Eötvös Loránd Tudományegyetem
Információs Rendszerek Tanszék

STAGE — Bin-packing Függvényapproximátor egy optimalizációs problémához

Diplomamunka

Bartalos Máté
Programtervező Matematikus hallgató
Témavezető: dr. habil. Lőrincz András
ELTE TTK Információs Rendszerek Tanszék

2002. június 15.

Tézisek

Munkám az optimalizációs problémák területéről származik. Az utóbbi időben teret nyert STAGE algoritmus és a pálcika-pakolási (bin-packing) probléma kapcsolatát vizsgálva a téziseim a következők:

- A kialakuló approximált függvényeket (FAPP-okat) tekintve a pálcika-pakolási probléma egyetlen problémacsaládot alkot, abban az értelemben, hogy a STAGE-futtatások alkalmával keletkező FAPP-ok alakja, irányítottsága hasonló.

- Egy nagyobb probléma FAPP-ja jobban teljesít egy kis problémán, mint annak saját FAPP-ja.

- Így egy megfelelően nagy FAPP-pal pálcika-pakolási problémák széles skálája oldható meg mindössze két darab lokális kereséssel.

1. Bevezetés

Számos mérnöki, illetve gazdasági probléma tartalmaz NP-beli optimalizációs problémákat, amelyekben az értékelő- vagy költségfüggvénynek egy optimális vagy közel optimális értékét valamilyen heurisztika segítségével találhatjuk meg. (Ezen dolgozatban mindvégig felteszem, hogy ez a költségfüggvény $Obj : X \to \mathbb{R}$, ahol X az állapotteret jelöli, és a cél az Obj függvény minimális értékének megtalálása.) Nagyon sok közelítő eljárás született arra, hogy lecsökkentsük ezen optimalizációs problémák időigényét. Jó áttekintést nyújt az ilyen közelítő eljárásokra [1].

Az utóbbi időben kezd elterjedni egy újfajta keresési eljárás: a STAGE [2, 3], amely felhasznál egy függvényapproximátort – például mesterséges neuronhálózatot – és felfogható a megerősítéses tanulás egy speciális közelítéseként (RL, [4]). A STAGE valóban nagyon jó tulajdonságokat mutat valós problémákon [5, 6]. Bár a STAGE képes megoldani nehéz feladatokat, vannak hátrányai is. A STAGE általában felismeri az állapottér struktúráját, és ezt felhasználva jobb régiók felé vezeti a keresést az állapottérben. Néhány esetben előfordulhat azonban, hogy a STAGE instabilitást mutat. Ami szintén hátrány, hogy a STAGE nem párhuzamosítható algoritmus.

Az alacsony memória- és időigénye miatt a STAGE-ben célszerű négyzetes regresszión alapuló függvényapproximátort használni. [2]-ben található számos teszt (pl.: láda-pakolási, VLSI-tervezési, kielégíthetőségi probléma,...) mutatja, hogy a quadratikus regresszión alapuló függvényapproximátor megbízható és jó tulajdonságokkal rendelkezik, a globális optimuma könnyen megtalálható.

Számos „benchmark" probléma ismert, amin az optimalizációs algorit-

musok tesztelhetőek, és a kapott eredmények összehasonlíthatóak. Ezek az, általában NP-beli, problémák egyszerűek és a kapott eredmények jól ellenőrizhetőek. Egyik ilyen probléma az 1-dimenziós láda-pakolási (bin-packing), a pálcika-pakolási probléma, ami ennek a dolgozatnak a tárgya. Megfigyeléseim szerint egy nagyobb láda-pakolási problémából nyert approximált függvény felhasználásával – a legtöbb esetben – jobb eredményt tudunk elérni egy kisebb problémán, mint annak saját approximált függvényével. Ez hasznos lehet, ugyanis a struktúra tár- és időigényes feltérképezése elhagyható, mert közelítése a rendelkezésünkre áll.

1.1. A dolgozat felépítése:

A 2. fejezetben részletesen bemutatom a STAGE algoritmust, részletesen ismertetve annak működését. A 3. fejezet leírja az pálcika-pakolási problémát. A kapott eredmények, azok elemzése és értelmezése a 4. fejezetben található. Az eredmények hátterébe enged némi bepillantást az 5. fejezet, a 6. fejezet pedig a dolgozat összefoglalása, illetve kitekintés a lehetséges további kutatásokra.

2. Stage

A STAGE algoritmust J. A. Boyan mutatta be dolgozatában [2]. A szerző több alkalmazási területen mutatja be az eljárás hatékonyságát. Mielőtt a konkrét alkalmazásokra rátérnék áttekintem a módszer működését és elméleti hátterét.

2.1. A Stage informális leírása

2.1.1. A Stage működése vázlatosan

A Stage egy többszintű globális keresőalgoritmus. A Stage fő ötlete az, hogy felosztja a keresést két fázisra. Az első szinten (angolul 'stage', innen a név) egy lokális keresést indít, amelynek a célja, hogy lokális optimumot találjon. A második szinten felhasználjuk azt a feltételezést, hogy ezeknek a lokális minimumoknak van egy átfogó struktúrájuk. A Stage-algoritmus közelíti a globális struktúrát ezen lokális minimumok simításával. A Stage hozzárendeli a lokális minimum értékét minden ponthoz azon a trajektórián, ami ehhez a lokális minimumhoz vezetett: Minden a lokális keresés közben érintett pont és a hozzájuk tartozó minimumérték egy-egy párt alkot. Minden ilyen pár egy *példa* a megfelelően választott függvényapproximátornak. A függvényapproximátort az adott példák alapján tanítjuk. A Stage következő lépése az, hogy felhasználjuk az approximált függvényt, hogy találjunk egy új „intelligens restart" pontot, amiből indíthatunk egy új lokális keresést ígéretesebb régiók felé. A Stage erőssége abban rejlik, hogy a függvényapproximátort optimalizálhatjuk általánosításra (olyan részeken is próbáljon jó értékeket jósolni, ahonnan még nem kapott tanítóadatokat) így segítve a globális keresést.

2.1.2. A függvényapproximátor és a jellemzők

Van egy másik fontos tulajdonsága is a STAGE-nek. A függvényapproximátor egy általános koncepció, amely input-output párok alapján állítja be paramétereit, így közelítve azt a függvényt, ami az input-output párokat létrehozta. A mi esetünkben ez a függvény nem más, mint az adott pontokból elérhető lokális minimumok értékének közelítése. A függvényapproximátor triviális inputja maga az állapot lehet, amihez az adott érték tartozik. De az input lehet más is, teljesen szabadon megválasztható. Azaz az input lehet egy transzformált formája, egy *jellemzője* (feature), az adott állapotnak. Akár gyarapíthatjuk is a bemenő változókat a transzformált inputadatokkal. Az irodalomban sok példa van arra, hogy az inputot teljes egészében helyettesíti néhány releváns transzformált változó (jellemző) [7].

2.1.3. A függvényapproximátor használata

A STAGE egy megoldást nyújt arra, hogy új, intelligens kezdőpontot találjunk, felhasználva ehhez a függvényapproximátort. A hagyományos lokális keresés (LK) befejezte után a STAGE a következőket teszi (lásd 1. ábra):

- A függvényapproximátor tanul az utolsó LK példapontjaiból
- Egy új lokális keresés végrehajtása, a *függvényapproximátor* által közelített *értékelőfüggvényt* minimalizálva (FLK)
- E lokális keresés trajektóriáján lévő utolsó pont lesz az „intelligens restart" pont, azaz a kezdőpontja a következő LK-nak

Megjegyzendő, hogy sajnos nem ismeretes módszer a jó jellemzők automatikus generálására. Pedig a jó jellemzők a legfontosabbak a keresés haté-

1. ábra. A STAGE sematikus működése

konyságának szempontjából [3]. Így a jellemzők kigondolása a tervezőre, ill. a programozóra marad.

2.2. A STAGE formális leírása

2.2.1. A keresési feladat definíciója

Legyen a keresési feladat állapottere X, és legyen $Obj(x)$ az X-en értelmezett valós értékű költségfüggvény[1]: $Obj : X \to \mathbb{R}$ minden $x \in X$-re. Az optimalizációs feladatokban a célunk az, hogy találjunk egy vagy több olyan $x \in X_{min}$ állapotot, amire a következő egyenlőtlenség teljesül tetszőleges $y \in X$ mellett: $Obj(y) \geq Obj(x)$. Jelöljük a lokális optimalizációs algoritmusunkat a következőképpen: $\pi_{loc} : X \to X$. Ez a lokális keresés a következő tulajdonságokkal kell bírjon: $\forall x$-re a $\pi_{loc}(x)$ szomszédja x-nek és $Obj(\pi_{loc}(x)) \leq Obj(x)$. A szomszédság egy olyan leképezés, ami minden x-hez hozzárendeli X-nek egy véges részhalmazát: $N : X \to 2^X$. A szomszédok, illetve a szomszédság konkrét megválasztása sokféleképpen történhet, és erősen függ tőle az algoritmus hatékonysága.

A π_{loc}-ra teljesülnie kell a Markov-tulajdonságnak. Ez azt jelenti, hogy

[1]Ezt szokás célfüggvénynek is hívni. Azért határoztam a költségfüggvény elnevezés mellett, mert ez utal arra, hogy ebben a dolgozatban csak minimalizációs problémákkal foglalkozom.

egy trajektória aktuális állapotától függ csak, hogy π_{loc} mely állapotba lép, és nem függ a trajektória korábbi állapotaitól. Erre azért van szükség, mert így a függvényapproximátor a π_{loc} által generált trajektória minden elemét felhasználhatja, mint tanítóadatot, nemcsak az elsőt. Ezáltal gyorsul az állapottér struktúrájának tanulása.

2.2.2. A tanítóadatok

A lokális keresés eredménye állapotok egy sorozata, az úgynevezett trajektória. A STAGE végrehajt egy simítást a költségfüggvényen a kapott trajektória mentén. Tegyük fel, hogy a trajektórián l darab pont van: $x_{traj} = \{x_1, \ldots, x_l\}$. Az $Obj(x_1), \ldots, Obj(x_l)$ pedig az ezekhez a pontokhoz tartozó költségértékek. A STAGE helyettesíti ezeket az értékeket a trajektória minimális értékével, $min_i(Obj(x_i)) : x_i \in x_{traj}$-jal, az ehhez tartozó állapot $x_{min} = argmin_i(Obj(x_i)) : x_i \in x_{traj}$[2]. Ezután a STAGE az x_{traj} trajektória minden pontjából származtat egy input-output párt a függvényapproximátora számára. Mindegyik pár egy x_i ($x_i \in x_{traj}$) input állapotból, és az $Obj(x_{min})$ értékből áll.

