Problém rozvrhování lidských zdrojů

Ing. Jan Smejkal, FEL ČVUT

Obsah

1 Rešerše na téma rozvrhování lidských zdrojů 3
1.1 Úvod . . . 3
1.2 Definice problému . . . 3
1.3 Rešerše možných řešení problému . . . 4
1.3.1 Tabu search . . . 4
1.3.2 Large neighborhood search . . . 5
1.3.3 Hyper–heuristika . . . 6
1.3.4 Heuristika založená na komponentách . . . 7
1.3.5 Simulované žíhání . . . 7
1.3.6 Řešení pomocí LP nebo ILP . . . 7
1.3.7 Column Generation . . . 8
1.3.8 Volba počáteční instance . . . 8
1.3.9 Zhodnocení řešerše . . . 9

2 Návrh algoritmu na základě předchozí rešerše 10
2.1 Scatter Search . . . 10
2.2 Scatter Search hyper–heuristika . . . 10
2.3 Návrh algoritmu . . . 10
2.3.1 Reprezentace a tvorba heuristik . . . 10
2.3.2 Lokální prohledávání . . . 12
2.3.3 Popis navrženého algoritmu . . . 12
2.4 Možná zdokonalení algoritmu . . . 13

3 Objektový model algoritmu 16

4 Zdroje 18

1 Rešerše na téma rozvrhování lidských zdrojů

Klíčová slova: timetabling, emloyee timetabling, rostering, large neighborhood search heuristic, multicriterial optimization, hyper heuristics, column generation

Výstup: Definice řešeného problému. Popis jednotlivých nalezených algoritmů, posouzení jejich vhodnosti k řešení stanoveného problému.

Popis: Provést rešerši algoritmů řešících zadaný problém.

1.1 Úvod

V minulosti vznikla řada systémů a postupů, které pomáhají rozvrhovat směny mezi zaměstnance tak, aby byl snížen čas a rozpočet potřebný na plánování. Zároveň je u nich kladen důraz také na uspokojení osobních požadavků jednotlivých zaměstnanců. Problém rozvrhování pracovních směn představuje již několik desítek let zkoumaný NP–úplný problém.

Tato rešerše se zabývá vyhledáním univerzálního postupu pro řešení podmnožiny rozvrhovacích problémů, které se zdají být komplikovanější než ostatní. Jedná se o rozvrhování směn ve společnostech, kde je zapotřebí lidská práce 24 hodin denně, 7 dní v týdnu, a to po celý rok. Navíc se zaměříme pouze na problémy, ve kterých je potřeba rozvrhnout zaměstnance z velkého počtu pracovních pozic do řádově několika desítek až stovek směn. Následuje definice problému, rešerše jednotlivých metod k jeho řešení a shrnutí spojené s výběrem nejvhodnějších metod.

1.2 Definice problému

Problém rozvrhování lidských zdrojů je definován v mnoha textech. Například [5] uvádí obecně formulovaný nurse rostering problem. Článek [12], ze kterého následující část textu vychází, zavádí problém rozvrhování lidských zdrojů obecně.

Problém je založen na 4 entitách: zaměstnancích, směnách, úkolech a pracovištích. Zaměstnanci jsou schopni na základě své kvalifikace vykonávat pouze některé úkoly. Ty provádí na různých pracovištích ve směnách. V rámci rozvrhování se uvažuje krátké opakující se období, například jeden týden nebo měsíc. Existuje celá řada nestandardních směn. Mezi ně patří víkendové, noční, sváteční a další. Dále jsou stanovena silná a slabá omezení, tedy ta, která musí být dodržena například kvůli zákonům a interním předpisům a ta, které popisují osobní preference zaměstnanců. Řešením je rozvrh, přiřazení zaměstnanců ke směnám, úkolům a pracovním místům, které splňuje všechna silná omezení a co nejvíce vyhovuje slabým omezením.

Článek [12] definuje tato silná omezení:

- **Zaplnění:** V každé směně je dostatečný počet pracovníků na každý úkol na každém pracovišti.
- **Schopnost:** Každý zaměstnanec má kvalifikaci k provádění jemu přiřazených úkolů.

- **Dostupnost:** Každý ze zaměstnanců je dostupný pouze na určitou podmnožinu všech směn.
- **Konflikty:** Žádnému ze zaměstnanců nesmí být přiřazen více než jeden úkol na směnu. Zaměstnanci nesmí pracovat ve dvou směnách, které jsou v konfliktu, tedy například v těch, které se překrývají nebo jsou obě ve stejný den.
- **Pracovní zatížení:** Každý ze zaměstnanců má definovaný maximální počet směn, které může odpracovat za rozvrhované období, přesněji průměr, kterému by se měl výsledný rozvrh co nejvíce blížit.

Článek [12] dále definuje tato slabá omezení:

- **Preferované úkoly a směny:** Každý ze zaměstnanců může preferovat některé úkoly, například kvůli míře svých schopností nebo osobním důvodům. Navíc každý ze zaměstnanců dává přednost určitým typům směn.
- **Flexibilita pracovního zatížení:** V rámci více rozvrhovacích období by se průměrné pracovní vytížení mělo co nejvíce blížit hodnotě definované silným omezením.

Různé podmnožiny problému rozvrhování lidských zdrojů přidávají další omezení, která je specifikují.

1.3 Rešerše možných řešení problému

Jak uvádí [1], většina algoritmů a přístupů k řešení problému je založena na postupném prohledávání nebo ořezávání stavového prostoru všech možných rozvrhů, dokud není nalezeno rozvržení zaměstnanců do směn splňující stanovené požadavky na kvalitu.

Mezi nejčastěji používané postupy a systémy patří matematické programování, metody umělé inteligence, expertní nebo znalostní systémy nebo meta–heuristiky (například genetické algoritmy, simulované žíhání, tabu search, memetický algoritmus, large neighborhood search nebo heuristiky založené na hledání komponent). V posledních letech se navíc objevují metody heuristik matematického programování a hyper–heuristik. Při řešení reálných problémů je často pro dosažení lepších výsledků využito rovnou několika metod najednou.

Následující část textu popisuje základní principy jednotlivých metod, shrnuje, pro jaké instance problému jsou vhodné a jakých výsledků bylo pomocí těchto metod dosaženo.

1.3.1 Tabu search

Tabu search (TS) je jedna ze nejpoužívanějších metod prohledávání stavového prostoru problému aplikovatelná na velké množství kombinatorických problémů. Iterativně prochází sousedy aktuálního částečného řešení a vybírá z nich ty, pro které hodnotící funkce vrací nejlepší výsledek. Sousední řešení k původnímu je přitom získáno aplikováním jedné z předem definovaných transformací. TS navíc využívá seznam tabu

transformací. V něm jsou uloženy ty, které nesmí být použity k vytváření sousedů a procházení stavového prostoru. Seznam je aktualizován v každé iteraci tak, aby nedocházelo například k zacyklení. Jeho použití slouží také k opuštění lokálního optima a nalezení lepšího výsledku. K tomu jsou využity ještě další postupy, například částečné poškození a opětovné na náhodě založené sestavení řešení. Jak základní teorii, tak i veškeré varianty popisuje podrobněji [3]. Konkrétní způsob aplikace na rozvrhování lidských zdrojů uvádí [2] a [12].

Srovnání nalezených postupů: [2] uvádí, že při využití kombinace heuristik bylo pro problém z reálného prostředí vždy nalezeno řešení v čase do 45 minut. Není zde však uveden detailní popis dat využitých k testu. Program implementovaný pomocí [12] rozvrhl 100 zaměstnanců s 449 požadavky během řádově stovek sekund. Je zde však poznamenáno, že samotná realizace aplikace byla velmi složitá.

Výhody: TS je velmi variabilní, poskytuje velký prostor k vyjádření různých omezení a přání týkajících se rozvržení směn. Je známý už od roku 1986, tudíž je velmi dobře popsaný v mnoha článcích.

