül kelljen választania.

Az előbbi két érdek egymásnak ellentmond, a dobozba pakoló személy célját ezért úgy szimulálhatjuk, hogy minden fedett ládáért pénzt kap (emiatt érdemes sok ládát telepakolnia, vagyis jó fedést csinálnia), de másrészt ahogy a nyitott ládák száma növekszik, egyre kevesebb pénzt kap egy-egy fedett ládáért (vagyis érdemes kevés nyitott ládával dolgoznia).

Az online algoritmusok hatékonyságát rendszerint versenyképességi analízissel mérik. Ez azt jelenti, hogy egy A online algoritmus által kapott $C_A(I)$ célfüggvényértéket (ahol I-vel jelölik az inputot) összehasonlítják (elosztják) az offline optimum $OPT(I)$ értékével. Maximalizálási feladatok esetén (ahogy a mi esetünkben is) a $C_A(I)/OPT(I)$ hányados infimumát (ahol az infimumot tetszőleges I inputon vesszük) az A algoritmus versenyképességi hányadosának nevezzük. Mit értünk „offline" feladaton és mit „offline optimum"-on? Offline feladatok esetén mindig feltételezzük, hogy az inputra vonatkozó összes információ előre ismert. Ha az (ismert) tárgyakat az offline feladat esetén tetszőleges sorrendben szabad pakolni a ládákba, akkor a G haszon-függvénynek semmi szerepe nincs.

(Ha a tárgyakat előre ismerjük, akkor – exponenciális időben – megadható az optimális fedés. Ezt a fedést egy lista szerint is el lehet készíteni – amelyik lista persze általában más lesz, mint az adott L lista – úgy, hogy egyszerre csak egy ládát kell nyitva tartani.)

Ezért úgy definiáljuk az offline modellt, hogy előre ismerjük a bemenetre vonatkozó összes információt, de a tárgyakat az adott L lista szerinti sorrendben kell a ládákba pakolnunk (abban a sorrendben, ahogy azok valójában, online módon érkeznek).

Természetesen ilyen offline optimális megoldásnak léteznie kell. Bármelyik pillanatban, amikor egy új tárgy érkezik, véges sok lehetőség közül lehet választani (legfeljebb K számú ládába lehet a következő tárgyat pakolni). A mindent összevetve is véges számú lehetőség között léteznie kell olyan megoldásnak, ami a legjobb értékét adja a célfüggvénynek. Természetesen az offline-optimum nemcsak a tárgykészlettől függ, de az adott L listától is és a G haszon-függvénytől is.

Bármely véges L tárgy-lista és G haszon-függvény esetén legyen $C_A(L,G)$ egy A algoritmus által kapott megoldási értéke. Ezt hasonlítjuk össze az offline-optimális megoldással, amit $OPT(L,G)$-vel jelölünk. Ekkor azt mondjuk, hogy az A algoritmus ρ-kompetitív, $(0 \leq \rho \leq 1)$ ha

$$\frac{C_A(L,G)}{OPT(L,G)} \geq \rho$$

teljesül bármely L, G esetén. Az előbbi ρ számok maximumát az A algoritmus versenyképességi hányadosának nevezzük. Másrészt, tegyük fel, hogy az L sorozat és G haszon-függvény esetén tetszőleges A online algoritmusra

$$\mu \geq \frac{C_A(L,G)}{OPT(L,G)}$$

teljesül valamilyen μ számmal, az ilyen μ számok legkisebbikét a feladat felső korlátjának nevezzük. Egy algoritmus optimális, ha a ρ versenyképességi-hányadosa megegyezik a feladat μ felső korlátjával.

3. Néhány (természetesen adódó) algoritmus

Ebben a fejezetben először megadjuk néhány klasszikus algoritmus természetes adaptációját. Először tekintsük a Duál Next-Fit (röviden DNF) algoritmust. DNF mindig csak egy ládát tart nyitva, és amint a láda megtelik, elszállítjuk. Az algoritmikus leírás az alábbi:

DNF algoritmus

– A következő tárgyat mindig a nyitott ládába pakoljuk. Csak akkor szállítjuk a ládát, ha az fedett lesz, ekkor egy új ládát nyitunk a további tárgyak számára. Ha már nem jön több tárgy, az algoritmus megáll.