2.2.3. Az értékelőfüggvény

Az $Obj(x)$ simított formáját értékelőfüggvénynek hívjuk, és $V(x)$-szel jelöljük. Ez a függvény azt az értéket mutatja, ami az adott x állapotból elérhető egy π_{loc} lokális keresés alkalmazásával. A STAGE a $V(x)$ függvényt egy függvényapproximátorral közelíti. A legtöbb esetben az x állapotot egy D dimenziós, valósértékű jellemzővektorral (featurevector) kódoljuk. A kódolás során tetszőleges transzformációt alkalmazhatunk: használhatunk nemlineáris vagy

[2]A korábban leírt tulajdonságokkal bíró π_{loc} esetén ez az utolsó elem: $x_{min} = x_l$.

akár nem invertálható transzformációkat. Az X állapottér F „jellemzőtérre" való kódolása bármilyen információt tartalmazhat az állapotokról. Egy tipikus példa maga az $Obj(x)$ függvény, ami az egyik jellemző lehet. A fenti leképezést jelölése legyen $F : X \rightarrow \mathbb{R}^D$, és így az értékelőfüggvény a következő alakot nyeri: $V(F(x))$.

A $V(F(x))$ függvény $\tilde{V}(F(x))$ approximációját arra használja a STAGE, hogy új kezdőpontot, „restart"-pontot találjon a lokális keresés számára. Az állapottér tetszőleges pontjától kezdve a STAGE sztochasztikus hegymászó algoritmust alkalmaz: egy pont egy vagy több szomszédját véletlenszerűen kiválasztjuk, és kiszámoljuk a hozzájuk tartozó $\tilde{V}(F(x))$ értéket. Kiértékelt állapotok közül kiválasztunk egyet – például mohó módon (a legnagyobb értékűt), vagy a szimulált hűtés paradigmája szerint, vagy más hasonló módszerrel –, és ez lesz a trajektória következő pontja.

2.2.4. A függvényapproximátor szükséges tulajdonságai

A $V(F(x))$-et közelítő függvényapproximátornak a következő tulajdonságokkal kell rendelkeznie:

- **Inkrementális:** A STAGE-ben keletkezik sok — akár több millió — tanítópélda. A függvényapproximátornak kezelnie kell ezt a nagy mennyiségű adatot, ezért a tanítóeljárásnak inkrementálisnak kell lennie ahelyett, hogy az összes előző input-output pár alapján direkt módon számítsa ki az approximációt: azaz függvényapproximátor gyors kell, hogy legyen.

- **Zajra érzéketlen:** A STAGE által tanult értékek hosszú sztochasztikus keresési eljárások eredményei, ezért a függvényapproximátornak a zajra

érzéketlennek kell lennie.

- **Extrapoláló:** A STAGE-nek, az általa használt függvényapproximátor segítségével, új kezdőpontot kell találnia egy lokális keresési algoritmus részére. Az a feltételezésünk, hogy az új kezdőpontból induló lokális keresés jobb értéket ad, mint az addigiak. Tehát a STAGE-nek olyan függvényapproximátor kell, amelyik jól extrapolálja a még nem látott állapotok értékeit is, így ilyen esetekben is ígéretes kezdőpontot adhat.

A [2]-ban leírt tapasztalok alapján a jellemzők terének egy quadratikus regresszión alapuló approximációja nagyon sikeres lehet, ugyanis teljesül rá a fenti három pont mindegyike. Ezenkívül nagyon nagy előnye, hogy a kialakult approximált függvény egy maximum másodfokú függvény, amin egy sztochasztikus hegymászó[3] algoritmus 1 valószínűséggel megtalálja a globális optimumot.

2.2.5. A függvényapproximátor működése

[2] szerint a közelítést célszerű maximum másodfokú függvények lineáris kombinációit felhasználva végezni:

$$V\ (x, \beta) = \sum_{k=1}^{K} \beta[k]\phi_k(x)$$

Itt β a hangolandó paramétervektor. A β i. komponense a $\beta[i]$. A ϕ_k, $i = 1, \ldots, K$ függvények gyorsan számolható bázisfüggvények. Tekintsük a $f_1, f_2, \ldots, f_D$ jellemzővektort. Egy ilyen jellemző megad egy valós értéket

[3]Persze a minimalizáció miatt az algoritmus a csökkenés irányába megy, tehát a „hegymászó" elnevezés félrevezető lehet.

minden egyes $x \in X$-re: $f_k(x) \in \mathbb{R}$. Felépíthető egy kvadratikus approximáció a jellemzők használatával, ahol a jellemzők maximum másodfokú kombinációit vesszük figyelembe. Az ilyen komponensek száma, azaz a bázisfüggvények száma $K = (D+1)(D+2)/2$. A ϕ vektorfüggvénynek $(D+1)(D+2)/2$ darab komponense van, azaz $\phi : X \to \mathbb{R}^{(D+1)(D+2)/2}$. ϕ-t írhatjuk a következő alakban:

$$\phi(x) = (1, f_1(x), f_2(x), \ldots, f_D(x), \\ f_1^2(x), f_1 f_2(x), \ldots, f_2^2(x), \ldots, f_2 f_D(x), \ldots, f_D^2(x))$$

A lineáris kombináción alapuló függvényapproximátor hangolása egyszerű. Tegyük fel, hogy a tanítóadatok halmaza a következő:

$$\{\phi_{(1)} \to y_1, \phi_{(2)} \to y_2, \ldots, \phi_{(N)} \to y_N\}$$

ahol $\phi_{(i)} = (1, f_1(x_i), \ldots, f_D^2(x_i))$ az i. input és az y_i az i. output, ezek alkotják az i. tanítandó input-output párt. A tanulás a β^* együtthatóvektor értékeinek beállítását jelenti, ahol

$$\beta^* = \underset{\beta \in \mathbb{R}^K}{argmin} \sum_{i=1}^{n} (y_i - \beta \cdot \phi_{(i)})^2$$

Az optimális β^* együtthatóvektor megtalálása egy legkisebb négyzetek módszerével megoldható probléma. β^* kiszámításához szükséges az

$$A = \sum_{i=1}^{N} \phi_{(i)} * \phi_{(i)}^T \quad (1)$$

$$b = \sum_{i=1}^{N} \phi_{(i)} y_i \quad (2)$$

ahol A egy $K * K$ méretű mátrix, b egy K dimenziós vektor és teljesül

$$A\beta = b. \quad (3)$$

Ennek az egyenletnek a megoldása

$$\beta^* = A^{-1}b. \quad (4)$$

Az A mátrix inverzét szinguláris felbontással lehet kiszámítani, amely robosztus, ha A szinguláris.

2.2.6. A kvadratikus függvényapproximátor időigénye

A STAGE futásakor minden iterációban számos új tanítópélda gyűlik össze és a függvényapproximátor minden iterációban újra tanul. Az újratanulás – kvadratikus approximáció esetén – azt jelenti, hogy az új tanítópéldák alapján frissítjük az A mátrixot és a b vektort, azaz az (1) és (2)-ben lévő szumma új tagjait kiszámítjuk, és hozzáadjuk az eddigi A mátrixhoz, illetve b vektorhoz. Az A mátrix frissítéséhez $O(K^2)$ lépés , míg a b vektor frissítéséhez $O(K)$ lépés szükséges tanítópéldánként. A β^* együtthatóvektor kiszámítása $O(K^3)$ lépést igényel.

A STAGE pszeudokódja az 1. táblázatban látható.

2.3. A STAGE alkalmazása egy egyszerű problémán

Ebben a szakaszban a STAGE működését egy 1-dimenziós függvény-minimalizációs problémán mutatom be [2] alapján. A függvény a következők szerint lett megválasztva:

- Egy dimenzióban jól szemléltethető
- A globális optimuma ismert
- Elegendően sok lokális optimuma van, így a keresés nehezített.

1. táblázat. A STAGE pszeudokódja

Legyen x_0 véletlen kezdőállapot
Hajtsd végre a következőket maximum N(=EvalNum)-szer
  Futtasd π_{loc}-ot az X-en x_0-ból
    A kapott trajektória: $(x_0, x_1, \ldots, x_k)$
  Legyen y a trajektória legjobb állapotának értéke
  A simítás végrehajtása:
    Add az (x_i, y) párokat a függvényapproximátor
      tanítóhalmazához
  Tanítsd újra a függvényapproximátort az új tanítóhalmazzal
  Hajts végre egy sztochasztikus lokális keresést
    a V-n az x_N állapotból kezdve és
    legyen x_0^{new} a keresés végpontja
  Ha $x_0^{new} \neq x_0$, akkor legyen $x_0 := x_0^{new}$
  különben legyen x_0 egy véletlen állapot
Térj vissza az eddigi legjobb értékű állapottal

A feladat az $Obj(x) = (|x| - 10) \cdot cos(2x)$ célfüggvény minimumának megtalálása a $[-10, 10]$ intervallumon. A függvény grafikonja a 2. ábra felső részén található. Jól látható, hogy a globális minimum az x-tengely 0 értékénél van.

2.3.1. A STAGE alkalmazásának részletei

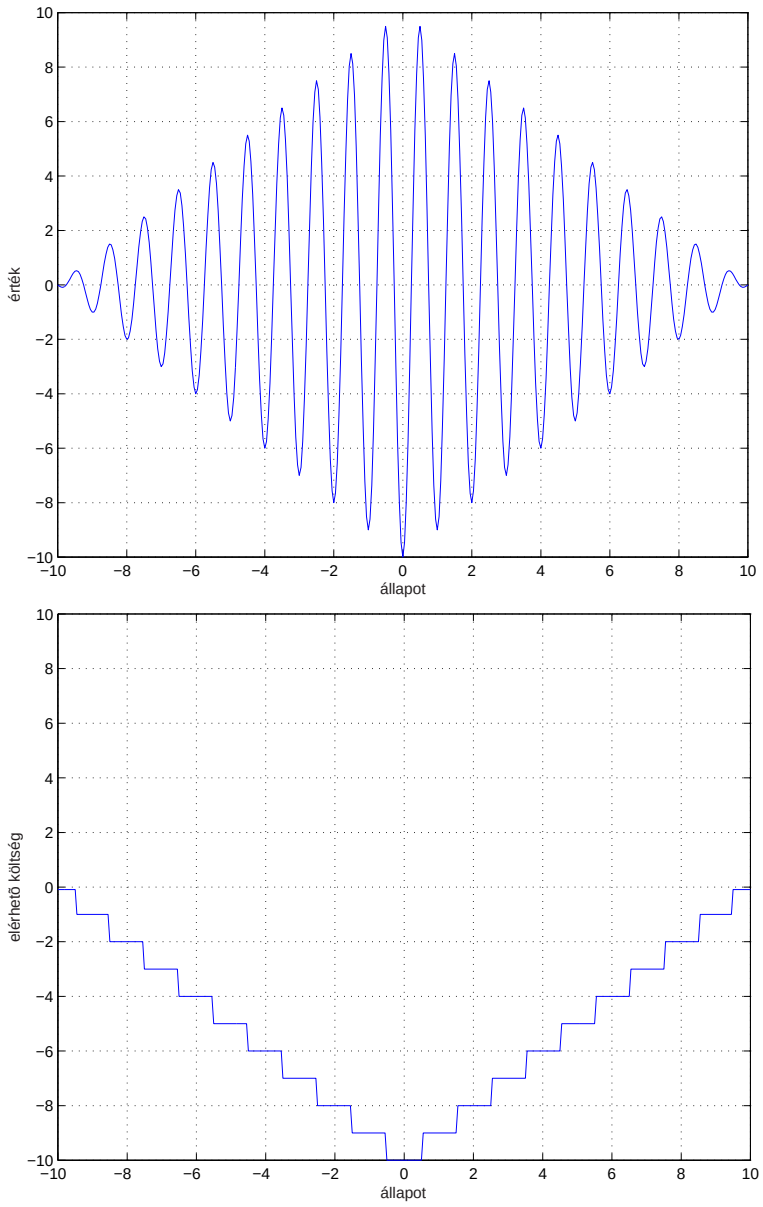
Ahhoz, hogy alkalmazni tudjuk a STAGE-et a problémán szükségünk van az állapottér és a szomszédsági struktúra definíciójára. Ebben az esetben, mivel folytonos feladattal állunk szemben, diszkretizálni kell állapotokra a keresési teret. Ezt például úgy tehetjük meg, hogy felbontjuk az x-tengelyt a $0,1$ többszöröseire, és így az állapotterünk az $X = \{-\frac{100}{10}, -\frac{99}{10}, -\frac{98}{10}, \ldots, \frac{99}{10}, \frac{100}{10}\}$ halmaz lesz, és ezek közül keressük azt, amelyiken a célfüggvény a legkisebb értéket veszi fel. Az állapottérnek megfelelően egy elem szomszédainak a tőle $\frac{1}{10}$ távolságra lévő elemeket definiálhatjuk.