Nevýhody: U složitějších problémů je potřeba aplikovat celou řadu heuristik k nalezení výsledku. Implementace řešení použitelného na rozvrhovací problémy z reálného prostředí je velmi složitá.

1.3.2 Large neighborhood search

Jak uvádí [6], large neighborhood search (LNS) podobně jako TS prohledává sousedství aktuálního částečného řešení k nalezení lepšího výsledku. K tomu využívá opakované poškozování a opětovné sestavení počátečního řešení. Spadá do rodiny very large neighborhood search heuristik. Ty se vyznačují tím, že prohledávají větší sousedství částečného řešení a díky tomu, že vybírají z více možností, rychleji nalézají kvalitnější výsledek. Pomocí LNS je možné najít řešení mnoha problémů kombinatorické optimalizace. Samotná metoda se dále člení podle toho, jakým způsobem je určeno sousedství aktuálního řešení a jak se toto okolí prochází. Dále jsou uvedeny dvě modifikace metody, konkrétně variable neighborhood search (VNS) [4] a adaptive neightborhood search (ANS) [5].

VNS definuje tři typy sousedství podle jejich velikosti. Nejjednodušší z nich transformuje částečné řešení na sousední stav pomocí přesunu jedné ze směn na jiného zaměstnance tak, aby byla stále splněna všechna silná omezení. Druhý typ sousedství je definován pomocí několika transformací, které nijak nezlepšují celkovou kvalitu výsledku, ale pouze zvyšují naplnění některého ze slabých omezení. Třetí pak zahrnuje sousedy, kteří vznikli mohutnější transformací skládající se z přesunů celých dnů nebo množin směn.

ANS oproti tomu prochází stavový prostor pomocí intenzivního, průměrného a diverzifikačního prohledávání. Intenzivní prohledávání představuje TS heuristika hledající lokální optimum. Průměrné prohledávání oproti tomu vybírá pouze z náhodné podmnožiny možných sousedů. Tím se snaží přesunout hledání do jiné části prostoru. Diverzifikační prohledávání je založeno stejně jako druhá transformace u VNS na uspokojení slabých omezení.

Srovnání nalezených postupů: Konkrétní postup aplikace VNS z článku [4] byl otestován pouze na malé instanci o 20 zaměstnancích a 4 typech směn. Použitím průchodu sousedstvím druhého typu bylo dosaženo časové optimalizace z původních 30 na necelé 4 minuty. ANS v článku [4] byl oproti tomu konstruován za účelem nalezení postupu, který předčí dosud existující metody řešení rozvrhovacích problémů z reálného prostředí. Testy v mnoha ohledech ukazují dosažení tohoto cíle. Výsledky potvrzuje i článek [18], který na VRP dokazuje, že použití více typů okolí v podobné formě, jako je tomu u ANS, je vhodnější, než rozdělení okolí podle velikosti tak, jak je tomu při použití VNS. ANS se na základě těchto faktů zdá vhodnější.

Výhody: LNS je velmi variabilní, dává velký prostor k vyjádření rúzných omezení a přání týkajících se rozvržení směn. Nastavení různých prohledávacích mechanismů umožňuje optimalizovat metodu na konkrétní problém.

Nevýhody: Je poměrně obtížné vybrat jednotlivé prohledávací mechanismy tak, aby se dohromady doplňovaly a umožnily procházení celého stavového prostoru. Články [4] a [5] k tomu ale poskytují určitý návod.

1.3.3 Hyper–heuristika

Jak uvádí [7], zatímco heuristika je metoda, která počítá výsledek přímou manipulací s daty, hyper–heuristika využívá vyšší úrovně abstrakce. Operuje s celou řadou heuristik nižšího stupně a až ty pracují s daty. Získává tedy výsledek nepřímo. Jedná se o nadřazenou metodu, která dokáže pomocí správných heuristik vyřešil velké množství problémů kombinatorické optimalizace.

Článek [7] uvádí hyper–heuristiku založenou na principu TS. Jednotlivé metody spolu soutěží o to, jaká z nich bude použita na prohledávání stavového prostoru. Pokud je navíc některá z nich delší dobu neúspěšná, je na pár iterací přidána do seznamu tabu heuristik. Tím je dočasně zabráněno jejímu dalšímu použití. Je zde uvedeno také mnoho heuristik nižší úrovně pro nurse rostering problem a university timetabling problem.

Oproti tomu článek [15] se snaží nalézt metodu, která by byla při hledání úspěšnější než TS hyper–heuristika. Místo TS je použit Scatter Search, metoda, která využívá malé množiny nejlepších nalezených jedinců, jejich mutací a 10 heuristik nižší úrovně, které jsou na mutace aplikovány. Tím jsou nejlepší řešení obměňována, dokud není nalezeno řešení spňující přijímací kritérium. Článek metodu aplikuje na rozvrhování zkouškových termínů.

Srovnání nalezených postupů: Podle [7] je možné výběrem správných heuristik dosáhnout výborných výsledků, například univerzitní rozvrh s přibližně 400 kurzy a 10 místnostmi byl rozvržen pouze s využitím 1166 operací. Cílem článku [15] je nalezení hyper–heuristiky, která by tento již i tak velmi kvalitní výsledek předčila. Test, který ji srovnává mimo jiné i s TS hyper–heuristikou vychází pro Scatter Search nejlépe.

Výhody: Metoda poskytuje opět mnoho možností a velký prostor k vyjádření rúzných omezení a přání týkajících se rozvržení směn.

Nevýhody: Opět je poměrně obtížné vybrat jednotlivé heuristiky tak, aby se navzájem podporovaly a na ně aplikovaná hyper–heuristika vedla k dobrému výsledku. Článek [7] však udává několik základních pro nurse rostering problem.

1.3.4 Heuristika založená na komponentách

Článek [8] definuje metodu řešící rozvrhovací problémy založenou na hledání komponent. Rozvrh směn je rozdělen podle zaměstnanců na jednotlivé části. Ty jsou pak pomocí evoluční selekce a mutace vylepšovány, dokud nevznikne sada jedinců dohromady představující výsledný rozvrh. Samotná mutace je aplikována na komponenty, pro které vychází nejhorší výsledky hodnotící funkce. Náhodná mutace cenných jedinců zajišťuje posuny ve stavovém prostoru a jeho důkladnější prohledání.

Výhody: Z testů v [8] vychází, že na dostupných datech je tato metoda jak z hlediska ceny, tak i časové složitosti přibližně stejně kvalitní jako LNS a hyper–heuristika.

Nevýhody: Metoda prohledávání stavového prostoru pomocí mutací poskytuje menší možnost úpravy postupů pro problémy se specifičtějšími podmínkami na rozvržení práce mezi zaměstnance.

1.3.5 Simulované žíhání

Simulované žíhání vychází z genetických algoritmů. Na začátku procesu je vytvořeno jedno částečné řešení představující první generaci. Následně iterativně vznikají další, střídavě pomocí ohřívání a ochlazování. Ohřívání slouží k získání generace s mnoha potenciálními řešeními, které nemusí být příliš kvalitní. Cílem ochlazování je oproti tomu vytvoření generace obsahující pouze nejlepší známé jedince. Celková teplota je navíc postupně snižována. Díky tomu jsou oběma metodami vytvářeny generace o stále menším počtu jedinců. Tím je dosaženo jak prohledání celého stavového prostoru, tak i výběru nejlepšího řešení. Článek [10] uvádí, jak aplikovat simulované žíhání na problém rozvrhování.

Výhody: Výsledky testů zaznamenané v [10] ukazují, že metoda simulovaného žíhání dosahuje výkonnostně obdobných výsledků jako LNS nebo hyper–heuristika.

Nevýhody: Transformace, které vytváří potomky z jedinců předchozí generace, jsou poměrně specifické. Pro různé obdoby problému by musely být často upravovány. K využití obecných transformací jako swap nebo n–opt je instance řešení problému příliš komplikovaná.