Ezen algoritmus alkalmazása esetén csak egy lehetőség van a soron következő tárgy pakolásához. Azonban ez az algoritmus mégis optimális tud lenni abban a speciális esetben, ha az elszállított ládák után kapott haszon nulla, ha több mint egy láda van egyszerre nyitva. Továbbá optimális abban az esetben is, ha a tárgyméretek majdnem egyformák, a következő lemma szerint:

3.1. Lemma. *Tegyük fel, hogy* $1/m \leq p_i < 1/(m-1)$ *teljesül minden tárgyméretre, ahol* $m \geq 2$ *egész szám. Ekkor a DNF algoritmus optimális.*

Bizonyítás. Minden fedett láda pontosan m tárgyat tartalmaz. Emiatt a legjobb választás, hogy egyszerre csak egy láda van nyitva. □

Természetesen általában nem feltételezhetjük a tárgyméretekre az előző lemma feltételének teljesülését. Ezért a DNF algoritmus alkalmazásakor veszteség keletkezik abban az értelemben, hogy számos láda „túl lesz pakolva", amin azt értjük, hogy néhány láda úgy lesz lefedve, hogy a bele pakolt tárgyak összmérete jóval nagyobb lehet 1-nél, tehát jóval több, mint ami éppen elegendő lenne. Ha ezt a veszteséget meg akarjuk „menteni", lehetővé kell tennünk, hogy egyszerre több láda legyen nyitható, ezáltal a ládaméret jobban közelíthetővé válik.

Tekintsük a következő klasszikus algoritmus, a *Harmonic(K)* algoritmus, vagy röviden H(K) adaptációját, ahol $K \geq 1$. Az algoritmus elve az, hogy csak hasonló méretű tárgyak kerülhetnek egy ládába. Egyszerre legfeljebb csak K láda lehet nyitva. A tárgyakat a méreteik alapján osztályokba csoportosítjuk, minden láda egy osztályt reprezentál. Ha a következő tárgy mérete az $I_k = \left(\frac{1}{k+1}, \frac{1}{k}\right]$ intervallumba esik, a k-dik ládatípusba kerül, ahol: $k = 1, \ldots, K-1$. A legkisebb tárgyak, vagyis az $I_K = \left(0, \frac{1}{K}\right]$ intervallumba esők pedig a K-dik típusú ládába kerülnek.

H(K) algoritmus

– Helyezzük a következő tárgyat a k-dik típusú ládába, ha van ilyen nyitott láda, és a tárgy mérete az I_k intervallumba esik. Ha nincs ilyen nyitott láda, akkor nyitunk számára egy ilyen típusút. Amint egy láda megtelik (fedetté válik), elszállítjuk. Ha nincs további tárgy, az algoritmus megáll.

Megfigyelhető, hogy a H(K) algoritmus jobban teljesít, mint a DNF, ha K „nem túl nagy" és a fedett ládák után járó profit-függvény „nem csökken túl gyorsan". Megjegyezzük, hogy a H(K) okosabbá tehető oly módon (vagyis nyerünk egy új algoritmust, amit Smart Harmonic(K)-nak, vagy röviden SH(K)-nak neveztünk el, ami H(K) „ügyes" változata), ha a következő szabályok szerint pakolja a tárgyakat:

SH(K) algoritmus

1. Ha a következő elem le tudja fedni valamelyik ládát, akkor tegyük a tárgyat ezek közül a legkisebb telítettségű ládába, és szállítsuk el a ládát.
2. Bármely más esetben az SH(K) algoritmus a H(K) algoritmus szabálya szerint működik.