A függvényapproximátor most magán a célfüggvényen dolgozik, azaz nem vezetünk be külön jellemzőket.

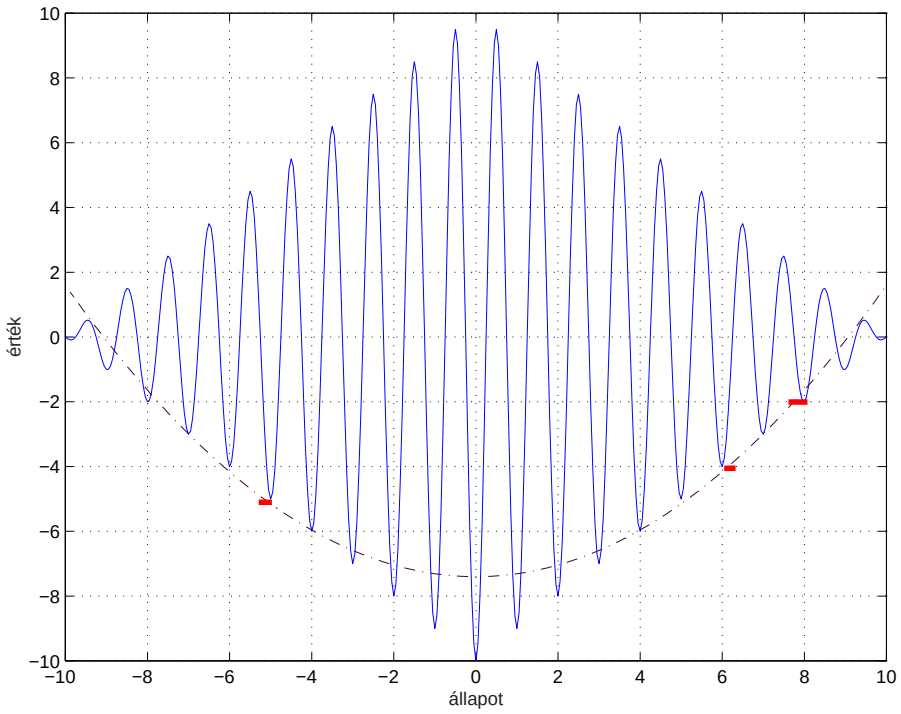
2.3.2. Az értékelő függvény

Az értékelőfüggvény minden ponthoz megadja, hogy az onnan indított lokális keresés milyen költségértéket tud elérni. Ebben az esetben könnyen látható, hogy a lokális minimumok körüli pontokból indított keresések visszakerülnek az adott lokális minimumba, így az értékelőfüggvény egy lépcsős függvény lesz. Ezt mutatja a 2. ábra alsó fele.

A függvényapproximátor ezt az értékelőfüggvényt közelíti kvadratikus regresszióval. Ez itt azt jelenti, hogy a megfigyelt értékekre egy parabolát illesztünk. Jól látható, hogyha három lokális keresés három különböző lokális minimumba érkezik, akkor az ezek alapján közelített parabola már



2. ábra. Az $Obj(x) = (|x| - 10) \cdot cos(2x)$ célfüggvényű 1-dimenziós optimalizációs probléma és a hozzá tartozó értékelőfüggvény.



3. ábra. A függvényapproximátor 3 iteráció után. A piros vonalak 3 lokális keresésből származó 3 tanítóhalmazt jeleznek, a szaggatott vonallal rajzolt görbe pedig az approximált értékelőfüggvény.

egyértelműen a 0-t jelöli ki, mint legígéretesebb kezdőpont. Erre láthatunk példát a 3. ábrán.

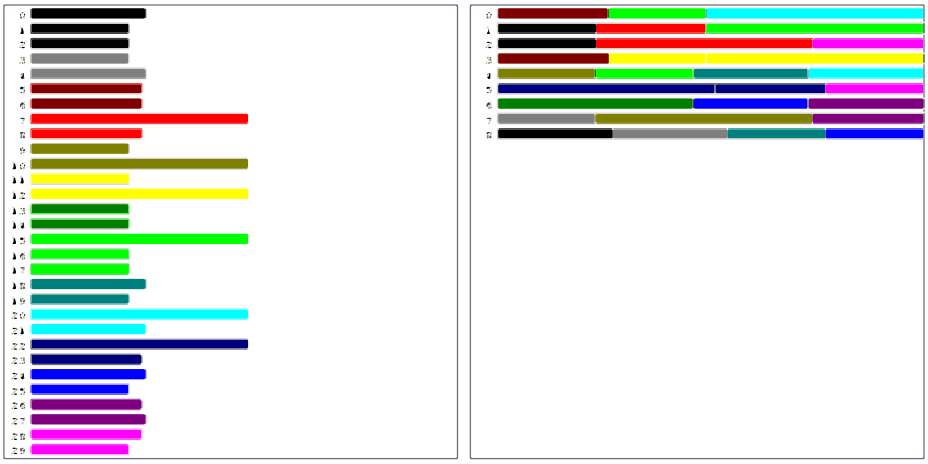
3. A pálcika-pakolási probléma

Sok fontos optimalizációs problémára nem ismerünk hatékony megoldóalgoritmust. Ezek a problémák általában NP-beli problémák, így nem is remélhetünk hatékony, egzakt megoldóalgoritmust. Egyik ilyen optimalizációs probléma a ládapakolási probléma, ami szintén NP-beli, sőt NP-teljes [8]. E problémának rengeteg gyakorlati alkalmazása van: Kamionfeltöltés méret- és súlykorlátozással, raktározási problémák, kábeldarabolás.

3.1. A definíció

E dolgozat további részében az 1-dimenziós láda-pakolási (pálcika-pakolási) problémával foglalkozom. Adva van a C *kapacitás* és n darab tárgy (pálcika) egy listában, $L = (a_1, \ldots, a_n)$, ahol s jelöli a tárgyak méretét: $s(a_i) > 0$. A célunk az, hogy ezeket a tárgyakat belepakoljuk a lehető legkevesebb ládába. Ez azt jelenti, hogy szét kell osztanunk a tárgyakat m halmazba $B_1, \ldots, B_m$ úgy, hogy mindegyik B_j halmazra $\sum_{a_i \in B_j} s(a_i) \leq C$. A 4. ábrán egy kisebb méretű ládapakolási probléma látható összesen 30 tárggyal.

Legyen X a probléma állapottere. Egy x állapot meghatározza, hogy melyik tárgy melyik ládában van. Ahhoz, hogy a STAGE-et alkalmazni tudjuk, definiálnunk kell a szomszédságot: $N : X \to 2^X$, ahol N a szomszédait rendeli egy állapothoz. Ebben az esetben egy állapot szomszédai azok az állapotok legyenek, amelyekben pontosan egy tárgy lett átpakolva (természetesen egy olyan ládába, amelyikben volt elég hely). A költségfüggvény a nemüres ládák száma.



4. ábra. Egy kis méretű ládapakolási probléma. Balra: kezdeti állapot (30 tárgy, mindegyik külön ládában). Jobbra: globális optimum (a tárgyak 9 ládába pakolva, maradék szabad hely nélkül).

3.2. A jellemzők

Az eddigieken kívül kell, hogy találjunk egy jó jellemző-vektort, mert a STAGE hatékonysága nagyban függ ettől. Boyan munkája [2] alapján válasszunk egy 2-dimenziós jellemzővektort. Az 1. dimenzió legyen maga a költségfüggvény, míg a második a ládatelítettség varianciája. E mögött a döntés mögött az áll, hogy az optimális állapot kirakása során azokban az állapotokban, amelyek közvetlen az optimális állapot előtt vannak, nagy a variancia (a ládák egy része tele van, más része meg csak egy pálcikát tartalmaz). Tehát a variancia releváns információ.

A jellemzők formálisan: Legyen $x = \{B_1, \ldots, B_{M_x}\}$ egy állapot. Ekkor

$$Obj(x) = M_x \tag{5}$$

$$Var(x) = \left(\frac{1}{M_x}\sum_{j=1}^{M_x} fullness(B_j)^2\right) - \left(\frac{1}{M_x \cdot C}\sum_{i=1}^{n} s(a_i)\right)^2 \tag{6}$$

ahol

$$fullness(B_j) = \frac{1}{C}\sum s(a_i).$$

4. A futtatások és a kapott eredmények

4.1. A FAPPN rövidítés

Ettől a ponttól kezdve használni fogom a FAPPN rövidítést, ami egy olyan approximált függvényt jelöl, amit egy N méretű problémán való STAGE-futtatásból kapunk. Ha a szövegkörnyezetből egyértelmű, akkor néha elhagyom az N-et és egyszerűen FAPP-ot írok. Egy probléma méretét értelmezhetjük a felhasznált ládák számaként a legjobb állapotban (persze rögzített ládakapacitás mellett). A „FAPP-méret"-en a FAPP-hoz tartozó probléma méretét fogom érteni.

