# Täysin polynomiaikaisia approksimaatioalgoritmeja kapsäkkiongelmaan

TURUN YLIOPISTO
Informaatioteknologian laitos
Tietojenkäsittelytiede
LuK-tutkielma
7.4.2010
Ville Salo

TURUN YLIOPISTO
Informaatioteknologian laitos
SALO, VILLE: Täysin polynomiaikaisia approksimaatioalgoritmeja kapsäkkiongelmaan
LuK-tutkielma, 32 sivua
Tietojenkäsittelytiede
Huhtikuu, 2010

---

Kapsäkkiongelma on optimointiongelma, jossa syötteenä on joukko objekteja, joilla jokaisella on kokonaislukuarvoinen paino ja arvo. Tarkoituksena on valita alijoukko, johon kuuluvien objektien yhteenlaskettu arvo on mahdollisimman suuri, mutta joiden yhteenlaskettu paino pysyy halutun kapasiteetin sisällä. Ajatellaan siis, että on annettu kapsäkki, joka kestää tietyn määrän painoa, ja pyritään pakkaamaan sinne esineet, joiden yhteenlaskettu arvo on mahdollisimman suuri, ilman että kapsäkki hajoaa. Approksimointiongelmassa on lisäksi annettu tarkkuusparametri, ja pyritään löytämään ratkaisu, joka on optimaalisesta korkeintaan halutun tarkkuuden päässä.

Tässä tutkielmassa käsitellään muutamia syötteen koon ja tarkkuusparametrin suhteen polynomisia approksimaatioalgoritmeja. Tällaisia kutsutaan täysin polynomiaikaisiksi approksimaatioskeemoiksi, TPAAS. Algoritmit perustuvat yksinkertaisiin heuristiikkoihin, kuten objektien listassa esiintyvien erisuurien arvojen määrän pienentämiseen jakamalla nämä arvot sopivalla vakiolla, ja erilaisten algoritmien käyttämiseen arvokkaille ja vähemmän arvokkaille objekteille. Kaikki algoritmit käsitellään tarkasti, ja lähes kaikki tulokset johdetaan alusta saakka.

Esitettävät algoritmit ovat peräisin kahdesta 1970- ja 1980-lukujen vaihteessa kirjoitetusta artikkelista, joista toinen painottaa aikakompleksisuutta syötteen suhteen, ja toisen vahvuus on tilakompleksisuudessa. Lisäksi mainitaan muutamia muita tutkimusartikkeleissa julkaistuja tuloksia.

---

Avainsanat: kapsäkkiongelma, täysin polynominen approksimaatioskeema

# Sisältö

# 1 Johdanto

Kapsäkkiongelmat ovat optimointiongelmia, jossa pyritään valitsemaan kapsäkkiin yhteisarvoltaan mahdollisimman suuret objektit, joiden yhteispaino ei ylitä kapsäkin kapasiteettia. Approksimointiongelmassa on lisäksi annettu tarkkuusparametri, ja pyritään löytämään ratkaisu, joka on optimaalisesta korkeintaan halutun tarkkuuden päässä. Itse ongelmat ovat NP-vaikeita, mutta approksimointi onnistuu usein polynomiajassa.

Tässä tutkielmassa käsitellään pääasiallisesti 0–1 -kapsäkkiongelman täysin polynomiaikaista approksimointia. Tutkielmassa käsitellään vain approksimaatioita, jotka ovat todistetusti nopeita aikakompleksisuudeltaan. Käytännössä toteutuvia ajoaikoja ei käsitellä. *Täysin polynomiaikainen approksimointi* tarkoittaa, että algoritmi on polynomiaikainen sekä syötteen koon, että tarkkuusparametrin käänteisluvun suhteen. Pelkästään syötteen koon suhteen polynomisia approksimaatioalgoritmeja sanotaan *polynomisiksi approksimaatioalgoritmeiksi*, niitä ei käsitellä tässä tutkielmassa lainkaan.

Tämän tutkielman tulokset ovat enimmäkseen ensimmäisistä aiheesta kirjoitetuista artikkeleista 1970-luvulta, ja myös uudempia tuloksia 1990-luvulta on mainittu. Tutkielman päälähteet ovat artikkelit [L] ja [MO]. Lähteessä [L] esitetyt skaalauksen ja suuriin ja pieniin objekteihin jakamisen ideat ovat alun perin peräisin lähteestä [IK], joka esitti ensimmäisen TPAAS:n kapsäkkiongelmalle. Suurinta osaa lähteessä [L] esitetyistä alkuperäisista ideoista ei käsitellä. Suuressa osassa tutkielmaa käytetään lähteen [L] esitystapaa.

Työ koostuu neljästä luvusta, joista ensimmäinen on johdanto. Toinen luku määrittelee kapsäkkiongelman ja approksimoinnin käsitteen. Lisäksi luvussa mainitaan muutamia kapsäkkiongelman muunnelmia. Toisen luvun aika- ja tilakompleksisuuksia käsittelevässä aliluvussa määritellään muutamia kompleksisuusteorian peruskäsitteitä. Luokkien P ja NP määritelmät oletetaan kuitenkin tunnetuiksi.

Kolmannessa luvussa alkaa ongelman varsinainen käsittely. Ensimmäinen askel täysin polynomisen approksimaatioalgoritmin löytämiseen on pseudopolynomiaikaisen, tarkan ratkaisun etsiminen. Tämä on kolmannen luvun tärkein sisältö. Ongelman tarkan ratkaisun arvon etsimiseen käytetään dynaamista ohjelmointia täysin suoraviivaisesti. Dynaaminen ohjelmointi perustuu muistin käyttöön apuna, joten hieman vaikeuksia ilmenee, jos myös itse optimiratkaisu halutaan löytää (eikä siis vain sen arvoa). Tähän tarkoitukseen esitetään kaksi algoritmia.

Neljäs luku on matemaattisesti vaativin, ja siinä esitellään varsinaiset approksimaatioalgoritmit. Ensimmäinen esiteltävä algoritmi, ‘ahne algoritmi’, on hyvin yksinkertaiseen heuristiikkaan perustuva lineaariaikainen algoritmi, jonka antamaa $50\%$ tarkkuuden ratkaisua käytetään myöhemmissä algoritmeissa. Toinen esiteltävä algoritmi perustuu objektien arvojen skaalaamiseen, eli olennaisesti niiden pyöristämiseen pienempään lukujoukkoon. Laskemalla todetaan, että sopivan skaalauskertoimen valinta (ja sen jälkeen kokonaislukuihin pyöristäminen) antaa halutun tarkkuuden täysin polynomisesti.

Paras tutkielmassa esitettävä tulos saadaan, kun objektit jaetaan vielä ‘suuriin’ ja ‘pieniin’, ja käytetään pienille objekteille suoraan ahnetta algoritmia. Suuret objektit ovat arvoltaan tietyn kynnysarvon ylittävät objektit, ja pienet arvoltaan tätä pienemmät objektit. Tässä käytetään artikkelin [L] mahdollisesti hieman harhaanjohtavaa terminologiaa.

Tutkielmassa esitetään pieni mutta epätriviaali korjaus erääseen lähteen [L] algoritmiin. Lisäksi tietyt perustelut muuttuvat selkeämmiksi tutkielmassa esitettävän Pareto-rintaman käsitteen avulla. Lisäksi lähteen [L] tarkan algoritmin yhteydessä käyttämän tietorakenteen havaitaan syntyvän automaattisesti, kun objektilistojen tallennukseen käytetään LISP-tyylisiä listoja.

# 2 Kapsäkkiongelman ja käytettyjen notaatioiden määrittely

## 2.1 Kapsäkkiongelman määrittely

Tässä tutkielmassa käsitellään pääasiallisesti *0–1 -kapsäkkiongelmaa*, joten ‘kapsäkkiongelma’ viittaa jatkossa siihen. Kapsäkkiongelma on optimointiongelma, jossa syötteenä on joukko objekteja, joilla jokaisella on kokonaislukuarvoinen paino ja arvo. Tarkoituksena on valita alijoukko, johon kuuluvien objektien yhteenlaskettu arvo on mahdollisimman suuri, mutta yhteenlaskettu paino pysyy halutun kapasiteetin sisällä. Ajatellaan siis, että on annettu kapsäkki, joka kestää tietyn määrän painoa, ja pyritään pakkaamaan sinne esineet, joiden yhteenlaskettu arvo on mahdollisimman suuri, ilman että kapsäkki hajoaa.

Siis syötteenä on kapasiteetti $K \in \mathbb{N}$ ja järjestämätön joukko objekteja

$$U = \{(p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)\} = \{u_1, u_2, \ldots, u_n\}$$

missä $p_i \in \mathbb{N}$, $a_i \in \mathbb{N}$ $\forall i \in \{1, \ldots, n\}$. Merkitään myös

$$p = (p_1, p_2, \ldots, p_n)$$

$$a = (a_1, a_2, \ldots, a_n)$$

$$N = \{1, \ldots, n\}$$

$$P = \sum_{i \in N} p_i$$

$$A = \sum_{i \in N} a_i,$$

ja tarkoituksena on etsiä sellainen osajoukko $S \subseteq N$, että

$$A^* = \sum_{i \in S} a_i$$

on mahdollisimman suuri niin, että

$$\sum_{i \in S} p_i \leq K.$$

Optimointiongelma voidaan esittää myös seuraavassa muodossa:

$$X = \underset{\substack{x \in \{0,1\}^n \\ p \cdot x \leq K}}{\arg\max} a \cdot x,$$

jolloin muuttujan $X$ suhde muuttujiin $A^*$ ja $S$ on

$$A^* = a \cdot X$$

$$S = \{i : X_i = 1\}.$$

Esitämme tässä siis alijoukkoa $S$ $\{0, 1\}$-arvoisella vektorilla $X$, missä alkio $i$ on 1 jos ja vain jos objekti $i$ kuuluu alijoukkoon. Samaistamme käsittelyssä tällaiset vektorit niiden esittämien alijoukkojen kanssa.

## 2.2 Kapsäkkiongelman likimääräinen ratkaiseminen

Kapsäkkiongelman päätösversiossa lisäparametrina on haluttu arvo, jota suuremmaksi alijoukon arvojen summa yritetään saada (toisin kuin optimointiversiossa, jossa se yritetään

saada mahdollisimman suureksi). Tämä on tunnettu esimerkki NP-täydellisestä ongelmasta, eli NP:ssä olevasta ongelmasta, johon mikä tahansa NP:ssä oleva ongelma voidaan redusoida polynomiajassa. Selvästi optimointiversio kapsäkkiongelmasta ratkoo tämän erikoistapauksenaan, ja on näin ollen NP-vaikea. Suuria ongelman instansseja ei siis ole järkevää ratkaista optimaalisesti. Tässä tutkielmassa käsitellään vain optimointiongelmia, mutta kaikista kapsäkkiongelman versioista on luonnollisesti olemassa myös päätösversio.

Likimääräis- ali approksimointiongelmassa pyritään löytämään riittävän hyvä ratkaisu optimointiongelmaan. Tämä tarkoittaa niin sanotun likimääräisalgoritmin etsimistä. Likimääräisalgoritmi etsii tietyssä ajassa ratkaisun, jonka suhteellista virhettä, eli eroavuutta optimiratkaisusta, rajaa ylhäältä jokin 'ongelman koon' funktio $f$. Tällöin ratkaisu on niin sanottu $f$-approksimaatio. Esimerkiksi graafiongelmissa ongelman kooksi voidaan käsittää kaarten lukumäärä. Likimääräisalgoritmi saattaa olla olennaisesti nopeampi kuin tarkan ratkaisun etsivä algoritmi.