A következő táblázatban az előző algoritmusokat hasonlítottuk össze. Soronként a különböző algoritmusok által kapott megoldásokat közöljük, soronként másmás feladatosztályokra alkalmazva őket. A probléma-osztályok az alábbiak:

A ládaméretet minden esetben 100-nak, a tárgyak számát pedig 1000-nek választottuk. A tárgyméretek soronként a következő intervallumokból kerülnek ki: $[10; 40]$, $[15; 25]$, $[1; 100]$, és ismét $[10; 40]$, $[15; 25]$, $[1; 100]$. A profit-függvény az első három esetben: $G(k) = 11 - k$ (pl. $G(1) = 10$, $G(2) = 9$ és így tovább), ami egy gyorsan csökkenő profit-függvény. A következő három esetben pedig

$G(k) = 10,1 - 0,1k$ (pl. $G(1) = 10$, $G(2) = 9,9$ és így tovább), vagyis egy lassan csökkenő profit-függvényt választottunk.

Így nyerünk hat lényegesen különböző feladatosztályt. A DNF algoritmus által kapott megoldás értékét mindig 100%-nak tekintettük, és ehhez hasonlítottuk a többi algoritmus által kapott megoldásokat. Tíz futás átlagát írtuk a táblázat megfelelő rublikáiba. (A program C-nyelven íródott, a kódot kérésre szívesen elküldjük.)

	DNF	**H(2)**	**H(3)**	**SH(2)**	**SH(3)**
1	$2080 = 100\%$	$98,6\%$	$97,4\%$	99%	$97,5\%$
2	$1770 = 100\%$	$101,1\%$	100%	$101,1\%$	$101,1\%$
3	$3460 = 100\%$	$93,6\%$	$83,1\%$	$99,2\%$	$100,6\%$
4	$2080 = 100\%$	100%	$101,2\%$	$98,6\%$	$101,8\%$
5	$1770 = 100\%$	$100,6\%$	100%	100%	$101,1\%$
6	$3460 = 100\%$	101%	$95,7\%$	$103,2\%$	$108,4\%$

1. táblázat. Természetes algoritmusok

Emlékezzünk vissza, hogy az első három esetben a haszon-függvény erősen csökkenő. Az első feladatosztály esetén a többi algoritmus nem képes a DNF algoritmust „legyőzni", de a második és a harmadik esetben már van olyan algoritmus, amelyik legyőzi a DNF-et. A többi esetben a haszon-függvény csak enyhén csökken, itt a DNF könnyen legyőzhető, de egy idő után H(K), ill. SH(K) hatékonysága nem javul tovább K növelésével, SH(K) bizonyos esetekben tényleg „okosabb", vagy „ügyesebb" mint a H(K) algoritmus egyszerű változata (ugyanazzal a K-val). Figyeljük meg, hogy a hatodik feladatosztály esetében az SH(K) lényegesen jobb, mint H(K), ennek az lehet az oka, hogy a tárgyméretek eloszlása (1 és 100 között) nagyobb lehetőséget ad az algoritmusnak, hogy „okos" legyen.

3.1. Egy új, rugalmas algoritmus-család

Most egy új algoritmus-családot definiálunk, amely kellően rugalmas ahhoz, hogy az előbbi algoritmusok bármelyikével sikeresen felvegye a versenyt. Ezt különböző stratégiai paraméterek beállításával érjük el. Az algoritmust Mask(α, β, K)-val jelöljük, és az alábbiak szerint működik. K az egyszerre megnyitható ládák maximális számát jelenti. α és β K-dimenziós nemnegatív vektorok, ahol minden koordináta kisebb, mint 1. Az algoritmus egy elfogadás-elutasítás politikát folytat: a következő tárgyat elfogadja, és valamely ládába pakolja, ha a ládába pakolt tárgyak (az aktuális tárgy méretével) megnövelt összmérete az „elfogadó" tartományba kerül, és elutasítja a tárgynak a ládába történő pakolását, ha a megnövelt összméret az „elutasító" tartományba kerül. Az elutasító, illetve az elfogadó tartományokat az α és a β paraméterek definiálják a következőképpen:

