Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Plánování nakládky jako součást aplikace DCIx

Plzeň 2013 Jakub Kotásek

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 26. června 2013

Jakub Kotásek

Abstract

Truck load planning for DCIx

The aim of this master thesis is to implement algorithm for solving bin packing problems to already existing warehouse management system (DCIx). I present mapping of existing algorithms, which are compared according to results from expert literature. The most suitable algorithm, which will be implemented, is closely described. Inputs and outputs for the algorithm are identified in the system DCIx. The results at the end of this work also show description of implementation.

Obsah

1 Úvod

Tato diplomová práce byla zadána firmou Aimtec, která chce do svého skladového systému DCIx začlenit plánování nakládky.

Plánování nakládky je podskupinou problému Bin Packing Problem (BPP). Řešení BPP má různorodé použití, slouží například k optimalizovanému rozmístění reklam v novinách; rozřezání skla na tabule, pro minimalizaci odpadu; optimalizaci nakládání zboží do kontejnerů, pro minimalizaci nevyužitého místa - tedy plánování nakládky, a další. BPP náleží do skupiny NP-těžkých problémů a pro jeho řešení existuje velké množství algoritmů.

Práce obsahuje shrnutí algoritmů řešících BPP, jejich porovnání z hlediska výkonosti a kvality řešení. Dále obsahuje výběr algoritmu, který odpovídá požadavkům na implementaci nakládky do systému DCIx.

Další část práce se poté zabývá analýzou stávajících struktur v aplikaci DCIx, jejich využitím pro algoritmus plánování nakládky , a nakonec popisem implementace vybraného algoritmu.

2 Bin Packing Problem (BPP)

2.1 Obecné popis problému

Obecně je vstupem algoritmů řešících BPP množina předmětů s udanými rozměry a množina homogenních kontejnerů s udanými rozměry. Cílem algoritmu je naložení co nejvíce předmětů do co nejmenšího počtu kontejnerů. Přesné vstupy a cíl algoritmu se liší od použití v reálném světě. Např.: Množina předmětů může být považována za nekonečnou a množina kontejnerů za konečnou - tedy máme omezený počet nákladních aut (kontejnerů), které budou předměty převážet, a počet skladových zásob (předmětů) je tak vysoký, že ho můžeme považovat za neomezený. Nebo naopak množina předmětů může být považována za konečnou a množina kontejnerů za nekonečnou - máme přesně vymezené, které předměty chceme převézt, ale nezáleží nám na počtu nákladních automobilů (kontejnerů), tento počet pouze chceme minimalizovat.

Jedná se o vícerozměrný problém (může být použit jako jedno-, dvou- nebo třírozměrný). Pro přehlednost budeme dále uvažovat o problému jako dvourozměrném, analogicky jde však problém popsat i jako třírozměrný.

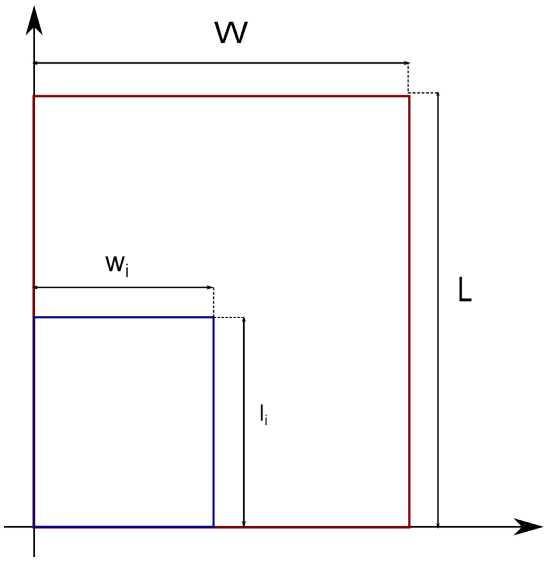
2.1.1 Vstupy

Předměty

Definujme tedy množinu předmětů jako $i \in I$, které mají rozměry a to šířka předmětu (w_i) a výška předmětu (h_i) viz obrázek 2.1. Později u 3D BPP ještě uvedeme třetí rozměr, hloubku předmětu (d_i).

Kontejnery

Množinu kontejnerů definujeme jako homogenní, všechny kontejnery mají tedy stejné rozměry: šířka kontejnerů (W), výška kontejnerů (H) (u 3D BPP opět ještě hloubka kontejnerů (D)) a dále máme definovaný parametr maximální počet kontejnerů (b_{max}) viz obrázek 2.1.



Obrázek 2.1: Předmět umístěný v kontejneru

Omezení

Předpokládáme omezení předmětů (viz vzorec 2.1) - předměty tedy nemohou přesáhnout velikost kontejneru.Dalším omezením je, že předměty nelze dělit a musí být do kontejneru umístěny jako celek, dále se předměty umístěné do kontejneru nesmějí překrývat.

$$w_i \leq W \wedge h_i \leq H \tag{2.1}$$

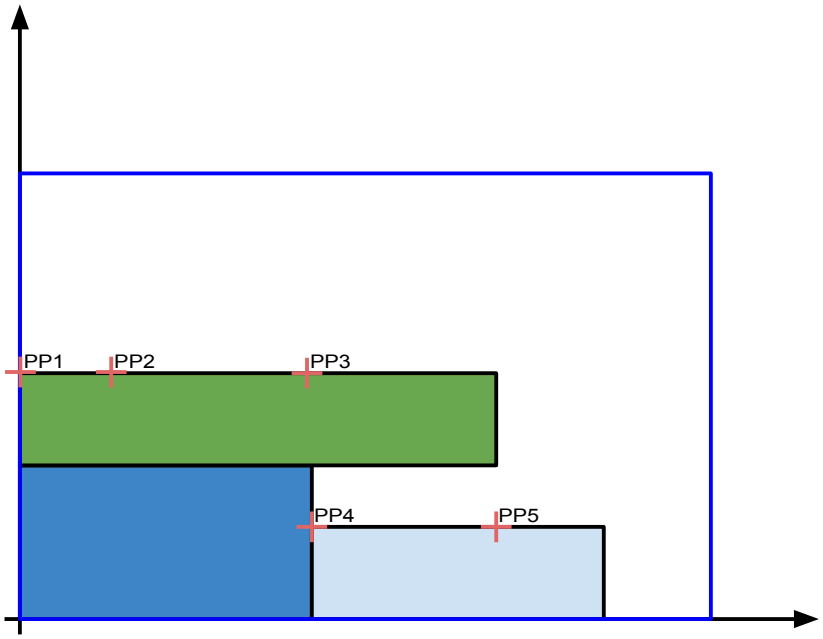
2.2 Existující algoritmy

BPP a jeho podmnožiny se řeší již několik desítek let. Jedna z prvních zmínek se datuje už k roku 1897 (jak uvádí [Peng and Zhang, 2012]). V posledních dekádách se objevilo velké množství algoritmů, které se na tento problém zaměřují. Následující kapitoly obsahují stručný popis jednotlivých algoritmů a jejich srovnání a to jak z hlediska výkonu, tak z hlediska modifikovatelnosti pro reálné použití.

2.3 Umístění předmětu do kontejneru

Jedním z hlavních problémů, který musí algoritmus řešit, je definování míst, kam může být předmět umístěn. Zatímco u 1D problému je umístění předmětu jasné (viz obrázek 2.2 (a)), neboť předmět se vloží ihned na první volné místo, u problému 2D není možné umístění jednoznačně určit (viz obrázek 2.2 (b)). Musíme zvolit heuristiku, která najde body, kam je možné předmět umístit a poté posoudí, které z míst je pro umístění nejvhodnější. U 3D problému se složitost problému zvyšuje, avšak je možné použít stejné heuristiky jako u 2D problému.

(a) značka se šipkou naznačuje jediné vhodné místo pro umístění předmětu



(b) body PPi znázorňují náhodně vybrané body pro umístění předmětu

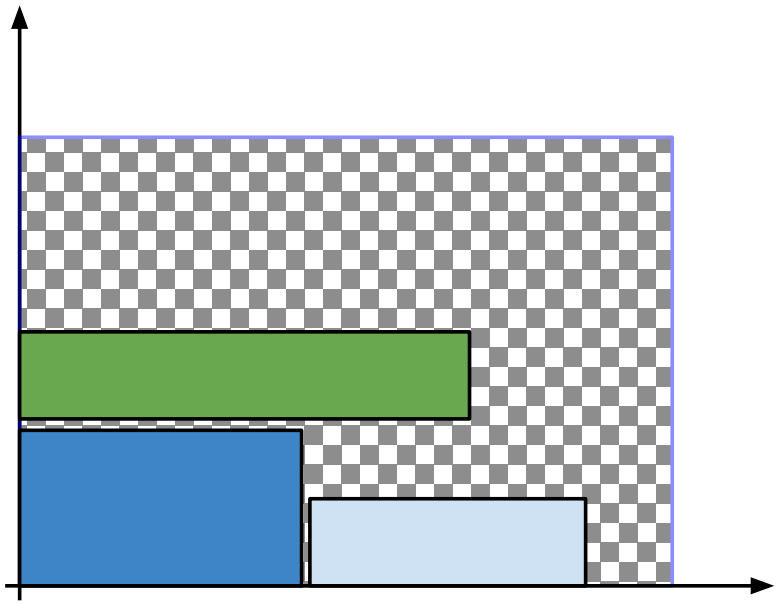
Obrázek 2.2: Umístění předmětu do kontejneru

2.3.1 Vzory naskladnění

Jeden z prvních pokusů heuristiky umístění je vytvoření množiny všech možných vzorů naskladnění předmětů do kontejneru v závislosti na omezení pro umístění předmětů (viz [Gilmore and Gomory, 1965]). Vzory se poté kombinují, tak aby odpovídaly předmětům, které mají být umístěny do kontejnerů, ve snaze minimalizovat počet použitých kontejnerů. Tento přístup lze použít jen pro malou podmnožinu BPP.

2.3.2 Obdélníkové oblasti

Dalším přístupem je rozdělit kontejner do více obdélníkových oblastí o rozměrech $p \times q$. Předměty jsou poté vždy svým levým spodním okrajem umístěny do levého spodního okraje obdélníkové oblasti (viz [Beasley, 1985]). Příklad umístění je možné vidět na obrázku 2.3. Při takovémto postupu záleží kvalita řešení na zvolení rozměrů p, q daných oblastí, avšak s větší granularitou narůstá složitost řešení.



Obrázek 2.3: Na obrázku je znázorněno rozdělení kontejneru na obdélníky, dále můžeme vidět vznikající mezery mezi předměty

2.3.3 Binární proměnné

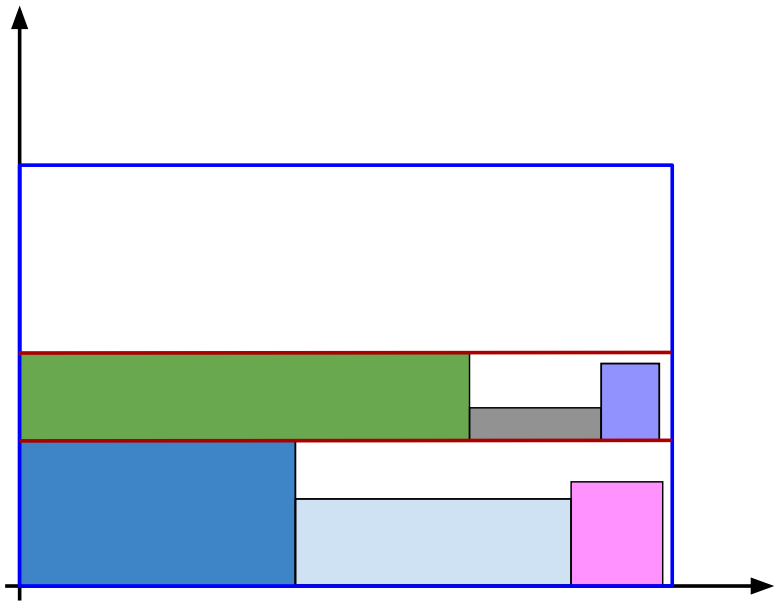
[Egeblad and Pisinger, 2009] vymysleli model, který reprezentuje přesah předmětů v kontejneru a to za pomoci binárních proměnných. Tento model je schopný se vypořádat s rotací předmětů, určením pevné pozice pro určitý předmět atd. Podobný přístup je možný vidět u [Baldi et al., 2011]; tento algoritmus se zabývá problémy, kde je nutné uvažovat o rozložení váhy nákladu v kontejneru z důvodu zachování rovnováhy. Omezení bylo například specifikováno tak, že předmět byl definován těžištěm a jeho rozměry. Předměty musí být umístěny do kontejneru tak, aby celkové těžiště nákladu leželo v předem definované zóně.

Všechny heuristiky uvedené v této a předchozích kapitolách se však musí potýkat s velkým množstvím proměnných vůči kvalitě řešení. Mohou být použity pouze pro skupiny případů s malým počtem předmětů (cca 20 druhů předmětů pro 3D problém).

Heuristika pro umístění předmětů do kontejneru musí být brána s ohledem k výpočetnímu času a složitosti datových struktur pro držení informací o již umístěných předmětech v daných kontejnerech. Rovněž se musí brát v potaz možnosti přidání dalšího omezení pro algoritmus, a to například pevné umístění některých předmětů v kontejnerech, nebo zohlednění řezů (při optimalizaci rozdělení skla na jednotlivé tabule apod.).

2.3.4 Zjednodušení problému

Pro 2D problém se často využívá rozdělení heuristiky na dvě části, tím dojde k převodu na 1D problém (viz [Chung et al., 1982], [Berkey and Wang, 1987] a [Bortfeldt and T.Winter, 2009]). Nejprve se umístí předměty do jednotlivých vrstev (polic), které mají jasně danou výšku h_s odpovídající nejvyššímu předmětu ve vrstvě a šířku W odpovídající šířce kontejneru. Jednotlivé vrstvy se pak naskládají na sebe do kontejneru (viz obrázek 2.4).