Yleensä approksimointialgoritmeja käytetään 'järkevien' ratkaisujen etsimiseen NP-vaikeisiin ongelmiin, koska optimiratkaisun etsimiseen ei tunneta polynomiaikaista algoritmia. Myös joitakin P:ssä olevia ongelmia voidaan approksimoida, jolloin niiden aikakompleksisuus laskee matalampiasteiseksi polynomiksi. NP-vaikeiden optimointiongelmien likimääräinen ratkaiseminen onnistuu kuitenkin usein polynomiajassa, kun taas nykytiedon mukaan tarkan ratkaisun kompleksisuus on eksponentiaalinen. Tämän vuoksi niiden approksimoituvuus on olennaisempi kysymys.

Tarkemmin, määrittelemme maksimointiongelman parina $(Z, H)$, missä mahdolliset ratkaisut ovat (diskreetin) avaruuden $Z$ pisteet ja $H : Z \to \mathbb{R}$ antaa kunkin ratkaisun hyvyyden. Kapsäkkiongelmassa tämä avaruus on $\{x \in 2^U : p \cdot x \leq K\}$, ja $H(x) = a \cdot x$. Maksimointiongelman optimiratkaisu on mikä tahansa $z \in Z$, jolle $\nexists z' \in Z$ siten, että $H(z') > H(z)$. Nyt algoritmi $A$ on tietyn ongelman $f$-approksimointialgoritmi, jos

$$\forall (Z, H) : H(z^*) - H(A(Z, H)) \leq f(H(z^*)) * H(z^*)^1,$$

[1]Koska tässä tutkielmassa käsitellään nimenomaan tehokkaita approksimaatioita, on mukavaa voida käyttää esimerkiksi termiä 0.1-approksimaatio. Käsiteltäessä approksimaatioalgoritmeja, jotka löytävät olennaisesti huonompia approksimaatioita, ja ratkaisu on aina vain esimerkiksi suurempi kuin optimiratkaisun neliöjuuri, olisi parempi määritelmä $f$-approksimaatiolle esimerkiksi $H(A(Z, H)) \geq f(H(z^*))$, jolloin voitaisiin puhua $\sqrt{n}$-approksimaatiosta meidän määritelmämme antaman $\frac{n-\sqrt{n}}{n}$-approksimaation sijaan.

missä $(Z, H)$ on mikä tahansa ongelman instanssi, ja $z^*$ sen jokin optimiratkaisu. Niin sanotussa $\epsilon$-approksimaatiossa $f$ on vakiofunktio $f(H(z^*)) = \epsilon$. Tällöin esimerkiksi $100$ on eräs $0.5$-approksimaatio ongelmaan, jonka optimaalinen ratkaisu on $150$, koska $150 - 100 \leq 0.5 * 150$.

Approksimointiongelmalle voidaan antaa lisäparametreja, jotka määräävät halutun approksimaatiotarkkuuden. Esimerkiksi tässä tutkielmassa esitettävät kapsäkkiongelman likimääräisalgoritmit ottavat syötteenä varsinaisen ongelman parametrien lisäksi $\epsilon$-arvon, ja tuottavat $\epsilon$-approksimaation polynomiajassa. Tällaisia algoritmeja kutsutaan *polynomiaikaisiksi approksimaatioskeemoiksi* (polynomial time approximation scheme). Jos algoritmi lisäksi toimii polynomiajassa $\epsilon$-arvon käänteisluvun suhteen, kutsutaan sitä *täysin polynomiaikaiseksi approksimaatioskeemaksi* (fully polynomial time approximation scheme). Esittelemämme algoritmit ovat myös tällaisia, käytämme jatkossa lyhennettä TPAAS.

Mainittakoon vielä, että kapsäkkiongelma on erityisen helposti approksimoituva. Olettaen, että $P \neq NP$, on olemassa ongelmia, jotka sallivat korkeintaan $0.5$-approksimaation, ja joissakin ongelmissa parhaan mahdollisen polynomiajassa laskettavissa olevan approksimaation $f$-raja lähestyy ykköstä ylipolynomisesti.

## 2.3 Ratkaistaessa tehtävät oletukset

Ongelmamme parametreina ovat $U$, $K$ ja $\epsilon$, missä

$$U = \{(p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)\}$$

on objektien $(p_i, a_i)$ joukko, $K$ on kapasiteetti, ja $\epsilon$ tarkkuusparametri. Tarkoitus on etsiä $\epsilon$-approksimaatio $U$:n ja $K$:n virittämän kapsäkkiongelman instanssin ratkaisulle. Kaikki muutkin edellä annetut merkinnät pätevät jatkossa, mutta käytämme paikoittain muuttujia $p$ ja $P$ myös muihin tarkoituksiin.

Aikakompleksisuuksien yhteydessä oletetaan, että aritmeettiset operaatiot vievät vakioajan $A$:n, $P$:n ja $n$:n kokoisilla luvuilla operoitaessa. Jos ongelmaa halutaan ratkoa mielivaltaisen suurille luvuille, ilmestyy kompleksisuuksiin näiden lukujen suhteen $\log$-kertoimia. Emme paneudu lukujen esityksen yksityiskohtiin tässä tutkielmassa.

Syötteenä olevien objektien järjestyksellä ei ole merkitystä algoritmin kannalta ja jär-

jestyksen suhteen ei seuraavassa tehdä mitään oletuksia. Syöte $U$ on annettu joukkona, mutta oletetaan, että voidaan viitata $O(1)$-ajassa yksittäisiin objekteihin $(p_i, a_i)$. Jatkossa oletamme seuraavat ominaisuudet objektien joukolta:

Ensinnäkin

$$p_i \leq K \text{ kaikilla } i \in \mathbb{N},$$

eli jokainen objekti mahtuu yksinään kapsäkkiin. Tämä voidaan helposti varmistaa lineaarisessa ajassa. Objektit käydään läpi yksi kerrallaan ja poistetaan objektit, joiden paino ylittää kapasiteetin.

Toisekseen

$$\sum_{i \in N} p_i > K,$$

eli kaikki objektit eivät yhtä aikaa mahdu kapsäkkiin. Jos ne mahtuvat, on optimiratkaisu triviaalisti $S = N$, ja tämä voidaan suoraan palauttaa algoritmin tuloksena. Selvästi tämäkin on tarkistettavissa lineaarisessa ajassa laskemalla yhteen objektien painot, sillä $K$:n kokoluokkaa olevien lukujen yhteenlaskun ajateltiin onnistuvan vakioajassa.

Kolmannekseen

$$p_i \neq 0, a_i \neq 0 \text{ kaikilla } i \in N,$$

eli painot ja arvot eivät ole nollia. Objektit, joiden arvo on nolla, voidaan ensin poistaa tarkastelusta, sillä selvästi yhteenlasketun arvon kannalta ei ole väliä otetaanko ne osajoukkoihin. Tämän jälkeen voidaan nollapainoiset objektit unohtaa tarkastelusta, ja lisätä lopuksi ratkaisuun. (Approksimoinnin yhteydessä tätä voisi optimoida suurentamalla virheparametria sen mukaan paljonko tässä saadaan ‘ilmaista arvoa’.)

## 2.4 Aika- ja tilakompleksisuudesta

NP-vaikealla ongelmalla tarkoitetaan ongelmaa, johon kaikki NP:ssä olevat ongelmat voidaan redusoida polynomiajassa. NP-täydellisyydellä ongelmalla tarkoitetaan NP-vaikeaa ongelmaa, joka on itse luokassa NP. Päätösongelmat, kuten kapsäkkiongelman päätösversio, ovat tyypillisiä NP-täydellisiä ongelmia. Niiden optimointiversiot ovat NP-vaikeita, sillä päätösongelmat ratkeavat optimointiongelman sivutuloksena, mutta ne eivät yleensä ole NP:ssä.

Yleensä kokonaislukuja käsittelevissä algoritmeissa oletetaan, että luvut on annettu $k$-kantaisessa lukujärjestelmässä, missä $k > 1$. Siis esimerkiksi yhteenlaskualgoritmin syötteen $45 + 83$ ajatellaan olevan kokoluokkaa $\lceil \log_{10} 45 \rceil + \lceil \log_{10} 83 \rceil = 4$, eikä kokoluokkaa $45 + 83 = 128$. Kutsumme lukuja parametrinaan ottavan ongelman unaariseksi versioksi ongelmaa, jonka syötteenä olevat luvut annetaan unaarisessa muodossa. Esimerkiksi laskettaessa yhteen luvut $45$ ja $83$ unaarisella yhteenlaskulla, olisi syöte muotoa $\underbrace{11\ldots1}_{45kpl} + \underbrace{11\ldots1}_{83kpl}$.

Määrittelemme *vahvasti NP-täydellisen ongelman* (strongly NP complete problem) ongelmana, joka on NP-täydellinen, ja jonka unaarinen versio on myös NP-täydellinen. Vastaavasti määritellään *vahvasti NP-vaikeat ongelmat*. *Pseudopolynomiaikaisella algoritmilla* tarkoitetaan algoritmia, joka ratkoo ongelman unaarisen version polynomisessa ajassa, toisin sanoen algoritmia, joka vie lukujen arvojen suhteen polynomisen ajan. Normaalistihan vaaditaan, että algoritmi vie polynomisen ajan lukujen esityksien pituuksien suhteen.

Vahvasti NP-vaikeille ongelmille on mahdotonta kehittää täysin polynomisia approksimaatioalgoritmeja [WP]. Myöskään kaikille heikosti NP-vaikeille ongelmille tällaista ei ole olemassa. Kapsäkkiongelma ovat erityisen helppo NP-vaikea ongelma siinä mielessä, että sille on olemassa TPAAS. Tällaisen löytämisessä on olennaista, että kapsäkkiongelmalle on olemassa pseudopolynominen ratkaisualgoritmi. Itse asiassa voidaan todistaa (tietyin ehdoin), että jollei tällaista ole olemassa, myöskään TPAAS:ta ei voida löytää [WP]. Seuraavassa esiteltävistä kapsäkkiongelman muunnelmista vain moniulotteiseen kapsäkkiongelmaan ei voida kehittää TPAAS:ta.

## 2.5 Kapsäkkiongelman muunnelmia

Ennen menetelmien käsittelyä mainitaan muutamia kapsäkkiongelman muunnelmia. Tässä tutkielmassa käsitellään näiden ongelmien approksimointia tai tarkkaa ratkaisemista hyvin kursorisesti. Kaikissa tapauksissa määrittelemme ongelmista optimointiversion.

Jos kapsäkkiongelmassa vaaditaan $p_i = a_i$ kaikilla $i \in N$, saadaan niin sanottu *alijoukon summa* -ongelma. Voidaan ajatella, että objekteilla ei ole erikseen painoa ja arvoa, vaan vain koko, ja pyritään maksimoimaan alijoukon yhteenlaskettu koko ylittämättä annettua kapasiteettia (tämä on myös järkevämpi määritelmä alijoukon summa -ongelmalle). Kolmas tapa ajatella asia on, että niin sanottu arvotiheys $\frac{a_i}{p_i} = 1$ jokaisella objektilla. Itse asiassa saamme olennaisesti saman ongelman jos oletamme arvotiheydet vain samoik-

si, emmekä välttämättä ykkösiksi. Alijoukon summa -ongelma on kapsäkkiongelman erikoistapaus, ja sitä voidaan tämän vuoksi approksimoida samalla algoritmilla (joskin tähän tarkoitukseen on olemassa tehokkaampikin algoritmi).