A k-dik láda elfogadó tartománya a következő: $[0; 1-\alpha_k] \cup [1; 1+\beta_k]$. Vagyis minden láda esetén két elfogadó és két elutasító tartomány van. Egyrészt, nem engedjük meg a tárgynak a ládába pakolását, ha az összméret ezáltal túlságosan nagy lenne, nagyon „túl lenne pakolva" a láda, vagyis, ha az összméret $(1+\beta_k)$-nál nagyobbá nőne. Másrészt azt sem engedjük meg, ha a megnövelt összméret már majdnem 1, de még kevesebb, mint 1 (vagyis a láda még nincs tele, de már majdnem tele van). Ilyenkor ugyanis, megfelelően nagy tárgyméretek esetén a következő tárgy, amelyik megtölti a ládát, egyben jelentősen „túlpakolná" azt. Ezek alapján a Mask(α, β, K) algoritmus formális leírása a következő:

***Mask*(α, β, K) algoritmus**

1. Ha a következő tárgy lefedi valamelyik ládát az elfogadó-tartományban, akkor pakoljuk a tárgyat abba a ládába, amelyik ezen ládák közül a legkisebb telítettségű. Szállítsuk el a ládát és menjünk az 5-ös pontra.
2. Ha a következő tárgy pakolható valamelyik ládába (az elfogadó-tartományban, de a láda meg nem lesz fedett), akkor pakoljuk azt egy ilyen ládába. Menjünk az 5-ös pontra.
3. Ha $k < K$, akkor nyissunk egy új ládát, az aktuális tárgy ebbe a ládába kerül, és menjünk az 5-ös pontra.
4. Ha $k = K$, akkor pakoljuk az aktuális tárgyat a legkisebb telítettségű ládába. Ha a láda fedett lesz, szállítsuk el és menjünk az 5-ös pontra.
5. Ha nincs több tárgy, az algoritmus megáll, különben menjünk az 1-es pontra.

Megjegyzés. A Mask algoritmusnak itt már rögtön az „ügyes" változatát definiáltuk, vagyis ha a következő tárggyal be tudunk fedni egy ládát, az elfogadási tartományon belül, ezek közül olyanba tesszük, amelyik a legkevésbé lesz túlpakolva.

A következő részben a Mask algoritmus stratégiai K, α, β paramétereinek „jó" megválasztásával foglalkozunk. Azt fogjuk látni, hogy lényegében minden feladatosztály esetén be lehet úgy állítani a paramétereket, hogy a megfelelő beállítás mellett Mask legalább olyan jó, vagy még hatékonyabb, mint az előző algoritmusok.

Ugyanazt a hat feladatosztályt vizsgáltuk, mint az előbb, a „max" oszlopában a korábbi hét algoritmus (DNF, H(2), H(3), H(4), SH(2), SH(3), SH(4)) által kapott legjobb megoldás értéke szerepel (%-os formában), ezzel versenyeztetünk különböző paraméterű Mask algoritmusokat. A paraméterek a következők:
$Mask_1$: $K = 2$, $\alpha = (15, 20)$, és $\beta = (10, 30)$.
$Mask_2$: $K = 1$, $\alpha = (10)$, és $\beta = (20)$.
$Mask_3$: $K = 2$, $\alpha = (15, 15)$, és $\beta = (30, 30)$.
$Mask_4$: $K = 3$, $\alpha = (20, 20, 20)$, és $\beta = (30, 30, 30)$.
$Mask_5$: $K = 3$, $\alpha = (10, 10, 10)$, és $\beta = (25, 25, 25)$.
$Mask_6$: $K = 3$, $\alpha = (10, 20, 30)$, és $\beta = (40, 50, 60)$.

	max	$\boldsymbol{Mask}_1$	$\boldsymbol{Mask}_2$	$\boldsymbol{Mask}_3$	$\boldsymbol{Mask}_4$	$\boldsymbol{Mask}_5$	$\boldsymbol{Mask}_6$
1	100 %	**100, 2**%	100, 1%	96, 2%	91%	93, 7%	88, 9%
2	101, 1%	91, 5%	**101, 1**%	94, 6%	86, 2%	88, 8%	88, 3%
3	100, 6%	100%	101, 5%	**106**%	100%	103, 9%	100, 6%
4	104, 3%	105, 9%	100, 5%	104%	**106, 6**%	102%	102, 8%
5	101, 1%	100, 8%	101, 1%	103, 1%	102, 6%	**103, 5**%	102, 3%
6	108, 4%	105%	101, 5%	108, 3%	113, 1%	110%	**114**%