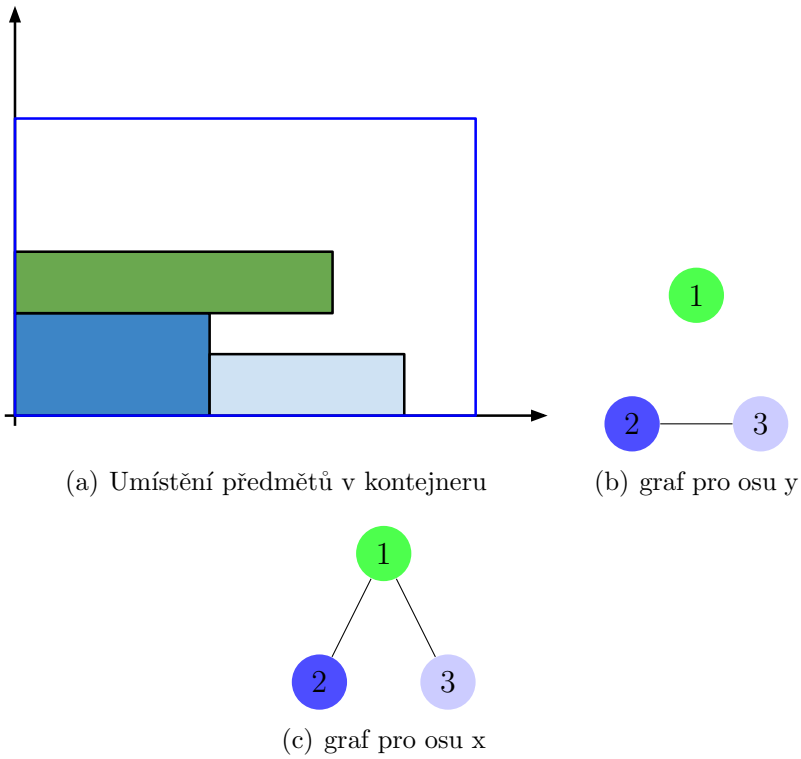


Obrázek 2.4: Rozdělení kontejneru do vrstev (polic)

Obdobný postup lze použít pro 3D problém, kdy se nejprve předměty vyskládají do tzv. zdí s rozměry W a H, podle kontejneru (tedy redukce na 2D problém). A poté se zdi vyskládají do kontejneru, kde hloubka zdi hz_i odpovídá nejhlubšímu balení v dané zdi, tedy redukce na 1D problém (viz [George and Robinson, 1980] a [Pisinger, 2002]).

2.3.5 Intervalové grafy

Za pomocí teorie grafů definovali umístění předmětů do kontejneru [Fekete and Schepers, 1997] a [Fekete and Schepers, 2004]. Umístění předmětů je znázorněno pomocí intervalových grafů viz obrázek 2.5 (pro každou ortogonální osu jeden intervalový graf). Tímto přístupem mohli pracovat s celou množinou již umístěných balení najednou. Každý předmět je v grafu znázorněn jako vrchol, hrana mezi vrcholy existuje pouze tehdy, pokud se v dané ose předměty překrývají. Ukázky grafů můžeme vidět na obrázcích 2.5(c) (intervalový graf osy x) a 2.5(b) (pro osu y).



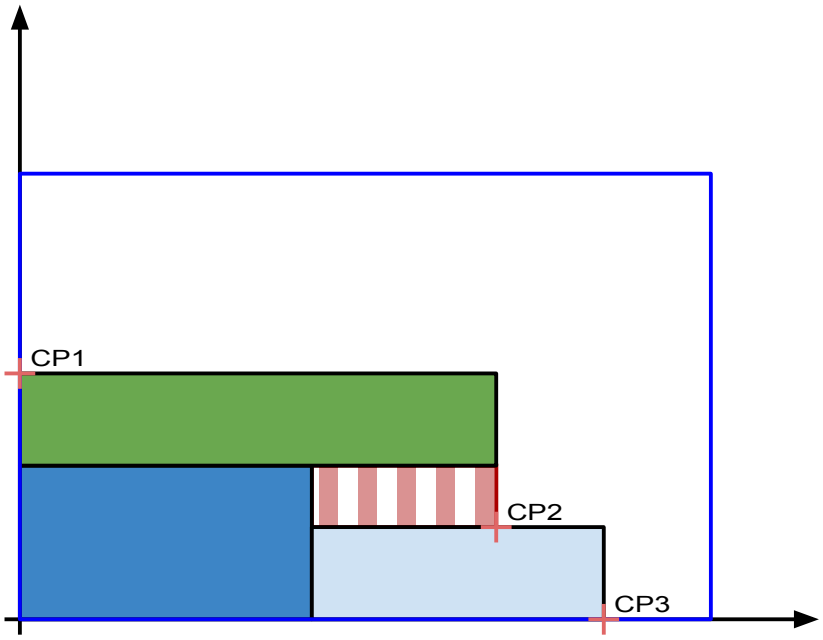
(a) Umístění předmětů v kontejneru (b) graf pro osu y

(c) graf pro osu x

Obrázek 2.5: Znázornění umístění předmětů pomocí intervalových grafů

2.3.6 Rohové body

Dalším přístupem je definování tzv. rohových bodů viz [Martello et al., 2000], kde může být předmět umístěn. Rohový bod je takový bod, kde může být předmět umístěn aniž by mohlo dojít k překrytí předmětu s jiným, již umístěným předmětem (viz obrázek 2.6). Výpočet těchto bodů má dokázanou složitost $O(n^2)$. [Martello et al., 2000] použili tento algoritmus jako základ rozhodnutí, zda daná množina předmětů může být naložena do kontejneru. V [den Boef et al., 2005] však bylo ukázáno, že pokud neexistuje žádný rohový bod, kam by se vešel daný předmět, tak přesto existuje místo v tomto kontejneru, kam tento předmět umístit lze, nevyužitelná oblast je na obrázku 2.6 vyšrafována.



Obrázek 2.6: Umístění rohových bodů

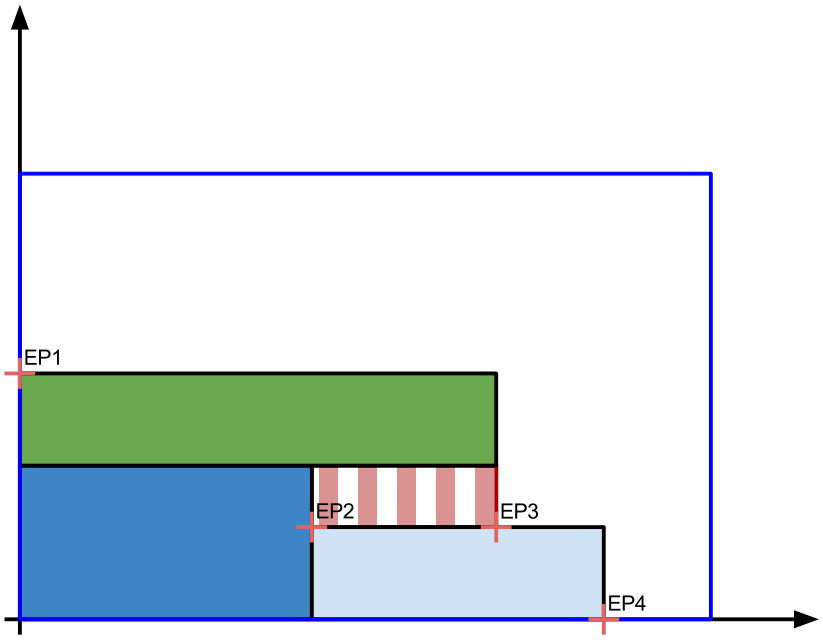
2.3.7 Extrémní body

Rozšířením přístupu z kapitoly 2.3.6 jsou tzv. extrémní body (EB), které byly představeny v [Crainic et al., 2008]. K rohovým bodům jsou přidány další takové body, kam je možné předmět umístit a sjednocením těchto množin dostáváme množinu extrémních bodů (viz obrázek 2.7). Použití EB využívá i místa, která při použití rohových bodů nelze použít. To můžeme vidět na obrázcích 2.6 a 2.7 jako červeně-bílo pruhovanou oblast.

2.4 Algoritmy řešící 2D a 3D BPP

V této kapitole budou popsány algoritmy, které obecně řeší problém, kde máme konečné množství předmětů, jež se snažíme uložit do co nejmenšího počtu kontejnerů stejných rozměrů.

Výsledky všech algoritmů byly shromážděny z příslušné literatury. Vzhledem k rozdílným procesorům, které byly použity jednotlivými autory pro



Obrázek 2.7: Umístění extrémních bodů

testování algoritmů, byly časy přepočítány podle SPEC CPU2006 (viz [cpu, 2006]) a přepočítané časy odpovídají procesoru Pentium 4 3000 MHz.

2.4.1 TSPACK

[Lodi et al., 1999] představil algoritmus TSPACK. Jedná se o Tabu Search (TS) algoritmus řešící 2D-BPP. Algoritmus využívá dvě heuristiky pro umístění předmětů do kontejnerů. TS pouze kontroluje pohyb mezi jednotlivými kontejnery. Mezi dvěma sousedy se algoritmus pokouší přesunout všechny předměty z nejslabšího kontejneru do silnějšího. Nejslabší kontejner je takový kontejner, ze kterého bude nejjednodušší přesunout všechny předměty do jiného, záleží na zvolení parametru, např.: nejmenší počet předmětů v kontejneru, nejmenší zabraná plocha předměty apod. Druhá heuristika slouží k umístění předmětů v daném kontejneru, vlastnosti algoritmu závisí na použití vnitřní heuristiky. Tento algoritmus je považován za jednu z nejlepších meta-heuristik, které byly pro 2D-BPP sestrojeny, avšak na jednotlivé výpočty potřebuje průměrně 60 CPU sekund.

2.4.2 Height first - Area second (HA)

Stejní autoři jako v kapitole 2.4.1 prezentovali algoritmus HA řešící 2D-BPP. HA je založen na umisťování předmětů do vrstev, viz kapitola 2.3 (obrázek 2.4), algoritmus vybírá lepší ze dvou řešení. První řešení vzniká seskupením předmětů podle výšky. Jednotlivé skupiny poté umístí do vrstev a ty vloží do kontejnerů podle algoritmu BaB, který slouží k řešení 1D-BPP (viz [Martello and Toth, 1990]). Druhé řešení vzniká seřazením balení do skupin podle plochy, kterou zaberou. Tyto skupiny opět umístí do vrstev, vrstvy umístí podle algoritmu BaB do kontejnerů. Algoritmus je rychlejší, avšak méně přesný než TSPACK.

2.4.3 Branch-and-Bound (BaB)

Dvou úrovňový BaB algoritmus, který je přímo určen pro řešení 3D-BPP, byl představen v [Martello et al., 2000]. První úroveň algoritmu přiřadí předměty do kontejnerů. V druhé úrovni se ověří zda předměty přiřazené k určitému kontejneru je opravdu možné do tohoto kontejneru naložit. V té samé práci uvádí autoři další dvě heuristiky pro řešení problémů, a to S-PACK, která je odvozená od umisťování předmětů do vrstev, a MPV-BS, ta plní postupně každý kontejner algoritmem BaB, který představili stejní autoři. V práci jsou uvedeny výsledky jejich algoritmů s výpočetním časem omezeným na 1000 CPU sekund.

2.4.4 Guided Local Search (GLS)

Dalším algoritmem řešícím 3D-BPP je GLS, který byl popsán v [Faroe et al., 2003]. Algoritmus si na začátku získá spodní a horní hranici počtu kontejnerů pomocí hladové heuristiky. Začne na horní hranici počtu a po každé iteraci sníží počet kontejnerů a poté se pomocí GLS snaží najít umístění předmětů v kontejnerech. Algoritmus končí, pokud mu dojde čas, nebo pokud se dostane na spodní hranici počtu kontejnerů. Výsledky algoritmu jsou kvalitní, avšak k jejich dosažení algoritmus počítal 1000 CPU sekund.

2.4.5 Extrémní body a Best Fit Decreasing (BFD)

V [Crainic et al., 2008] byly představeny extrémní body (EB) v kombinaci s algoritmem BFD. Rozšíření BFD pro použití v 3D-BPP je složité například z důvodu, že ve vyšších dimenzích můžeme řadit podle více atributů (objem, obsah pokládající strany, šířka, délka a výška předmětu). Po provedení testů zahrnujících nejrůznější varianty byly ty nejlepší z nich použity pro algoritmus Crainic's Extreme Points Best Fit Decreasing (C-EPBFD). K výpočtu metoda potřebuje zanedbatelný čas (řády jednotek sekund) a výsledkem předčila jiná komplexnější řešení jako například BaB ([Martello et al., 2000]).

2.4.6 TS^{2}PACK

Dvouúrovňová úprava metody TSPACK byla navržena v [Crainic et al., 2009] (TS^{2}PACK), a slouží zejména k řešení 3D-BPP. První úroveň se stará o přiřazení předmětů do kontejnerů. Každý předmět, který je přiřazen do určitého kontejneru, je umístěn druhou úrovní algoritmu, která používá reprezentaci pomocí intervalových grafů (viz obrázky 2.5). Přesnost tohoto algoritmu je vylepšena za pomocí *k-chain-move* procedury, která rozšiřuje vzdálenost mezi sousedními prvky bez zvýšení algoritmické náročnosti. TS^{2}PACK v současné době podává nejlepší výsledky v řešení 3D-BPP, avšak metoda konverguje k výsledku příliš pomalu a vyžaduje výpočetní čas přibližně 300 CPU sekund k dosažení nejlepšího výsledku.

2.4.7 Greedy Adaptive Search Procedure (GASP)

Metoda GASP představená v [Perboli et al., 2011], je určena jak pro řešení 2D-BPP, tak pro 3D-BPP. Pro dosažení výsledku využívá tato metoda jednoduchosti hladových algoritmů ve spojení s mechanismem učení. GASP je schopna dosáhnout kvalitního výsledku v zanedbatelném čase oproti ostatním metodám a tyto metody překonat i kvalitou řešení.

2.4.8 Testovací třídy příkladů

Všechny porovnávané algoritmy byly testovány na standardních třídách příkladů.

Pro 2D-BPP třídy I-VI pocházejí od [Berkey and Wang, 1987] a pro třídy VII-X od [Martello et al., 2000]. Každá třída obsahuje jiné rozložení velikostí předmětů , počty předmětů v každé třídě odpovídají 20, 40, 60, 80 a 100. Pro každou třídu je výpočet proveden desetkrát.