Alijoukon summa-ongelma voidaan määritellä seuraavasti:

$$\text{On annettu joukko lukuja } U_s = \{k_1, k_2, \ldots, k_n\} \text{ ja kokonaisluku } K$$

$$\text{Määrättävä } X = \underset{\substack{x \in \{0,1\}^n \\ k \cdot x \leq K}}{\arg\max}\, k \cdot x$$

Tässä tutkielmassa pääasiallisesti käsiteltävä versio kapsäkkiongelmasta on niin sanottu 0–1 -kapsäkkiongelma. Kutakin objektia saadaan siis ottaa kapsäkkiin enintään yksi kappale. Kun samaa objektia saadaan ottaa useampi kappale, saadaan ongelmalle kaksi variaatioita. *Rajoitetussa kapsäkkiongelmassa* etsitään multijoukkoa, johon sama objekti voidaan valita useita kertoja. Jokaisella objektilla on painon ja arvon lisäksi määrä, joka kuvaa sitä, montako kertaa sen saa käyttää. *Rajoittamaton kapsäkkiongelma* puolestaan määritellään kuten 0–1 -kapsäkkiongelma, mutta valintojen määrä saa olla mikä tahansa luonnollinen luku.

Sekä rajoittamaton, että rajoitettu kapsäkkiongelma voidaan selvästi redusoida tavalliseen kapsäkkiongelmaan. Yksi tapa tehdä tämä on tehdä objektista niin monta kopiota kuin mitä kapsäkkiin maksimissaan mahtuu, eli $\left\lfloor \frac{K}{p_i} \right\rfloor$ kappaletta. Rajoittamattomassa tapauksessa tehokkaampi tapa on ottaa objektista $(p, a)$ kahden potensseja $2^k$ kohti painotettuja kopioita $(p*2^k, a*2^k)$ mukaan, kunnes tällainen yksinään ylittää kapasiteetin. Tällöin kopioita tarvitaan vain logaritminen määrä. Rajoitetun tapauksen optimoiminen on hieman monimutkaisempaa, nyt tehdään kuten rajoittamattomassa tapauksessa, mutta viimeisen painotetun kopion $2^k$ sijaan valitaankin sellainen kerroin, että kaikkien kopioiden ottaminen alijoukkoon yhtä aikaa vastaa korkeintaan $m_i$:n objektin valintaa.

Tarkastellaan esimerkiksi tilannetta $m_i = 13$. Sanotaan, että kertoimien joukko $I$ generoi joukon $J$, jos $J = \{j : \exists x \in 2^I \text{ s.e. } j = \sum_{i \in x} i\}$. Nyt valittaisiin objektille $i$ kertoimet $\{1, 2, 4, 6\}$, sillä $1 + 2 + 4 + 6 = 13$. Luvut $\{1, 2, 4\}$ generoivat koko välin $[0, 7]$, joten selvästi luvut $\{1, 2, 4, 6\}$ generoivat välin $[0, 7] \cup [0+6, 7+6] = [0, 13]$, kun $6$ joko valitaan tai ei valita mukaan. Selvästi tällä menettelyllä saadaan rajoitettu tapaus redusoitua 0–1 -kapsäkkiongelman instanssiksi.

Rajoitetun kapsäkkiongelman määrittely:

$$\text{On annettu joukko monikkoja } U_r = \{(p_1, a_1, m_1), \ldots, (p_n, a_n, m_n)\} \text{ ja kokonaisluku } K$$

$$\text{Määrättävä } X = \underset{\substack{x \in \prod_{i \in N}\{0,\ldots,m_i\} \\ p \cdot x \leq K}}{\arg\max} \; a \cdot x$$

Rajoittamaton kapsäkkiongelma lyhyesti:

$$\text{On annettu joukko pareja } U_r = \{(p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)\} \text{ ja kokonaisluku } K$$

$$\text{Määrättävä } X = \underset{\substack{x \in \mathbb{N}^n \\ p \cdot x \leq K}}{\arg\max} \, a \cdot x$$

*Monivalintaongelmassa* objektin tilalla on luokka, eli joukko objekteja, joilla kaikilla on paino ja arvo. Jokaisesta luokasta valitaan korkeintaan yksi objekti. *Moniulotteisessa kapsäkkiongelmassa* taas on kapsäkkejä useampi kappale. Kukin objekti vie tietyn verran jokaisen kapsäkin kapasiteetista, ja pyritään maksimoimaan niiden yhteenlaskettu arvo siten, ettei minkään kapsäkin kapasiteetti ylity. Moniulotteiselle kapsäkkiongelmalle ei ole olemassa pseudopolynomiaikaista ratkaisualgoritmia [BD], minkä vuoksi sille ei ole myöskään täysin polynomista approksimaatioskeemaa [WP]. Monivalintaongelmalle on olemassa täysin polynomiaikainen approksimaatioalgoritmi, mutta emme käsittele sitä tässä tutkielmassa [BV].

# 3 Tarkka ratkaiseminen

## 3.1 Miksi tarkkaa ratkaisua tarvitaan approksimaatioalgoritmissa

Seuraavassa luvussa esiteltävän kapsäkkiongelman TPAAS:n perusidea on seuraava: Ensin etsitään kohtuullisen tarkka yläraja optimiratkaisulle. Tämän jälkeen skaalataan kaikki objektit, eli jaetaan niiden arvot jollain luvulla $s$, ja pyöristetään saadut skaalatut arvot, jolloin erilaisia arvoja on pienempi joukko. Sopivan skaalauskertoimen valinnassa tarvitaan optimiratkaisun ylärajaa. Tämän jälkeen ratkaistaan ongelma näitä pyöristettyjä arvoja $b$ käyttäen optimaalisesti dynaamisella ohjelmoinnilla. Saadusta ratkaisusta $r'$ muodostetaan alkuperäisen ongelman ratkaisu $r$ valitsemalla objektit, joiden skaalatut versiot $r'$:ssä olevat objektit ovat. Koska skaalaamme arvoja $a$, emmekä painoja $p$, on painojen summan oltava kapasiteetin rajoissa myös $r$:ssä.

Skaalauskertoimen valinta on tärkein osuus skaalausalgoritmin suunnittelua. Saatavan ratkaisun on pysyttävä $\epsilon$-tarkkuuden sisällä, mikä rajaa kertoimen valintaa ylhäältä. Toisin

sanoen liian suuri kerroin johtaa siihen, että vastaus on liian huono, koska skaalattu ongelma on liian kaukana alkuperäisestä. Toisaalta algoritmin on myös oltava polynomiaikainen. Tämä rajaa kertoimen valintaa alhaalta. Meidän on valittava riittävän suuri kerroin, jotta saatujen skaalattujen arvojen joukon koko olisi polynominen alkuperäisen syötteen pituuden suhteen. Jos kerroin on liian pieni, pysyy aikakompleksisuus eksponentiaalisena. Esimerkiksi jos käytetään kerrointa $1$, on ratkaistava skaalattu ongelma identtinen alkuperäisen kanssa, jolloin arvojen joukon koko on eksponentiaalinen syötteen pituuden suhteen. Käy ilmi, että nämä ongelmat voidaan ratkaista sopivalla kertoimen valinnalla, kunhan tiedetään yläraja optimiratkaisun arvolle.

## 3.2 Perusrekursio ja dynaaminen ohjelmointi

Tässä luvussa esitettävä algoritmi perustuu pääosin lähteen [L] tarkkaan algoritmiin, mutta esitystä ei seurata tarkasti.

Tarkan ratkaisun etsiminen dynaamisella ohjelmoinnilla perustuu seuraavaan rekursioyhtälöön:

$$Opt(0,k) = 0, k = 1, 2, \ldots, K$$

$$Opt(j,k) = \max\{a_j + Opt(j-1, k-p_j), Opt(j-1,k)\},$$

missä $Opt(j,k)$ on suurin arvo, joka voidaan saavuttaa ensimmäisillä $j$ objektilla, kun käytössä oleva kapasiteetti on $k$. Yhtälö seuraa siitä, että $j$:s objekti joko otetaan tai ei oteta mukaan. Jos objekti otetaan mukaan, täytyy jäljellä olevasta kapasiteetista vähentää $j$:nnen objektin paino $p_j$ laskettaessa optimiarvoa $j-1$:nnen tason objekteille, mutta arvo $a_j$ voidaan lopuksi lisätä tähän arvoon. Jos objektia ei oteta mukaan, palautuu optimin etsiminen suoraan $j-1$ objektin tapaukseen. Rekursion perustana on, että kapasiteetin $0$ alle ei mahdu yhtään objektia, ja nollasta objektista saavutettu arvo on nolla. Selvästi kaavan suora kääntäminen algoritmiksi tuottaisi eksponentiaaliaikaisen algoritmin, mutta koska parametrit $(j,k)$ kuuluvat aina joukkoon $N \times \{1, \ldots, K\}$, saadaan algoritmi välittömästi $O(nK)$-tilaiseksi ja -aikaiseksi taulukoimalla.

Koska skaalausvaiheessa skaalaamme arvoja, emmekä painoja, täytyy rekursioyhtälön antamaa ideaa hieman muokata. Ideana on nyt joka hetki pitää muistissa kaikki mahdolliset ratkaisut ensimmäisille $j$ objektille, ja lisätä objekteja ratkaisuehdokkaisiin yksi kerrallaan. Muuttujan $j$ annetaan käydä läpi arvot $0, \ldots, n$, ja kullakin iteraatiolla päivitetään lista $L$, jossa säilytetään optimiratkaisuja. Algoritmi on seuraavanlainen:

1: $L \leftarrow \{(0,0)\}$
2: **for** $j = 1$ to $n$ **do**
3: $\quad L \leftarrow L \cup \{(p + p_j, a + a_j) | (p,a) \in L\}$
4: $\quad L \leftarrow \{(p,a) \in L : p <= K\}$
5: $\quad L \leftarrow \{(p,a) \in L |\ \nexists (p',a') \in L :\ Dominoi((p',a'),(p,a))\}$
6: **end for**

missä Dominoi$((p',a'),(p,a)) \iff (p' < p \wedge a' \geq a) \vee (p' \leq p \wedge a' > a)$.

$L$ sisältää siis aluksi pelkän tyhjän (askel 1), ja tämän jälkeen $j$:nnellä iteraatiolla lisätään uusi objekti jokaiseen $j-1$ objektin ratkaisuun (askel 3). Tämän jälkeen poistetaan liian painavat ratkaisut (askel 4), ja lopuksi sellaiset ratkaisut poistetaan, jotka ovat jotain muuta ratkaisua huonompia sekä arvon että painon suhteen (askel 5). Uusi algoritmi on selvästi tilakompleksisuudeltaan luokkaa $O(\min\{K, A\})$, koska listassa ei voi olla yhtäaikaa kahta samanpainoista tai -arvoista ratkaisua.