2. táblázat. Mask algoritmusok különböző stratégiai paraméterekkel

Természetesen nem mindegyik Mask „jó" minden esetben. Ez nem is lenne célunk. Azonban megfigyelhetjük a következőt: mindegyik feladatosztály esetén **van olyan Mask algoritmus** (van olyan paraméter-beállítás), amelyik versenyképes a korábbi legjobb algoritmussal, sőt legyőzi azt. Azt kell még „kitalálnunk", hogy hogyan tudjuk megtalálni, egy adott feladatosztály esetén a paraméterek megfelelő beállítását. Ezzel a kérdéssel foglalkozunk a következő fejezetben.

4. Algoritmusok evolúciója - EOA

Megállapítottuk tehát, hogy a Mask algoritmus paramétereinek helyes megválasztásával képesek vagyunk a feladatot jól megoldani és Mask felülmúlja a többi, korábban tárgyalt algoritmust. Az egyetlen problémát a paraméterek „helyes" megválasztása jelenti. Ebben a fejezetben egy új módszert adunk a ládafedési feladatunk megoldására, a módszert „Algoritmusok evolúciójának" (EOA) nevezzük. E módszer más (nehéz) online feladat megoldására is alkalmazható. Módszerünk nem azonos azzal, amit „Evolúciós algoritmus"-nak (EA) neveznek. Mi a fő különbség a kettő között? Az evolúciós algoritmusok (EA) (vagy más lokális kereső módszerek), valamilyen **offline** feladat megoldására szolgálnak. Legyen S a feladat megengedett megoldásainak halmaza, ezek között egy szomszédsági struktúrát definiálunk (a megoldásokat általában $0-1$, vagy valós vektorokkal reprezentáljuk). Az EA először meghatározza a feladat egy x_0 megoldását, majd kiindulva ebből az x_0-ból, ennek a környezetéből választ egy (másik) x_1 megoldást, ha bizonyos kritériumok teljesülnek, akkor x_0-át x_1-re cseréli és ezt a lépést iterálja.

Most másról van szó, először is, feladatunk nem offline, hanem **online**. (Vagyis bizonyos értelemben az EOA az EA online megfelelője.) Nem az offline feladat megengedett megoldásai között, hanem az online feladat megoldó algoritmusai között hozunk létre szomszédsági struktúrát. Megengedett megoldások helyett algoritmusokon „lépegetünk". A szerzők legjobb tudomása szerint ilyen módszert korábban még nem adtak meg és nem vizsgáltak. Reméljük, hogy módszerünk (meghatározunk egy kellően rugalmas algoritmus-családot valamely online feladat megol-

dására, az algoritmusok között egy szomszédsági struktúrát definiálunk, eztán egy lokális kereső módszer segítségével kiválasztjuk az algoritmusok „legjobbikát") alkalmazható lesz a jövőben más (nehéz) feladatok megoldására is. A főbb különbségeket EA és EOA között az alábbi táblázat szemlélteti.

	EA	**EOA**
a feladat jellege	offline	online
„amin lépegetünk"	megengedett megoldás	megoldó algoritmus

3. táblázat. EA és EOA összehasonlítása

A szomszédsági struktúra természetes módon határozható meg a különböző Mask algoritmusok között: Egy rögzített $Mask(K, \alpha, \beta)$ algoritmusnak módosítjuk valamelyik paraméterét: K-t 1-gyel növeljük vagy csökkentjük, illetve az α vagy β vektorok valamelyik komponensét csökkentjük, vagy növeljük egy előre rögzített kicsi Δ pozitív konstanssal (úgy, hogy a megváltozott érték ismét pozitív legyen és kisebb, mint 1). A megfelelő szomszéd választásához szimulált hűtést alkalmaztunk, amely módszert az alábbiakban röviden összefoglaljuk.