1.3.6 Řešení pomocí LP nebo ILP

Článek [9] uvádí metodu formulující rozvrhovací problém s jeho silnými a slabými omezeními jako instanci binárního ILP. Ta je následně vyřešena pomocí metody branch and bound. Článek [11] popisuje jinou možnost formulace a doporučuje vyřešení instance pomocí komerčního solveru CPLEX nebo PBS.

Pokud bychom vycházeli z definice instance rozvrhovacího problému pomocí ILP, mohli bychom využít například postupů z článku [17]. Zde je použita hyper–heuristika, která pracuje s heuristikami řešícími SAT problém a pomocí nich hledá výsledek.

Srovnání nalezených postupů: Z hodnot naměřených během testování a zveřejněných v [9] a [11] se zdá, že řešení založené na využití solverů je výrazně rychlejší. Na obdobných testovacích datech skládajících se z přibližně 20 zaměstnanců a 5 typů směn je schopné

vypočítat výsledný rozvrh rámcově během několika minut. Vlastní implementace metody branch and bound této rychlosti nedosahuje. Jak uvádí [9], výpočet řešení trvá přibližně 40 minut.

Výhody: Metoda LP poskytuje velké vyjadřovací možnosti pro různá omezení směn a požadavky zaměstnanců.

Nevýhody: Popsané postupy se hodí spíše pro menší rozvhrhovací problémy. Hlavním důvodem je fakt, že s počtem zaměstnanců velmi rychle roste i počet podmínek instance LP. Z tohoto důvodu není k rozumnému řešení rozvrhovacího problému z reálního prostředí možné použít ani hyper-heuristiku založenou na SAT heuristikách.

1.3.7 Column Generation

Další variantu řešení problému rozvrhování představuje podle článku [15] Column Generation (CG). Jedná se o metodu, která snižuje časovou náročnost výpočtu rozvrhu postupným rozšiřováním a sestavováním celého problému. Je definován hlavní problém a podproblém. Hlavní problém obsahuje pouze nejdůležitější část podmínek celé instance. Po nalezení řešení splňujícího podmínky hlavního problému jsou do instance přidány také omezení podproblému. Vzniká tak nový hlavní problém, který je opět vyřešen a rozšířen. Nakonec je nalezen výsledný rozvrh. K výpočtu řešení částečného problému je přitom možné použít libovolnou jinou heuristiku nebo jejich kombinaci.

Článek [15] dále popisuje pokus, při kterém k nalezení postupných řešení využívá ILP a genetického algoritmu. Instance problému je postupně rozšiřována a řešení generováno pomocí genetického algoritmu. Pokud tento postup selže, je část instance vyřešena jako ILP metodou branch and bound. Následně je opět aplikován genetický algoritmus.

Článek [16] oproti tomu navrhuje jiný postup. Počáteční část problému se postupně rozšiřuje o další zaměstnance, směny a dny. Řešení je nalezeno pomocí relaxace na celočíselnost instance a následné nalezení nejlepšího celočíselného výsledku.

Srovnání nalezených postupů: Podle experimentů v článku [15] se podařilo s využitím implementace v jazyce C vyřešit instance o velikosti 27 až 130 zaměstnanců v čase do 30 minut. Článek [16] definuje testovací instanci podobné velikosti (86 zaměstnanců, 5 druhů směn, 28 dní, 7 pracovních pozic), která byla opět vyřešena řádově v desítkách minut. Přístup druhé metody založený na postupném přidávání zaměstnanců, směn a dnů se však zdá přirozenější a výhodnější.

Výhody: K nalezení postupných řešení je možné využít libovolnou heuristiku, tím je možné postup optimalizovat pro konkrétní problém.

Nevýhody: Je poměrně obtížné rozdělit počáteční problém na části podle jejich důležitosti.

1.3.8 Volba počáteční instance

U metod založených na postupném prohledávání stavového prostoru je pro nalezení kvalitního řešení velmi důležitý výchozí stav. Ten může být určen mnoha způsoby. Mezi nejběžnější patří náhodné vygenerování, použití rozvrhu z předchozího období a vytvoření počátečního rozvrhu pomocí hladového algoritmu. Pro každou metodu řešení

rozvrhovacích problémů je přitom nejvýhodnější jeden konkrétní postup. Například heuristika založená na koponentách [8] a hyper-heuristika [7] pracují nejlépe, pokud jako počáteční instanci použijeme náhodně vygenerovaný validní rozvrh. Oproti tomu ANS [5] a VNS [4] využívají k sestavení počátečního řešení hladový algoritmus.

Článek [18] navrhuje postup vytvoření počáteční instance pomocí ILP. Instance obsahující všechna silná omezení a slabá omezení jednoduchá na formulaci je částečně vyřešena pomocí SAT solveru. Výsledek je použit jako výchozí bod pro další hledání řešení. Článek popisuje konkrétní způsob, kdy je počáteční řešení spočítané pomocí CPLEXu vylepšováno VNS heuristikou. Testy ukazují, že pomocí popsané metody bylo dosaženo výrazně lepších výsledků než pomocí samotného VNS. Obecnost tohoto postupu zajišťuje podobné výsledky například i při použití společně s ANS nebo hyper–heuristikou.

1.3.9 Zhodnocení řešerše

LP je i přes svoji velkou obecnost z důvodu časové složitosti na řešení větších rozvrhovacích problémů nepoužitelné. Ostatní metody (TS, LNS, hyper–heuristika, heuristika založená na komponentách a simulované žíhání) mají až na rozdíly způsobené vlastní implementací téměř shodnou časovou náročnost. Z hlediska univerzality je méně použitelné simulované žíhání a metoda založená na komponentách. Obecnější jsou TS, obě dvě varianty LNS a Column Generation. Nejuniverzálnější je pak hyper–heuristika, která odděluje samotný algoritmus a heuristiky určené k vyřešení problému do dvou úrovní abstrakce.

Pokud bychom měli zhodnotit jednotlivé konkrétní metody popsané výše, mezi nejlepší na základě výsledků testů patří ANS a Scatter Search hyper–heuristika. Obě dvě metody jsou skoro stejně kvalitní. Scatter Search hyper–heuristika je jistě obecnější, ANS však tento nedostatek překonává lepšími výsledky testů, kterých dosahuje právě konkrétnějším zaměřením na množinu rozvrhovacích problémů a zdá se proto být nejlepší nalezenou metodou. Za pokus by jistě stálo i zdokonalení metody o postup, který je v článku [18] úspěšně aplikován na VNS. Počáteční řešení, ze kterého bude ANS vycházet, může být spočítáno pomocí ILP solveru.

2 Návrh algoritmu na základě předchozí rešerše

Výstup: Detailní popis navrženého řešení.
Popis: Důkladné studium vybraného algoritmu, postupů, které s ním souvisí. Zdůvodnění výběru. Popis návrhu případných uzpůsobení vzhledem ke konkrétnímu problému. Analýza problému z hlediska datové reprezentace a následně její návrh.

Následující část textu se zabývá na základě řešerše vybranou Scatter Search hyper–heuristikou. Nejdříve popisuje principy samotné heuristiky, následně definuje navržený algoritmus a vysvětluje jeho jednotlivé části.

2.1 Scatter Search

Jak uvádí článek [15], Scatter Search je metaheuristika procházející stavový prostor řešení, jejíž hlavní podstatou je ukládání malého množství referenčních řešení neboli takzvané referenční množiny. Nejdříve je vytvořena počáteční množina co nejvíce rozdílných řešení. Pomocí lokálního prohledávání a následné selekce nejlepších nalezených řešení je vytvořena počáteční referenční množina. Z té je náhodně vybrána podmnožina, její prvky jsou navzájem většinou po dvou zkombinovány a na každé z nových řešení je aplikováno lokální prohledávání. Pokud jsou nově nalezená řešení lepší, nahrazují méně kvalitní prvky v referenční množině. Postup výběru podmnožiny, kombinace prvků a lokálního prohledávání se stále opakuje, dokud není nalezeno dostatečně kvalitní řešení nebo se prvky referenční množiny již delší dobu nemění.