Aikakompleksisuudeltaan algoritmi on myös luokkaa $O(n \min\{K, A\})$, kun $L$ pidetään muistissa sekä painon että arvon suhteen kasvavasti järjestettynä listana. Rivit 3 ja 4 vievät aikaa selvästi $O(\min\{K, A\})$. Rivillä 5 väite seuraa, kun rivin 3 yhdisteen yhteydessä suodatetaan dominoidut alkiot pois. Molemmissa yhdistettävissä listoissa $L$ ja $\{(p + p_j, a + a_j) | (p,a) \in L\}$ alkiot ovat kasvavassa paino- ja arvojärjestyksessä, ja jos ratkaisu $a$ dominoi ratkaisua $b$, on $a$:n ja $b$:n oltava eri listoissa. Näin ollen riittää valita listojen aluista alkioita kasvavassa järjestyksessä, ja poistaa ratkaisu, jos toisen listan ensimmäinen alkio dominoi sitä.

Edellä esitetty algoritmi löytää vain optimaalisen arvon, eikä objektijoukkoa, jolla se saavutetaan. Kun lisäksi pidetään yllä tietoa kunkin ratkaisun antavasta objektijoukosta, saadaan tilakompleksisuudeksi $O(n \min\{K, A\})$, sillä jokaiseen ratkaisuun listassa $L$ voi nyt liittyä $O(n)$-kokoinen lista objekteja. Aikakompleksisuus ei kasva, kun listoja pidetään esimerkiksi LISP-tyylisissä listoissa: Uuden alkion voi lisätä cons-operaatiolla listan alkuun $O(1)$-ajassa, eikä listoja koskaan tarvitse kopioida, sillä useampi lista voi jakaa saman hännän. Lawler [L] tekee tämän eksplisiittisellä tietorakenteella, mutta tulos on luonnollisesti täysin identtinen. Voidaan ajatella, että listassa $L$ säilytetään tässä objektijoukon löytävässä algoritmissa monikkoja $(p, a, O)$, missä $O$ on osoitin listaan objekteja.

Koska $O(n \min\{K, A\}) \subseteq O(nA)$, voimme tästä lähtien unohtaa $K$:n aika- ja tilakompleksisuuksista, sillä skaalatessa pienennämme nimenomaan $A$:ta, jolloin $A$:n arvo yleensä dominoi $K$:ta. Huomataan vielä, että algoritmin suorituksen aikana missään rat-

kaisussa ei arvo voi olla suurempi kuin optimaalisen ratkaisun arvo $A^*$, joten kompleksisuudet voidaan merkitä myös muodossaa $O(nA^*)$.

## 3.3 Pareto-optimaalisuus ja Pareto-rintama

Tässä luvussa esittelemme käsitteen, joka selittää $L$:n roolin tarkemmin. Oletetaan, että $X$ on mielivaltainen joukko, ja $<_X$ on sille täydellinen järjestys, toisin sanoen $\forall x, y \in X : x \neq y \implies x < y \vee y < x$. Sanotaan, että $(X, <_X)$ on täydellisesti järjestetty joukko. Merkitään loppuluvussa $<_X$:n sijasta $<$ selkeyden vuoksi.

Nyt voimme tehdä $X^m$:stä osittain järjestetyn joukon asettamalla

$$(x_1, \cdots, x_m) < (y_1, \cdots, y_m) \iff \forall i : x_i \leq y_i \wedge \exists i : x_i < y_i.$$

Jos ajatellaan, että $<$ on 'huonompi kuin'-relaatio, tarkoittaa määritelmä, että tässä osittaisessa järjestyksessä paremmuus tarkoittaa paremmuutta kaikilla mittareilla, eli jokaisessa vektoreiden indeksissä. Sanotaan, että osajoukolle $Y \subseteq X^m$ on $y \in Y$ *Pareto-optimaalinen* ($Y$:ssä), jos $\forall x \in Y : x \leq y$. Nyt voidaan määritellä $Y$:n *Pareto-rintama* kaikkien sen Pareto-optimaalisten alkioiden joukkona. Esimerkiksi jos molempien koordinaattien suhteen käytetään tavallista kokonaislukujen vertailua, niin joukon $(2,1), (1,2), (1,1)$ Pareto-rintama on joukko $\{(2,1), (1,2)\}$, sillä sen vektorit ovat vertailukelvottomia, ja dominoivat vektoria $(1,1)$.

Määritellään nyt, että *paino $a$ on parempi kuin paino $b$*, jos $a < b$. *Arvoille* taas määritellään, että *$a$ on parempi kuin $b$*, jos $b < a$. Tällöin dominoi-relaatio tarkoittaa yksinkertaisesti paremmuutta paino-arvo-pareille painon ja arvon täydellisistä järjestyksistä saatavalla osittaisella järjestyksellä. Esimerkiksi objekti $(p_1, a_1) = (1, 2)$ on parempi kuin objekti $(p_2, a_2) = (2, 1)$, sillä $p_1 = 1 < 2 = p_2$ ja $a_1 = 2 > 1 = a_2$, ja siten objekti $(p_1, a_1)$ dominoi objektia $(p_2, a_2)$.

Tällöin $L$:lle saadaan tarkka karakterisointi: iteraatiolla $k$ $L$ sisältää ensimmäisten $k$ objektin osajoukkojen paino-arvo-parien summien joukon koko Pareto-rintaman. Tämä on selvästi totta ennen yhtäkään iteraatiota. Jos väite on totta $k-1$ ensimmäisellä iteraatiolla, on se totta myös $k$:nnellä. Tämä seuraa siitä, että jos valitaan mielivaltaisen Pareto-optimaalinen osajoukon summa ensimmäisistä $k$ objektista, merkitään $Z = (p_{i_1} + \cdots + p_{i_l} + p_k, a_{i_1} + \cdots + a_{i_l})$, niin väite on triviaali, jos $k$ ei kuulu joukkoon. Muussa tapauksessa, merkitään $p_{i_l} = k$, täytyy myös osasumman $(p_{i_1} + \cdots + p_{i_{l-1}} + p_k, a_{i_1} + \cdots + a_{i_{l-1}})$ olla

Pareto-optimaalinen, jolloin se löytyy $L$:stä iteraatiolla $k-1$, ja siten Pareto-optimaalinen joukko $Z$ lisätään iteraatiolla $k$.

## 3.4 Vaihtoehtoinen tapa löytää optimaalisen arvon antava objektijoukko

Edellä esitelty työssä [L] käytettävä tapa löytää optimaalisen arvon antava objektijoukko ei ole ainoa mahdollinen. Toinen mahdollisuus olisi tarkistaa kutakin objektia kohti muuttaako sen poistaminen optimaalista ratkaisua.

Tässä algoritmissa etsitään ensin optimaalinen ratkaisu $A^*$ ongelmalle (jollakin tarkalla algoritmilla, joka ei löydä itse objekteja). Sitten käydään läpi objektit, $i$:nnen objektin kohdalla se poistetaan listasta, ja ratkaistaan ongelma optimaalisesti muille objekteille. Jos ratkaisu pysyy samana, voidaan $i$:s objekti unohtaa kokonaan, koska on oltava jokin optimaalinen ratkaisu, jossa sitä ei esiinny. Jos taas optimaalinen ratkaisu huononee, on $i$:s objekti mukana jokaisessa optimaalisessa ratkaisussa, jolloin se voidaan lisätä lopulliseen ratkaisuun, ja vähentää sen paino ja arvo seuraavissa iteraatioissa käytettävistä painosta ja arvosta. Selvästi algoritmi löytää optimaalisen arvon antavan objektijoukon ajassa $O(n^2 * \min\{K, A\})$ ja tilassa $O(\min\{K, A\})$, koska edellisen luvun algoritmi suoritetaan $O(n)$ kertaa, ja ratkaisujen listassa ei tarvitse säilyttää objektijoukkoja.

Lähteessä [MO] tehdään vaihtokauppa aika- ja tilakompleksisuuden välillä, eikä osaratkaisuihin tallenneta objektilistaa, vaan se etsitään myöhemmin peräytymishaulla. Arvojen suhteen algoritmi toimii yhä samoin, mutta nyt listaan $L$ tallennetaan joukon $O$ sijasta vain viimeinen siihen lisätty objekti $o$. Merkitään tätä viimeisen lisätyn objektin muistavaa algoritmia $A_{yksi}$, ja seuraavaksi kehitettävää sitä käyttävää koko objektijoukon löytävää algoritmia $A_{kaikki}$. $A_{kaikki}$ toimii seuraavasti:

Ensin $A_{yksi}$ suoritetaan kerran koko joukolle, ja valitaan listasta $L$ monikko $(p, a, o)$, jonka arvo on suurin. Koska tällöin eräässä optimaalisessa ratkaisussa viimeinen lisätty objekti on $o$, voidaan kaikki objektilistassa $o$:n jälkeen (siinä järjestyksessä, jossa objektit käydään tarkassa algoritmissa läpi) tulevat objektit pudottaa pois, ja $o$ voidaan lisätä lopulliseen ratkaisuun. Nyt riittää saada ratkaistuksi ongelma, jossa objektien joukko on $\{1, \ldots, o-1\}$ ja kapasiteetti on $K - p_o$. Tätä ei suoraan ratkaista rekursiivisesti, vaan objektien joukko jaetaan ensin mahdollisimman tarkasti kahtia muuten mielivaltaisiin osajoukkoihin $J'$ ja $J''$. $A_{yksi}$ suoritetaan näille, olkoot ratkaisut sisältävät listat $L'$ ja $L''$. Huomaa, että tämä

kutsu ei ole rekursiivinen.

Tarkastellaan nyt löydettyä monikkoa $(p, a, o)$ vastaavaa alkuperäisen ongelman mielivaltaista optimaalista ratkaisua $\{j_1, \ldots, j_k\}$, missä $j_k = o$ on viimeinen valittu objekti, ja jonka paino siis on $p$ ja arvo $a$. Objektin $o$ painoa ja arvoa merkitään normaalisti $p_o$ ja $a_o$. Joukon $\{j_1, \ldots, j_{k-1}\}$ elementit ovat jakaantuneet jollakin tavoin joukkoihin $J'$ ja $J''$, merkitään menettämättä yleisyyttä $\{j_1, \ldots, j_m\} \subseteq J'$ ja $\{j_{m+1}, \ldots, j_{k-1}\} \subseteq J''$. Tällöin paras $L'$:ssa oleva monikko, jonka paino on korkeintaan $p_{j_1} + \ldots + p_{j_m}$, on arvoltaan korkeintaan $a_{j_1} + \ldots + a_{j_m}$, koska muuten ratkaisu $\{j_1, \ldots, j_k\}$ ei olisi optimaalinen. Siis $(p_{j_1} + \ldots + p_{j_m}, a_{j_1} + \ldots + a_{j_m}, o') \in L'$ jollekin $o'$. Samoin $(p_{j_{m+1}} + \ldots + p_{j_{k-1}}, a_{j_{m+1}} + \ldots + a_{j_{k-1}}, o'') \in L''$ jollekin $o''$.

Siis listoista $L'$ ja $L''$ löytyy varmasti sellaiset $(p', a', o') \in L'$ ja $(p'', a'', o'') \in L''$, että $p' + p'' + p_o = p$, $a' + a'' + a_o = a$, ja siten riittää rekursiivisesti ratkaista $A_{kaikki}$ joukoille $J'$ ja $J''$ kapasiteeteilla $p'$ ja $p''$. Löydetyt objektijoukot $O'$ ja $O''$ liitetään sitten yhteen, ja lopullinen palautettava objektijoukko on $O' \cup O'' \cup \{o\}$, joka on varmasti optimaalinen.