A szimulált hűtés (simulated annealing) egy népszerű, sok feladat megoldására alkalmazott, hatékony metaheurisztika, közeli rokonságban áll a Tabu-kereséssel és a genetikus algoritmusokkal. Szimulált hűtés esetén adott egy (nehéz) kombinatorikus optimalizálási offline feladat, amelynek ismerjük egy x megengedett megoldását, esetleg egy ilyen megoldást valamilyen heurisztikával állítunk elő. Feltételezzük, hogy minden megengedett megoldásnak könnyen elő tudunk állítani véletlenszerűen valamilyen y „szomszédját", vagyis egy olyan másik megengedett megoldást, amely az előbbinek egy „kicsi" változtatásával adódik, vagyis kicsi környezetéből való. Szimulált hűtés esetén, ezután, a két megoldás közül az egyiket választjuk majd, megfelelő szabály szerint. Így, iterációnként egy-egy megengedett megoldásunk van, és ezekkel igyekszünk valamilyen optimális megoldást megközelíteni.

A szimulált hűtés azt szimulálja, amikor valamilyen anyagot felmelegítünk, majd lassú hűtés folyamán a részecskék valamilyen alaphelyzetbe kerülnek. A kulcs most az, hogy hogyan választunk az előbbi két megengedett megoldás, x és y közül. Egyrészt, ha y jobb megoldás, mint x, vagyis maximalizálandó célfüggvény esetén nagyobb célfüggvény-érték tartozik hozzá, akkor mindenképpen a jobb megoldást, y-t választjuk. Azonban y-t akkor is elfogadjuk, ha csak egy „kicsivel" rosszabb megoldás mint x, de egyre csökkenő valószínűséggel, és egyre kisebb célfüggvényromlást engedünk meg. A pontos képletek helyett, az érdeklődő olvasó figyelmébe inkább az [5] összefoglaló cikket ajánljuk. Szimulált hűtés esetén is sok múlik bizonyos stratégiai paraméterek megfelelő beállításán, amelyek az algoritmus hatékonyságát jelentősen befolyásolják.

Alább ismét közlünk egy összehasonlító tesztet, ahol azt vizsgáltuk, hogy vajon az EOA algoritmus által jobb eredményeket kapunk-e, mint a korábbi algoritmu-

sokkal (vagyis tényleg sikerül-e a Mask algoritmus stratégiai paramétereit helyesen beállítani, „kézi vezérlés", vagyis próbálkozás nélkül).

A problémaosztályok ugyanazok voltak, mint korábban is. Lássuk a kapott eredményeket. (Egy „sima" lokális keresést (LS) is lefuttatunk minden esetben, amelynél csak akkor lépünk át a szomszéd algoritmusra, ha az az előzőnél jobb eredményt produkál.)

	max	**LS**	**EOA**
1	100,2%	100,6%	101%
2	101,1%	101,1%	101,7%
3	106%	106,2%	106,9%
4	106,6%	108,7%	109,6%
5	103,5%	103,6%	104,7%
6	114%	116,9%	116,9%

4. táblázat. Algoritmusok evolúciója

Amint a táblázatból látható, az EOA módszerünk az adott feladat esetén tényleg működik, vagyis alkalmas arra, hogy a feladatot hatékonyan megoldó algoritmust konstruáljon. A más algoritmusok által kapott megoldásokat helyenként lényegesen sikerült javítani. A lokális keresés is sok esetben hatékonynak bizonyul, de ahogy várható volt, ennél EOA (szimulált hűtést alkalmazva) még hatékonyabb.

Természetesen módszerünk csak akkor képes jól működni, ha a tárgyak eloszlása az időben nem változik. Az algoritmus-család (Mask) tagjait pedig nem a teljes tárgyhalmazra, hanem annak egy-egy $m < n$ darabból álló ugyanakkora szeletére alkalmazzuk.