2.2 Scatter Search hyper–heuristika

Scatter Search hyper–heuristika využívá popsaný postup na vyšší úrovni abstrakce. Scatter Search slouží ke správě referenční množiny heuristik, které definují, jakým způsobem bude při lokálním prohledávání procházen stavový prostor. Samotné heuristiky jsou tvořeny posloupností heuristik nižší úrovně a jsou navzájem kombinovány pro nalezení lepších řešení. Heuristiky, které jsou pro lokální prohledávání nejvhodnější, jsou uloženy v referenční množině.

2.3 Návrh algoritmu

Následující podkapitola popisuje nejdříve jednotlivé dílčí části algoritmu. Dále obsahuje jak slovní popis celého navrženého algoritmu, tak i jeho vyjádření pomocí pseudokódu.

2.3.1 Reprezentace a tvorba heuristik

Heuristiky se skládají z heuristik nižší úrovně, které představují postupy pro procházení stavového prostoru během lokálního prohledávání. Kvalita výsledného rozvrhu záleží na jejich počtu a povaze.

V následujícím seznamu jsou uvedeny některé základní heuristiky nižší úrovně, je však možné navrhnout i další. Podstatou heuristik nižší úrovně je vždy vyhledání změny rozvrhu, která snižuje hodnotu kritéria. Změna přitom může být nalezena mnoha postupy. Ve většině případů nezbývá nic jiného než vybrat a vyzkoušet některé z možností. V určitých případech je ale možné aplikovat jiný postup. Například u heuristiky nižší úrovně č. 3 lze za tímto účelem využít algoritmu pro nalezení párování v bipartitním grafu mezi uzly představujícími směny a zaměstnance.

1. Vyber den s nejvyšší hodnotou kritéria. Najdi prohození úvazku dvou zaměstnanců v tento den, které snižuje hodnotu kritéria rozvrhu.

2. Vyber den s nejvyšší hodnotou kritéria. Najdi změnu úvazku zaměstnance v tento den, která snižuje hodnotu kritéria rozvrhu.

3. Vyber den s nejvyšší hodnotou kritéria. Najdi nové rozvržení směn, které snižuje hodnotu kritéria rozvrhu.

4. Vyber blok směn od pondělí do pátku s nejvyšší hodnotou kritéria. Najdi prohození úvazků mezi dvěma zaměstnanci ve vybraném bloku, které snižuje hodnotu kritéria.

5. Vyber víkend s nejvyšší hodnotou kritéria. Najdi spojení úvazků dvou zaměstnanců během vybraného víkendu, které přiřazením pouze jednomu snižuje hodnotu kritéria.

6. Vyber dva zaměstnance s nejvyšší hodnotou kritéria. Najdi prohození jejich úvazků v libovolném dni, které snižuje hodnotu kritéria.

7. Vyber dva libovolné zaměstnance. Najdi prohození jejich úvazků v libovolném dni, které snižuje hodnotu kritéria.

8. Vyber jednoho zaměstnance s nejvyšší hodnotou kritéria a jednoho libovolně. Najdi prohození jejich úvazků v libovolném dni, které snižuje hodnotu kritéria.

9. Vyber jednoho zaměstnance s nejvyšší hodnotou kritéria a jednoho libovolně. Najdi prohození jejich úvazku během libovolného víkendu, které snižuje hodnotu kritéria.

10. Vyber jednoho zaměstnance s nejvyšší hodnotou kritéria a jednoho libovolně. Najdi prohození jejich úvazku během libovolného bloku od pondělí do pátku, které snižuje hodnotu kritéria.

11. Vyber zaměstnance s nejvyšší hodnotou kritéria. Najdi změnu jeho úvazku v libovolném dni, které snižuje hodnotu kritéria.

Většina výše uvedených heuristik nehledá zlepšení v rámci celého rozvrhu, ale zaměřuje se pouze na oblast (den, zaměstnance nebo jejich skupinu), která nejméně odpovídá požadovanému výsledku a její hodnota kritéria je tudíž nejvyšší. Některé heuristiky nižší úrovně se od této oblasti více či méně odchylují a operují libovolně nad celým rozvrhem.

Tím může být nalezen zlepšující tah, který ostatní heuristiky nižší úrovně nejsou schopny odhalit.

Samotné heuristiky jsou pak tvořeny posloupností heuristik nižší úrovně a reprezentovány jejich číselnými hodnotami. Pokud bychom uvažovali například heuristiky o délce 5, byla by heuristika (1, 5, 4, 2, 9) tvořena heuristikami nižší úrovně, které v seznamu odpovídají uvedeným číslům. Nové heuristiky jsou tvořeny pomocí jednoduchého překřížení dvou už existujících heuristik. Počáteční heuristiky, se kterými algoritmus výpočet začíná, jsou vygenerovány náhodně.

2.3.2 Lokální prohledávání

S využitím heuristiky můžeme prohledávat okolní stavy řešení a hledat vhodnější rozvrhy. Hledání probíhá pomocí varianty steepest descent lokálního prohledávání. Na vstupní rozvrh jsou cyklicky aplikovány heuristiky nižší úrovně obsažené v heuristice. Pokud je heuristika nižší úrovně úspěšná, je původní rozvrh nahrazen novým. V případě, že po určitý počet kroků nedojde v dalšímu zlepšení rozvrhu, bylo dosaženo lokálního optima a hledání končí. Scatter Search hyper–heuristika si během výpočtů uchovává množinu nejlepších řešení. Heuristika je vždy pomocí lokálního prohledávání otestována na všech rozvrzích z této množiny. Výsledkem tohoto procesu jsou nové vylepšené varianty původních rozvrhů. Na základě výsledku lokálního prohledávání je nakonec heuristika ohodnocena. Nejjednodušším způsobem je určit kvalitu heuristiky jako součet rozdílu hodnot kritérií všech vstupních a výstupních rozvrhů.

2.3.3 Popis navrženého algoritmu

Hlavním vstupem algoritmu je soubor dat popisující parametry požadovaného rozvrhu, tedy počet zaměstnanců, typy směn, délka rozvrhovaného období, požadavky na pokrytí období směnami a podobně. Dalšími vstupy jsou parametry, které slouží k nastavení samotného algoritmu, určují například počet prvků v referenční množině nebo počet uchovávaných průběžných řešení. Výstupem je nejlepší nalezený rozvrh.

Algoritmus průběžný stav ukládá pomocí dvou již zníměných množin. Referenční množina obsahuje dosud nejúspěšnější nalezené heuristiky. Dále je využívána množina nejlepších řešení, ze které se vychází při lokálním prohledávání. Počáteční nastavení algoritmu spočívá v založení těchto dvou množin. Do množiny nejlepších výsledků je umístěn prázdný rozvrh, referenční množina zůstává prázdná.

Pro úspěšnou inicializaci musí být proveden první krok, který celý algoritmus připraví. Jsou náhodně vygenerovány heuristiky, jejichž délka a počet odpovídají vstupním parametrům. Každá z heuristik je využita k lokálnímu prohledávání nad prázdným rozvrhem. Odpovídající počet nejlepších nalezených řešení je uložen. Nakonec jsou do referenční množiny vybrány heuristiky, které dosáhly nejlepších výsledků v lokálním prohledávání.