Ennen rekursiivista kutsua tarvitaan aikaa $O(n * \min\{K, A\})$ ja tilaa $O(\min\{K, A\})$ algoritmin $A_{yksi}$ suorittamiseen kolme kertaa. Kahtiajako vie aikaa $O(n)$. Parien $(p', a', o')$ ja $(p'', a'', o'')$ löytäminen listoista $L'$ ja $L''$ vie $O(\min\{K, A\})$, koska ne ovat kasvavassa järjestyksessä sekä painojen että arvojen suhteen. Kutsutaan tätä etsintää skannaukseksi.

Rekursion tasolla $k$ ongelmia on ratkaistavana korkeintaan $2^k$, ja näitä tasoja voi olla korkeintaan $\log n$. Jokaisella tasolla ongelman $A_{yksi}$ ratkaisemiseen käytetään yhä aikaa $O(n * \min K, A)$, sillä $\{1, \ldots, n\}$ on partitioitu tasolla olevien kutsujen välille, ja $\min\{K, A\}$ on sama joka kutsulla, ja siten $A_{yksi}$-aliongelmien ratkaisemiseen kuluu aikaa yhteensä $O(n \min\{K, A\} \log n)$.

Jokaisella rekursion tasolla $k$ skannaus suoritetaan $2^k$ kertaa. Siis yhteensä skannauksia suoritetaan korkeintaan

$$\sum_{i=1}^{\lfloor \log n \rfloor + 1} 2^i < 2n,$$

ja siten skannaukset vievät aikaa $O(n \min\{K, A\})$. Yhteensä aikakompleksisuus on siis $O(n \min\{K, A\} \log n)$, ja tilakompleksisuus on $O(n + \min\{K, A\})$, missä $\min\{K, A\}$ on kunkin yksittäisen kutsun käyttämä väliaikainen tila, ja $n$ tarvitaan objektijoukon tallentamiseen. Kukin kutsu voi esimerkiksi merkitä alkuperäiseen taulukkoon mitkä objektit ratkaisuun otetaan.

| Lähde | Aikakompleksisuus | Tilakompleksisuus |
|---|---|---|
| Lawler | $O(n \log \frac{1}{\epsilon} + (\frac{1}{\epsilon})^4)$ | $O(n + (\frac{1}{\epsilon})^3)$ |
| Magazine & Oguz | $O(n^2 \log n * \frac{1}{\epsilon})$ | $O(n * \frac{1}{\epsilon})$ |
| Kellerer & Pferschy | $O(n \log n + \min\{n, \frac{1}{\epsilon} \log \frac{1}{\epsilon}\} * (\frac{1}{\epsilon})^2)$ | $O(n + (\frac{1}{\epsilon})^2)$ |

Taulukko 1: Kapsäkkiongelman approksimaatioalgoritmien aika- ja tilakompleksisuuksia

# 4 Approksimointi

## 4.1 Kapsäkkiongelman kompleksisuustuloksia

Tässä tutkielmassa käsitellään julkaisuissa [L] ja [MO] annettuja tuloksia. Näistä osa on alun perin julkaisusta [IK]. Taulukko 1 listaa näiden artikkelien sekä lähteen [KP] esittämien algoritmien aika- ja tilakompleksisuudet.

Lähteessä [L] rakennetaan approksimaatioalgoritmi seuraavissa vaiheissa: Ensimmäiseksi arvot jaetaan tietyllä skaalauskertoimella, ja ratkotaan ongelma tarkasti tälle pienemmälle arvojoukolle. Tällä tekniikalla saavutetaan aika- ja tilakompleksisuus $O(n^2/\epsilon)$. Jakamalla objektit 'suuriin ja pieniin' saadaan tehokkaampi approksimaatio. Ongelma ratkaistaan skaalaavalla algoritmilla suurille objekteille, ja pienille objekteille myöhemmin esiteltävällä ahneella $50\%$-approksimaatiolla. Näin saatavan algoritmin aika- ja tilakompleksisuuksille ovat molemmille sekä $O(\frac{n}{\epsilon^2})$ että $O(\frac{n^2}{\epsilon})$ ylärajoina.

Koska skaalaamisen jälkeen objektien erilaisten arvojen joukko pienenee, esiintyy samoja arvoja niin useita kertoja, että osa niistä voidaan unohtaa. Tähän perustuu seuraava approksimaatio, jolla aika- ja tilakompleksisuudet putoavat luokkiin $O(n \log \frac{1}{\epsilon} + (\frac{1}{\epsilon})^4 \log \frac{1}{\epsilon})$ ja $O(n + (\frac{1}{\epsilon})^4 \log \frac{1}{\epsilon})$. Seuraava idea on skaalata arvoja sitä enemmän, mitä suurempia ne ovat. Tämä tuottaa aika- ja tilakompleksisuudet $O(n \log \frac{1}{\epsilon} + \frac{1}{\epsilon^4})$ ja $O(n + (\frac{1}{\epsilon})^4)$. Lopullinen kompleksisuus saavutetaan pienellä lisämodifikaatiolla suurten objektien laskemisvaiheeseen.

Lawlerin [L] tavoite oli siis selvästi pudottaa $n$:n kerroin mahdollisimman pieneksi. Tämä suoritetaan jossain määrin $\frac{1}{\epsilon}$-kompleksisuuden kustannuksella, sillä $\frac{1}{\epsilon^4}$ on hyvin epäkäytännöllinen kompleksisuus. Tässä tutkielmassa käsitellään lähteen [L] esittämät approksimaatioalgoritmit suuriin ja pieniin jakavaan algoritmiin asti.

Julkaisun [MO] approksimaatioalgoritmi perustuu samaan ideaan kuin julkaisun [L],

paitsi että nyt käytetään dynaamisessa ohjelmoinnissa kappaleessa (Vaihtoehtoinen tapa löytää optimaalisen arvon antava objektijoukko) esiteltyä tekniikkaa. Kompleksisuudet $O(\frac{n^2}{\epsilon})$ ja $O(\frac{n}{\epsilon})$ ovat julkaisun [L] rajoja huonompia $n$:n suhteen, mutta alempaa astetta $\frac{1}{\epsilon}$:n ja $n$:n polynomina. Tilakompleksisuus $O(\frac{n}{\epsilon})$ on varsin hyvä. Tämä algoritmi esitellään myös tässä tutkielmassa.

## 4.2 Ahne algoritmi

Tarkoituksena tässä ja seuraavissa aliluvuissa on löytää approksimaalinen ratkaisu, jonka arvo on $D^*$ siten, että

$$A^* - D^* \leq \epsilon A^*,$$

missä $A^*$ on annetun ongelmainstanssin optimaalinen ratkaisu.

Ensimmäinen approksimaatio, jota käytämme myöhemmin apuna paremmissa approksimaatiossa, on niin sanottu ahne algoritmi. Algoritmi antaa aina $50\%$-approksimaation, ja se voidaan suorittaa lineaarisessa ajassa $n$:n suhteen. Esitämme aluksi yksinkertaisemman $O(n \log n)$-aikaisen version algoritmista.

Ahne algoritmi etsii kaksi yksinkertaista approksimaatiota, joiden antamista arvioista valitaan suurempi, merkitään $A_{hne}$. Ensiksi se etsii objektien listasta arvokkaimman, merkitään sitä $a_{max}$. Tämä onnistuu ajassa $O(n)$. Toinen vaihe on hivenen monimutkaisempi: Objektin $(p, a)$ *arvotiheydeksi* sanotaan lukua $a/p$. Ahne algoritmi käy objektit läpi laskevassa arvotiheysjärjestyksessä, joten aluksi objektien lista järjestetään $O(n \log n)$-ajassa tähän järjestykseen. Tämän jälkeen algoritmi poimii objekteja listan alusta, kunnes seuraavan objektin ottaminen mukaan saisi valittujen objektien yhteispainon ylittämään kapasiteetin. Merkitään $j$:llä viimeistä näin valittua objektia.

Siis

$$A_{hne} = \max\{a_{max}, a_1 + \ldots + a_j\}$$
$$p_1 + \ldots + p_j \leq K < p_1 + \ldots + p_{j+1},$$

mistä saadaan

$$\begin{aligned} 2A_{hne} &\geq a_1 + \ldots + a_j + a_{max} \\ &\geq a_1 + \ldots + a_j + a_{j+1} \\ &> A^*, \end{aligned}$$

missä viimeinen arvio seuraa siitä, että objektit ovat laskevassa arvotiheysjärjestyksessä, joten jos $a_1 + \ldots + a_{j+1}$ saavutettaisiin jollain muulla objektijoukolla kuin $1, \ldots, j+1$, niin tämän joukon paino olisi vähintään $p_1 + \ldots + p_{j+1} > K$. Siis ahne algoritmi tuottaa $50\%$-approksimaation. Käytännössä tulos on yleensä pikemminkin $10\%$ luokkaa.

## 4.3 Yksinkertainen skaalaus

Ensimmäisen skaalausapproksimaation suoritamme skaalaamalla arvot vakiokertoimella $s$, skaalattuja arvoja merkitsemme $b_i = \lfloor a_i/s \rfloor$. Merkitään $E = A^*/s$. Koska tarkan algoritmin aika- ja tilakompleksisuus olivat molemmat $O(nA^*)$, ovat kompleksisuudet tälle skaalatulle versiolle $O(nE)$. Pyöristysvirheet eivät luonnollisesti vaikuta aikakompleksisuuksiin.

Lattiafunktion määritelmästä nähdään

$$b_i s \leq a_i < (b_i + 1)s,$$

joten

$$0 \leq \sum_{i \in J} a_i - s b_i \leq s|J| \leq sn \tag{1}$$

minkä tahansa indeksijoukon $J \subseteq N$ yli (jokainen virhe korkeintaan $s$, ja joukossa korkeintaan $n$ objektia). Nyt kun $A^*$ on alkuperäisen ongelman optimiratkaisu (pelkkä arvo), $B^*$ alkuperäisen ongelman optimaalisen ratkaisun antavien objektien skaalattujen arvojen summa, $C^*$ optimaalinen ratkaisu skaalatulle ongelmalle (pelkkä arvo), ja $D^*$ skaalatun ongelman optimaalisen ratkaisun antavia objekteja vastaavien skaalamattomien objektien arvojen summa, saadaan

$$A^* - sB^* \leq sn, \tag{2}$$

$$C^* \geq B^*, \tag{3}$$

$$D^* \geq sC^*, \tag{4}$$

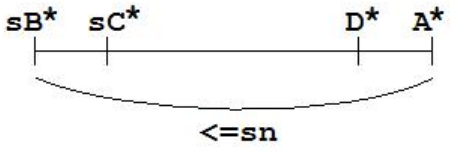
joten

$$A^* - sC^* \leq sn,$$

mistä seuraa

$$(0 \leq)\ A^* - D^* \leq sn,$$

katso kuva 1.




Kuva 1: Ratkaisujen keskinäiset suhteet.

Epäyhtälöt (2) ja (4) seuraavat epäyhtälöstä (1). Epäyhtälö (3) seuraa siitä, että $B^*$ on eräs skaalatun ongelman ratkaisu, ja $C^*$ on sen optimaalinen ratkaisu.