4.2. A felhasznált problémák

Az általam használt problémák egy része a következőképp készült. A ládák számát lerögzítettem, majd valamilyen eloszlás szerint szétdaraboltam őket részekre, így kaptam meg a tárgyakat. Ez azért hasznos, mert a kapott problémára tudjuk a globális optimumot, s így a futtatási eredmények könnyen ellenőrizhetők és összehasonlíthatók. A többi problémát a [9] Falkenauer-féle Operations Research Library-ről töltöttem le. A problémákat általában nem az eredeti, hanem kis zajjal perturbált formájukban használtam fel.

4.3. Az egyiterációs összehasonlítási módszer

Ebben a szakaszban a FAPP-okat összehasonlító módszerek közül az egyik legegyszerűbbet alkalmaztam, amiben a STAGE egyetlen iterációja alapján hasonlítom össze a FAPP-okat.

4.3.1. Az egyiterációs összehasonlító algoritmus

Ez az algoritmus úgy hasonlítja össze a FAPP-okat, hogy mindegyikhez rendel egy jóságot. Működését tekintve az algoritmus először egy lokális keresést hajt végre az adott FAPP felhasználásával egy véletlen kezdőállapotból. Majd a kapott trajektória végpontjából (azaz az „intelligens restart"-pontból) egy újabb lokális keresést hajt végre a költségfüggvényt minimalizálva. E második keresés végpontjához tartozó költségérték lesz az adott FAPP jósága. (Úgy értve, hogy annál jobb, minél kisebb ez a költségérték.) Miután a lokális keresési algoritmusok ugyanúgy, mint a kezdőpont-kiválasztás sztochasztikusak, ezt az algoritmust sokszor kellett lefuttatnom és vennem az átlagot. Az algoritmus pszeudokódja a 2. táblázatban olvasható.

4.3.2. FAPP-ok összehasonlítása az egyiterációs módszerrel

Kutatásaim elején próbáltam a ládapakolási problémákat osztályozni a FAPP-jaik szerint. Két nagyon különböző probléma és FAPP-jaik összehasonlításakor azt találtam, hogy az egyik FAPP jobb eredményt adott mindkét problémára, mint a másik. A 3. táblázat egy példát mutat erre. Mindkét problémánál (ahogy e dolgozat legtöbb problémájánál is) a ládakapacitás 150. A *bin*40-es problémánál a legjobb megoldás 40, a *bin*80-asnál 80. Megfigyelhető, hogy a FAPP80 majdnem a legjobb megoldást találta meg a *bin*40-es problémához, miközben mindössze 2 lokális keresést használt (1 FLK, 1 LK, tanítás nélkül). Másik példa látható a 4. táblázatban. A fenti és a többi hasonló eredmény alapján elkezdtem keresni, hogy mely FAPP-ok a jobbak, és miért.

2. táblázat. Az egyiterációs FAPP-értékelő algoritmus pszeudokódja

Töltsd be a kérdéses FAPP-ot
Legyen $sum := 0$
Tedd a következőket N-szer
   Legyen x_0^{FAPP} egy véletlen állapot
   Hajts végre egy sztochasztikus lokális keresést a V-n x_0-ból
      A kapott trajektória $(x_0^{FAPP}, \ldots, x_m^{FAPP})$
   Legyen $x_0 := x_m^{FAPP}$
   Futtasd π_{loc}-ot x_0-ból az $Obj(x)$-et minimalizálva
   Add hozzá az eredmény költségértékét sum-hoz
Térj vissza az átlaggal: sum/N.

3. táblázat. A FAPP40 és FAPP80 teljesítménye a *bin*40 problémán. A FAPP80 jobban teljesít mindkét problémán. A táblázat értékeit 100 futtatás átlagából kaptam.

	FAPP40	FAPP80
bin40	63.46	41.18
bin80	159.40	140.80

4. táblázat. A FAPP50-50 és FAPP100-50 teljesítménye a bin50-50 problémán. A FAPP100-50 jobban teljesít mindkét problémán. A táblázat értékeit 5 futtatás átlagából kaptam.

	FAPP50-50	FAPP100-50
bin50-50	66	51
bin100-50	200	185

4.3.3. A Futtatási sorozat

Azt találtam, hogyha a ládakapacitás megegyezik és a pálcikák mérete is nagyjából egyforma, akkor a nagyobb probléma FAPP-ja jobb teljesítményt nyújt. Hogy ezt megmutassam, futtatási sorozatokat készítettem. Egyet mutatok be az elkövetkezendőkben. Ebben a sorozatban 9 ládapakolási probléma van, 150-es ládakapacitással: a $bin10, bin20, \ldots, bin90$-es probléma. A probléma nevében lévő szám azt jelenti, hogy hány ládába lehet a tárgyakat pakolni a legjobb megoldásban. A pálcikák méretét az $[1, 50] \subset \mathbb{N}$-es intervallumból egyenletesen választottam. Mindegyik problémán futtattam a STAGE-et, és elmentettem a kapott FAPP-okat. Ezután teszteltem minden FAPP-ot minden problémán. A kapott eredmények az 5. táblázatban láthatóak.

Megfigyelhető, hogy a táblázat főátlóját alatt a FAPP-ok nagyon rosszul teljesítenek. Ezalapján levonhatjuk, hogy egy kisebb problémából kapott FAPP használhatatlan egy nagyobb problémára. A főátlón már jobb eredményt érnek el a FAPP-ok, de még nem elég jót. Ez azt jelenti, hogy egy FAPP a saját méretéhez hasonló méretű problémára már jobb, de egy iterá-

5. táblázat. A FAPP10...FAPP90 teljesítménye a növekvő $bin10 \ldots bin90$ problémákon. A nagyobb FAPP-ok jobban teljesítenek, mint a kisebbek. A táblázat értékeit 100 futtatás átlagából kaptam.

	FAPP10	FAPP20	FAPP30	FAPP40	FAPP50	...
bin10	11.65	21.83	10.96	11.00	11.00	
bin20	37.87	31.97	21.05	21.00	21.00	
bin30	61.48	61.55	41.31	31.21	31.23	
bin40	81.28	81.82	81.07	63.46	42.21	
bin50	95.74	96.13	96.07	95.65	71.88	
bin60	119.6	118.8	118.9	118.6	119.1	
bin70	141.4	142.3	140.6	141.4	141.2	
bin80	159.5	158.4	158.9	159.4	159.3	
bin90	178.9	178.6	179.0	179.5	178.2	

...	FAPP60	FAPP70	FAPP80	FAPP90
bin10	11.00	10.99	11.00	10.97
bin20	21.01	21.00	21.00	20.99
bin30	31.31	31.01	31.06	31.00
bin40	42.03	41.02	41.18	41.02
bin50	52.89	51.10	51.54	54.03
bin60	86.75	61.55	62.40	61.09
bin70	140.9	132.0	73.97	71.05
bin80	158.7	159.5	140.8	81.99
bin90	178.0	179.1	179.6	179.0

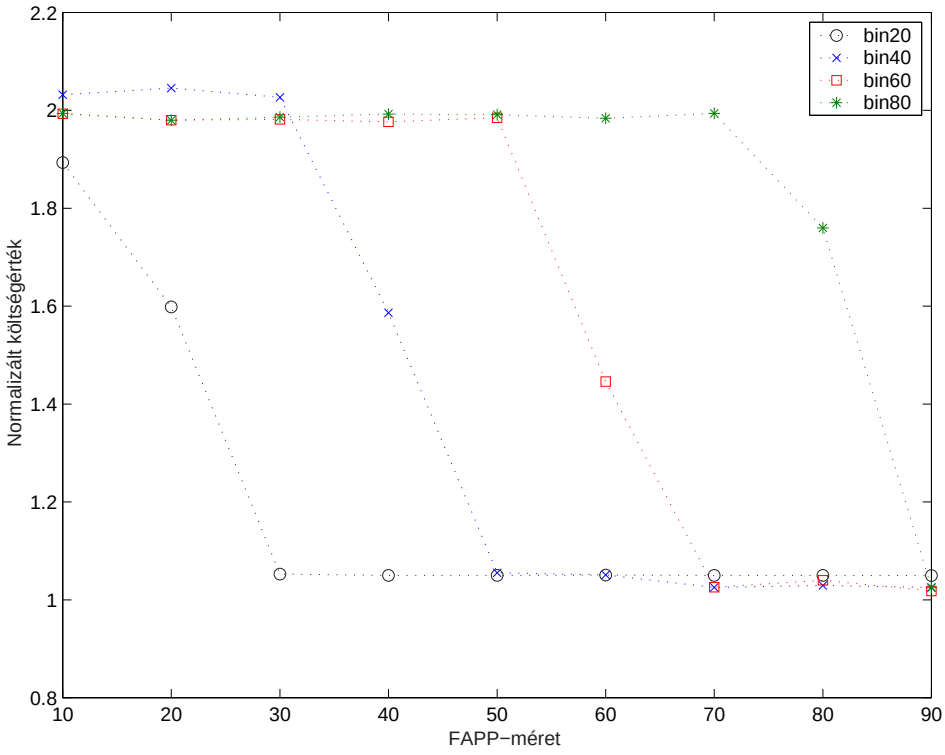
ció még nem elég, hogy az optimum közelébe jusson. Viszont az átló felett a teljesítmény kiváló. Itt a keresés 1 iteráció alatt, azaz 2 lokális kereséssel (LK,FLK) elért egy majdnem optimális állapotot. Például nézzük meg a FAPP90-et. A *bin*70-es problémán talált egy 71 ládából álló megoldást. Ez csak eggyel nagyobb, mint az ideális megoldás, míg a STAGE $600 \cdot 2$-nél is több lokális keresést végrehajtva szintén 71-et adott. Ez azt jelenti, hogy a megfelelő FAPP-pal drasztikusan csökkenthető a számítási idő.

4.3.4. A normalizált teljesítmény

A 5. ábra egy másik nézőpontból mutatja az eredményeket. Ezen az ábrán 4 problémát láthatunk. Minden vonal az adott problémára a FAPP-méret függvényében mutatja a teszteredményt. Az ábrán normalizált értékeket használtam, mert az értékek így függetlenek a problémák méretétől. A normalizált költségértéket az eredeti költségérték és a problémaméret hányadosaként definiáltam. (Pl.: a *bin*40-es problémánál FAPP20-szal $81.82/40 \approx 2.05$-et kapunk.) Megfigyelhetjük, hogy egy nagyobb problémán a kisebb FAPP körülbelül kétszer rosszabb eredményt ad, mint az ideális megoldás. Ha a FAPP-méret és a problémaméret egyforma, akkor a normalizált érték $1.4 - 1.8$ között van. Végül ha a FAPP-méret nagyobb, akkor a hányados kicsivel van 1 fölött. További kutatás tárgya lehetne egy sűrűbb problémaméretű futtatási sorozat végrehajtása és analízise.

4.4. A többiterációs összehasonlítási módszer

Ebben a szakaszban a FAPP-okat egy olyan algoritmussal hasonlítom össze, amellyel nemcsak a vizsgált FAPP-ok eredményei, hanem az eredeti STAGE-futtatások eredményei is összehasonlíthatóak.