Nyní už jsou datové struktury algoritmu naplněny výchozími daty a je možné přejít na hledání nejlepšího rozvrhu. To probíhá v cyklu, který skončí, pokud po předem určený počet iterací nebyl přidán žádný výsledek do množiny nejlepších řešení. V každém cyklu se pak aplikuje následující postup. Z referenční množiny je vybrána podmnožina. Z

této podmnožiny jsou postupně po dvou náhodně odebírány jednotlivé heuristiky. Z každé dvojice jsou překřížením vytvořeny nové heuristiky, která jsou postupně použity k lokálnímu prohledávání nad množinou nejlepších výsledků. Pokud je možné kvalitu množiny zvýšit pomocí nově nalezených výsledků, proběhne její aktualizace. V případě, že je nově vytvořená heuristika lepší než nejhůře ohodnocená heuristika v referenční množině, je původní heuristika zahozena a nová vložena do referenční množiny. Průchod hlavním cyklem algoritmu končí po využití všech nově vytvořených heuristik. Následně je algoritmus buď ukončen nebo je vybrána nová podmnožina a celý postup se opakuje.

2.4 Možná zdokonalení algoritmu

Vysoká úroveň abstrakce Scatter Search hyper–heuristiky umožňuje vkládání nových heuristik nižší úrovně, které mohou způsobit rychlejší průchod stavového prostoru směrem ke kvalitním rozvrhům. Dalším místem, kde by mohl být výše navržený algoritmus zdokonalen, je jeho počáteční nastavení. Pokud bychom místo prázdného rozvrhu použili například rozvrh vytvořený pomocí SAT solveru nebo hladového algoritmu, přiblížili bychom se rychleji lepším výsledkům. Výsledek algoritmu závisí také na způsobu tvorby nových heuristik, jednoduché překřížení se ale zdá postačující.

Algorithm 1 Pseudokód algoritmu založeného na Scatter Search hyper-heuristice

1. Vstup

   $schedulingPeriod$ ▷ Požadavky na výstupní rozvrh
   $numberOfInitHeuristics$ ▷ Počet heuristik k inicializaci
   $lengthOfHeuristic$ ▷ Počet heuristik nižší úrovně v jedné heuristice
   $numberOfSolutions$ ▷ Počet průběžných nejlepších nalezených řešení k uložení
   $sizeOfReferenceSet$ ▷ Velikost referenční množiny heuristik
   $maxIdleIterations$ ▷ Maximální počet neúspěšných iterací, po kterých algoritmus ukončen

2. Výstup

   $roster$ ▷ Výsledný nejlepší nalezený rozvrh

3. Počáteční nastavení algoritmu

   $solutions \leftarrow \{empty\ roster\}$ ▷ Množina nejlepších řešení obsahuje pouze prázdný rozvrh
   $referenceSet \leftarrow \{\}$ ▷ Referenční množina heuristik je zatím prázdná

4. Vytvoření počáteční množiny heuristik

   $initHeuristics \leftarrow \{\}$ ▷ Množina počátečních heuristik je zatím prázdná
   for $i = 0$ to $numberOfInitHeuristics$ **do**
   $newHeuristic \leftarrow \{\}$ ▷ Založení nové heuristiky
   for $j = 0$ to $lengthOfHeuristic$ **do**
   $newHeuristic[j] \leftarrow random\ low\ level\ heuristic$
   end for
   $initHeuristics[i] \leftarrow newHeuristic$ ▷ Uložení nově vytvořené heuristiky
   end for

5. Získání počátečních řešení

   $newSolutions \leftarrow \{\}$ ▷ Množina nových řešení
   for $i = 0$ to $numberOfInitHeuristics$ **do**
   $newSolutions \leftarrow newSolutions \cup localSearch(initHeuristics[i], solutions)$
   ▷ Prohledání stavového prostoru pomocí heuristiky, uložení nově nalezených rozvrhů
   end for
   $solutions \leftarrow numberOfSolutions\ best\ solutions\ from\ newSolutions$

6. Vytvoření referenční množiny heuristik

   $referenceSet \leftarrow sizeOfReferenceSet\ best\ heuristics\ from\ initHeuristics$

Algorithm 2 Pseudokód Scatter Search hyper-heuristiky (pokračování)

7 Nalezení nejvhodnějšího rozvrhu

    $idleIterations \leftarrow 0$
    while idleIterations < maxIdleIterations **do**
        $SubSet \leftarrow random\ subset\ of\ ReferenceSet$
        $newHeuristics \leftarrow makeCrossOvers(SubSet)$
        for all $newHeuristicinnewHeuristics$ **do**
            $betterSolutions \leftarrow localSearch(newHeuristic)$
            if betterSolutions not empty **then**
                $solutions \leftarrow updateSolutions(betterSolutions, solutions)$
                $worstHeuristic \leftarrow worst\ heuristic\ from\ referenceSet$
                if newHeuristic is better than worstHeuristic **then**
                    $replace\ worstHeuristic\ with\ newHeuristic\ in\ referenceSet$
                end if
                $idleIterations \leftarrow 0$
            else
                $idleIterations \leftarrow idleIterations + 1$
            end if
        end for
    end while
        return $best\ solution\ from\ solutions$

3 Objektový model algoritmu

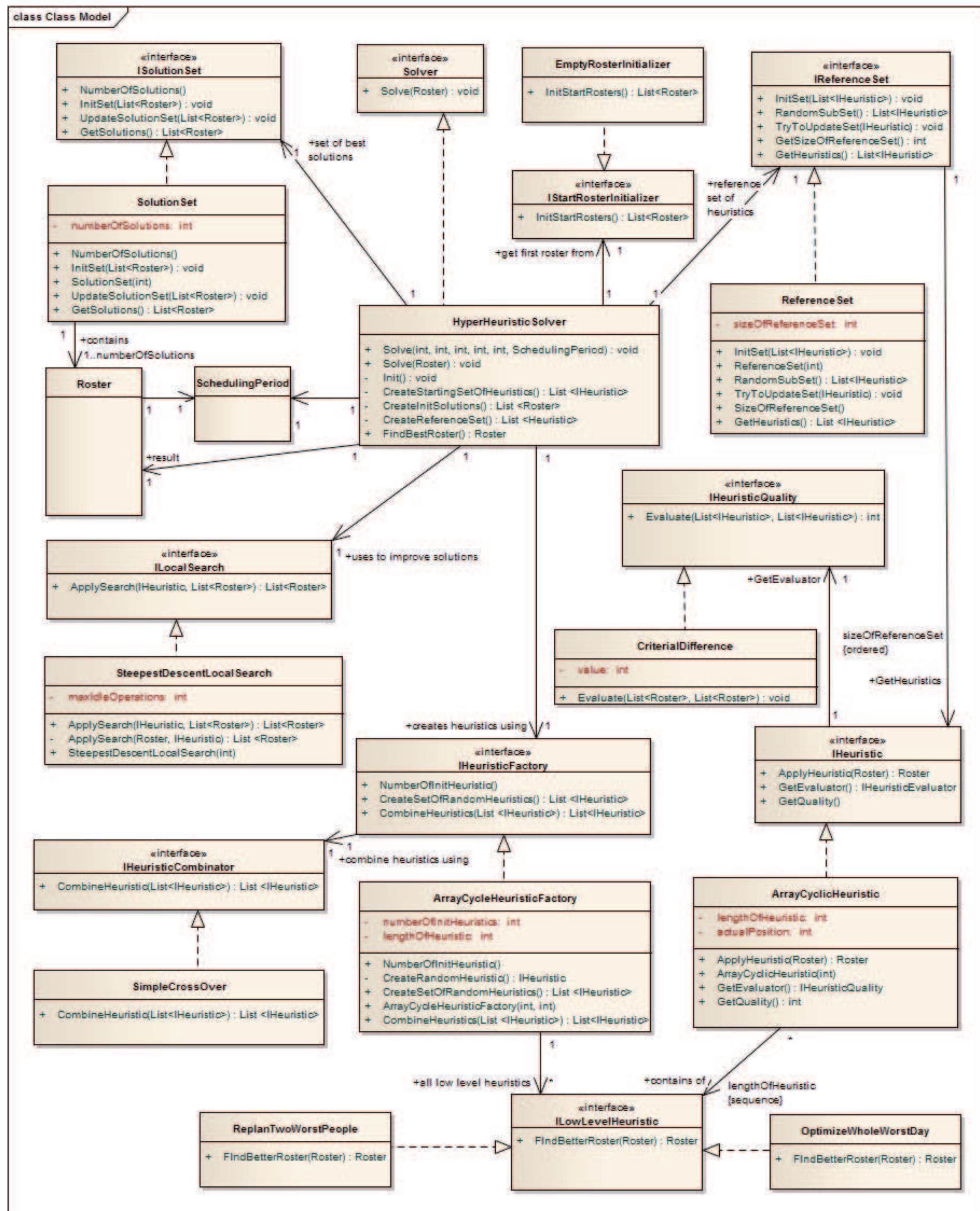
Objektový model algoritmu popisuje soustavu tříd a rozhraní vycházející z podoby Scatter Search algoritmu. Během návrhu struktury byl kladen důraz především na dosažení co největší modifikovatelnosti a rozšiřitelnosti algoritmu. Model zobrazuje objektový návrh na úrovni detailu odpovídající popisu algoritmu, neobsahuje žádné další detaily související s implementací.