Siis kun ratkaisemme skaalatun ongelman optimaalisesti, ja valitsemme saatua ratkaisua vastaavat objektit, saadaan approksimaalinen ratkaisu $D^*$ alkuperäiseen ongelmaan, ja epäyhtälöistä näemme, että se on korkeintaan $sn$ huonompi kuin alkuperäisen ongelman ratkaisu.

Koska haluttiin $\epsilon$-approksimaatio, vaaditaan, että ratkaisu täyttää ehdon

$$A^* - D^* \leq \epsilon A^*.$$

Siis riittää valita $s$ siten, että $sn \leq \epsilon A^*$. Halutaan tietenkin valita mahdollisimman suuri arvo skaalausvakiolle $s$, jotta aikakompleksisuus olisi mahdollisimman pieni. Arvoa $A^*$ ei kuitenkaan tunneta, joten optimaalista arvoa $\frac{\epsilon A^*}{n}$ ei voida valita. Ahneen algoritmin ratkaisusta $A_{hne}$ saadaan kuitenkin arvio $A_{hne} > A^*/2$, ja siten voidaan valita

$$s = \frac{\epsilon A_{hne}}{n}, \tag{5}$$

jolloin $sn = \epsilon A_{hne} \leq \epsilon A^*$, mutta toisaalta on $nE = nA^*/s = n^2 A^*/A_{hne}\epsilon <= 2n^2/\epsilon$ joten on löydetty $\epsilon$-approksimaatio $O(\frac{n^2}{\epsilon})$-tilassa ja -ajassa! Huomataan vielä, että ahne algoritmi voitiin suorittaa ajassa $O(n \log n) \subseteq O(n^2/\epsilon)$, eikä se siis vaikuta aikakompleksisuuteen. Algoritmi on siis TPAAS, koska se on polynomiaikainen ja -tilainen sekä syötteen koon $n$ että tarkkuusparametrin käänteisluvun $\frac{1}{\epsilon}$ suhteen.

Jos skaalausparametrille $s$ saadaan kaavassa (5) ykköstä pienempi arvo, skaalaa algoritmi nyt objektien arvot suuremmiksi, kuin ne alun perin olivat, ennen tarkan algoritmin ajoa. Tämä ei vaikuta kompleksisuuksiin, mutta oletamme, että tässä tapauksessa $s$ asetetaan ykköseksi. Tällöin saadaan optimaalinen ratkaisu edellä johdetussa ajassa ja tilassa. Siis lopullinen $s$:n valinta on

$$s = \max\{\frac{\epsilon A_{hne}}{n}, 1\}$$

## 4.4 $O(n)$-aikainen versio ahneesta algoritmista

Tämä luku perustuu lähteeseen [L]. Seuraavassa kappaleessa approksimaatioalgoritmista tehdään lineaarisen $n$:n suhteen. Tätä varten ahnetta algoritmia on muokattava, sillä objektien lajittelemista laskevaan arvojärjestykseen ei enää voi suorittaa, sillä lajittelu vie vähintään ajan $O(n \log n)$. Käytämme apuna algoritmia, joka etsii järjestämättömän listan mediaanin ajassa $O(n)$. Tämä on tunnetusti helppoa tehdä satunnaistetussa lineaariajassa muokkaamalla pikalajittelua, ja mahdollista suorittaa hieman suuremmalla näkymättömällä kertoimella myös absoluuttisessa lineaariajassa. Oletamme, että kaikkien objektien arvotiheys on eri. Jos kahdella objektilla on sama arvotiheys, vertaillaan indeksejä alkuperäisessä objektilistassa. Ahne algoritmi etsii siis lineaariajassa sellaisen joukon $J$, että $J$:ssä olevat objektit ovat arvotiheydeltään suurimmat $|J|$ objektia, ja yhdenkin objektin $o$ lisääminen siihen saisi $(J \cup \{o\})$:n yhteispainon ylittämään kapasiteetin $K$.

Ensin algoritmi etsii listan mediaaniarvotiheyden $r$. Merkitään $J' = \{j | a_j/p_j >= r\}$. Nyt on kolme vaihtoehtoa: Jos $J_p = \sum_{j \in J'} p_j = K$, on löydetty varmasti optimaalinen joukko, ja algoritmi voi palauttaa joukon $J'$. Jos $J_p > K$, voidaan kaikki painotiheydeltään $r$:ää pienemmät objektit unohtaa, ja kutsua algoritmia rekursiivisesti objekteilla $J'$ ja samalla kapasiteetilla. $J_p > K$ nimittäin tarkoittaa, että kaikki arvotiheydeltään $r$:ää suuremmat objektit eivät mahdu samaan ratkaisuun, joten koska valitsemme ratkaisun objektit laskevassa arvotiheysjärjestyksessä, ahne algoritmi ei tule koskaan valitsemaan mitään arvotiheydeltään $r$:ää pienempää objektia. Jos taas $J_p < K$, voidaan koko $J'$ ottaa mukaan ratkaisuun. Tällöin kutsutaan algoritmia rekursiivisesti joukolla $N - J$ ja kapasiteetilla $K - J_p$.

Algoritmi löytää selvästi saman joukon $J$ kuin aiempi versio ahneesta algoritmista, mutta nyt se toimii ajassa $O(n)$: Mediaanin laskeminen ja $J'$:n etsiminen voidaan suorittaa ajassa $O(n)$. Rekursion tasolla $k$ tehdään työtä $O(\frac{n}{2^k})$, koska $J'$:n koko on puolet syötteen koosta mediaanin määritelmän mukaan, ja jokaisella tasolla on tasan yksi kutsu. Siten koko algoritmi tekee työtä yhteensä $O(n)$.

## 4.5 Objektien jako suuriin ja pieniin

Tämä luku perustuu lähteeseen [L], jonka esitystapaa seurataan melko tarkasti, joskin käsittely on yksityiskohtaisempaa. Merkitsemme edelleen $s$:llä skaalauskerrointa, ja merkitsemme niin sanottua *kynnysarvoa* $k$:lla. Objekteja, jotka ovat arvoltaan vähemmän kuin $k$,

sanotaan pieniksi, ja muita objekteja suuriksi. Suurille objekteille käytetään aiemmin esitettyä skaalaavaa algoritmia, pienille taas ahnetta algoritmia. Merkitään $\phi(K')$:lla ahneen algoritmin tulosta kapasiteetilla $K'$ skaalaamattomilla pienillä objekteilla.

Tarkemmin, algoritmi suorittaa ensin ahneen algoritmin, joka tuottaa $50\%$-approksimaation $A_{hne}$, ja määrittää sen perusteella arvot $s$ ja $k$. Tämän jälkeen suuret objektit skaalataan $k$:lla, ja tarkka algoritmi suoritetaan näin saatavalle listalle, mikä tuottaa listan $L$ ratkaisuja. Tämän jälkeen etsitään

$$A_{ppro} = \max_{(p,b)\in L} \{sb + \phi(K - p)\},$$

eli etsitään paras tapa jakaa kapasiteetti suurten ja pienten objektien kesken.

Koska tavoitteena on saada voimaan $A^* - A_{ppro} \leq \epsilon A^*$, tarvitaan $A_{ppro}$:lle alaraja, ja $A^*$:lle yläraja. $A_{ppro}$:lle saadaan alaraja seuraavasti: Oletetaan, että $A_s$ ja $A_p$ ovat erästä optimaalista ratkaisua vastaavat suurten ja pienten objektien yhteisarvot, $P_s$ ja $P_p$ näiden painot, ja $B_s$ ja $B_p$ vastaavien skaalattujen objektien yhteisarvot. Tällöin listasta $L$ löytyy varmasti paria $(P_s, B_s)$ dominoiva pari $(P', B')$, koska muuten tarkka algoritmi ei olisi löytänyt koko Pareto-rintamaa, toisin sanoen

$$P' \leq P_s, B' \geq B_s.$$

Nyt määritelmänsä mukaan

$$A_{ppro} \geq sB' + \phi(K - P') \geq sB_s + \phi(K - P'), \tag{6}$$

mistä saadaan alaraja approksimaatioalgoritmin antamalle arvolle.

Koska jokainen suuri objekti on arvoltaan vähintään $k$, on niitä optimaalisessa ratkaisussa korkeintaan $A_s/k \leq A^*/k$ kappaletta. Siis (1) antaa $A^*$:lle ylärajan

$$A^* = A_s + A_p < s(B_s + A^*/k) + A_p,$$

mistä saadaan yhtälön (6) kanssa erotukselle $A^* - A_{ppro}$ yläraja

$$\begin{aligned} A^* - A_{ppro} &< s(B_s + A^*/k) + A_p - sB_s - \phi(K - P') \\ &= \frac{s}{k}A^* + A_p - \phi(K - P') \end{aligned} \tag{7}$$

Epäyhtälö $P' \leq P_s \leq K - P_p$ on voimassa, koska $P_s + P_p \leq K$. Näin ollen $K - P' \geq P_p$. Koska ahneen algoritmin virhe voi olla korkeintaan suurimman yksittäisen objektin arvo,

on se pienillä objekteilla korkeintaan kynnysarvon suuruinen, eli $\phi(K - P') \geq \phi(P_p) \geq A_p - k$, koska $\phi$ on selvästi ei-laskeva funktio. Siis erotus saadaan muotoon

$$A^* - A_{ppro} < \frac{s}{k}A^* + k$$

arvioimalla $A_p - \phi(K - P')$ ylöspäin. Nyt riittää siis saada voimaan

$$A^* - A_{ppro} < \frac{s}{k}A^* + k \leq \epsilon A^*.$$

Tätä varten asetetaan

$$\frac{s}{k} = \lambda\epsilon, k = (1 - \lambda)\epsilon A_{hne} \tag{8}$$

mielivaltaiselle $0 < \lambda < 1$, jolloin

$$\begin{aligned} A^* - A_{ppro} &< \frac{s}{k}A^* + k \\ &= \lambda\epsilon A^* + (1 - \lambda)\epsilon A_{hne} \\ &\leq (\lambda + 1 - \lambda)\epsilon A^* \\ &= \epsilon A^*. \end{aligned}$$

Ratkaisemalla yhtälöistä (8) $s$ saadaan

$$s = \lambda(1 - \lambda)\epsilon^2 A_{hne},$$

joten koska skaalauskerroin halutaan mahdollisimman suureksi, kannattaa valita $\lambda = \frac{1}{2}$, jolloin

$$s = \frac{1}{4}\epsilon^2 A_{hne}, k = \frac{1}{2}\epsilon A_{hne}.$$

Nyt

$$\left\lceil \frac{A^*}{s} \right\rceil \leq \frac{2A_{hne}}{\frac{1}{4}\epsilon^2 A_{hne}} = \frac{8}{\epsilon^2},$$

mikä on yläraja ‘arvoavaruuden koolle’. Tämä tarkoittaa, että Pareto-rintamassa $L$ on aina $O(\frac{1}{\epsilon^2})$ objektia suoritettaessa tarkkaa algoritmia, mistä seuraa, että $O(\frac{n}{\epsilon^2})$ on yläraja suurten objektien laskemisajalle, koska tarkka algoritmi käy Pareto-rintaman $L$ läpi $n$ kertaa.