5. ábra. 4 probléma normalizált megoldási hatékonysága a FAPP-méret függvényében.

4.4.1. A többiterációs összehasonlító algoritmus

Ez az algoritmus is úgy hasonlítja össze a FAPP-okat, hogy mindegyikhez rendel egy jóságot, de itt valójában egy olyan STAGE-et használok, amely nem tanul, hanem ugyanazt a kezdetben betöltött FAPP-ot használja mindig az intelligens kezdőpont meghatározására. Ez azért lehet jó, mert nemcsak azt tudjuk megállapítani, hogy melyik FAPP mennyire jó, hanem a keresések időbeli fejlődését is vizsgálni tudjuk. Miután maga a STAGE is erősen sztochasztikus, ezért itt is sok futtatásra van szükség. Az algoritmus pszeudokódja a 6. táblázatban olvasható.

4.4.2. FAPP-ok összehasonlítása a második módszerrel

Ezt a módszert is a már ismert $bin10, \ldots, bin90$-es problémákon alkalmaztam. Mindegyik problémán lefuttattam az új algoritmust 100-szor és a kapott eredmények alapján készült el a 6., 7. és 8. ábra. Ezeken az ábrákon a vízszintes tengelyen a lépésszám van feltüntetve, függőlegesen pedig egy 0 és 1 közötti valószínűségérték. Mindegyik ábrán 5 FAPP és a STAGE görbéi láthatók, amelyeket egy konkrét problémán történő futtatásokat figyelembe véve kaptunk. Egy FAPP (vagy a STAGE) görbéje azt mutatja, hogy a vizsgált problémán a FAPP- (vagy STAGE)-futtatásokból milyen valószínűséggel találta meg az optimumot az adott lépésszám alatt a keresés.[4] Ha tehát 1000 lépésszámot véve egy FAPP 100 futtatásból 40-szer találja meg az optimumot, akkor 0.4-es érték áll a FAPP görbéjén az 1000-es abszcisszaérték felett.

[4]Valójában az optimálisnál eggyel nagyobb értéket állítottam be az ábrák létrehozásakor, ugyanis a $bin10$-nél nagyobb problémákon gyakori, hogy 100 futtatásból egyszer sem találja meg az adott algoritmus az optimumot.

6. táblázat. A többiterációs FAPP-értékelő algoritmus pszeudokódja

Töltsd be az adott FAPP-ot

Legyen x_0^{FAPP} egy véletlen állapot

Ismételd a következőket maximum K(=EvalNum)-szor

  Hajts végre egy sztochasztikus lokális keresést

    x_0^{FAPP}-ból a V-t minimalizálva,

    a kapott trajektória $(x_0^{FAPP}, \ldots, x_m^{FAPP})$

  Legyen $x_0 := x_m^{FAPP}$

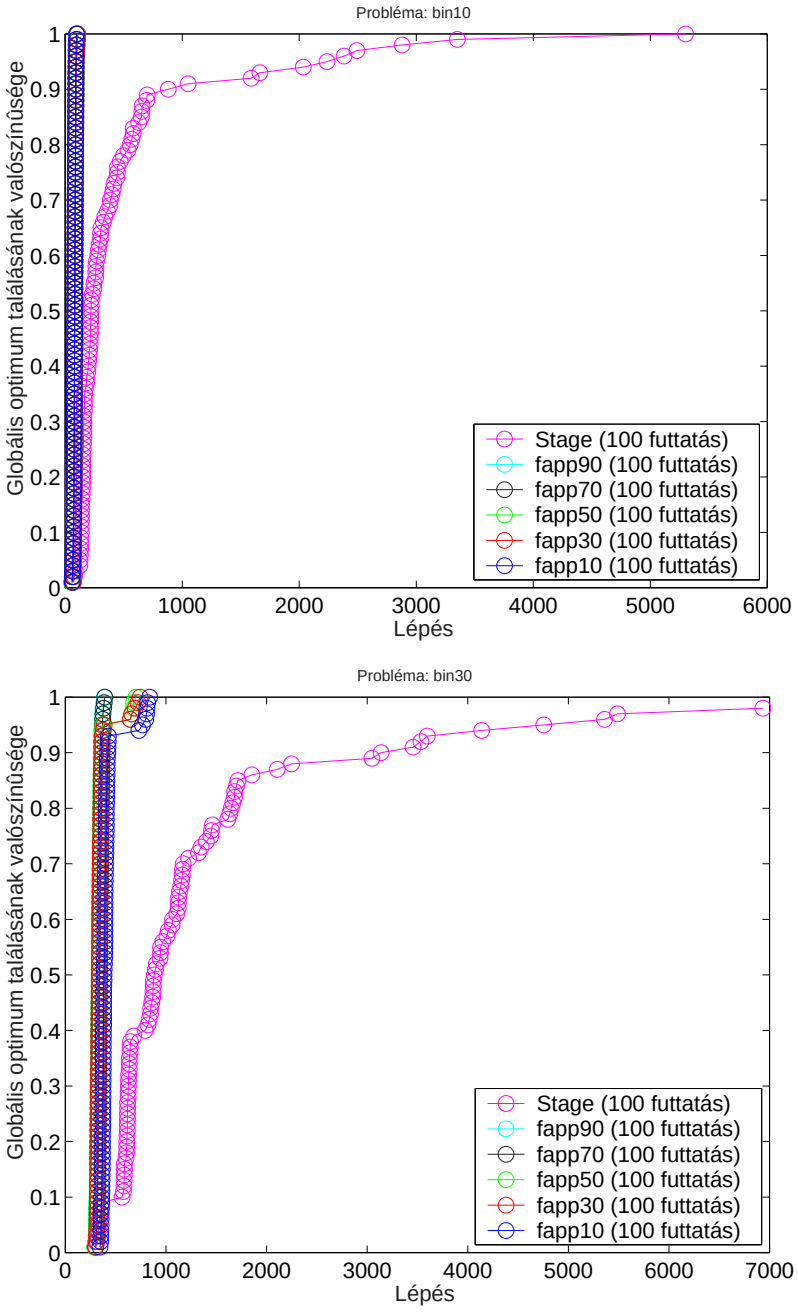
  Futtasd π_{loc}-ot x_0-ból az $Obj(x)$-et minimalizálva

    A kapott legjobb állapot legyen z

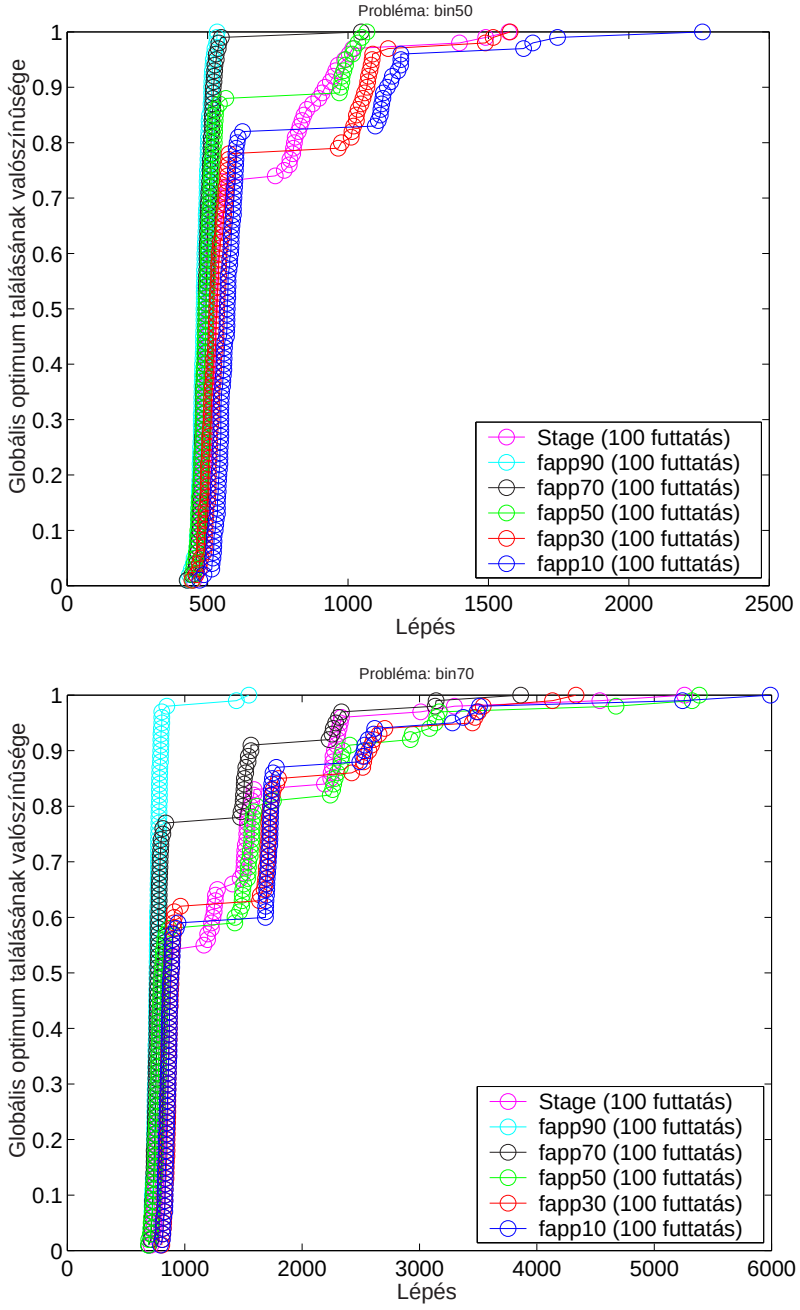
  Ha $z \neq x_0^{FAPP}$, akkor legyen $x_0^{FAPP} := z$

  különben lépj ki a ciklusból

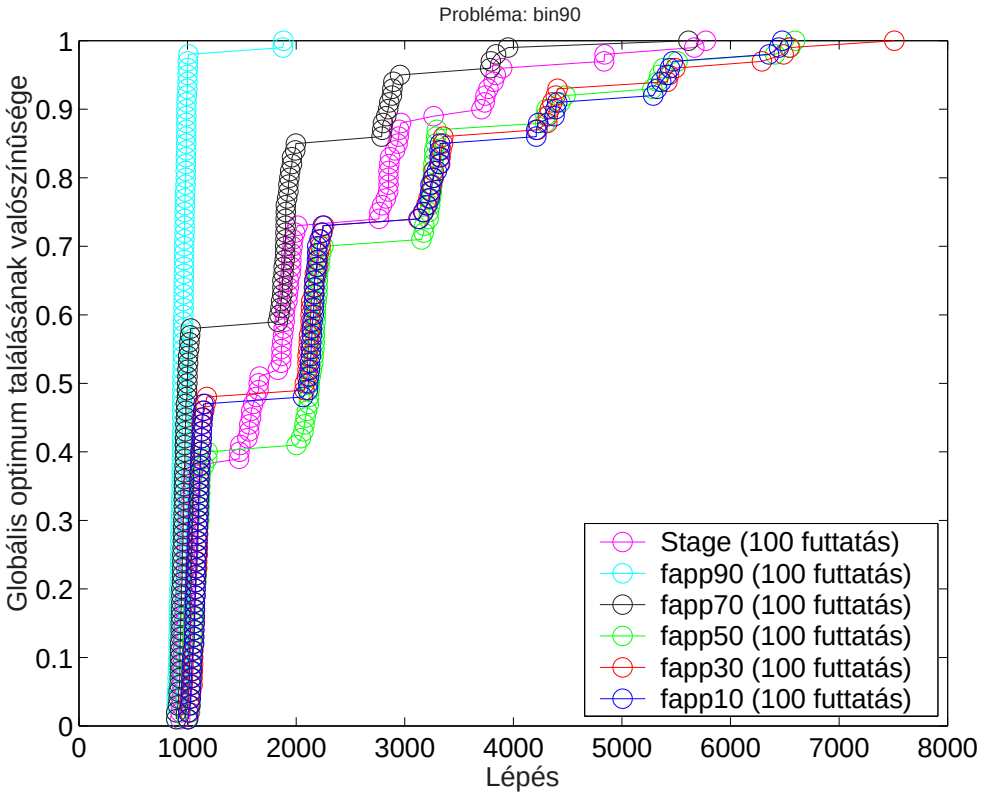
Térj vissza a legjobb költségű állapottal.