Třídy pro 3D-BPP jsou rozděleny na dvě skupiny, a to I-III. Velikost kontejnerů je rovna $W = H = D = 100$, v každé této třídě je umísťováno pět typů balení s rozměry rozprostřenými od malých do velkých.V ostatních třídách jsou rozměry kontejnerů a předmětů rozděleny podle následujících pravidel:

- Class IV - $w_i, l_i, h_i \sim U[1, 10]$ and $W = L = H = 10$
- Class V - $w_i, l_i, h_i \sim U[1, 35]$ and $W = L = H = 40$
- Class VI - $w_i, l_i, h_i \sim U[1, 100]$ and $W = L = H = 100$

Počty předmětů jsou stanoveny na 50, 100, 150 a 200 pro každou třídu. Pro každou kombinaci třídy a počtu předmětů je výpočet prováděn desetkrát.

2.4.9 Porovnání kvality algoritmů

2D BPP

Všechny porovnávané metody byly testovány na třídách uvedených v kapitole 2.4.8 a výsledky porovnání byly získány v [Crainic et al., 2012b]. Nejprve se zaměříme na algoritmy řešící 2D-BPP, výsledky můžeme vidět v tabulce 2.1. V prvním sloupci je uvedena třída, která byla řešena, v druhém a třetím sloupci jsou hodnoty výsledků testovaných metod (počty zaplněných kontejnerů) a ve čtvrtém sloupci je uvedeno nejlepší známé řešení (vzaté z literatury). V posledním sloupci je procentuální rozdíl mezi metodou TSPACK a nejlepším známým řešením oproti metodě GASP(záporné procento znamená lepší řešení metody GASP).

	Kvalita řešení			GASP vs	
Třída	**GASP**	**TSPACK**	**NZR**	**TSPACK**	**NZR**
I	100,1	101,5	99,7	-1,40%	0,40%
II	12,9	13,0	12,4	-0,81%	4,03%
III	70,6	72,3	68,6	-2,48%	2,92%
IV	13,0	12,6	12,4	3,23%	4,84%
V	90,1	91,3	89,1	-1,35%	1,12%
VI	11,8	11,5	11,2	2,68%	5,36%
VII	83,1	84,0	82,7	-1,09%	0,48%
VIII	83,6	84,4	83,0	-0,96%	0,72%
IX	213,0	213,1	213,0	-0,05%	0,00%
X	51,4	51,8	50,4	-0,79%	1,98%
Celkem	729,6	735,5	722,5	-0,82%	0,98%

Tabulka 2.1: 2D-BPP kvalita řešení

V tabulce 2.1, můžeme vidět, že algoritmus GASP má o $0,82\%$ lepší výsledky než metoda TSPACK. A to i přes kratší dobu výpočtu, která byla u GASP stanovena na $3s$ a u metody TSPACK na $12s$. Metoda GASP je také oproti TSPACK modifikovatelnější a její algoritmus je jednoduší na implementaci.

3D BPP

Porovnání metod řešících 3DBPP můžeme vidět v tabulkách 2.2 a 2.3. V tabulce 2.2 se časy potřebné k řešení danou metodou významně liší, hodnoty jsou uvedeny vždy pod metodou v druhém řádku tabulky. Pokud porovnáme časy jednotlivých metod můžeme si všimnout, že metoda GASP potřebovala k výpočtu $5s$, metody GLS a TSPACK $300s$, tedy $60\times$ více než metoda GASP. Metoda MPV (upravený algoritmus BaB) potřebovala k výpočtu $1000s$, tedy $200\times$ více než GASP.

V prvním sloupci je uvedena třída řešeného problému, v druhém sloupci se nachází rozměr kontejnerů, v třetím je uveden počet předmětů, které budou umístěny do kontejnerů. Ve čtvrtém sloupci se nachází výsledky metody GASP, v pátém - sedmém sloupci jsou procentuální rozdíly oproti GASP (opět záporné hodnoty znamenají lepší výsledek GASP). V posledním sloupci je uveden procentuální rozdíl metody GASP oproti nejlepšímu známému řešení.

Třída	**Kontejner**	**n**	**GASP** 5 s	**MPV** 1000 s	**GLS** 300 s	**TSPACK** 300 s	**NZR**
I	100	50	13,4	-1,47%	0,00%	0,00%	3,88%
		100	26,9	-1,47%	0,75%	0,75%	5,08%
		150	37	-3,14%	0,00%	0,00%	3,35%
		200	51,6	-1,34%	0,78%	0,98%	3,82%
II	100	50	29,4	0,00%	0,00%	0,00%	1,38%
		100	59	-0,17%	0,00%	0,17%	0,85%
		150	86,8	-0,46%	0,00%	0,00%	0,46%
		200	118,8	-0,59%	-0,17%	0,00%	0,42%
III	100	50	8,4	-8,70%	1,20%	1,20%	10,53%
		100	15,1	-13,71%	0,00%	-0,66%	7,86%
		150	20,6	-14,17%	1,98%	2,49%	9,57%
		200	27,7	-12,89%	1,84%	1,09%	6,54%
IV	10	50	9,9	1,02%	1,02%	1,02%	5,32%
		100	19,1	-1,55%	0,00%	0,00%	3,80%
		150	29,5	-0,34%	0,34%	1,03%	3,51%
		200	38	-0,52%	0,80%	0,80%	3,54%
V	40	50	7,5	-8,54%	1,35%	1,35%	10,29%
		100	12,7	-16,99%	3,25%	3,25%	10,43%
		150	16,6	-15,74%	5,06%	5,06%	15,28%
		200	24,2	-13,88%	2,98%	2,98%	6,61%
VI	100	50	9,3	-7,92%	1,09%	1,09%	6,90%
		100	19	-5,94%	0,53%	1,06%	3,26%
		150	24,8	-9,16%	3,77%	3,77%	10,22%
		200	31,1	-10,89%	4,01%	3,67%	10,28%
Total			736,4	-4,35%	0,85%	0,90%	3,89%

Tabulka 2.2: 3D-BPP kvalita řešení, rozdílné časy

Z tabulky můžeme vyčíst, že metody GLS a TSPACK podávají lepší výsledky než metoda GASP. Avšak kvalita výsledků se v průměru liší pouze o $0,85\%$ oproti GLS a $0,90\%$ oproti TSPACK. Metoda MPV je v průměru horší o $4,35\%$ a podává tedy nejhorší výsledky v řádově vyšším čase oproti ostatním metodám.

V tabulce 2.3 můžeme vidět porovnání metod GASP, GLS a TSPACK ve srovnatelných časech. Metoda GASP potřebovala k výpočtu $5s$, metody GLS a TSPACK potřebovaly $18s$. V prvním sloupci je opět třída problému, v druhém sloupci rozměr kontejnerů a v třetím počet předmětů. Ve čtvrtém sloupci se nachází výsledky metody GASP, v pátém a šestém sloupci jsou procentuální rozdíly oproti GASP. V posledním sloupci je uveden procentuální rozdíl metody GASP oproti nejlepšímu známému řešení.

Třída	**Kontejner**	**n**	**GASP** 5 s	**GLS** 18 s	**TSPACK** 18 s	**NZR**
I	100	50	13,4	0%	0%	3,88%
		100	26,9	0%	-0,37%	5,08%
		150	37	-1,33%	-1,86%	3,35%
		200	51,6	-2,27%	-2,64%	3,82%
IV	100	50	29,4	0%	0%	1,38%
		100	59	0%	-0,34%	0,85%
		150	86,8	-0,34%	-0,57%	0,46%
		200	118,8	-0,92%	-0,34%	0,42%
V	100	50	8,4	1,2%	1,2%	10,53%
		100	15,1	0%	-1,95%	7,86%
		150	20,6	-0,48%	-1,44%	9,57%
		200	27,7	-0,36%	-1,07%	6,54%
VI	10	50	9,9	1,02%	0%	5,32%
		100	19,1	-1,04%	-2,05%	3,8%
		150	29,5	0%	0,34%	3,51%
		200	38	-1,3%	-1,81%	3,54%
VII	40	50	7,5	1,35%	1,35%	10,29%
		100	12,7	3,25%	3,25%	10,43%
		150	16,6	5,06%	3,75%	15,28%
		200	24,2	-0,82%	-2,42%	6,61%
VIII	100	50	9,3	1,09%	1,09%	6,9%
		100	19	0,53%	-1,04%	3,26%
		150	24,8	1,22%	0,81%	10,22%
		200	31,1	1,63%	0,97%	10,28%
Total			736,4	-0,23%	-0,57%	3,89%

Tabulka 2.3: 3D-BPP kvalita řešení, srovnatelné časy

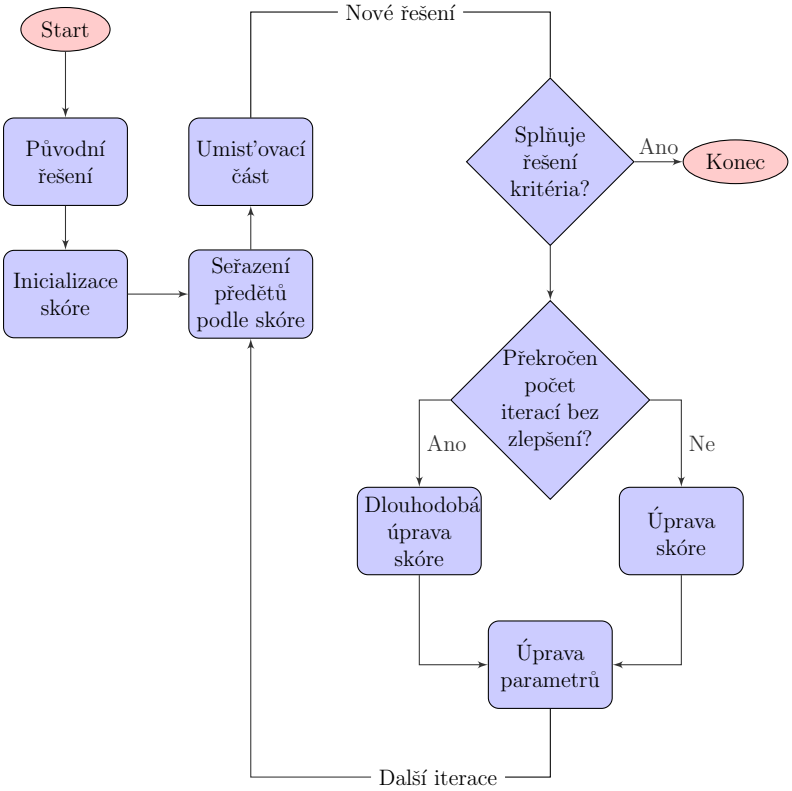
Metoda GASP dává nejlepší výsledky v průměru o $0,23\%$ (oproti GLS) a $0,57\%$ oproti TSPACK. Navíc metoda GASP k získání lepších výsledků potřebovala méně než třetinový čas v porovnání s ostatními metodami.

Zhodnocení

Z porovnání vychází jako nejkvalitnější metoda GASP. Tato metoda dokáže v zanedbatelném čase ($5s$) dosáhnout obdobných výsledků jako ostatní metody. Metoda GASP je také lehce modifikovatelná a díky tomu je optimální pro implementaci. Detailní popis této metody můžeme vidět v kapitole 2.5.

2.5 Greedy Adaptive Search Procedure (GASP)

Hlavní myšlemkou GASP algoritmu je rozdělit umisťování předmětů do kontejneru (umisťovací část) a řazení předmětů, tedy pořadí v jakém budou předměty do kontejneru umístěny (optimalizační část), schéma můžeme vidět na obrázku 2.8. O umisťovací část se stará hladový algoritmus, použití konkrétního hladového algoritmu záleží na reálném použití algoritmu a proto je popsán v kapitole 3.4.2.



Obrázek 2.8: Schéma metody GASP

V optimalizační části se předměty seřadí podle skóre. Skóre jednotlivých předmětů se v průběhu algoritmu upravuje, což slouží jako učící algoritmus. Metoda byla detailně popsána v [Crainic et al., 2012a].

Úprava skóre záleží na konkrétním případu využití GASP algoritmu. Může záležet na atributech předmětů a kontejnerů (např. rozměry, obsah základny, váha ...) a cíli algoritmu (např. optimalizace využití kontejneru, zvýšení zisku z reklam, minimalizace odřezků ...). Skóre předmětů je zohledňováno

vůči jejich atributům a změna skóre je prováděna každou iteraci v závislosti na kvalitě výsledku předchozí iterace.

Skóre je nejprve inicializováno podle původního řešení (*Inicializace skóre*), poté je upravováno pomocí *úpravy skóre*, nebo *dlouhodobé úpravy skóre*. *Úprava skóre* provádí pouze malé změny skóre jednotlivých předmětů, to je upraveno v závislosti na kvalitě řešení z předchozí iterace. *Dlouhodobá úprava skóre* provádí zásadnější změny skóre u předmětů, funguje jako dlouhodobá paměť a provádí se méně častěji. *Dlouhodobou úpravou skóre* se algoritmus vyhýbá cyklení na stejné množině řešení a snaží se posunout k jiné množině řešení.

Výpočet skóre a jeho úprava záleží na počtu parametrů, pro lepší výkonnost je dobré snažit se minimalizovat počet zohledněných parametrů. Pokud je to nutné, dynamická úprava parametrů je možná pomocí *úpravy parametrů*.

Posloupnost hlavních kroků algoritmu je vyobrazena na obrázku 2.8, jednotlivé části algoritmu budou popsány v následujících kapitolách. Kroky můžeme rozdělit do následujících skupin:

- Získání původního řešení
  - Původní řešení se získá za pomoci již existujícího algoritmu např.: C-EPBFD nebo C-EPFFD
  - Toto řešení se poté uloží jako nejlepší známé řešení
- Skóre
  - Inicializace skóre
  - Dokud nejsou splněna kritéria řešení
    - Seřaď předměty podle skóre.
    - Aplikováním hladového algoritmu získej nové řešení
    - Pokud nebylo získáno lepší řešení po stanovený počet iterací aplikuj *dlouhodobou úpravu skóre*. V opačném případě aplikuj *úpravu skóre*.
    - Pokud nově získané řešení je lepší než stávající nejlepší řešení, ulož si ho jako nejlepší řešení a vynuluj počet iterací bez zlepšení.