Třídou, jejíž účelem je nalezení nejlepšího rozvrhu pomocí Scatter Search hyperheuristiky, je HyperHeuristicSolver. Tato třída dědí z rozhraní Solver z datové struktury problému a kromě metody Solve se skládá ještě z metod odpovídajících jednotlivým částem algoritmu. HyperHeuristicSolver obsahuje asociaci na Roster a SchedulingPeriod, další dvě třídy z datové struktury problému odpovídající rozvrhu a vstupní datové sadě.

Třída SolutionSet představuje množinu nejlepších dosud nalezených výsledků, váže se na ni maximálně vstupním parametrem daný počet instancí třídy Roster. Třída EmptyRosterInitializer slouží k získání počátečního prázdného rozvrhu. K rychlé změně inicializace pomocí jiných postupů, například vytvoření počátečního rozvrhu pomocí SAT solveru, stačí znovu implementovat rozhraní IStartRosterInitializer. Ze stejných důvodů model obsahuje i rozhraní ILocalSearch, které popisuje vlastnosti lokální prohledávání požadované algoritmem, aby v budoucnu bylo možné nahradit třídu SteepestDescentLocalSearch implementací jiného lokálního prohledávání.

K uložení referenční množiny heuristik a práci s ní je určeno rozhraní IReferenceSet a třída ReferenceSet. Referenční množina uchovává informaci o uložených heuristikách, tedy instancích implementujících rozhraní IHeuristic. Navržený algoritmus konkrétně pracuje s implementací ArrayCyclicHeuristic, využívající cyklicky procházené pole heuristik nižší úrovně. Výše zníměná heuristika se pak, jak je uvedeno i v návrhu algoritmu, skládá z posloupnosti heuristik nižší úrovně popsaných rozhraním ILowLevelHeuristic. Jeho implementace reprezentují jednotlivé heuristiky nižší úrovně. Každá heuristika zná své ohodnocení. Nový způsob hodnocení a porovnávání heuristik z hlediska kvality je možné realizovat pomocí nové implementace rozhraní IHeuristicQuality. Instance implementací rozhraní IHeuristic jsou vytvářeny pomocí implementací rozhraní IHeuristicFactory. K tvorbě instancí ArrayCyclicHeuristic a jejich jednoduchému překřížování je zapotřebí ArrayCyclicHeuristicFactory. Z důvodu konstrukce heuristik vyšší úrovně z heuristik nižší úrovně obsahuje tato třída reference na všechny existující implementace rozhraní ILowLevelHeuristic.

Obrázek 1: Objektový model algoritmu



4 Zdroje

1. Jingpeng Li, Edmund K. Burke, Tim Curtois, Sanja Petrovic, Rong Qu, The falling tide algorithm: A new multi-objective approach for complex workforce scheduling, Omega, Volume 40, Issue 3, June 2012, Pages 283-293, ISSN 0305-0483, 10.1016/j.omega.2011.05.004. (http://www.sciencedirect.com/science/article/pii/S0305048311000697)

2. Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, A Hybrid Tabu Search Algorithm for the Nurse Rostering Problem, Proceedings of the Second Asia-Pacific Conference on Simulated Evolution and Learning, 1998, Pages 187-194, Springer (http://ingenieur.kahosl.be/Vakgroep/it/publications/SEAL98.pdf)

3. Michel Gendreau, An introduction to tabu search, 2002, (http://opim.wharton.upenn.edu/~sok/papers/g/Gendreau_ANINTRODUCTIONTOTABUSEARCH.pdf)

4. Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, Sanja Petrovic, Variable neighborhood search for nurse rostering problems, Metaheuristics, 2004, Pages 153-172, Kluwer Academic Publishers (http://dl.acm.org/citation.cfm?id=982417)

5. Zhipeng Lü, Jin-Kao Hao, Adaptive neighborhood search for nurse rostering, European Journal of Operational Research, 2012, Pages 865-876 (http://smart.hust.edu.cn/admin/papers/1336449003.pdf)

6. David Pisinger, Stefan Ropke, Large neighborhood search (http://www.diku.dk/~sropke/Papers/lns.pdf)

7. Edmund K. Burke, G. Kendall, E. Soubeig, A Tabu-Search Hyperheuristic for Timetabling and Rostering, 2003, (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.9539)

8. Jingpeng Li, Uwe Aickelin, Edmund K. Burke, A Component Based Heuristic Search Method with Evolutionary Eliminations for Hospital Personnel Scheduling (http://dl.acm.org/citation.cfm?id=982417)

9. Briggite Jaumard, Frédéric Semet, Tsevi Vovor, A generalized linear programming model for nurse scheduling, European Journal of Operational Research, 1998, Pages 1-18 (http://www.sciencedirect.com/science/article/pii/S0377221797003305)

10. Mohammed Hadwan, Masri Ayob, A Constructive Shift Patterns Approach with Simulated Annealing for Nurse Rostering Problem, Information Technology (ITSim), 2010 International Symposium, 2010, Pages 1-6 (http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5561304&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D5561304)

11. Fadi Aloul, Bashar Al-Rawi, Anas Al-Farra, Basel Al-Roh, A Constructive Shift Solving Employee Timetabling Problems Using Boolean Satisfiability (http://www.aloul.net/Papers/faloul_iit06_sch.pdf)

12. Marco Chiarandini, Andrea Schaerf, Fabio Tiozzo, Solving Employee timetabling problems with flexible workload using tabu search (http://www.diegm.uniud.it/satt/papers/ChST00.pdf)

13. André Gustavo dos Santos, Geraldo Robson Mateus, General hybrid column generation algorithm for crew scheduling problems using genetic algorithm, Proceeding CEC'09 Proceedings of the Eleventh conference on Congress on Evolutionary Computation, 2009, Pages 1799-1806, IEEE Press Piscataway (http://dl.acm.org/citation.cfm?id=1689836)

14. Andrew J Mason, Mark C Smith, A Nested Column Generator for solving Rostering Problems with Integer Programming (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.1832)

15. Nasser R. Sabar, Masri Ayob, Examination timetabling using scatter search hyper-heuristic, 2nd Conference on Data Mining and Optimization, 2009 (http://80.ieeexplore.ieee.org.dialog.cvut.cz/stamp/stamp.jsp?tp=&arnumber=5341899)

16. David Pisinger, Stefan Ropke, A general heuristic for vehicle routing problems, Computers and Operations Research, 2007, Pages 2403-2435 (http://www.sciencedirect.com/science/article/pii/S0305054805003023)

17. Mohamed Bader-El-Den, Riccardo Poli, Evolving Effective Incremental Solvers for SAT with a Hyper-Heuristic Framework Based on Genetic Programming, Genetic Programming Theory and Practice VI, 2009, Pages 1-16 (http://www.springerlink.com/content/v1ln03380286g059/)