6. ábra. A FAPP-ok hatékonysága a *bin*10 és *bin*30-as problémán.



7. ábra. A FAPP-ok hatékonysága a $bin50$ és $bin70$-es problémán.



8. ábra. A FAPP-ok hatékonysága a *bin*90-es problémán.

4.4.3. A második módszer eredményei, és azok értelmezése

Az eddigi eredmények alapján mindenképpen az várható a 6, 7, 8-as grafikonoktól, hogy a kisebb problémákon a FAPP-ok jobban teljesítenek, mint a nagyobbakon (ez azon egyszerű okból is kell következzen, hogy a nagyobb feladatokban egy véletlen állapotból egy optimális állapot eléréséhez több lépés kell, mert több pálcikát tartalmaznak a nagyobb problémák). Hasonlóan az is várható, hogy egy konkrét problémán a nagyobb FAPP-ok jobban teljesítenek, mint a kisebbek. (Ez a grafikonon azt jelenti, hogy a nagyobb FAPP-ok görbéi gyorsabban nőnek, mint a kisebbek.) Ez eddigiek alapján ezt csak az első iterációra tudjuk, a kérdés az az, hogy igaz-e ez általában, azaz optimumot is gyorsabban talál-e a nagy FAPP a kicsinél. Abból a meggondolásból, hogy egy adott problémán a STAGE-nek tanulnia kell, míg a megfelelő FAPP használatával ez a tanulás kiküszöbölhető, várható az is, hogy a STAGE görbéje lassabban nő, mint a feladathoz tartozó (illetve a nála nagyobb összes) FAPP görbéje.

A grafikonok teljesen az elvárásoknak megfelelően alakultak. Az 5 ábrát összehasonlítva észrevehetjük, hogy a kisebb problémákon nagyon jól teljesítenek a FAPP-ok, míg a nagyobbakon rosszabbul: a *bin*10 problémán mindegyik FAPP 100 lépés alatt megtalálta az optimumot, míg a *bin*90-es problémán a legjobb FAPP, a FAPP90 is csak 1900-2000 lépésben tudta biztosan megtalálni az optimumot.

Az is jól látható, hogy a nagy FAPP-ok jobbak, mint a kicsit. Persze amikor a feladathoz képest jóval nagyobb FAPP-okat vizsgálunk, akkor ezen FAPP-ok eredménye nem nagyon különböznek. Erre mutat példát az egész *bin*10-es és *bin*30-as probléma a 6. ábrán vagy a bin50-es problémán a FAPP70 és FAPP90 a 7. ábra felső felén. Megfigyelhető az is, hogy a

nagyobb FAPP-ok görbéi gyorsabban emelkednek.

Harmadszor pedig az is leolvasható, hogy a STAGE rosszabbul teljesít, mint a probléma méretének megfelelő FAPP. Az ábrákon a STAGE futtatásaiból kapott görbét a lila szín jelölte. Látható, hogy a lila görbe a kicsi problémákon a legrosszabbul teljesít, míg a többin a középmezőnyben található.

4.4.4. Következmény

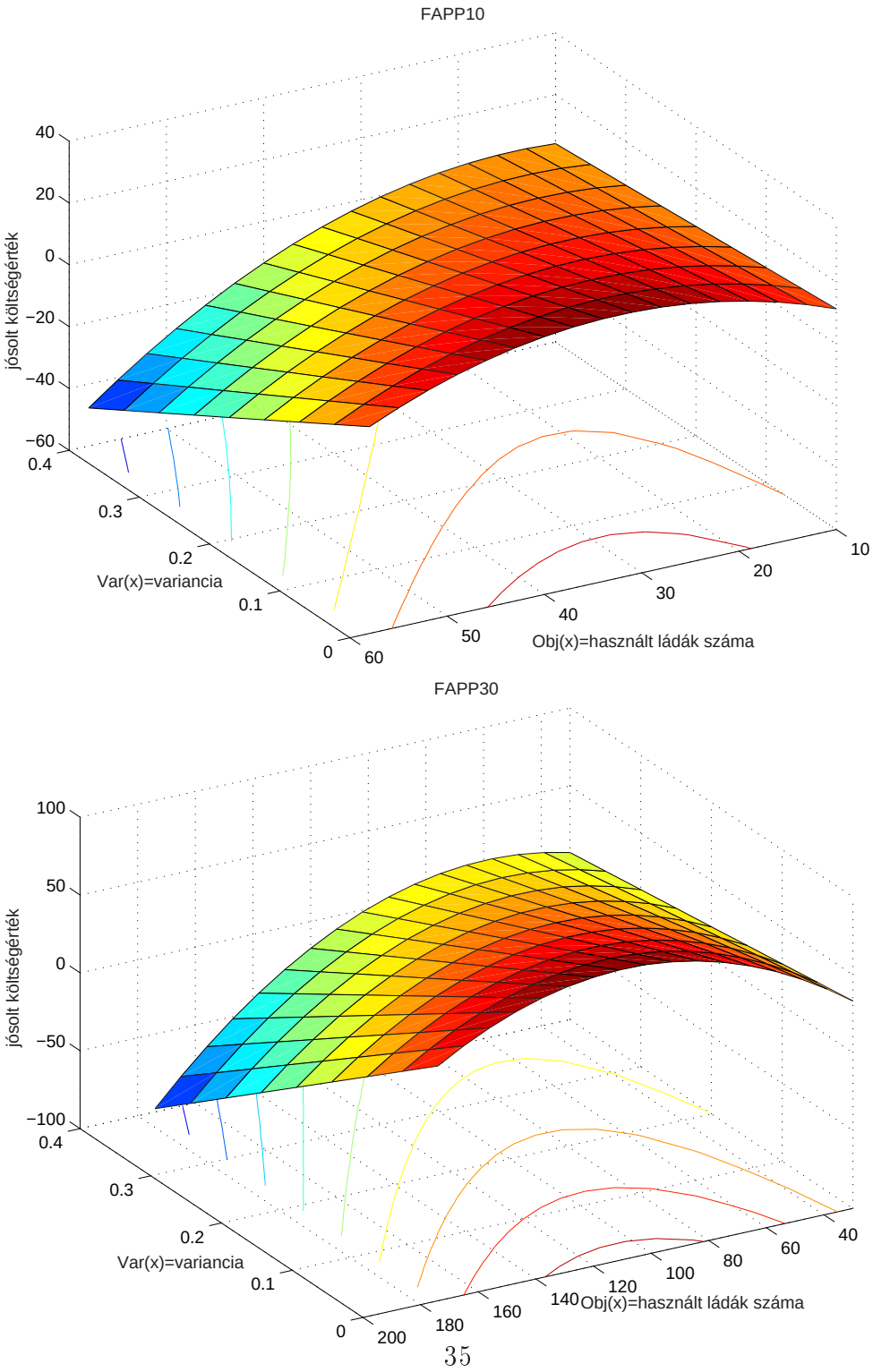
Az eredményekből egyértelműen látható, hogy tárolt FAPP-ok használata nemcsak az első iterációkban, hanem az egész STAGE-futtatás alatt jelentős teljesítménynövekedést adhat. Tehát, ha nincsenek túl nagy FAPP-jaink, hanem csak az éppen vizsgált probléma méretéhez hasonló méretű FAPP-unk van, akkor is érdemes lehet felhasználni ezt a FAPP-ot. További kutatási feladat lehet annak a vizsgálata, hogy melyik a hatékonyabb: az adott FAPP-ot tanulás nélkül használni (mint az általam használt második módszerben), vagy betölteni az adott FAPP-ot, de megengedni a tanulást (azaz STAGE-et használni, de úgy, hogy az elején mi inicializáljuk a kezdő-FAPP-ot a megadott FAPP-ra).