2.5.1 Inicializace skóre

Skóre je inicializováno podle původního řešení, vezme se první předmět, který byl umístěn do kontejneru, a přiřadí se mu skóre rovno n (počtu předmětů, které jsou umístěny v kontejnerech). Následujícímu balení se přiřadí skóre $(n-1)$ atd. Předmětům, které nebyly umístěny v řešení původního algoritmu, se přiřadí skóre 1.

2.5.2 Umist'ovací část

Určí se předmět s nejvyšším skóre a ten je umístěn do kontejneru podle zvoleného algoritmu, který odpovídá požadavkům reálného použití. Obecně (pro 2D-BPP) se vezme předmět s rozměry h_i a w_i a aplikuje se na něj nějaký z již existujících algoritmů pro umist'ování např.: C-EPBFD. Poté se určí další předmět a takto se pokračuje dokud nejsou všechny předměty umístěny.

2.5.3 Dlouhodobá úprava skóre

Dlouhodobá úprava skóre funguje podobně jako *inicializace skóre*, avšak místo původního řešení se použije dosavadní nejlepší řešení.

2.5.4 Úprava skóre

Obecně algoritmy řešící BPP dosahují skvělých výsledků u prvních kontejnerů a špatných u posledních. "Chyby", které jsou v seřazeném seznamu předmětů pro umístění, se většinou nalézají uprostřed seznamu předmětů, avšak tento seznam není znám dopředu. Hlavním důvodem pro úpravu skóre je tedy donutit algoritmus, aby zpřeházel předměty mezi kontejnery, které jsou lépe naložené a těmi, které jsou hůře. Aby toho bylo docíleno, tak je skóre upraveno podle následujícího vzorce (2.2).

$$s_i = \begin{cases} s_i(1-m) & b(i) \in \mathcal{B}' \subset \mathcal{B} \\ s_i(1+m) & \text{jinak} \end{cases} \tag{2.2}$$

Kde m je kladný parametr, který může být modifikován, $b(i)$ je kontejner v kterém je umístěn předmět i, $\mathcal{B}$ je množina všech kontejnerů do kterých jsou uloženy předměty a $\mathcal{B}'$ je taková podmnožina kontejnerů, které můžeme považovat za lépe naložené.

Tedy vzorec 2.2 penalizuje předměty, které jsou v lépe naložených kontejnerech, naopak zvýhodňuje předměty v hůře naložených kontejnerech. Množina $\mathcal{B}$ je seznam kontejnerů od 1 do $|\mathcal{B}|$, kde 1 je první kontejner, který byl vytvořen a $|\mathcal{B}|$ je poslední vytvořený. Z testů provedené v [Crainic et al., 2012a] vychází, že dobře naložené kontejnery se nacházejí v první polovině tedy $\mathcal{B}' = \left\{1, \ldots, \lfloor \frac{|\mathcal{B}|}{2} \rfloor \right\}$.

Hodnota m silně ovlivňuje prohazování předmětů a tím i výsledky algoritmu. Čím větší je hodnota m, tím více se předměty prohazují a tím odlišnější výsledky algoritmus produkuje. Naopak čím je hodnota menší, tím jsou změny v pořadí menší a je prozkoumáváno blízké okolí současného řešení. Pokud bychom volili m jako konstantu, museli bychom si vybrat, zda chceme prohledávat pouze více náhodné řešení z velké oblasti, nebo zlepšit stávající již nalezené řešení. Avšak oba tyto přístupy jsou nutné pro nalezení co nejlepšího řešení. Nejprve je důležité prohledat velké okolí (větší změny ve skóre) pro nalezení dobrého řešení a to pak ještě více optimalizovat za pomoci lokálního hledání (tedy menších změn ve skóre). Z toho vyplývá, že algoritmus musí hodnotu m upravovat dynamicky v závislosti na tom v jaké fázi se nachází.

V [Crainic et al., 2012a] navrhli změnu parametru m v závislosti na dvou parametrech, které jsou měněny na základě průběhu algoritmu a to:

- Parametr p
  - Počáteční hodnota je nastavená na $p = 1$
  - Po každé *dlouhodobé úpravě skóre* je hodnota p zvýšena o jedna, tedy $p = p + 1$
  - Maximální hodnota změny $\overline{s}$ je volitelná a je zvolena jako $\overline{s} = 10\%$
- Parametr k
  - Počáteční hodnota je nastavená na $k = 1$
  - Po každé *dlouhodobé úpravě skóre* je hodnota k opět nastavena na jedna, tedy $k = 1$

– Hodnota k je zvýšena o jedna, pokud bylo nalezeno nové nejlepší řešení

– Maximální hodnota k_{max} je opět volitelná a je zvolena jako $k_{max} = 4$

Pokud se nyní podíváme na vzorec 2.3 můžeme vidět, že algoritmus začíná s největšími změnami (obě proměnné jsou nejmenší) a v průběhu celého běhu algoritmu se změny zmenšují (p se postupně zvyšuje v rámci celého běhu algoritmu). Proměnná k se po každé *dlouhodobé úpravě skóre* nastaví na původní hodnotu. A s každým lepším nalezeným řešením se zvýší, tedy omezí se prohazování předmětů v seznamu a menší okolí aktuálního řešení je prohledáváno.

$$m = \frac{\overline{s}}{p}(k_{max} - k) \tag{2.3}$$

Výsledný vzorec úpravy skóre (2.4) umožňuje metodě GASP velké změny v pořadí předmětů a tím prohledání velké oblasti řešení a také přesnější hledání s menšími změnami v pořadí předmětů, tedy prohledání menšího okolí u nejlepších nalezených řešeních.

$$s_i = \begin{cases} s_i\big(1 - \frac{\overline{s}}{p}(k_{max} - k)\big) & b(i) \in \mathcal{B}' \subset \mathcal{B} \\ s_i\big(1 + \frac{\overline{s}}{p}(k_{max} - k)\big) & \text{jinak} \end{cases} \tag{2.4}$$

2.5.5 Úprava parametrů

V tomto kroku se upravují parametry p a k v závislosti na průběhu algoritmu; a to tak že:

- p
  - je zvětšeno o jedna, pokud proběhla *dlouhodobá úprava skóre*
- k
  - je nastaveno na jedna, pokud proběhla *dlouhodobá úprava skóre*
  - je zvýšeno o jedna, pokud bylo nalezeno nové nejlepší řešení a nebylo dosaženo k_{max}

3 Implementace BPP pro DCIx

3.1 DCIx

DCIx je systém určený pro řízení materiálových pohybů a správu dodavatelského řetězce. Ke své činnosti systém používá jednoznačné identifikování reálných objektů ze skladu (balení, skladové pozice, obaly atd.) a to za pomocí technologií čárkových a 2D kódů, Radio Frequency Identification (RFID) a kanbanu. Informace o baleních (množství, datum přijetí, pozice uložení) a jiných reálných objektech jsou uloženy v databázi systému. Bližší popsání detailní části databázového schématu, která je využita pro algoritmus nakládky, je uvedena v kapitole 3.3. Informace o systému DCIx byly čerpány z [dci, 2013].

Systém DCIx je postaven na frameworku Apache Struts, tedy Java EE (Java platform, Enterprise Edition). Pro databázovou část systému je použit Microsoft SQL Server, jazyk Transact-SQL, což je rozšíření jazyka SQL.

Implementace logistických procesů je v systému zabezpečena za pomocí tzv. transakčních definic, tyto definice se skládají z posloupnosti modulů, které jsou během logistického procesu vykonávány. Jednotlivé moduly zabezpečují práci s daty v systému, jakožto například načtení balení z databáze, úprava množství, opětovné uložení balení a vytištění etikety. Nastavení procesů za pomocí transakčních definic, zabezpečuje systému DCIx velkou flexibilitu

V následujících kapitolách se seznámíme s objekty, které jsou používány v systému DCIx a které budou využity jako vstupní a výstupní data pro algoritmus nakládky. Tedy řešení problému BPP za pomoci algoritmu GASP, který byl vybrán na základě porovnání algoritmů uvedeném v kapitole 2.4.9. Dále budou uvedeny nové objekty, které byly vytvořeny pro algoritmus, a nakonec popis samotné implementace jednotlivých částí algoritmu.

3.2 Požadavky na algoritmus

Nejprve si musíme uvést požadavky na daný algoritmus, který bude umisťovat předměty do kontejnerů daných rozměrů. Vstupem bude konečný seznam předmětů, které mají být umístěny, a rozměry kontejneru. Předměty, které mají být do kontejnerů naloženy, budou načteny podle dodacích listů 3.3).

Pouze některé předměty bude možné skládat do sloupců na sebe, a takovéto předměty budou označovány za kompatibilní. U sloupce kompatibilních předmětů budeme předpokládat shodné rozměry základny předmětů, které budou odpovídat prvnímu předmětu umístěnému ve sloupci. Výška sloupce bude odpovídat součtu výšek předmětů v daném sloupci. Tímto způsobem je redukován 3DBPP na 2D (umístění nového sloupce do kontejneru) a 1D (umístění předmětu do kompatibilního sloupce) BPP.

Algoritmus se pokusí naložit do kontejneru všechny předměty z dodacích listů. Pokud se předměty nevejdou do jednoho kontejneru, odebere se poslední řádka z posledního dodacího listu, opět se načtou předměty a opět se pokusí naložit všechny předměty do jednoho kontejneru.

3.3 Vstupní a výstupní data

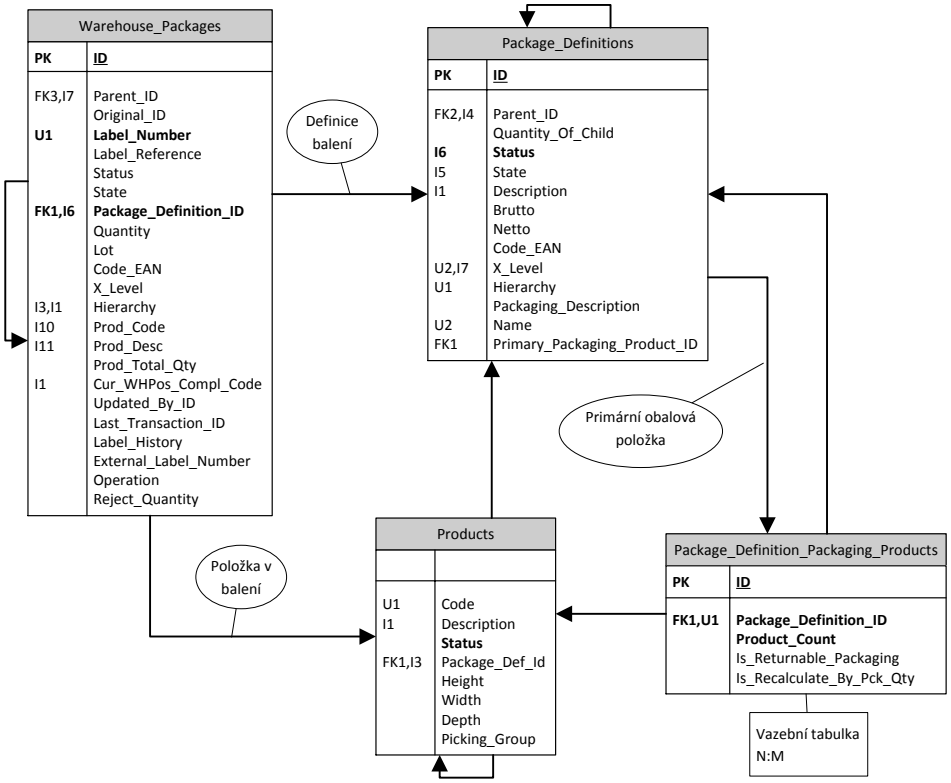
Vstupy pro algoritmus (viz kapitola 2.1.1) jsou jednotlivé předměty a kontejnery. Tyto vstupy je nutné identifikovat na reálných objektech, viz následující kapitoly. Jako výstupy budou opět využity již existující objekty v systému, avšak bude nutné vytvořit i nové výstupní objekty.

3.3.1 Existující objekty a jejich využití

Předměty

Předměty, které budou použity jako vstup pro algoritmus nakládky, jsou v DCIx označovány jako balení. Balení jsou uložena v tabulce *Warehouse_Packages*, viz schéma 3.1. Balení v DCIx mohou být víceúrovňová, to znamená, že například balení, které představuje paletu, může obsahovat další balení, které představují boxy apod. Definice toho, jak jsou jednotlivá balení uspo-

řádána, se nacházejí v tabulce *Package_Definitions* (schéma 3.1). Dalším objektem, který úzce souvisí s balením, je položka (tabulka *Products*). Položka určuje o jaký reálný výrobek nebo materiál se jedná. Dále jsou také položky využity pro vedení evidence o obalech (jakožto obalová položka). Na primární obalovou položku je odkazováno z balící definice.



Obrázek 3.1: Zjednodušené databázové schéma balení v systému DCIx

Ke každému balení musíme přiřadit příslušné rozměry, tedy šířku, hloubku a výšku. Tyto údaje budou využity z tabulky položek *Products*, z vazby primární obalová položka. Aby bylo v budoucnu možné tyto hodnoty vzít z jiného objektu (např. balící definice aj.), aniž by bylo nutné přepisovat implementovaný kód, byl vytvořen databázový pohled (*Warehouse_Packages_Loading_View*) obsahující příslušné údaje z primární obalové položky (viz schéma 3.1).

Dodací list

Je dokument, který vystavuje dodavatel, podle kterého příjemce zboží může potvrdit, zda všechno zboží bylo doručeno. V systému DCIx je dodací list reprezentován objektem příkazu, který má hlavičku, podle níž je identifikován. Řádky příkazu, které jsou navázány na hlavičku, obsahují informace o položce a jejím množství, každá řádka obsahuje pouze jednu položku.

Seznam dodacích listů je hlavním vstupem, podle kterého budou načteny balení, které mají být naloženy do kontejneru.

Dodávka

Dodávka je objekt, který reprezentuje naložený kontejner. Na dodávku se vážou dodací listy, které jsou v ní naloženy. Dodací listy nelze dělit mezi jednotlivé dodávky, pouze lze zrušit řádky dodacího listu. Z těchto zrušených řádek vytvořit nový dodací list a ten uložit do jiné dodávky. V DCIx je dodávka reprezentována opět objektem příkazu.