Myös vanha raja $O(n^2/\epsilon)$ saadaan yhä voimaan, jolloin algoritmin aika- ja tilakompleksisuus tulee olemaan näistä kahdesta parempi. Tämä tehdään seuraavasti: Yhtälössä (7),

eli ensimmäisessä ylärajassa algoritmin virheelle, voidaan termin $A^*/k$ tilalle vaihtaa $n$, sillä myös $n$ on yläraja suurten objektien määrälle. Siis saadaan

$$A^* - A_{ppro} < sn + A_p - \phi(K - P') \leq sn + k.$$

Jos $k$ määritellään edelleen samoin, voidaan nyt asettaa

$$s = \frac{\epsilon A_{hne}}{2n},$$

jolloin saadaan

$$A^* - A_{ppro} < sn + k \leq \frac{\epsilon A_{hne}}{2} + \frac{\epsilon A_{hne}}{2} = \epsilon A^*,$$

mikä taas tekee algoritmista $\epsilon$-approksimaation, ja nyt suurten objektien arvoavaruuden koolle saadaan yläraja

$$\left\lceil \frac{A^*}{s} \right\rceil \leq \frac{2A_{hne}}{(\frac{\epsilon A_{hne}}{2n})} = \frac{4n}{\epsilon},$$

ja siis $O(n/\epsilon)$ on nyt yläraja suurten objektien laskennan arvoavaruuden koolle, jolloin itse algoritmi toimii ajassa ja tilassa $O(n^2/\epsilon)$. Koska siis kumpi tahansa näistä $s$:n valinnoista tuottaa $\epsilon$-approksimaation, voidaan valita niistä suurempi, jolloin molemmat $O$-rajat saadaan voimaan yhtä aikaa.

Jos kynnysarvolle $k$ on $k < 1$, voidaan ongelma ratkaista optimaalisesti ajassa $O(n/\epsilon)$, sillä

$$k = \frac{\epsilon}{2} A_{hne} < 1 \implies A_{hne} < \frac{2}{\epsilon} \implies A^* < \frac{4}{\epsilon}.$$

Siis kun yhdistetään kaikki tämä, saadaan valinnat

$$s = \max\{\frac{\epsilon^2 A_{hne}}{4}, \frac{\epsilon A_{hne}}{2n}, 1\}$$

$$k = \begin{cases} \frac{\epsilon A_{hne}}{2} & \text{jos } s > 1 \\ \epsilon A_{hne} & \text{jos } s = 1 \end{cases},$$

Lopuksi vaihda $s \leftarrow 1, k \leftarrow 0$, jos $k < 1$ äskeisessä.

Edellä $k$:n jälkimmäinen määritelmä $k = \epsilon A_{hne}$ seuraa siitä, että jos $s = 1$, niin suurten objektien ongelma ratkaistaan tarkasti. Tällöin suurista objekteista ei tule virhettä lainkaan, ja pienistä korkeintaan kynnysarvon $\epsilon A_{hne}$ verran, jolloin kokonaisvirhe on $\epsilon A_{hne} \leq \epsilon A^*$, ja taas saadaan $\epsilon$-approksimaatio.

## 4.6 $\phi$-arvojen laskeminen

Myös tämä luku perustuu enimmäkseen lähteeseen [L]. Lähteen algoritmissa on uskoakseni epätriviaali virhe, jonka vuoksi $\phi$-arvot lasketaan väärin. Myös aikakompleksisuus lasketaan nähdäkseni väärin. Lisäksi kehitettävä algoritmi ei käsittääkseni ole sama ahne algoritmi, jota käytettiin aiemmin, mistä ei ole mainintaa lähteessä [L]. Näistä korjauksista huolimatta.

Edellisen luvun algoritmi käyttää apunaan $\phi$-funktiota, jonka arvo syötteellä $k$ on ahneen algoritmin tulos pienille objekteille tällä kapasiteetilla $k$. Huomataan aluksi, että ahnetta algoritmia voidaan tässä tapauksessa hieman yksinkertaistaa, sillä algoritmilta vaaditaan edellisessä analyysissä vain, että sen virhe on korkeintaan kynnysarvon $k$ suuruinen. Tällöin riittää, että ahne algoritmi vain täyttää kapsäkin ahneesti laskevassa arvotiheysjärjestyksessä, eikä lopuksi tarvitse etsiä enää suurinta yksittäistä arvoa. Tämä on suoraviivaista nähdä ahneen algoritmin analyysistä. Edellisessä approksimaatioalgoritmissa tarvitsemme vain $\phi$-arvot arvot $\phi(K - P_i)$ jokaiselle $P_i \in L$ suurten objektien osasummien Pareto-rintamassa $L$. Arvoja on siis $O(\frac{1}{\epsilon^2})$ kappaletta, ja nämä voidaan tallentaa esimerkiksi listaan $L$. Tässä luvussa kehitetään tehokas algoritmin näiden etsimiseen.

Koska objektien lajittelua laskevaan arvotiheysjärjestykseen ei enää voi suorittaa, ei ole selvää, että $\phi$-arvot voidaan laskea lineaarisessa ajassa. Jos arvot olisivat järjestyksessä, voitaisiin tämä suorittaa käymällä yhtä aikaa lista $L$ läpi käänteisessä järjestyksessä (suurimmasta pienimpään), ja objektit $(p_i, a_i)$ läpi laskevassa arvotiheysjärjestyksessä, ja kunkin $(P_j, A_j) \in L$ kohdalla täyttämällä $\phi(K - P_j)$ mahdollisimman täyteen. Selvästi tähän riittää yksi läpikäynti, ja siten kompleksisuus on listojen koista suurempi, eli luokkaa $O(n + \frac{1}{\epsilon^2})$. Jos objektit eivät ole arvojärjestyksessä, voidaan arvot laskea seuraavalla tavalla ajassa $O(n \log \frac{1}{\epsilon})$:

Algoritmin syöte on suurten objektien ratkaisujen lista $L$, pienten objektien lista $L_p$, ja kokonaiskapasiteetti $K$. Asetetaan aluksi $\phi(K - P_j) = 0$ kaikille $(P_j, A_j) \in L$. Merkitään arvotiheydeltään vähintään $r$ olevia objekteja $J_r = \{(i | i \in L_p, \frac{a_i}{p_i} >= r)\}$. $J_r$ voidaan selvästi laskea ajassa ja tilassa $O(n)$. Lasketaan nyt objektien listan arvotiheyksien mediaani $r$ ajassa $O(n)$, ja tätä vastaava $J_r$, ja asetetaan

$$\phi(K - P_j) \leftarrow \sum_{i \in J_r} a_i$$

kaikille sellaisille $(P_j, A_j) \in L$, että

$$\sum_{i \in J_r} p_i \leq K - P_j.$$

Etsitään samalla suurin mahdollinen $P_j$, merkitään $P_{max}$. Siis sen sijaan, että painoa $K - P_j$ kohti etsittäisiin mahdollisimman suuri joukko $J_r$, etsitäänkin sopiva objektien joukon puolittava $J_r$, ja laitetaan sen arvo mahdollisimman pieneen kapasiteettiin (ja kaikkiin sitä suurempiin). Tällöin ei välttämättä saada yhtään oikeaa $\phi$-arvoa, eikä voida sanoa mitään hyödyllistä virheen suuruudesta, vaan saadaan vain eräs alaraja $\phi$:lle.

Kun tämä yksi $\phi$-arvo on asetettu, kutsutaan funktiota kahdesti rekursiivisesti. Ensimmäinen kutsu tehdään suurten objektien listalla parista $(P_{max+1}, A_{max+1})$ listan $L$ loppuun asti, pienten objektien joukolla $J_r$, ja kapasiteetilla $K$. Koska koko $J_r$ mahtui kapasiteettiin $\phi(K - P_{max})$, ei $J_r$:n osajoukoilla voi saada sille parempaa arvoa. Samoin $J_r$:n aidot osajoukot eivät ole hyödyllisiä, jos täytetään suurempaa kapasiteettia kuin $K - P_j$ (ei jos painot ovat pienempiä). Tämä rekursiokutsu etsii siis arvot $\phi(K - P_k)$ kaikille $k > j$.

Toinen rekursiokutsu tehdään suurten objektien ratkaisujen listan $L$ alusta ratkaisuun $(P_{max}, A_{max})$ asti ja $J_r$:n komplementtiin $L_p - J_r$. Nyt $J_r$ otetaan mukaan joka ratkaisuun, joten kapasiteettiparametrista $K$ on poistettava sen paino. Jonkinlaisella lisäparametrilla on pidettävä huolta, että myös $\phi$-arvojen asetuksessa lisätään $J_r$:n arvo rekursiivisissa kutsuissa. Tämä voidaan tehdä neljännellä parametrilla, joka lisätään jokaiseen asetettavaan $\phi$-arvoon. On vielä huomattava, että rekursiokutsun suurten objektien lista todella sisältää ratkaisun $(P_{max}, A_{max})$, sillä saattaa olla, että lisäämällä jokin osajoukko $J_r$:n komplementista löydetään aiemmin saatua parempi arvo $\phi(K - P_{max})$:lle.

Vaikka algoritmi ei välttämättä löydä oikeaa $\phi$-arvoa heti, nähdään että jokainen $\phi$-arvo tulee löydetyksi. Tämä johtuu siitä, että kaikki erisuuret joukot $J_r$:t käydään läpi algoritmin suorituksen aikana, jolloin jokaista $K - P_j$ kohti myös sitä vastaava optimaalinen $J_r$ asetetaan siihen, sillä algoritmin tullessa $J_r$:n kohdalle asetetaan sen arvo jokaiseen kapasiteettiin, johon se mahtuu, myös kapasiteettiin $K - P_j$. Lisäksi oikeaa ratkaisua ei koskaan korvata väärällä, sillä $\phi$-arvon asetuksen jälkeen kaikki mahdolliset sen asetukset sisältävät aina myös joukon $J_r$ arvon.

Analysoidaan nyt algoritmin kompleksisuutta. Merkitään $s$:llä suurten objektien ratkaisujen määrää, ja $p$:llä pienten objektien määrää. Algoritmi käyttää mediaanin etsimiseen lineaarisen ajan $p$:n suhteen, ja sitten sopivan suurten objektien ratkaisun löytämiseen, ja $\phi$-arvojen täyttämiseen lineaarisen ajan $s$:n suhteen. Tämän jälkeen tehdään rekursiivinen

kutsu, jossa pieniä objekteja on $\frac{p}{2}$, ja suuret jaetaan kahtia näiden kutsujen välille.

Siis parametreilla $s$ ja $p$ ajankäytölle $T(s,p)$ on voimassa rekurrenssi

$$T(s,p) \leq cs + cp + \max_{1 \leq q \leq s} \{T(s-q, \frac{p}{2}) + T(q, \frac{p}{2})\},$$

missä $c$ on jokin riittävän suureksi valittu vakio. Mistä voidaan induktio-oletuksella $T(s,p) <= cp \log s + cs \log p$ ratkaista

$$\begin{aligned}
T(s,p) &\leq cs + cp + \max_{1 \leq q \leq s} \{T(s-q, \frac{p}{2}) + T(q, \frac{p}{2})\} \\
&= cs + cp + \max_{1 \leq q \leq s} \{\frac{cp}{2}(\log(s-q) + \log q)\} \\
&\qquad + \max_{1 \leq q \leq s} \{c(s-q+q) \log \frac{p}{2}\} \\
&\leq cs + cp + \frac{cp}{2}(2 \log \frac{s}{2}) + cs \log \frac{p}{2} \\
&= cs + cp + cp \log \frac{s}{2} + cs \log \frac{p}{2} \\
&\leq cp \log s + cs \log p.
\end{aligned}$$

Koska $s$ on luokkaa $O(\frac{1}{\epsilon^2})$ ja $p$ on luokkaa $O(n)$, on algoritmi aika- ja tilakompleksisuudeltaan luokkaa $O(n \log(\frac{1}{\epsilon^2}) + \frac{1}{\epsilon^2} \log n)$, minkä edellisen algoritmin kompleksisuus $O(\frac{n}{\epsilon^2})$ sisältää osajoukkonaan. On kuitenkin huomattava, että vaihtoehtoinen raja $O(\frac{n^2}{\epsilon})$ kynnysarvoa käyttävän approksimaatioalgoritmin kompleksisuudelle muuttuu nyt muotoon $O(\frac{n^2}{\epsilon} + \frac{1}{\epsilon^2} \log n)$. Joka tapauksessa algoritmi on $n$:n suhteen lineaarinen.