18. Edmund K. Burke, Jingpeng Li, Rong Qu, A hybrid model of integer programming and variable neighbourhood search for highly-constrained nurse rostering problems, European Journal of Operational Research, 2010, Pages 484-493 (http://www.sciencedirect.com/science/article/pii/S0377221709005396)

Implementace NRRP na GPU

Jan Dvořák

I. Rešerše

V rámci studia dané problematiky a aplikace algoritmů na frameworku CUDA byly nastudovány dále detailněji popsané články. Pro implementaci na GPU můžeme články rozdělit dle předpokládaného modelu implementace:

- heterogenní:
  - Competitive nurse rostering and rerostering[1]
  - Bipartite Graph Matching Computation on GPU[9]
  - GPU Implementation of the Branch and Bound method for knapsack problems[11]
- homogenní:
  - A genetic algorithm approach to a nurse rerostering problem[3]
  - Solving a bi-objective nurse rerostering problem by using a utopic Pareto genetic heuristic[4]
  - Parallel implementation of flow and matching algorithms[10]

Z pohledu způsobů řešení daného problému můžeme zavést následující taxonomii:

- optimální:
  - An Integer Multicommodity Flow Model Applied to the Rerostering of Nurse Schedules[2]
  - An optimal network-based approach to scheduling and re-rostering continuous heterogeneous workforces [8]
  - Bipartite Graph Matching Computation on GPU[9]
  - GPU Implementation of the Branch and Bound method for knapsack problems[11]
  - Parallel implementation of flow and matching algorithms[10]
- heuristické:
  - Competitive nurse rostering and rerostering[1]
  - A genetic algorithm approach to a nurse rerostering problem[3]
  - Solving a bi-objective nurse rerostering problem by using a utopic Pareto genetic heuristic[4]
  - An evolutionary approach for the nurse rerostering problem[7]

A. *Competitive nurse rostering and rerostering[1]*

Článek autora Michaela Vincenta Chiaramontea hledí na danou problematiku z pohledu agentních technologií. Základem jeho algoritmu pro řešení přerozvrhování směn je algoritmus pro rozvrhování, který je jen lehce obměněn. Proto nejdříve ve zkrácené verzi popíšu algoritmus pro rozvrhování. Pro jeho správnou funkci je již z počátku potřeba iniciálního rozvrhu, ze kterého vychází a dále jej upravuje. Jak jsem již psal, vychází algoritmus z agentních technologií a tyto agenti se dělí na Broker Agent (BA), kteří zastupují roli jednotlivých sester a Auction Control Agent (ACA), který zastupuje roli centrálního ovládání aukcí. Celý algoritmus pracuje na principu simulace aukce, kdy jednotlivé sestry (BA) se snaží vyměnit si vzájemně služby tak, aby to bylo pro ně výhodnější, než-li je aktuální stav. Každý BA si uchovává rozvrh pro danou sestru, ze kterého odvozuje give list (seznam služeb, které by bylo vhodné předat) a take list (seznam služeb, které by bylo vhodné převzít). Autor pracuje jen se dvěma typy služeb: denní a noční. Tyto služby jsou dále spojovány do shluků o maximálně třech službách. Tyto shluky je možné nabízet do aukce (směnit s ostatními sestrami). Z aktuálního rozvrhu je vypočtena užitková funkce, kterou se snaží algoritmus maximalizovat. V úvahu se berou preference volného dne v týdnu, předem požadovaných volných dní a preferovaného počtu volných dní za sebou. Každý BA má také přiřazené číslo označující hodnost sestry (novic, normální sestra, vrchní sestra). Pro výpočet užitkové funkce se ještě započítávají postihy za on-off-on vzor služeb a za nevybalancovaný rozvrh. ACA agent se stará o průběh aukce a snaží se zajistit, aby nově vytvořené rozvrhy po úpravě byli platné.

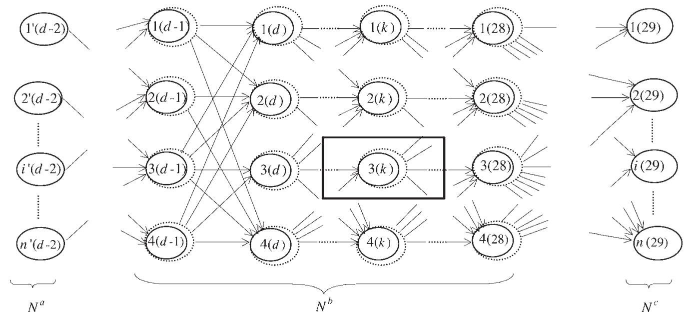
Na počátku Competitive Nurse Roasting (CNR) algoritmu ACA načte inicializační rozvrh a pro každou sestru vytvoří jeden BA a předá mu její rozvrh. Poté si každý BA vytvoří give list a take list a zinicializuje si vlastní struktury. Následně ACA cyklicky vyzývá BA, aby vytvořily nabídku výměny směny. Daná nabídka obsahuje Sale item, což je shluk směn, které se snaží BA předat někomu jinému a Currency list, což je shluk směn, které je ochoten přijmout výměnou za tyto služby. Tuto nabídku ACA odešle ostatním BA a ty zpět posílají příhozy. Každý příhoz obsahuje nabídku shluku směn, které chce daný BA za nabízené služby vyměnit. Přitom nabízí ty shluky, které jsou pro něj co nejvýhodnější a naplňují požadavky. Poté jsou příhozy seřazeny v klesajícím pořadí dle užitkové funkce upraveného rozvrhu nabízejícího BA dle výměny. Cyklicky jsou ostatní BA poptáváni po příhozech, které více zvýší užitkovou funkci než je dosud nejvyšší příhoz. V případě, že již ostatní BA nemají co nabídnout a nebo by jejich nabídky překročily práh změny jejich vlastní užitkové funkce (pro CNR nastaven na 0), aukce končí. Po konvergenci je přezván příhoz s nejvyšším zlepšením užitkové funkce a provedena výměna služeb mezi BA. Algoritmus opakuje aukci tak dlouho, dokud již nelze tímto postupem zlepšit užitkové funkce BA či dokud se nepřekročí stanovený maximální počet iterací. ACA po celou dobu aukcí hlídá, aby transakce nevyvolali neplatnost rozvrhu. Daný algoritmus je dále zlepšen pomocí Iterative local search (ILS) algoritmu a spojení těchto dvou principů je pak označováno za CNR-ILS. ILS řeší problém, kdy se může rozvrh zlepšit nejen výměnnou služeb mezi dvěma sestrami (BA), ale také přesunutím služeb v rozvrhu jedné sestry. Tento případ nastává ve chvíli, kdy je sestra v daný den nadbytečná

a je možné si tedy službu z jednoho dne přesunout na den jiný. BA postupně prochází kombinace v give listu a take listu a snaží se najít takovou, pro kterou by po výměně byl rozvrh platný a co nejvíce by se mu zvýšila užitková funkce. Toto opakuje, dokud již neexistuje taková kombinace.

Competitive Nurse Reroasting (CNRR) využívá metodu pokročilých transakcí (AT - advanced trading). Tato metoda upravuje currency list z nabídek tak, že již neobsahuje jen shluky stejné délky jako je nabídka, ale jednotlivé směny, kterých je stejný počet jako směn ve shluku v nabídce. Je tedy možné nahradit službu ve shluku i službami z jiných dní, které ale nejsou přímo za sebou. Tím se stává CNRR flexibilnější co se aukcí týče a pro konvergenci jí stačí méně iterací, které jsou ale časové náročnější. Dále je upravena užitková funkce všech BA, která nyní penalizuje změny služeb. CNRR pak pracuje ve třech fázích. V první (nazvané Schedule improvement) je spuštěno CNR s AT a práh změny užitkové funkce pro příhozy je nastaven na 0. Pokud se tímto vyřeší problém, je řešení navráceno. Pokud ne, spustí se fáze nazvaná Impact Isolation, která zavolá CNR-ILS a snaží se problém vyřešit přerovnáním služeb sestry, která požaduje uvolnění. Pokud se tímto problém vyřeší, je řešení navráceno. V opačném případě přechází algoritmus do třetí fáze a tou je Impact minimalisation. Tato fáze je stejná s první, jen využívá metody linear rollback, která iterativně snižuje práh změny užitkové funkce pro příhozy. Tím se algoritmus snaží najít řešení s co nejmenším snížením užitkové funkce jednotlivých BA.