3.3.2 Nové objekty

V rámci výstupu algoritmu bylo nutné navrhnout nové objekty, které budou reprezentovat umístěné předměty (balení v kontejneru).

Ložní sloupec

Ložní sloupec bude reprezentovat sloupec balení v dodávce. Na ložní sloupce budou alokována jednotlivá balení. Výstupem algoritmu bude seznam ložních sloupců spolu se souřadnicemi sloupce v daném kontejneru. Tato data poté bude možno zobrazit za pomoci Reporting Services (viz příloha B), které se pro zobrazování dat v systému DCIx používají.

3.4 Implementace algoritmu

Algoritmus bude implementovaný jako samostatný modul (viz kapitola 3.1). Nejprve bude nutné načíst balení z dodacích listů, dalším krokem bude umístění balení do kontejneru za pomoci algoritmu GASP. Poté se ověří zda se balení vešla do jednoho kontejneru, pokud ne, tak algoritmus zruší poslední řádku posledního dodacího listu a bude opakovat umisťování.

Nakonec algoritmus vytvoří ložní sloupce a objekt nakládky a data potřebná pro zobrazení reportu umístění sloupců v kontejneru. A tyto objekty uloží do databáze.

V následujících kapitolách jsou uvedeny zkrácené ukázky kódu, které osvětlují provedenou implementaci nakládky do systému DCIx. Jsou zde popsány pouze zásadní části implementovaného algoritmu.

3.4.1 Načtení balení

Načtení balení z dodacích listů může být ovlivněno mnoha parametry, které záleží na daném zákazníkovi. Daný zákazník například může vyskladňovat položky podle FIFO (popř. FEFO), počty položek v jednotlivých baleních se také mohou přepočítávat podle různých parametrů a podobně.

Není tedy možné bez přesných požadavků daného zákazníka implementovat načtení balení dopředu. Také je velmi pravděpodobné, že každý zákazník bude mít speciální požadavky na načítání balení, a proto je vhodné navrhnout kód tak, aby byl lehce modifikovatelný a udržovatelný.

V následujících kapitolách jsou rozebrány zvažované alternativy pro implementaci načítání balení.

Použití Java enumu

Java enum umožňuje jednotlivým hodnotám z výčtu definovat i chování, metody které definují chování jsou deklarované v hlavním těle enumu a jednotlivé hodnoty z výčtu mohou tyto metody přepsat (viz příklad použití v listingu 3.1, převzáno z tutoriálu [enu, 2013]). Použití výčtu zpřehledňuje kód, neboť ve výčtu jsou uvedeny hodnoty se stejnými metodami a oproti

klasickému dědění se dá přes výčet iterovat. Od enumů se nedá dědit a neboť enum implicitně dědí od *java.lang.Enum* nemůže dědit ani od jiné třídy (více viz [enu, 2013]).

Listing 3.1: Java enum, příklad použití

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6), //vycet prvku
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;
    private final double radius;
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }


    public static final double G = 6.67300E-11;

    double surfaceGravity() { // definice metody vyctu
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Planet <earth_weight>");
            System.exit(-1);
        }
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())    // iterovani pres vycet
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
}
```

Výhodou tohoto řešení je přehlednost kódu, možnost iterovat přes výčet, tedy v nastavení modulu, lze jednoduše vygenerovat selectbox s hodnotami

výčtu. Uživatel si tak jednoduše může vybrat příslušnou metodu pro načtení balení. Nevýhodou však je při přidání nové metody nutnost restartovat produkční prostředí, což by v některých případech mohlo být nevyhovující. Další nevýhodou je nemožnost oddělit zákaznický kód, neboť hodnoty výčtu musí být součástí jedné třídy. A tak by zákaznický kód musel být součástí standardního, což by bylo nepřehledné a nežádoucí.

Použití Java procedur

Java procedury je framework vyvinutý a používaný v DCIx. Slouží ke spouštění sql procedur, funkcí (popř. jiného sql kódu) z Javy. Java procedury se překládají za běhu serveru, tedy při vytvoření, nebo úpravě Java procedury není nutné překompilovat celý projekt a restartovat server, což je v produkčním prostředí velká výhoda.

Hlavní částí Java procedury je vykonávaný kód (viz listing 3.2). Jako první můžeme vidět anotaci *@Event(EventEnum.TRANSACTION_XP_READ_PACKAGES_FOR_LOADING)*, tato anotace specifikuje, kde má být procedura použita (v jakém modulu, části systému DCIx). Metoda *readPackagesByOrder* má parametr *TransactionMessengerXP messenger*, což je objekt, který slouží pro uložení dat v transakci (předávání dat mezi moduly), získání konekce do databáze, uložení chyb a varování.

Listing 3.2: Java procedura

```
@Override
@Event(EventEnum.TRANSACTION_XP_READ_PACKAGES_FOR_LOADING)
public void readPackagesByOrder(TransactionMessengerXP messenger) {

        // ziskani parametru pro sql proceduru
        ...
        // konec ziskani parametru


        List<Item> itemsToLoad = new ArrayList<Item>();

        // spusteni procedury
        ResultSet rs = Table.selectResultsetFromStoredProcedure(
            messenger.getConnection(),
            DciDatabase.decorateWithSchema(PROCEDURE_NAME),
            valueList);
        try {
```

```
                while (rs.next()) {
                // ziskani seznamu baleni z result setu
                        WarehousePackageItem packageItem = new
                            WarehousePackageItem();
                        packageItem.setId(rs.getInt(WarehousePackageMapping
                            .ID.getName()));
                        ...
                // konec ziskani seznamu baleni z result setu
                }

        } catch (SQLException sqle) {
                throw new ApplicationRuntimeException("Can't read
                    operation.", sqle);
        } finally {
                Table.closeResultSet(rs);
        }

        // ulozeni seznamu baleni do hidden values
        // pass into HV
        messenger.addHiddenValue(
            TransactionHiddenFieldDescriptor.PACKAGES_FOR_LOAD,
            itemsToLoad);
}
```

Další částí Java procedury je získání parametrů procedury z kontextu skriptu a zavolání výkonného kódu procedury (viz listing 3.3).

Listing 3.3: Získání parametru a zavolání výkonného kódu

```
public static void setScriptContext(ScriptContext ctx) {
        ReadPackagesByProductQuantityOnOrders proc = new
            ReadPackagesByProductQuantityOnOrders();

        proc.readPackagesByOrder((TransactionMessengerXP)
            ctx.getAttribute("messenger"));
}
```

Zavolání Java procedury z Javy, tedy z akce nebo v našem případě modulu, se provede následovně. Nejprve je nutné proceduru získat za pomoci jejího názvu z cache procedur (pomocí statické metody), to je možné vidět v listingu 3.4. Pokud nebude procedura v cache nalezena vrátí se chyba.

Listing 3.4: Načtení procedury z cache

```
//read procedure from cache
Procedure procedure = ProcedureCache.getProcedure(procedureName);
if(AlfaObject.hasNotValidKey(procedure)) {
        DciErrorCodes.procedureNotFound.addError(messenger.getErrors(),
            procedureName);
        return null;
}
```

Poté je připraven kontext procedury (uložen do mapy objekt messengeru) a procedura je spuštěna za pomoci frameworku (viz listing 3.5). Pokud ještě nebyl kód procedury přeložen do bytecode, framework se o její přeložení postará.

Listing 3.5: Příprava kontextu procedury a spuštění za pomoci frameworku

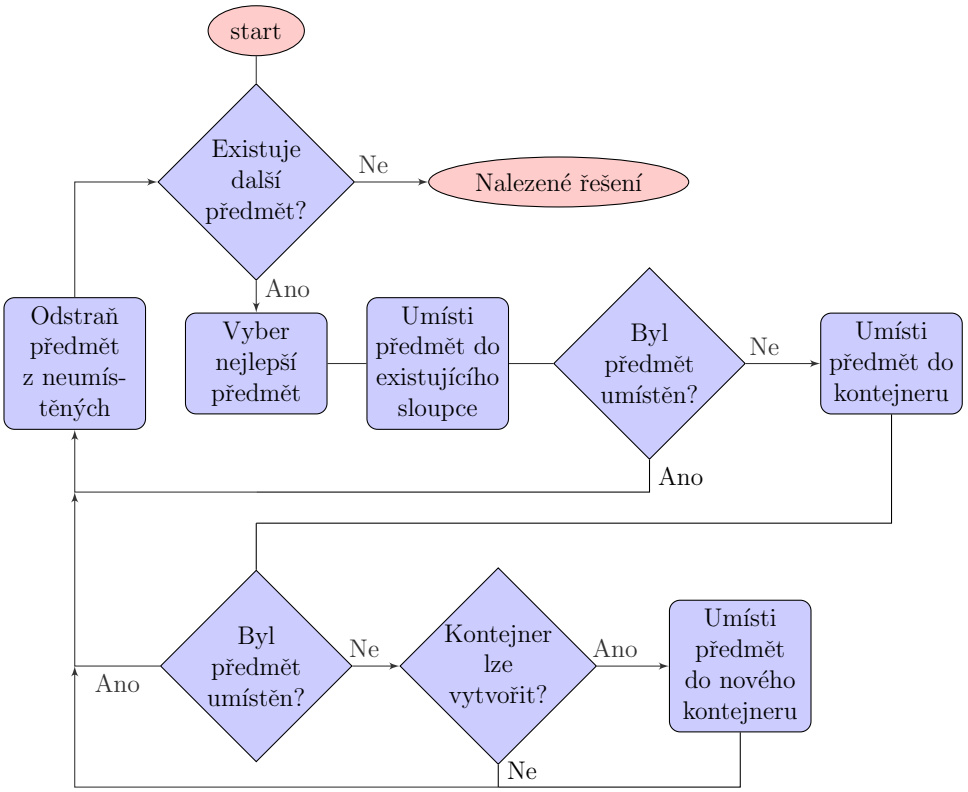
```
Map<String, Object> bean = new HashMap<String, Object>();
bean.put("messenger", messenger);

// read packages run procedure
ProcedureDispatcher.executeScript(procedure, bean,
    messenger.getErrors(), messenger.getWarnings(),
    messenger.getConnection());
```

3.4.2 Získání řešení ze seznamu předmětů

Jednou ze základních částí algoritmu GASP je vnitřní heuristika pro umístění balení do algoritmu. Pro tento účel byla použita C-EPBFD, tedy heuristika využívající EB (viz kapitola 2.3.7). Tato vnitřní heuristika je použita jak v kroku *Původního řešení* (pro jeho získání), tak v kroku *Umist'ovací část* metody GASP (viz schéma 2.8 uvedené na stránce 17). Algoritmus C-EPBFD byl doplněn o umístění předmětů do kompatibilních sloupců (viz schéma 3.2).

Algoritmus začíná s prázdným řešením, tedy neexistuje žádný vytvořený kontejner a všechny předměty jsou v seřazeném seznamu neumístěných předmětů.

V každém kroku se vezme první balení z neumístěných a projdou se všechny sloupce, zda existuje kompatibilní sloupec, do kterého je možné před-



Obrázek 3.2: Hladová metoda

mět umístit. Pokud takový sloupec existuje, předmět je do něj umístěn a algoritmus pokračuje dalším předmětem.

Pokud vhodný sloupec neexistuje, algoritmus se snaží umístit předmět do existujícího kontejneru na nejvhodnější místo (nejlepší poměr obsahu základny předmětu oproti obsahu základny volného místa pro umístění). Pokud se nenajde vhodné místo, volná místa v kontejneru mají menší rozměry než umist'ovaný předmět, tak je vytvořen nový kontejner, pokud již není dosaženo maximálního počtu kontejnerů. V této části algoritmu je nutné upravit rozměry oblastí extrémních bodů, to je popsáno v následující kapitole.

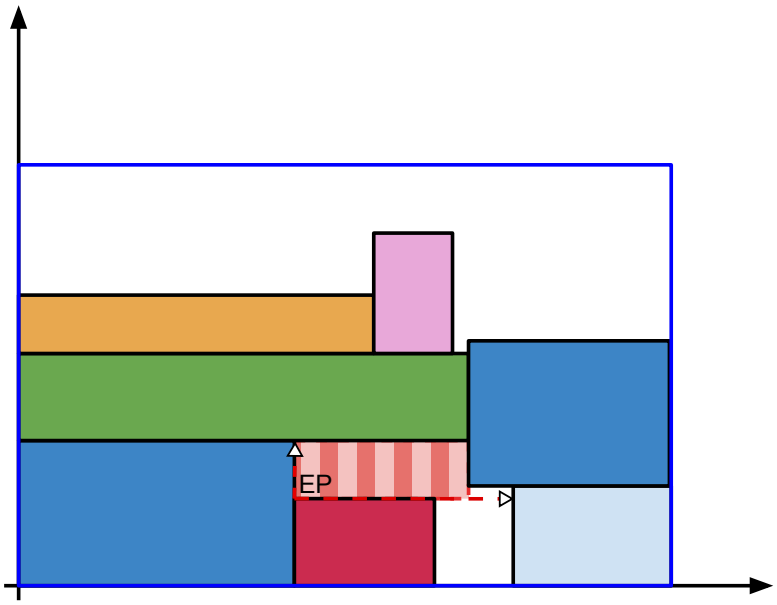
Vznik a úprava rozměrů oblastí extrémních bodů

Při každém vložení nového sloupce do kontejneru vzniknou maximálně dva extrémní body (v případě 2D problému, který řešíme), pokud se jedná o

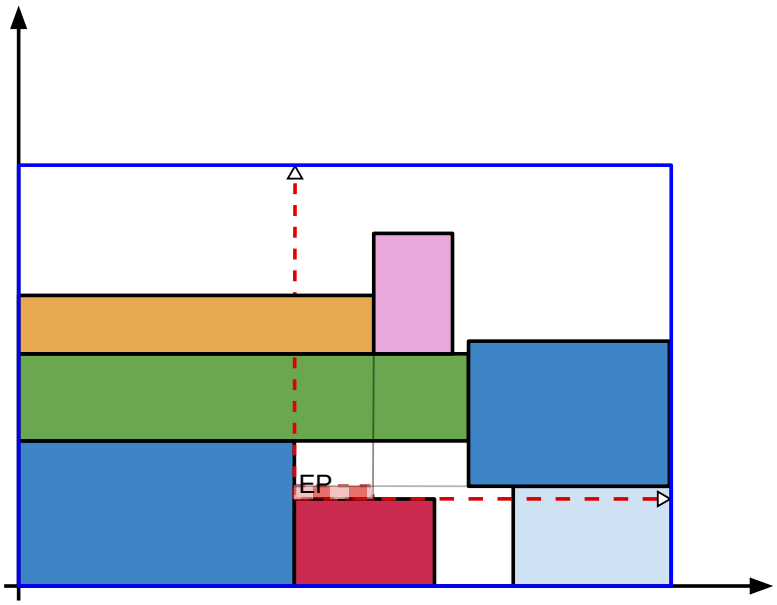
prázdný kontejner s právě jedním vloženým sloupcem extrémní body budou mít souřadnice $EB_x = (w_i, 0), EB_y = (0, h_i)$.

Po vložení sloupce do kontejneru, kde již existují i jiné sloupce je generování extrémních bodů složitější. Pro $EB_x = (w_i + x_i, y_{EB_x})$ je x souřadnice lehce spočítána. Souřadnice y_{EB_x} je získána promítnutím na stěnu nejbližšího sloupce ve směru k nulté souřadnici, pokud se na této straně žádný sloupec nevyskytuje bude souřadnice $y_{EB_x} = 0$. Pro $EB_y = (x_{EB_y}, h_i + y_i)$ je souřadnice x_{EB_y} získána obdobně.

Každý extrémní bod dále obsahuje informace o šířce a výšce, tedy maximálních rozměrech sloupce, jaký může být vložen na tento extrémní bod. Ty jsou opět při generování nových extrémních bodů dopočítány. Nejprve je šířka a výška oblasti extrémního bodu odhadnuta, a to tak že je zjištěn průmět extrémního bodu v souřadnících x a y ve směru od 0 (viz obrázek 3.3). A poté jsou dopočítány konečné rozměry oblasti extrémního bodu a to tak, že je získán průmět stran oblasti extrémního bodu opět ve směru od 0 (viz obrázek 3.3). Pokud bychom neudělali prvotní aproximaci a rozměry oblasti by byly odhadnuty podle rozměrů kontejneru, získali bychom menší oblast než ve skutečnosti můžeme pro umístění sloupce použít (viz obrázek 3.4).



Obrázek 3.3: Určení oblasti EB, úvodní aproximace průmětem



Obrázek 3.4: Určení oblasti EB, úvodní aproximace velikostí kontejneru

Dále je nutné projít již existující extrémní body, zda přidávaný sloupec nezasahuje do jejich oblastí a pokud ano, tak upravit jejich rozměry. Posledním krokem je seřazení extrémních bodů, podle jejich pozice, odstranění duplicit a nevalidních oblastí (s rozměry rovnými nule).

3.4.3 Porovnání kvality řešení

Důležitou částí algoritmu je porovnání, zda je jedno řešení lepší než druhé. Porovnat řešení je možné podle různých parametrů. Ať už se jedná o porovnání z hlediska využití prostoru (např. počet použitých kontejnerů, využití objemu apod.), nebo z hlediska procentuálního splnění dodacích listů.

Do implementovaného kódu by mělo být později možné jednoduše přidat další atribut, podle kterého se dají řešení porovnat. A dále musí být umožněno určit pořadí atributů, při porovnávání.

Jednotlivé metody porovnání podle atributů byly implementovány jako hodnoty výčtu (viz listing 3.6). Tyto metody jsou poté vkládány do kolekce komparátoru.

Listing 3.6: BinsComparatorMethods - výčet prvků