5. A kapott eredmények okának vizsgálata

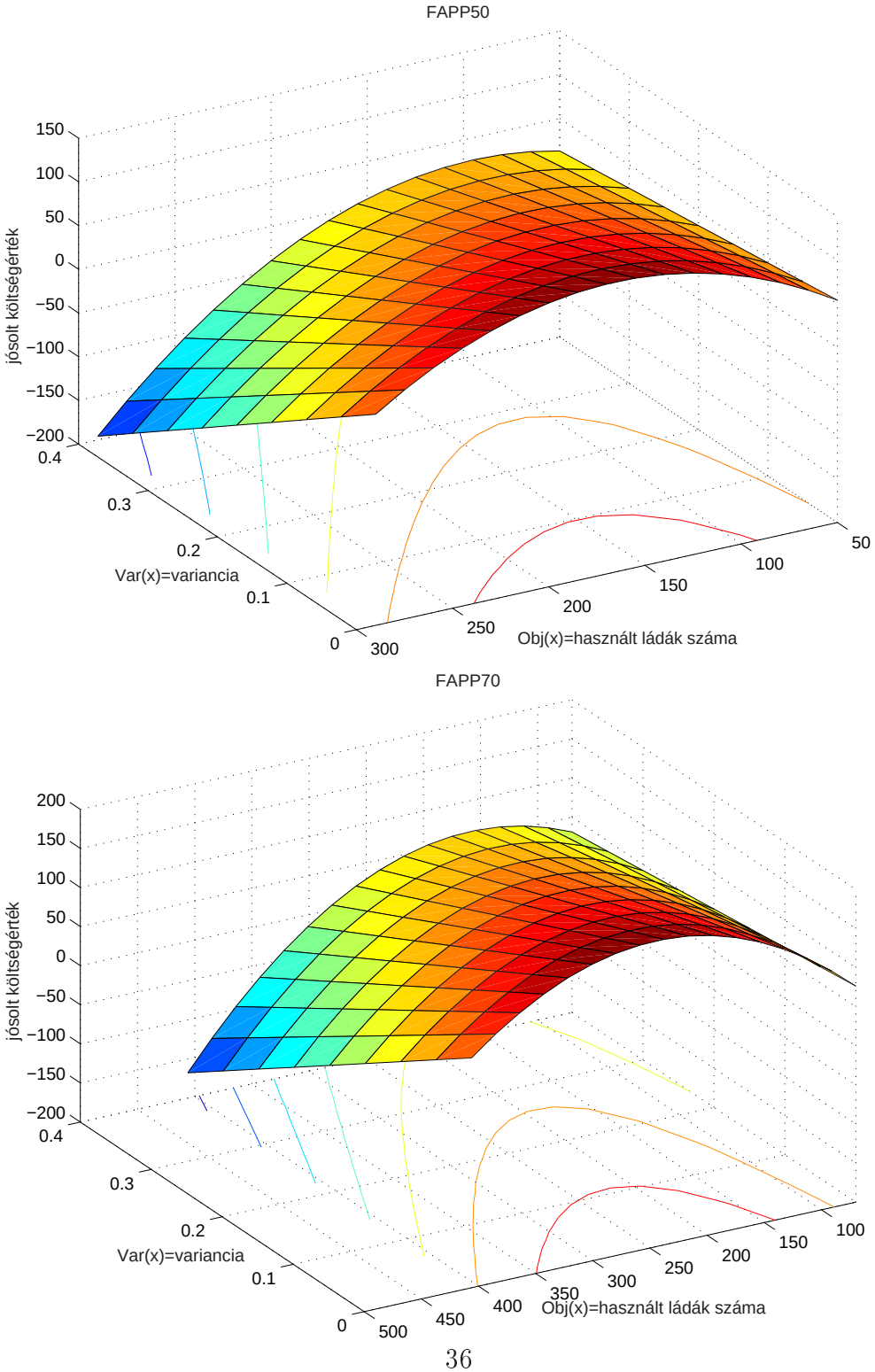
5.1. A FAPP-ok vizsgálata

Hogy megértsem, miért jobb a nagyobb probléma FAPP-ja, vettem STAGE-futtatásokat és ábrázoltam a kapott FAPP-okat. Ezek a 9, 10, 11. ábrán láthatóak. Ezeken az ábrákon a $bin10 \ldots bin90$-es problémák FAPP-jai közül ábrázoltam néhányat. A FAPP egy olyan függvény, amelyik a jellemzők teréből képez a valós számok halmazába, és egy adott jellemző-vektorhoz annak a költségnek a várható értékét rendeli, amelyik egy olyan lokális keresés végpontjának költségértéke, amely az adott jellemzőkkel rendelkező állapotból indul. A variancia egy 0 és 1 közötti valós érték, amely a vizsgált problémákon nem haladta meg a $0,4$-es értéket. Ezalapján a FAPP-okat csak a $[0; 0,4]$ intervallumbeli varianciaértékeknél ábrázoltam. A másik jellemző a költségfüggvény. Ez nem lehet több, mint a pálcikák száma, a minimuma pedig az ideális megoldás költségértéke, ami ismert az általam használt problémákban.

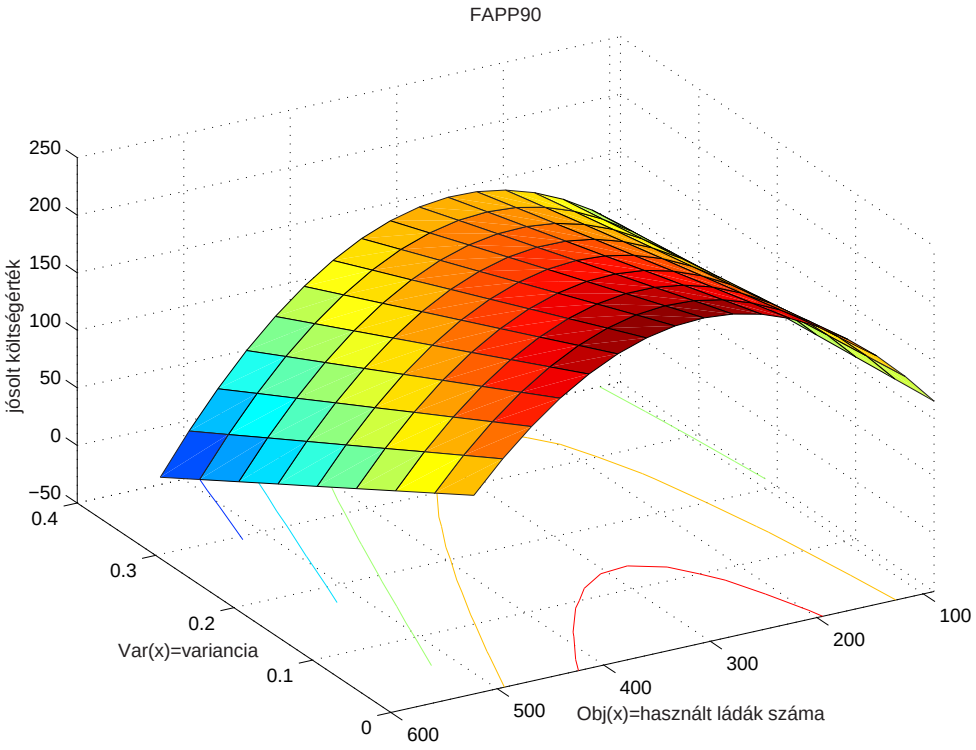
Ezen az 5 ábrán megfigyelhetjük, hogy a függvények irányultsága, alakja nagyon hasonló. Mindegyiknek van egy globális maximuma az ábrázolt régióban ott, ahol a variancia 0 és a költségérték körülbelül 3-szorosa a minimálisnak. Tehát egy ponttól kezdve a csökkenő költségérték mellett csökken a FAPP értéke, valamint növekvő varianciaérték mellett szintén csökken a FAPP értéke, de ez már az ábrázolt régió egészén igaz. Az is látszik, hogy a maximumhely eltolódik, ha nő a problémaméret.



9. ábra. A *bin*10 és *bin*30-as probléma FAPP-ja: a FAPP10 és FAPP30.



10. ábra. A *bin*50 és *bin*70-es probléma FAPP-ja: a FAPP50 és FAPP70.



11. ábra. A *bin*90-es probléma FAPP-ja: a FAPP90.

5.2. A trajektóriavizsgálatok

5.2.1. A futtatások bemutatása

A következő részben analizálni fogom az FLK trajektóriáit a FAPP-on. A FAPP60-at felhasználva lefuttattam 5-ször a teszt-algoritmust a $bin30, bin60$ és $bin90$-es problémákon, így kaptam 5-5 trajektóriát. Ezeket ábrázoltam a FAPP60-on. Az eredmények a 12. és 13. ábrán láthatóak. Egy trajektória két részből áll: egy pirosból, ami a FAPP-on való lokális keresés (FLK) trajektóriáját mutatja; valamint egy kékből, ami pedig az költségfüggvényen való lokális keresés (LK) trajektóriáját ábrázolja a FAPP-on. Miután az FLK végpontjából indul az LK, ez a két rész egy folytonos görbét alkot. Az LK végpontjához tartozó állapot költsége nem más, mint az első teszt-algoritmus eredménye.

5.2.2. FAPP-trajektóriák a kis problémán

Az 12. ábra felső részén, ahol a FAPP60-at teszteltem a $bin30$-as problémán, látható, hogy az FLK egy kiváló helyre vezette a keresést olyan állapotokon keresztül, amelyeknek nagy a varianciája. Ezután az LK-nak csak néhány lépést tett (esetleg egyet sem).

Ennél a problémánál az állapottérbeli állapotok a FAPP maximumától „jobbra” helyezkednek el. Így az FLK, a FAPP-ot minimalizálandó, a csökkenő költség irányába próbál menni, de az állapotok szomszédsági struktúrája miatt ezt csak egy, a nagyobb varianciájú pontokat érintő kerülővel képes megtenni. Persze eközben a FAPP értéke folyamatosan csökken.

Boyan disszertációjában [2] olvasható, hogy kis költségű megoldás olyan állapotokból indított kereséssel érhető el, amelyiknek relatív nem túl nagy a

költségértéke, viszont a varianciája nagy. Hasonló teljesül a mi esetünkre is, bár ebben az esetben maga az FLK végigviszi a keresést, és az LK-ra nem marad szinte semmi.

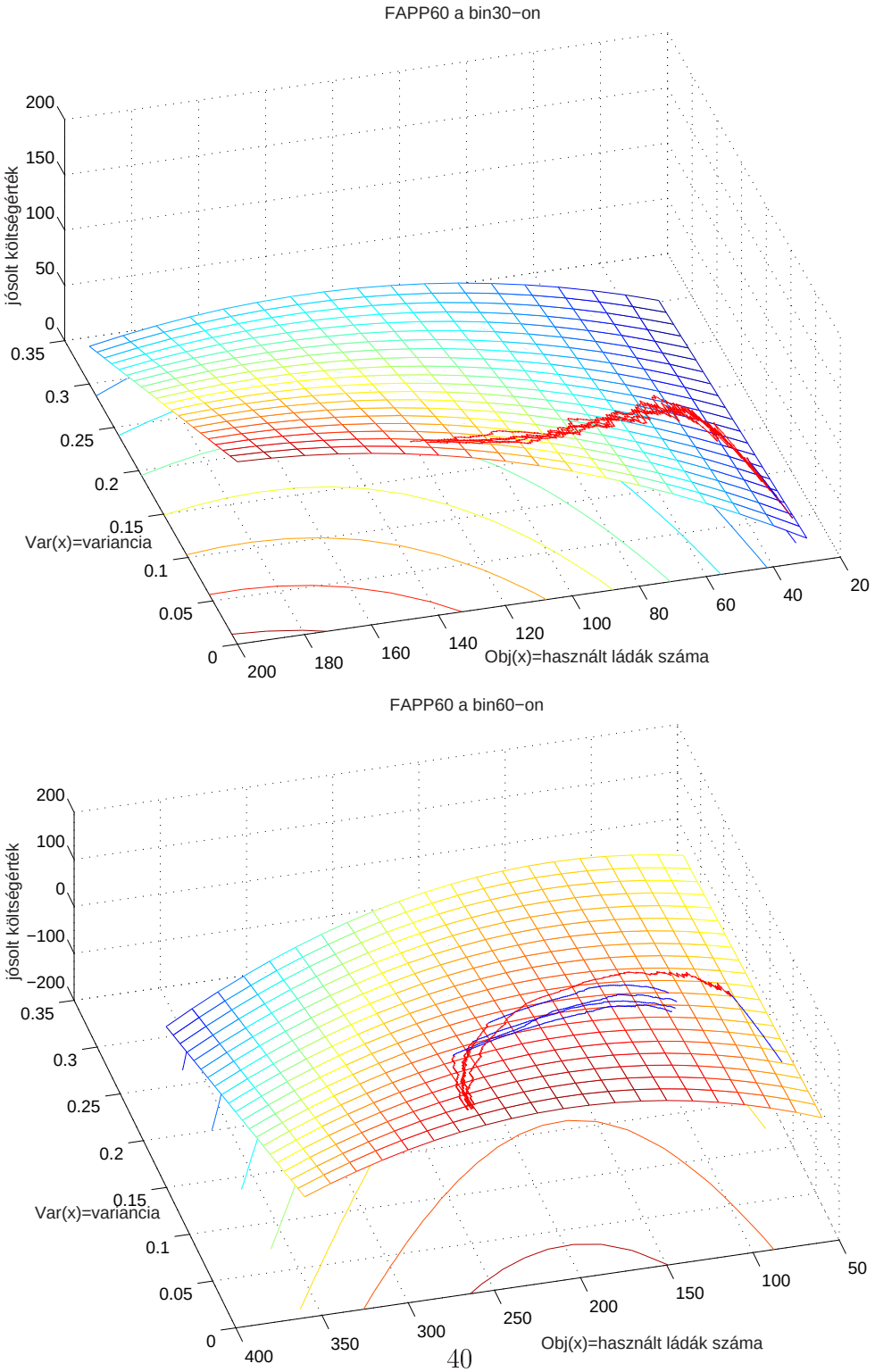
5.2.3. FAPP-trajektóriák a saját problémán