Lawler [L] esittää vastaavan algoritmin ilman mainintaa vaihtoehtoiseen kompleksisuuteen ilmaantuvasta lisätermistä $\frac{1}{\epsilon^2} \log n$. Koska algoritmin kompleksisuus suurten objektien ratkaisujen määrän ja pienten objektien lukumäärän suhteen on symmetrinen (sillä pahin tapaus puolittaa molemmat suureet rekursiossa), on selvää, että $\frac{1}{\epsilon^2} \log n$-termi on kompleksisuudessa, jos $n \log \frac{1}{\epsilon}$ on. Kompleksisuuden johtamisen sen osan, jossa $\frac{1}{\epsilon^2} \log n$ ilmestyy, Lawler [L] ohittaa lyhyellä maininnalla, ettei se vaikuta kompleksisuuteen.

Tämän lisäksi lähteen [L] algoritmi asettaa kunkin $\phi$-arvon vain kerran. Kyseinen algoritmi ei näin ollen välttämättä löydä tarkalleen oikeaa $\phi$-arvoa, vaan ensimmäisen siihen sopivan arvon, joka on siis käytännössä satunnainen luku. Tämä on ongelmallista, sillä koko kynnysarvoa käyttävä approksimaatio perustuu siihen, että virhe on korkeintaan kynnysarvon suuruinen. Lisäksi artikkelissa ei mainita, ettei uusi ahne algoritmi ole sama kuin aiemmin esitetty, vaan suurimman yksittäisen arvon etsiminen voidaan pudottaa pois. Alkuperäinen ahne algoritmi saadaan käymällä lopuksi läpi pienten objektien lista ja

asettemalla se jokaiseen soveltuvaan $\phi$-arvoon. On mahdollista, että tätä lisäystä pidettiin lähteessä [L] triviaalina.

## 4.7 Vaihtoehtoiseen tarkkaan algoritmiin perustuva tilatehokas approksimaatio

Muistetaan, että [MO]:ssä esitetty vaihtoehtoinen tarkan algoritmin implementaatio oli vaihtokauppa ajan ja tilan välillä. Peräytymishaulla kehitettiin tarkalle algoritmille implementaatio, jossa suurimman arvon saavuttava objektijoukko saatiin ajassa $O(nA^* \log n)$ ja tilassa $O(n + A^*)$. Lähteessä [MO] kehitetään tämän pohjalta skaalausalgoritmi, joka eroaa lähteen [L] skaalausalgoritmista vain siinä, että tarkan ratkaisun etsivänä rutiinina käytetään lähteen [MO] tarkkaa algoritmia, eikä lähteen [L] tarkkaa algoritmia. Tässä luvussa mainitaan tämä idea, ja selvitetään miten lähteen [MO] algoritmin kompleksisuudet seuraavat lähteen [L] kompleksisuuksista.

Tarkastelemalla skaalaavan approksimaatioalgoritmin kompleksisuuksia huomataan, että analyysin lopussa tarkan algoritmin aika- ja tilakompleksisuus $O(nA^*)$ yksinkertaisesti sijoitetaan paikalleen. Näin ollen saadaan aiemmin esitetyille algoritmeille samoilla arvioilla vaihtoehtoisia kompleksisuuksia, kun yksinkertaisesti käytetään tarkalle algoritmille vaihtoehtoista implementaatiota. Tällöin aikakompleksisuudeksi saadaan $O(n^2 \log n \frac{1}{\epsilon})$ ja tilakompleksisuudeksi $O(\frac{n}{\epsilon})$ siinä, missä lähteen [L] vastaavat kompleksisuudet ovat molemmat $O(\frac{n^2}{\epsilon})$. Käyttämällä vaihtoehtoista implementaatiota tarkalle algoritmille saadaan myös muille approksimaatioalgoritmeille vaihtoehtoisia kompleksisuuksia, joita ei listata tässä.

# 5 Päätelmiä

Tämän tutkielman kohteena olivat kapsäkkiongelman täysin polynomiset approksimaatioalgoritmit. Tutkielmassa esiteltiin ensimmäisten näistä kirjoitettujen julkaisujen tuloksia yksityiskohtineen. Päätavoite oli selkeyttää tutkimusartikkelien lyhyttä esitystä esittämällä tarkemmat perustelut väitteille, ja selittämällä paikoittain tarkemmin miksi tietyt valinnat tehdään ja mikä niiden tarkoitus on.

Tutkielmassa esitettiin pieni mutta epätriviaali korjaus erääseen lähteen [L] algorit-

miin. Lisäksi tietyt perustelut muuttuivat selkeämmiksi tutkielmassa esitettävän Pareto-rintaman käsitteen avulla. Lähteissä ei esitetä tarkkaa selitystä ratkaisujen listan $L$ luonteelle, vaan esimerkiksi lähteessä [L] vedotaan sen sijaan intuitioon, kun todistuksessa tarvitaan sitä seikkaa, että $L$ on Pareto-rintama. Lisäksi lähteen [L] tarkan algoritmin yhteydessä käyttämän tietorakenteen havaittiin syntyvän automaattisesti, kun objektilistojen tallennukseen käytetään LISP-tyylisiä listoja.

Kapsäkkiongelma on esimerkki hyvin approksimoituvasta ongelmasta, ja vaikka kaikki approksimointitekniikat perustuvatkin samoihin perusideoihin, on erilaisia saavutettuja kompleksisuuksia eri lähteissä monia. Täysin polynomisten algoritmien yhteydessä on välillä vaikea sanoa kumpi kahdesta algoritmista on tehokkaampi edes kompleksisuuden suhteen, sillä algoritmien kompleksisuudet mitataan sekä syötteen koon, että tarkkuusparametrin vaikutuksen suhteen. Tämä näkyy hyvin kapsäkkiongelmassa, kolmen Taulukko 1:ssä esitetyn artikkelin kompleksisuudet eivät yksikään ole toista aidosti parempia. Lisäksi kirjallisuudessa esiintyy monimutkaisempia rajoja, joita on vielä vaikeampi vertailla toisiinsa.

Nykyään NP-vaikeiden ongelmien ratkaisemiseen on olemassa yleisempiä tekniikoita, kuten kokonaislukuohjelmointi, johon redusoimalla saadaan monille NP-vaikeille ongelmille approksimaatioalgoritmeja suoraviivaisesti, ilman suurempaa erillistä analyysiä. Tämä luonnollisesti tarkoittaa, että uudet tiettyyn ongelmaan räätälöidyt approksimaatioalgoritmit joutuvat kilpailemaan tällaisten ratkaisujen kanssa. Lisäksi yhä enemmän käytetään yleisiä ratkaisuja, jotka eivät välttämättä takaa mitään tarkkaa polynomia kompleksisuudelle, vaan algoritmeja testataan käytännön tilanteissa esiintyvillä syötteillä.

1970-luvun tutkimusraporteissa ei puhuta tällaisesta käytännön tutkimuksesta mitään, ja usein artikkeleissa olevista huolimattomuusvirheistä saa sellaisen kuvan, ettei algoritmia koskaan ole implementoitukaan. Myös tämän tutkielman lähestymistapa on puhtaasti teoreettinen, sillä kapsäkkiongelman yksinkertaisuuden vuoksi heuristiikat tuottavat sille matemaattisesti mielenkiintoisia kompleksisuusrajoja, jotka on siten järkevää myös käsitellä tarkasti. On jopa yllättävää miten suoraviivaista skaalaukseen perustuva approksimaatioalgoritmi on johtaa algebrallisesti.

Kattavammassa tutkielmassa olisi ollut mielenkiintoista ottaa laajempi näkökulma, ja tarkastella sitä, miten samankaltaisia muiden NP-täydellisten ongelmien approksimaatioideat (erityisesti TPAAS:t) ovat, ja onko niiden johtaminen yhtä suoraviivaista heuristiikkojen kehittämistä sekä laskemista. Lähes kaikki kapsäkkiperheen ongelmat ratkeavat hy-

vin samankaltaisilla algoritmeilla, ja esimerkiksi alijoukon summa -ongelmaan saadaan vastaavilla ideoilla usein yhden $n$- tai $\frac{1}{\epsilon}$-kertoimen verran tehokkaampia algoritmeja. Tällaiset 'dimensiot' ovat ajatuksena kiehtovia, ja olisi esimerkiksi mielenkiintoista tietää, onko kapsäkkiongelmaan luonnollisia monimutkaistuksia, jotka kasvattavat mahdollisten TPAAS:ien kompleksisuuksia tietynlaisilla kertoimilla.

Vaihtokaupat ajan ja tilan välillä ovat myös mielenkiintoinen aihe, josta ei vielä tiedetä kovin paljoa. [L]:n ja [MO]:n erilaiset implementaatiot tarkalle algoritmille ovat tällainen vaihtokauppa, [MO]:ssa aikakompleksisuutta kasvatetaan $\log n$:llä, ja tilakompleksisuus putoaa $n$:n verran. Vastaavankaltainen vaihtokauppa voidaan usein tehdä samankaltaisella peräytymishaulla nimenomaan dynaamisen ohjelmoinnin yhteydessä, sillä usein ratkaisun arvon löytäminen on selvästi tilatehokkaampaa kuin itse ratkaisun löytäminen.

# Lähteet


[L] E. Lawler, 'Fast Approximation Algorithms for Knapsack Problems', Mathematics of Operations Research 4, 339–356, 1979

[MO] M.J. Magazine, O. Oguz, 'A Fully Polynomial Approximation Algorithm for the 0–1 Knapsack Problem', European Journal of Operational Research, 8, 270–273, 1981

[HP] H. Kellerer, U. Pferschy, 'A New Fully Polynomial Approximation Scheme for the Knapsack Problem', Technical Report, Faculty of Economics, University Ghaz, 1997

[WP] Wikipedia, Strongly NP-complete, http://en.wikipedia.org/wiki/Strongly_NP-complete, Viitattu 4.7.2010

[IK] O.H. Ibarra, C.E. Kim, 'Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems', Journal of the ACM, (22), (1975): 462–468

[BD] D- Bertsimas, R. Demir, 'An Approximate Dynamic Programming Approach to Multidimensional Knapsack Problems', Management Science, Vol. 48, No. 4, 550–565, 2002

[BV] M.S. Bansal, V.C. Venkaiah, 'Improved Fully Polynomial time ApproximationScheme for the 0–1 Multiple-choice Knapsack Problem', International Institute of Information Technology, Hyderabad, India, 2004