```
public enum BinsComparatorMethods implements Comparator<List<Bin>>
    {
        VOLUME {
                @Override
                public int compare(List<Bin> o1, List<Bin> o2) {
                        // vypocet objemu kontejneru a sploupcu
                        ...
                        //konec vypoctu
                        double difference = (o1BinsVolume -
                            o1ColumnsVolume) - (o2BinsVolume -
                            o2ColumnsVolume);
                        return (int) Math.signum(difference);
                }
        },
        BASE_AREA {
                @Override
                public int compare(List<Bin> o1, List<Bin> o2) {
                        // vypocet obsahu ázkladen kontejneru a
                            sploupcu
                        ...
                        //konec vypoctu
                        double difference = (o1BinsArea -
                            o1ColumnsArea) - (o2BinsArea -
                            o2ColumnsArea);
                        return (int) Math.signum(difference);
                }
        },
        BINS_COUNT {
                @Override
                public int compare(List<Bin> o1, List<Bin> o2) {
                                return o1.size() - o2.size();
                }
        };
```

Tělo výčtu (viz listing 3.7) obsahuje abstraktní metodu *compare* z rozhraní *Comparator*, kterou musí každý z prvků výčtu implementovat. A dále obsahuje metodu *contains*, která vrací *true* pokud v enumu existuje výčet s daným jménem, v opačném případě vrací *false*.

Listing 3.7: BinsComparatorMethods - tělo výčtu

```
        @Override
```

```
        public abstract int compare(List<Bin> o1, List<Bin> o2);

        /**
         * @param name enum name
         * @return <code>true</code> if contain enum with this name
         */
        public static boolean contains(String name) {
                BinsComparatorMethods[] values = values();
                for (BinsComparatorMethods binsComparatorMethods :
                    values) {
                        if
                            (binsComparatorMethods.toString().equals(name))
                            {
                                return true;
                        }
                }
                return false;
        }
}
```

Hlavní třída pro porovnání kvality řešení se nazývá *SolutionComparator* (viz listing 3.8) opět implementuje rozhraní *Comparator*. Obsahuje metodu *add* pro přidání metody pro porovnání, tedy hodnoty z výčtu *BinsComparatorMethods*. Další metodou je *compare* z rozhraní *Comparator*, která porovnává řešení podle hodnot výčtu v kolekci *comparators* a to v pořadí v jakém byly vloženy. Poslední metoda *isEmpty* slouží k ověření zda je kolekce *comparators* prázdná, tedy neobsahuje žádnou metodu pro porovnání.

Listing 3.8: SolutionComparator

```
public class SolutionComparator implements Comparator<List<Bin>> {
        List<BinsComparatorMethods> comparators = new
            ArrayList<SolutionComparator.BinsComparatorMethods>();

        public boolean add(BinsComparatorMethods comparator) {
                return comparators.add(comparator);
        }

        @Override
        public int compare(List<Bin> o1, List<Bin> o2) {
                for (BinsComparatorMethods comparator : comparators)
                    {
```

```
                    int currentCompare = comparator.compare(o1,
                        o2);
                    if (currentCompare != 0) {
                            return currentCompare;
                    }
            }
            // solutions are equal
            return 0;
    }

    public boolean isEmpty() {
            return this.comparators.isEmpty();
    }

}
```

3.4.4 Implementace metody GASP

Podstatnou částí implementovaného kódu je metoda GASP (viz listing 3.9). Jedná se o statickou metodu s parametry *List<Item> items* seznam předmětů, které mají být umístěny do kontejnerů, dále *Bin bin* objekt kontejneru obsahující maximální rozměry kontejnerů. Dalšími parametry je *int maxBinCount* - maximální počet naložených kontejnerů a jako poslední parametr je *Comparator<List<Bin>> solutionComparator*, který slouží k porovnání dvou řešení viz kapitola 3.4.3.

Nejprve je nutné algoritmus inicializovat za pomocí získání úvodního řešení a to pomocí algoritmu uvedeném v kapitole 3.4.2. Podle tohoto řešení je určeno skóre jednotlivých předmětů. Nakonec jsou nainicializovány proměnné, které budou použity v cyklu (viz listing 3.9).

Listing 3.9: Implementace metody GASP

```
public static List<Bin> getSolutionByGASP(List<Item> items, Bin
    bin, int maxBinCount, Comparator<List<Bin>> solutionComparator)
    {
        // get initial solution
        List<Bin> bestSolution = getSolutionFromOrderedItems(items,
            bin, maxBinCount);
        // initialize score
        setColumnsScoresBySolution(items, bestSolution);
```

```
Comparator<Item> itemsScoreComparatorDesc =
    Collections.reverseOrder(new ItemScoreComparator());

int withoutImprovmentCount = 0;
int withoutImprovmentCycleCount = 0;
int p = 1;
int k = 1;
```

Na další ukázce kódu (listing 3.10) je hlavní tělo cyklu, před každou iterací je nejprve ověřeno, zda současné řešení nesplňuje konečné podmínky, nebo zda již neproběhl končený počet cyklů bez zlepšení řešení. Poté je seznam předmětů seřazen podle skóre sestupně a získáno nové řešení ze seřazeného seznamu.

Listing 3.10: Implementace metody GASP začátek cyklu

```
while (!isSolutionSufficient(bestSolution,
    withoutImprovmentCycleCount)) {
        // sort items by score descending
        Collections.sort(items, itemsScoreComparatorDesc);
        // get solution by greedy algorithm
        List<Bin> currentSolution =
            getSolutionFromOrderedItems(items, bin,
            maxBinCount);
```

Pomocí *solutionComparator* je zjištěno, jestli bylo nalezeno lepší řešení, pokud ano, tak se uloží jako nejlepší řešení a vynulují se proměnné *withoutImprovmentCount* a *withoutImprovmentCycleCount*, které udávají počet cyklů bez zlepšení. Pokud nebylo nalezeno lepší řešení, tak se upraví parametry k a p podle průběhu algoritmu viz kapitola 2.5.

Proměnná *withoutImprovmentCycleCount* slouží jako ochrana proti zacyklení algoritmu, tedy pokud se již daný počet iterací nenašlo lepší řešení, algoritmus je ukončen.

Listing 3.11: Implementace metody GASP úprava parametrů

```
        //compare if current solution than bestSolution
        if (solutionComparator.compare(currentSolution,
            bestSolution) < 0) {
                // currentSolution was better
```

```
                    bestSolution = currentSolution;
                    withoutImprovmentCount = 0;
                    withoutImprovmentCycleCount = 0;
                } else {
                    ++withoutImprovmentCount;
                    ++withoutImprovmentCycleCount;
                }
                if (withoutImprovmentCount > NOT_IMPROVMENT_LIMIT) {
                    //long term update score
                    setColumnsScoresBySolution(items,
                        bestSolution);
                    withoutImprovmentCount = 0;
                    ++p;
                    k = 1;
                } else {
                    setColumnsScoresByParameters(currentSolution,
                        SCORE_PERCENT_CHANGE, p, k, K_MAX);
                    k = k < K_MAX ? k + 1 : k;
                }
        }
        return bestSolution;
}
```

3.4.5 Implementace transakčního modulu

Začlenění algoritmu nakládky je provedeno pomocí transakčního modulu. Transakční modul je součástí transakční definice, kterou jsou v systému DCIx modelovány procesy (viz kapitola 3.1).

Nejprve je načteno nastavení transakčního modulu (viz listing 3.12). Nastavení modulu obsahuje název procedury, která bude použita pro načtení balení a seznam komparátorů pro rozhodnutí o lepším řešením. Dále nastavení obsahuje definici a stav příkazu, které jsou důležité pro vytváření ložního sloupce.

Listing 3.12: Načtení nastavení modulu

```
String procedureName =
    attributes.getAttributeValue(LoadContainerByPackagesAttributes
    .PROCEDURE_FOR_READ_PACKAGES);
String columnOrderDefinition =
    attributes.getAttributeValue(LoadContainerByPackagesAttributes
    .COLUMN_ORDER_DEFINITION);
String columnOrderState =
    attributes.getAttributeValue(LoadContainerByPackagesAttributes
    .COLUMN_ORDER_STATE);

// get comparators - this will decide which solution is better
String comparators =
    attributes.getAttributeValue(LoadContainerByPackagesAttributes.
    SOLUTION_COMPARATORS);
```

Následně jsou načteny vstupy z předchozích modulů (např.: zadaných uživatelem, načtených z databáze apod.). Vstupy modulu jsou šířka, hloubka a výška kontejneru, příkaz dodávky a kolekce dodacích listů. Všechny vstupy jsou povinné a proto jsou z transakce získávány za pomocí metody *extractValueWithCheck*, která v případě nenalezení vstupu v transakci vrátí chybu do kolekce *errors*. Po načtení vstupů se zkontroluje kolekce, zda neobsahuje chyby, pokud ano vyskočí se z modulu a o ošetření chyby se postará transakce.

Listing 3.13: Získání vstupů z transakce

```
CustomDecimal width = (CustomDecimal)
    messenger.getFinalValues().extractValueWithCheck(
    TransactionFieldDescriptor.width, errors);
CustomDecimal depth = (CustomDecimal)
    messenger.getFinalValues().extractValueWithCheck(
    TransactionFieldDescriptor.depth, errors);
CustomDecimal height = (CustomDecimal)
    messenger.getFinalValues().extractValueWithCheck(
    TransactionFieldDescriptor.height, errors);

Order loadNote = (Order)
    messenger.getFinalValues().extractObjectWithCheck(
    TransactionFieldDescriptor.orderNumber, errors);
```