5. Következtetések

Cikkünkben definiáltunk egy új feladatot, aminek a „Ládafedés szállítással" nevet adtuk. Megadtunk néhány természetesen adódó algoritmust a feladat megoldására, aztán egy sokkal hatékonyabb módszert, amit „Algoritmusok evolúciója"-nak, vagy röviden EOA-nak nevezünk. Ez különbözik az evolúciós algoritmusoktól abban az értelemben, hogy itt nem egy offline feladat megengedett megoldásain lépegetünk, hanem egy online feladat megoldó algoritmusain (a stratégiai paraméterek változtatásával).

A lépegetéshez szimulált hűtést alkalmaztunk, hogy megtaláljuk a megfelelő paramétereket. Természetesen ehelyett tabu-keresés vagy genetikus algoritmus alkalmazását is érdemes lenne kipróbálni, ez további kutatás tárgya.

Mivel a feladat nagyon nehéz, a szokásos versenyképességi analízis elvégzése szinte lehetetlen, ezért számítógépes tesztekkel igazoltuk algoritmusunk hatékony-

ságát. Az EOA során egy kellően rugalmas algoritmus-család (esetünkben a Mask) közül kiválasztjuk azt, amelyik a „legalkalmasabb" a család tagjai közül a feladat megoldására egy adott feladatosztályon belül.

Kíváncsian várjuk, hogy a valós életbeli problémák mely területein, milyen egyéb „nehéz" problémák megoldására lesz sikeresen alkalmazható az újonnan bemutatott módszer.

Hivatkozások

[1] G. Dósa, L. Epstein: *Online scheduling with a buffer on related machines*, J. of Comb. Optim., DOI 10.1007/s10878-008-9200-y

[2] M. Englert, D. Ozmen, M. Westermann: *The power of reordering for online minimum makespan scheduling*, In: Proc 48th symp foundations of computer science (FOCS), (2008) 603–612.

[3] M. R. Garey, D. S. Johnson: *Computers and intractability*, W. H. Freemand and Company, New York (1979)

[4] W. Zhong, G. Dosa, Z. Tan: *On the machine scheduling problem with job delivery coordination*, European Journal of Op. Res., **182**, (2007) 1057–1072.

[5] Peter J. M. van Laarhoven, Emile H. L. Aarts: *Simulated annealing: theory and applications*, Mathematics and its applications, Kluwer Academic Publishers (1987)

[6] Ed: Kaisa Miettinen, Marko M. Makela, Pekka Neittannmaki, Jacques Périaux: *Evolutionary algorithms in engineering and computer science*, John Wiley and Sons Ltd (1999)

[7] T. Nemeth, C. Imreh: *Parameter Learning Algorithms in Online Scheduling*, Conference of PhD Students in Computer Science, inf.u-szeged.hu

[8] A. Benkő, G. Dósa, Zs. Tuza: *Bin packing/covering with delivery*, manuscript (2009)

(Beérkezett: 2009. november 25.)

BENKŐ ATTILA
Pannon Egyetem
Matematika Tanszék
benko.attila@almos.vein.hu

DÓSA GYÖRGY
Pannon Egyetem
Matematika Tanszék
dosagy@almos.vein.hu

BIN COVERING WITH DELIVERY AND SOLVING IT WITH EVOLUTION OF ALGORITHMS

Attila Benkő, György Dósa

We deal with a new problem called Bin Covering with Delivery. Mainly we mean under this expression that we look for not only a good, but a *"good and fast"* covering. There are several ways to treat such a problem, but we investigate here only one of them, an online problem, which has real-life application, as well. We apply the appropriate version of some classical bin covering algorithms for the problem, and also propose a new method that we call "Evolution of Algorithms", to solve this (algorithmically very hard) problem. In case of such methods a neighborhood structure is defined among the algorithms in a flexible algorithm-family, and using a metaheuristic (simulated annealing now) in some sense the best algorithm is chosen from the set of the algorithms, to solve the problem. We show the efficiency of the proposed method by several computer tests.