A 12. ábra alsó felén a FAPP60-at a saját problémáján teszteltem. Itt az látható, hogy 5-ből 4 alkalommal az FLK rossz irányba próbálta vinni a keresést, aminek a következménye, hogy az LK nem volt képes jó állapotot találni. Az 5. esetben viszont az FLK-nak sikerült eljutnia a FAPP maximumának „jobb oldalára", és onnan elért egy megfelelően nagy varianciájú pontot. Innen pedig LK már egy nagyon jó megoldáshoz elvezet. Ez már egy az egyben megegyezik a fentebb említett [2]-ben találhatókkal. Ez nem is meglepő, hiszen a STAGE-futtatásokban pont az történik, hogy egy FAPP-ot a saját problémáján használunk.

Annak, hogy rosszabbul teljesít a FAPP, az az oka, hogy a véletlen kezdőpont választás a FAPP maximumhelyének „bal" oldalára esik (bár a maximumhelyhez közel) és onnan a FAPP rossz irányba próbálja vinni a keresést. Persze a szomszédsági struktúra miatt a keresés néha elérheti a „jobb" oldalt, és onnan pedig ígéretes LK-kezdőpontba juthat, mint azt láttuk az 5. esetben. A STAGE azért működik, mert az első LK végpontja már általában a maximumhely „jobb oldalán" van és onnan a következő iterációk már jó irányba viszik a keresést, és így a STAGE elér egy megfelelően jó állapotot.

5.2.4. FAPP-trajektóriák a nagy problémán

A 13. ábrán azt figyelhetjük meg, hogy az FLK kezdőpontja a FAPP rossz oldalán van, a maximumhelytől elég távol. Ezért az FLK a lehető legrosszabb



12. ábra. A FAPP60 trajektóriái a *bin*30 és *bin*60-as problémán, a FAPP60 felületén ábrázolva.

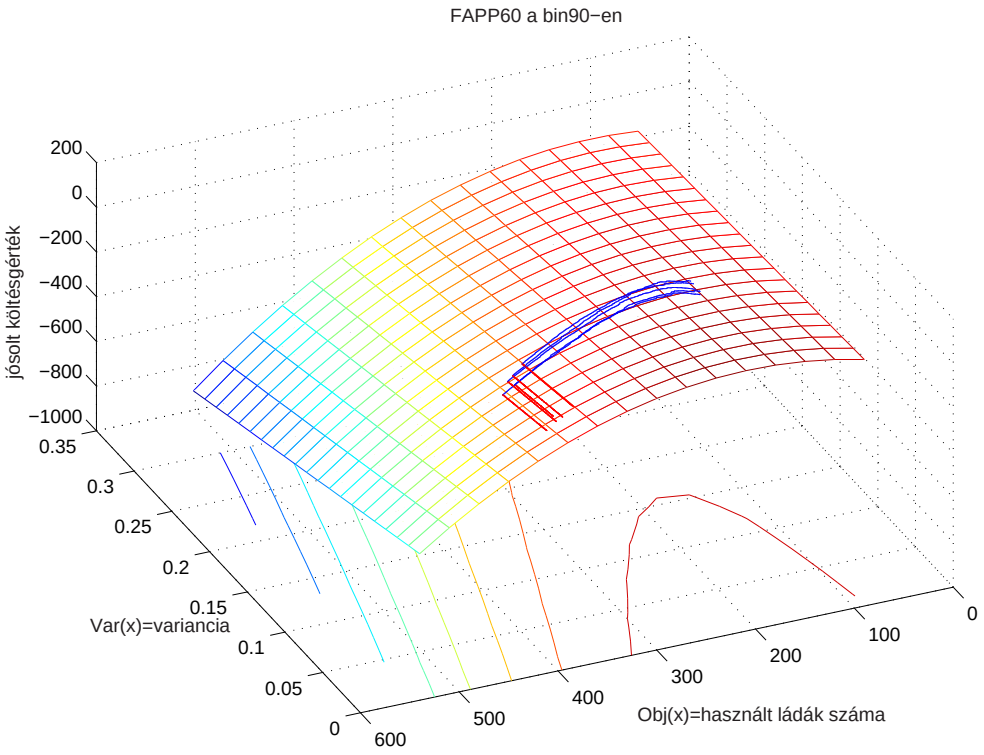
irányba próbál vinni. Persze egy véletlen kezdőállapotban a költségérték valószínűleg magas, így FLK azt nem nagyon tudja növelni, ezért a másik dimenzióban próbál süllyedni, aminek az az eredménye, hogy a végén kapunk egy nagy varianciájú, de viszonylag nagy költségű állapotot, amiből az LK nem tud jó állapotba vinni.

Az is látható, hogy az ezután végrehajtott LK végpontja nem biztos, hogy a „jobb oldalon" lesz, így az esetleges későbbi iterációkban előfordulhat, hogy az FLK visszafele vezeti keresést. Ez azt is magyarázza, hogy a STAGE jobban teljesít, mint a túl kicsi FAPP-ok, hiszen a STAGE előbb utóbb átalakítja a saját FAPP-ját, ami így már jó régiókat próbál elérni.

5.2.5. Trajektóriavizsgálatok összegzése

Az egyértelműen kiderült az eddigiekből, hogy a nagyobb problémákon miért teljesít rosszul a kisebb FAPP: A kis FAPP maximumhelye „túl jobbra" van, azaz túl kicsi a maximumhely költségérték-jellemzője. Így, miután ez az érték a nagy probléma egy véletlen állapotára nagy eséllyel magas, a FAPP rossz irányba viszi a keresést, ami után az LK nem képes megfelelően kis költségű állapotot elérni.

Az, hogy miért teljesít a nagy FAPP jól a kis problémákon, már nem ennyire egyértelmű. Az igaz, hogy a véletlen kezdőállapot a maximumhely megfelelő oldalára esik, de az egyelőre nem tisztázott, hogy a nagy FAPP erősebb lejtése, és a két dimenzió lejtésének kialakuló aránya miért kedvez a keresésnek, és miért van az, hogy már a FAPP-on való keresés magában képes egy majdnem optimális állapotot elérni.



13. ábra. A FAPP60 trajektóriái a *bin*90-es problémán, a FAPP60 felületén ábrázolva.

6. Konklúzió

Láttuk, hogy a nagyobb FAPP-ok kiváló teljesítményt nyújtanak a kis problémákon. Ezért, ha ismerünk egy nagyon nagy FAPP-ot, akkor jó megoldást találhatunk láda-pakolási problémák széles skálájára mindössze 2 lokális keresés felhasználásával. A STAGE használatához képest ezzel a módszerrel sok számítási idő spórolható meg, ugyanis nincs szükség az állapottér feltárására és tanulására.

Az eredményeim nagy része empirikus, ezért szükség van további kutatásokra: Egyik érdekes terület lehetne kapcsolatot teremteni a problémaméret és a megfelelő FAPP-együtthatók között. Ígéretes lehet megpróbálkozni a kvadratikus FAPP-ok lineárisakra történő cseréjével. Ez azért lehet jó, mert ha nagy FAPP segítségével kis problémát oldunk meg, akkor a megfigyelt trajektóriák a FAPP „jobb oldali", kis költségértékű részén vannak, és ezen a részen a FAPP közelítőleg lineáris.

Köszönetnyilvánítás

Köszönetemet szeretném kifejezni Lőrincz Andrásnak munkámhoz nyújtott segítségéért és útmutatásaiért. További köszönet illetti Hévizi Györgyöt és Grad Lászlót, mert bármikor fordulhattam hozzájuk problémáimmal és ők szívesen segítettek.

Hivatkozások

[1] D.S. Hochbaum. *Approximation Algorithms for NP - Hard Problems.* PWS Publishing Co., Boston, Editor, 1995.

[2] J. A. Boyan. *Learning Evaluation Functions for Global Optimization.* PhD thesis, School of Computer Science Carnagie Mellon University, Pittsburgh PA 15213, 1998.

[3] J. A. Boyan and A. W. Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–122, November 2000.

[4] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT Pess, 1998.

[5] Zs. Palotai, T. Kandár, Z. Mohr, T. Visegrády, G. Ziegler, P. Arató, and A. Lőrincz. Value prediction in the allocation problem of high level synthesis with IPs. *Journal on Applied Artifical Intelligence*, 2001. Accepted paper.

[6] G. Ziegler, Zs. Palotai, T. Cinkler, P. Arató, and A. Lőrincz. Value prediction in engineering applications. In L. Monostori, J. Váncza, and M. Ali, editors, *Engineering of Intelligent Systems. Proceedings of AIE/IEA 2001, Budapest Hungary*, volume 2070 of *LNAI*, pages 25–34. Springer, June 4-7 2001. ISBN 3-540-42219-6.

[7] W. Zhang and T.G. Dietterich. Solving combinatorial optimization tasks by reinforcement learning: General methodology applied to resource–

constrained scheduling. *Journal of Artificial Intelligence Research*, 2000. to appear.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990. ISBN 396-16-3029-3.

[9] E. Falkenauer. Operational Research Library. `http://www.ms.ic.ac.uk/info.html`.

Táblázatok jegyzéke

Ábrák jegyzéke

Tartalomjegyzék