```
OrderCommonCollection<? extends OrderInterface, ?> orders =
               (OrderCommonCollection<? extends OrderInterface, ?>)
                  messenger
               .getFinalValues().extractObjectWithCheck(
                  TransactionFieldDescriptor.ordersForBatch,
                  messenger.getErrors());

if (!errors.isEmpty()) {
       return null;
}
```

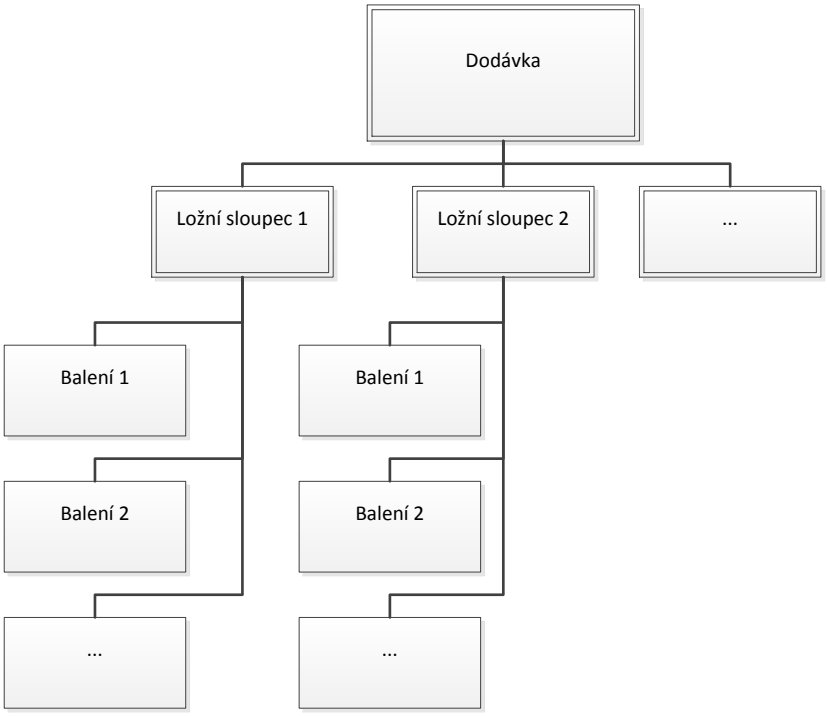
Pak již následuje cyklus (viz listing 3.14) v kterém jsou načtena balení za pomoci Java procedury (viz kapitola 3.4.1), ze seznamu balení je získáno řešení metodou *getSolutionByGASP* (kapitola 3.4.4), pokud byl naložen více než jeden kontejner, přidá se do kolekce vynechaných řádek dodací listů další řádka (metoda *addSkippedLine*). Cyklus končí pokud je naložen jeden nebo žádný kontejner.

Listing 3.14: Cyklus modulu

```
do {
        // get packages data
        itemList = readPackagesByOrder(procedureName, messenger);
        if (!errors.isEmpty()) {
                return null;
        }
        // get solution by algorithm
        actualSolution =
            ContainerLoadingCommon.getSolutionByGASP(itemList, bin ,
            MAX_BIN_COUNT, solutionComparator);
        if (actualSolution.size() > SUITABLE_BIN_COUNT) {
                Connection connection = messenger.getConnection();

                //add last line to skipped order line
                skippedIds = (HashSet<Integer>)
                    messenger.getHiddenValue(
                    TransactionHiddenFieldDescriptor
                    .SKIPPED_ORDER_LINES_IDS);
                skippedIds = addSkippedLine(skippedIds, orders,
                    connection);
                // pass into HV
                messenger.addHiddenValue(
                    TransactionHiddenFieldDescriptor
                    .SKIPPED_ORDER_LINES_IDS, skippedIds);
        }
} while (actualSolution.size() > SUITABLE_BIN_COUNT);
```

Po cyklu se řešení rozpadne na objekty v systému DCIx. Pro každý sloupec v kontejneru se vytvoří příkaz ložního sloupce a na ten se navážou balení pomocí vazební tabulky *Warehouse_Packages_Orders*. Ložní sloupce se dále navážou na příkaz nakládky vazbou *Original_Order*. Celkové provázání objektů můžeme vidět na schématu 3.5.

Poslední částí výstupu transakčního modulu jsou data do tabulky *Columns_In_Bins*, která obsahuje data pro zobrazení rozložení sloupců v kontejneru za pomoci Reporting Services. Strukturu tabulky můžeme vidět na listingu 3.15. *Load_Note_ID* je cizí klíč odkazující na příkaz dodávky, *Column_Order_ID* odkazuje na příkaz ložního sloupce. Sloupec *Order_In_Bin* určuje pořadí sloupce v daném kontejneru, tedy v jakém pořadí mají být sloupce ukládány do kontejneru, *Bin_Order* určuje číslo kontejneru, pokud



Obrázek 3.5: Provázání objektů výstupů

by byla dodávka tvořena více kontejnery. Sloupec *Spatial_Data* obsahuje geometrická data pro zobrazení sloupce na dané pozici. Geometrická data se v MS SQL Serveru ukládají za pomocí takzvaných *Spatial Data* (viz [spa, 2013]).

Listing 3.15: Struktura tabulky Columns_In_Bins

```
create table dciowner.Columns_In_Bins (
       ID                                  int identity not null,
       Load_Note_ID                        int not null,
       Spatial_Data                        geometry not null,
       Column_Order_ID                     int,
       Order_In_Bin                        int not null,
       Bin_Number                          int not null,

       constraint PK_Columns_In_Bins primary key (ID),
       constraint FK_Columns_In_Bins__Load_Note_ID__Generic_Orders
```

```
            foreign key (Load_Note_ID) references
                dciowner.Generic_Orders (ID),
        constraint
            FK_Columns_In_Bins__Column_Order_ID__Generic_Orders
            foreign key (Column_Order_ID) references
                dciowner.Generic_Orders (ID)
) on dataFile;
```

Výsledný report zobrazující uložení sloupců v kontejneru můžeme vidět v příloze na obrázku C.1. Čísla u jednotlivých sloupců představují čísla příkazu ložních sloupců. Každý sloupec v reportu je zobrazení řádku z tabulky *Columns_In_Bins*.

3.5 Automatické testy

Nedílnou součástí implementace algoritmu nakládky do systému DCIx jsou automatické testy. V rámci této práce byl implementovány jednotkové testy, které testují podstatné logické celky implementace nakládky. Dále byl vyvinut integrační test, který testuje transakční modul jako celek.

3.5.1 Jednotkový test

Základním testem, který byl v rámci implementace vytvořen je jednotkový test nad třídou *ContainerLoadingCommon*, která obsahuje následující statické metody:

addItemToExtremePoint vložení předmětu/sloupce do kontejneru na konkrétní extrémní bod

updateExtremePoints úprava extrémních bodů po vložení nového předmětu/sloupce do kontejneru

getSolutionFromOrderedItems získání řešení ze seznamu seřazených předmětů

getSolutionByGASP implementace algoritmu GASP

Tyto metody jsou otestovány za pomocí testovacího frameworku JUnit (viz JUn [2013]) a v následujících podkapitolách budou popsány některé z testů.

Test metody addItemToExtremePoint

V tomto testu je ověřeno správné vytváření extrémních bodů a rozměrů příslušných oblastí. Test obsahuje více scénářů pro ověření funkčnosti metody, v listingu 3.16 a 3.17 můžeme vidět první z nich. Nejprve je připraven první extrémní bod na pozici $[0, 0]$ s rozměry odpovídajícími rozměrům celého kontejneru, následně je vytvořen předmět, který bude vkládán do kontejneru. Poté je vytvořen seznam extrémních bodů. Na konci listingu můžeme vidět spuštění metody *addItemToExtremePoint* s připravenými daty.

Listing 3.16: Příprava dat a spuštění metody addItemToExtremePoint

```
ExtremePoint ep = new ExtremePoint();
ep.setX(0.0);
ep.setY(0.0);
ep.setResidualSpaceX(1000.0);
ep.setResidualSpaceY(150.0);

FakeItem item = new FakeItem();
item.setX(0.0);
item.setY(0.0);
item.setWidth(50.0);
item.setDepth(50.0);

ArrayList<ExtremePoint> extremePoints = new
    ArrayList<ExtremePoint>();
extremePoints.add(ep);

Bin bin = new Bin();
bin.setMaxX(1000);
bin.setMaxY(150);

bin.setExtremePoints(extremePoints);

ContainerLoadingCommon.addItemToExtremePoint(item, ep, bin);
```

V listingu 3.17 se nachází příprava očekávaných hodnot, tedy vytvoření extrémních bodů na pozicích $[0, 50]$ a $[50, 0]$ s příslušnými rozměry oblastí, tedy podle pravidel jak je popsáno v kapitole 3.4.2. Poté je ověřeno, že očekávané hodnoty jsou shodné s hodnotami vytvořenými metodou *addItemToExtremePoint*.

Listing 3.17: Vytvoření očekávaných hodnot a ověření funkčnosti metody addItemToExtremePoint

```
ExtremePoint epToPlace = new ExtremePoint();

epToPlace.setX(0.0);
epToPlace.setY(50.0);
epToPlace.setResidualSpaceX(1000.0);
epToPlace.setResidualSpaceY(150.0);
expectedEPs.add(epToPlace);

ep = new ExtremePoint();
ep.setX(50.0);
ep.setY(0.0);
ep.setResidualSpaceX(1000.0);
ep.setResidualSpaceY(150.0);
expectedEPs.add(ep);
```

Test metody updateExtremePoints

Aktualizace rozměrů oblastí extrémních bodů je jednou z velmi důležitých částí algoritmu. Před samotným testem jsou připraveny dva extrémní body a předmět, který bude vložen do kontejneru (viz listing 3.18).

Listing 3.18: Test metody updateExtremePoints, příprava dat

```
ArrayList<ExtremePoint> extremePoints = new
    ArrayList<ExtremePoint>();
ExtremePoint ep1 = new ExtremePoint();
ep1.setX(0.0);
ep1.setY(50.0);
ep1.setResidualSpaceX(1000.0);
ep1.setResidualSpaceY(150.0);
extremePoints.add(ep);
```

```
ExtremePoint ep2 = new ExtremePoint();
ep2.setX(50.0);
ep2.setY(0.0);
ep2.setResidualSpaceX(1000.0);
ep2.setResidualSpaceY(150.0);
extremePoints.add(ep);

FakeItem item = new FakeItem();
item.setX(0);
item.setY(50);
item.setWidth(150);
item.setDepth(50);
```

Po spuštění metody jsou vytvořeny očekávané hodnoty. Předmětem bude omezena oblast obou bodů. Velikost oblasti *ep2* bude nulová, proto bude odstraněna ze seznamu extrémních bodů. Následně jsou očekávané hodnoty porovnány s oblastmi extrémních bodů získané metodou (viz listing 3.19).

Listing 3.19: Test metody updateExtremePoints, validace hodnot

```
ContainerLoadingCommon.updateExtremePoints(item, extremePoints);

ArrayList<ExtremePoint> expectedEPs = new
    ArrayList<ExtremePoint>(1);
ep = new ExtremePoint();
ep.setX(50.0);
ep.setY(0.0);
ep.setResidualSpaceX(1000.0);
ep.setResidualSpaceY(50.0);
expectedEPs.add(ep);
assertListEquals(expectedEPs, extremePoints);
```

Test metody getSolutionFromOrderedList

Výstupem této metody jsou balení umístěná do sloupců v kontejnerech, tento výstup je však velmi komplexní a příprava očekávaných hodnot by byla velmi obtížná. Další nevýhodou validace přesných očekávaných hodnot této metody by byla nepřehlednost a prakticky nemožná údržba testu.

Ověření funkčnosti metody je docíleno pouze tím, že je řešení validní a že jsou všechny předměty umístěny tak, jak můžeme vidět v listingu 3.20. Metodou *prepareItemsInList* se připraví seznam předmětů, které mají být umístěny do kontejnerů, poté je vytvořen objekt reprezentující kontejner a spuštěna metoda *getSolutionFromOrderedItems*. Následně je zkontrolováno, zda jsou všechna balení z přípravy dat umístěna v kontejnerech (*isAllItemsLoaded*). Metodou *isSolutionValid* se ověří, zda je řešení validní a to následujícím způsobem (viz listing 3.21).

Listing 3.20: Test metody getSolutionFromOrderedList

```
List<Item> itemList = prepareItemsInList();

Bin bin = new Bin();
bin.setMaxX(800);
bin.setMaxY(150);
bin.setMaxZ(150);
List<Bin> actualSolution =
    ContainerLoadingCommon.getSolutionFromOrderedItems(itemList,
    bin , 20);
isAllItemsLoaded(actualSolution, itemList);
isSolutionValid(actualSolution);
```

V cyklu je pro každý kontejner ověřeno, že se sloupce vzájemně nepřekrývají (*columnsNotOverlapping*). Poté jsou zkontrolovány souřadnice každého sloupce, zda leží uvnitř kontejneru a nepřesahuje z něj (*columnInContainer*). Nakonec se ověří, že ve sloupci jsou pouze kompatibilní předměty (*itemsInColumnCompatible*) a ty se navzájem nepřekrývají (*itemsInColumnNotOverLap*).

Listing 3.21: Test metody getSolutionFromOrderedList, validita řešení

```
for (Bin bin : bins) {
        List<Column> columns = bin.getColumns();
        columnsNotOverlapping(columns);
        for (Column column : columns) {
                columnInContainer(bin, column);
                itemsInColumnCompatible(column);
                itemsInColumnNotOverLap(column);
        }
}
```

Obdobným způsobem je otestována i metoda *getSolutionByGASP*.

3.5.2 Integrační test

Integrační testy v systému DCIx jsou implementovány za pomocí frameworku Jameleon (viz [jam, 2008] a [Ašenbrener, 2010]). Test má za účel ověřit funkčnost modulu pro provedení nakládky jako celku. V testu je potvrzeno, že je modul schopný převzít vstupy z transakce, dále je otestováno, že jsou vytvořeny správné výstupy (ložní sloupce, data pro report a seznam řádek dodacích listů, které byly přeskočeny). Na závěr je ověřeno, že nevalidními vstupy a nastavením jsou vyvolány chybové stavy.

V následujících ukázkách kódu rozebereme jeden scénář integračního testu. Nejprve jsou nastaveny atributy modulu (viz listing 3.22), a to definice příkazu ložního sloupce (*COLUMN_ORDER_DEFINITION*) a jeho stav (*COLUMN_ORDER_STATE*), procedura pro načtení balení z dodacích listů (*PROCEDURE_FOR_READ_PACKAGES*) a seznam komparátorů řešení (*SOLUTION_COMPARATORS*).

Listing 3.22: Scénář integračního testu, nastavení atributů

```
<jm:submodule code="loadContainerByPackages">
       <jm:attribute key="COLUMN_ORDER_DEFINITION"
          value="TestUODefinition"/>
       <jm:attribute key="COLUMN_ORDER_STATE" value="Created"/>
       <jm:attribute key="PROCEDURE_FOR_READ_PACKAGES"
          value="STANDARD/ReadPackagesByProductQuantityOnOrders.java"/>
       <jm:attribute key="SOLUTION_COMPARATORS"
          value="BINS_COUNT,BASE_AREA,VOLUME"/>
</jm:submodule>
```

Poté jsou zadány vstupy (viz listing 3.23), které byly předem připraveny a jsou použity pro více scénářů. Prvním vstupem je číslo příkazu dodávky (*orderNumber*), dalším je seznam dodacích listů (*ordersForBatch*) a nakonec rozměry kontejneru (*width*, *depth* a *height*).

Listing 3.23: Scénář integračního testu, vstupy

```
<jm:fvBefore>
        <jm:fvEntry field="orderNumber"
            value="${universalOrderNumber}">
                <jm:attribute key="DEFINITION_NAME"
                    value="TestUODefinition"/>
        </jm:fvEntry>
        <jm:fvEntry field="ordersForBatch"
            value="${universalOrdersIDs[0]},
            ${universalOrdersIDs[1]}">
                <jm:attribute key="DEFINITION_NAME"
                    value="TestUODefinition"/>
        </jm:fvEntry>

        <jm:fvEntry field="width" value="2.5"/>
        <jm:fvEntry field="depth" value="2.0"/>
        <jm:fvEntry field="height" value="1.2"/>
</jm:fvBefore>
```

Validaci výstupů můžeme vidět v listingu 3.24. Nejprve jsou zkontrolovány stejné hodnoty jako na vstupu modulu, zda nejsou smazané nebo změněné. Dále se ověří existence identifikátoru transakce (*TRANSACTION_ID*), neboť se každý zápis do databáze pomocí transakce v systému DCIx eviduje. Následně se provede validace seznamu vynechaných řádek (*SKIPPED_ORDER_LINES_IDS*).

Listing 3.24: Scénář integračního testu, výstupy transakce

```
<jm:fvAfter>
        <jm:fvEntry field="width" value="2.5"/>
        <jm:fvEntry field="depth" value="2.0"/>
        <jm:fvEntry field="height" value="1.2"/>
        <jm:fvEntry field="orderNumber"
            value="${universalOrderNumber}" hasObject="true"/>
        <jm:fvEntry field="ordersForBatch"
            value="[${universalOrdersNumbers[0]},
            ${universalOrdersNumbers[1]}]" hasObject="true"/>
        <jm:fvEntry field="TRANSACTION_ID"
            enumClass="cz.aimtec.dci.transaction.xp
            .TransactionHiddenFieldDescriptor"
            isNotForReport="true"/>
</jm:fvAfter>
```

```
<jm:hvAfter>
                <jm:fvEntry field="SKIPPED_ORDER_LINES_IDS"
                    value="[${universalOrderID + 2}]"
                    enumClass="cz.aimtec.dci.transaction.xp
                    .TransactionHiddenFieldDescriptor"
                    hasObject="true"/>
</jm:hvAfter>
```

Poslední částí je kontrola dat v databázi, ověření, že se vytvořily ložní sloupce a že jsou navázány na dodávku (vis listing 3.25). Také jsou zkontrolovány odpovídající stavy a definice příkazu, které byly získány z nastavení modulu.

Listing 3.25: Scénář integračního testu, výstupy DB

```
<jm:readAndValidateDataFromDB functionId="Zkontroluj ze bylo
    vytvoreno 22 loznich ůsloupc a jsou navazany na dodavku"
                tableName="dciowner.Generic_Orders"
                rowsExpected="22">
                <jm:simpleParam name="Original_Order_ID"
                    value="${universalOrderID}" type="ID" />
                <jm:simpleParam name="Status" value="E"
                    type="STRING" />
                <jm:simpleParam name="State" value="Created"
                    type="STRING" />
                <jm:simpleParam name="Order_Definition_Name"
                    value="TestUODefinition" type="STRING" />
</jm:readAndValidateDataFromDB>
```

4 Závěr

Účelem této práce bylo vybrat nejvhodnější algoritmus pro řešení BPP, tuto metodu popsat a poté implementovat do systému DCIx.

Podle informací získaných z literatury byl jako nejvhodnější algoritmus vybrán GASP, který v reálných časových podmínkách podává nejlepší výsledky. Metody TSPACK a GLS sice podávaly lepší výsledky, avšak k jejich dosažení potřebovaly 60x větší čas než metoda GASP.

Specifikaci požadavků na implementaci nakládky v systému DCIx byla věnována velká pozornost. Jednou z nejdůležitějších částí byla identifikace stávajících objektů, které byly použity jako vstupy a výstupy algoritmu. Při tomtTo výběru musela být zohledněna především návaznost na další procesy.

Metoda GASP byla úspěšně začleněna do systému, a to jako transakční modul. Tím je umožněna velká flexibilita procesu v rámci kterého bude algoritmus využit. Během implementace byly také vytvořeny automatické testy, které usnadní budoucí úpravu kódu, neboť hlídají chyby, které by touto úpravou mohly nastat.

Literatura

(2006). Spec cpu2006. http://www.spec.org/cpu2006/results/.

(2008). Jameleon an automated testing tool. http://jameleon.sourceforge.net/.

(2013). Enum types. http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

(2013). Junit test framework. https://github.com/junit-team/junit/wiki.

(2013). Logistický informační systém dcix. http://www.aimtec.cz/cs/produkty/dcix.html.

(2013). Spatial data. http://msdn.microsoft.com/cs-cz/library/bb964711(v=sql.105).aspx.

Ašenbrener, L. (2010). Upgrade testovacího frameworku jameleon. Master's thesis, Západočeská Univerzita.

Baldi, M., Perboli, G., and Tadei, R. (2011). The tree-dimensional knapsack problem with balancing constraint. CIRRELT-2011-51.

Beasley, J. E. (1985). An exact two-dimensional non-guillotine cutting. *Operations Research 33*, pages 49–64.

Berkey, J. O. and Wang, P. Y. (1987). Two dimensional finite bin packing algorithms. *Journal of the Operations Research Society 38*, pages 423–429.

Bortfeldt, A. and T.Winter (2009). A genetic algorhitm for the two-dimensional knapsack problem with rectangular pieces. *International Transactions in Operatioal Research 16*, pages 685–713.

Chung, F. K. R., Garey, M. R., and Johnson, D. S. (1982). On packing two-dimensional bins. *Journal of Algebraic and Discrete Methods 3*, pages 66–76.

Crainic, T. G., Perboli, G., and Tadei, R. (2008). Extreme point-based heuristics for three-dimensional bin packing. *INFORMS Journal on Computing 20*, pages 368–384.

Crainic, T. G., Perboli, G., and Tadei, R. (2009). A two-level tabu search for the three-dimensional bin packing problem. *European Journal of Operational Research 195*, page 744–760.

Crainic, T. G., Perboli, G., and Tadei, R. (2012a). A greedy adaptive search procedure for multi-dimensional multi-container packing problems. `https://www.cirrelt.ca/DocumentsTravail/CIRRELT-2012-10.pdf`.

Crainic, T. G., Perboli, G., and Tadei, R. (2012b). Recent advances in multi-dimensional packing problems. *New Technologies - Trends, Innovations and Research*, pages 91–110.

den Boef, E., Korst, J., S. Martello, S. P., and Vigo, D. (2005). Erratum to "the three-dimensional bin packing problem": Robot-packable and orthogonal variants of packing problems. *Operations Research 53*, pages 735–736.

Egeblad, J. and Pisinger, D. (2009). Heuristic approaches for the two- and three-dimensional bins packing problem. *INFORMS Computers & Operations Research 36*, pages 1026–1049.

Faroe, O., Pisinger, D., and Zachariasen, M. (2003). Guided local search for the three-dimensional bin packing problem. *INFORMS Journal on Computing 15(3)*, page 267–283.

Fekete, S. P. and Schepers, J. (1997). A new exact algorithm for general orthogonal d-dimensional knapsack problems. *Springer Lecture Notes in Computer Science 1284*, page 144–156.

Fekete, S. P. and Schepers, J. (2004). A combinatorial characterization of higher-dimensional orthogonal packing,. *Mathematics of Operations Research 29(2)*, page 353–368.

George, J. A. and Robinson, D. F. (1980). A heuristic for packing boxes into a container. *INFORMS Computers & Operations Research 7*, pages 147–156.

Gilmore, P. C. and Gomory, R. E. (1965). Multistage cutting problems of two and more dimensions. *Operations Research 13*, pages 94–119.

Lodi, A., Martello, S., and Vigo, D. (1999). Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal onCOmputing 11*, pages 345–357.

Martello, S., Pisinger, S., and Vigo, D. (2000). The three-dimensional bin packing problem. *Operations Research 48*, pages 256–267.

Martello, S. and Toth, P. (1990). *Knapsack Problems - Algorithms and computer implementations*. Wiley-Interscience series in discrete mathematics and optimization. John Wiley & Sons.

Peng, J. and Zhang, B. (2012). Knapsack problem with uncertain weights and values. `http://www.orsc.edu.cn/online/120422.pdf`.

Perboli, G., Crainic, T. G., and Tadei, R. (2011). An efficient metaheuristic for multi-dimensional multi-container packing. *Proceedings of seventh annual IEEE Conference on Automation Science and Engineering (IEEE CASE 2011)*, pages 1–6.

Pisinger, D. (2002). Heurisctics for the container loading problem. *European Journal of the Operational Research 141*, pages 382–392.

A Seznam zkratek

BPP Bin Packing Problem 3

w_i šířka předmětu 2

h_i výška předmětu 2

d_i hloubka předmětu 2

W šířka kontejnerů 2

H výška kontejnerů 2

D hloubka kontejnerů 2

b_{max} maximální počet kontejnerů 2

NP Nedeterministicky Polynomiální 1

EB extrémní body 9

TS Tabu Search 10

HA Height first - Area second 3

BaB Branch-and-Bound 3

GLS Guided Local Search 3

BFD Best Fit Decreasing 3

C-EPBFD Crainic's Extreme Points Best Fit Decreasing 12

C-EPFFD Crainic's Extreme Points First Fit Decreasing 18

GASP Greedy Adaptive Search Procedure 3

RFID Radio Frequency Identification 23

B Seznam pojmů

Reporting Services Nástroj na zobrazování dat z databáze, za pomocí tabulek, formulářů, grafů apod.

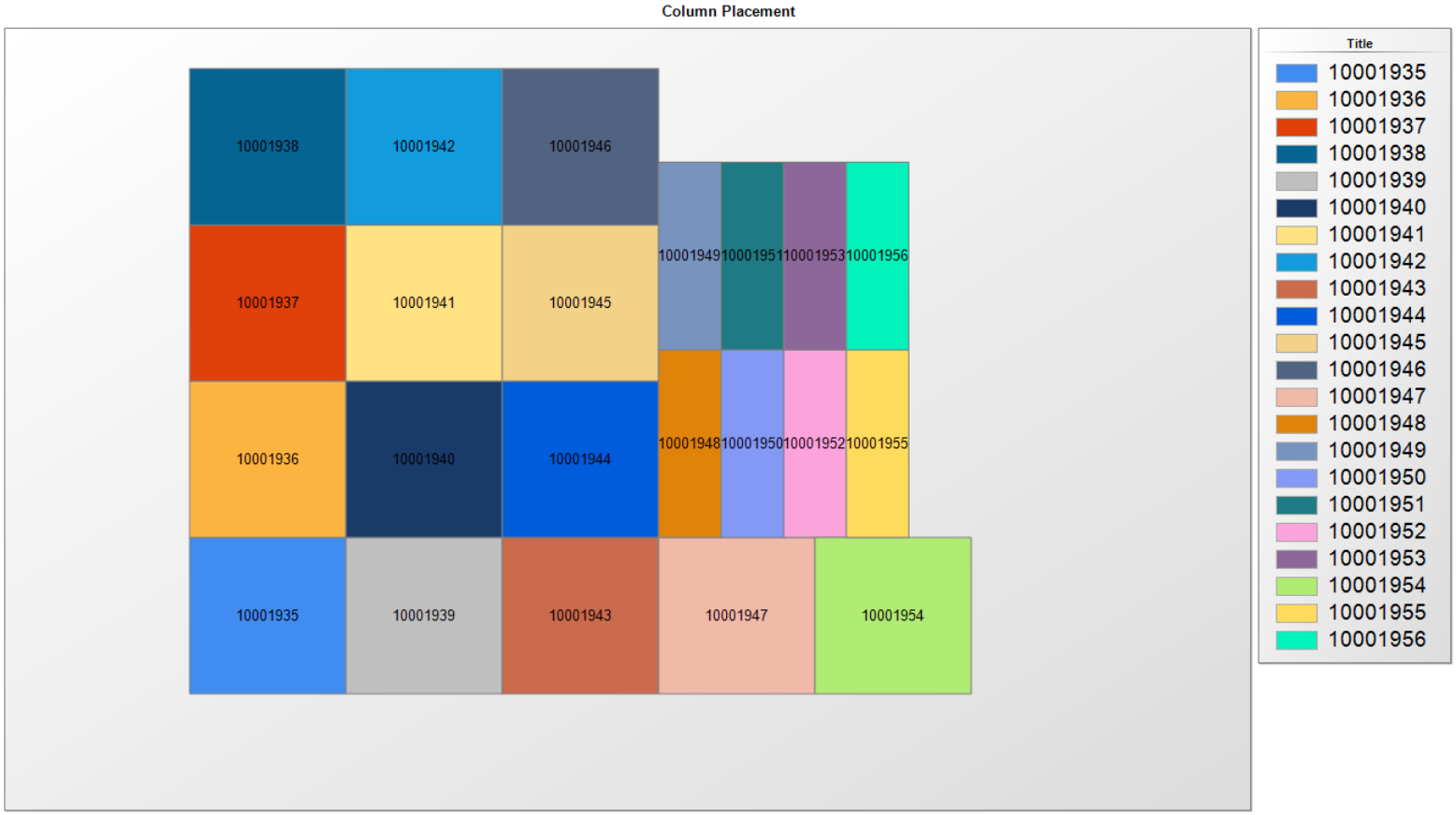
FIFO First In First Out - strategie vyskladňování balení, podle doby jejich naskladnění, tedy položka, která byla první naskladněna, je také první vyskladněna

FEFO First Expiration First Out - strategie vyskladňování balení, podle doby jejich expirace, tedy položka, která bude první expirovat, je první vyskladněna

Java enum výčtový datový typ, jednotlivé hodnoty ve výčtu mohou mít definované i chování (metody)

Java procedura framework vyvinutý a používaný v DCIx více v kapitole 3.4.1

C Zobrazení ložních sloupců



Obrázek C.1: Zobrazení ložních sloupců pomocí Reporting Services