Algoritmus je díky svým multiagentním vlastnostem vhodný pro paralelní zpracování. Míra paralelizace však je závislá na počtu agentů, které v algoritmu vystupují. V našem případě se tedy jedná o počet sester + 1. Pro malé počty sester je tedy možná paralelizace minimální. Avšak je také nutno podotknout, že paralelizace v dané úloze dává smysl až v případě většího počtu sester. Problém v paralelním zpracování také může způsobovat ACA agent, který se může stát úzkým hrdlem, jelikož je centrem veškeré komunikace mezi BA a také pro svoji práci má větší pamět'ové nároky.

Z pohledu CUDA architektury je třeba rozdělit iterace do dvou fází - BA vytváří své příhozy a fáze kdy ACA zpracovává příhozy či nabídky od BA. Zatímco první fázi, kdy pracují BA odděleně je možné efektivně na CUDA paralelizovat, druhou fázi je třeba vykonávat na CPU, jelikož je pouze jediný ACA agent a lze předpokládat, že jeho vykonání na GPU by bylo pomalé. Tím ovšem naroste komunikace mezi GPU a CPU, jelikož je třeba přenášet v každé iteraci sale item a currency list na GPU a několikrát přenášet příhozy od BA z GPU na CPU. Problém v implementaci BA nastává při alokování paměti pro take a give list, která by měl být dynamická, jelikož pro každé BA se mohou velikosti těchto seznamů v průběhu měnit. Díky (v praktických případech) malému počtu BA (každý BA by bylo zastoupeno jedním vláknem) je vhodné pro zrychlení uložit rozvrhy jednotlivých BA do sdílené paměti. Algoritmus je vhodný spíše pro architektury MIMD, nežli pro SIMD. Z důvodů časté komunikace mezi CPU a GPU a relativně malému počtu vláken v paralelním zpracování nemusí tento algoritmus implementovaný na CUDA zaznamenat výrazné zrychlení.



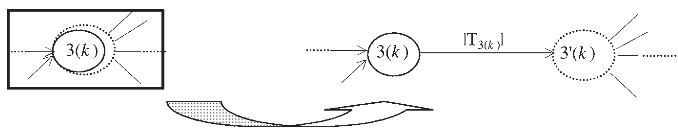
Obr. 1. Síť pro řešení NRRP[2]

B. An Integer Multicommodity Flow Model Applied to the Rerostering of Nurse Schedules[2]

Autoři Margarida Moz a Margarida Vaz Pato řeší problém přerozvrhování z pohledu multikomoditních toků. Jejich problém spočíval v přerozdělování směn, které se dělily na ranní, odpolední, noční a prázdný den v periodě 28 dní. Do požadovaného řešení byla zahrnuty i pevná a flexibilní omezení. Mezi pevná omezení byly zahrnuty například požadavky na minimální počet jednotlivých směn v daný den, zakázání určitých sekvencí směn či minimální počet volných dní za týden. Mezi flexibilní omezení byly zařazeny požadavky, aby sestry v co nejmenším měřítku měly dvě a více směn za sebou nočních, měly více než tři dny za sebou volných či více nežli tři dny za sebou pracovaly na ranní směně.

Problém je formulován pomocí multikomoditních toků. Graf sítě (Obr. 1) obsahuje $8 * (30 - d) + 2n$ uzlů, kde d je den v měsíci kde se vyskytuje první změna/absence a n je počet sester, které jsou k dispozici. Na začátku je n zdrojů toku, kde každý zdroj vytváří jednu komoditu, která zastupuje jednu sestru a na konci je n cílů komodit. Mezi těmito dvěma skupinami umělých uzlů je vytvořená síť, kde pro každý den je vytvořena osmice uzlů. Pro každou směnu v daný den jsou dva uzly (Obr. 2). Tyto dva uzly jsou propojeny hranou s omezením minimálního toku. Toto omezení vymezuje minimální počet sester, které se v daný den musejí dané směny účastnit. Síť začíná dnem $d - 1$ a končí dnem 28. Počet hran v síti je maximálně $20(28 - d) + 5n + 24$. Zdrojové uzly jsou pevně spojeny se službou v $d - 1$ den. Tento přístup povoluje změny nejdříve v den první absence. Dále jsou vytvořeny hrany pro každou možnou kombinaci směn vždy mezi daným a následujícím dnem. Každá dvojce uzlů jedné směny je propojena jednou hranou a nakonec jsou připojeny uzly cílů toku komodit. Takto vytvořená síť je poté inicializována parametry (omezení toku a cenou toku) v korespondenci s původním rozvrhem. Omezení je pro všechny hrany a všechny komodity (s výjimkou hran mezi dvěma uzly stejné služby v jeden den) nastaveno zespoda na 0 a z vrchu na 1. Tím při řešení celočíselného nejlevnějšího toku posloupnost hran, které mají nastavenu hodnotu toku dané komodity na 1, určí rozvrh pro danou sestru. Ceny hran v grafu odrážejí podobnost rozvrhu s původním grafem. Hrany, které korespondují s původním grafem mají tedy nastavenu cenu na 0, zatímco ostatní na 1.

Tímto přístupem je tedy namodelováno jádro problému.



Obr. 2. Dvojce uzlů jedné směny[2]

Není však možné takto jednoduše namodelovat i všechna omezení, které jsou k problému přidružena. Zbylá omezení jsou tedy přidána až do ILP formulace. Výše popsaná síť je tedy převedena do ILP formulace problému nejlevnějšího toku v multikomoditních sítích a následně obohacena o daná omezení. Detailní zápis této ILP formulace je v [2].

Díky převodu do ILP formulace a možnosti řešení problému pomocí například metody větví a mezí je tato formulace paralelizovatelná. Pro maximální délku rozvrhu (28 dní) a pro 19 sester se již v dané formulaci objevuje více nežli 10000 proměnných. Autor sám připouští, že i CPLEX Optimizer při řešení daného problému přesáhl i maximální využití paměti(128MB RAM) a proto nejsou všechny testovací instance vyhodnoceny. Pro paralelizaci na GPU by bylo tedy třeba tento problém vysokých paměťových nároků řešit.

Z pohledu implementace na CUDA se tedy tato metoda mění na implementaci ILP Solveru. Je možné využít paralelního zpracování Branch and Bound algoritmu, ale jelikož by optimalizace pro daný typ úlohy byla minimální, bylo by účelnější použít již odladěné implementace ILP Solverů.

C. A genetic algorithm approach to a nurse rerostering problem[3]

V článku An Integer Multicommodity Flow Model Applied to the Rerostering of Nurse Schedules [2] je metoda formulace NRRP pomocí multikomoditních toků srovnávána s heuristickou metodou. Z této metody vychází aktuální článek, který se tuto heuristickou metodu snaží křížit s genetickým programováním.

Heuristická metoda je založena na přerozvrhnutí celého rozvrhu se snahou co nejvýše se přiblížit původnímu. V první fázi algoritmus vezme všechny úlohy, které je třeba přiřadit a v náhodném pořadí je uloží do listu. Z tohoto listu poté postupně vybírá jednu úlohu za druhou, dokud se mu nepodaří všechny úlohy přiřadit nějaké sestře. Při přiřazování postupuje následujícím způsobem.

1) Nejprve se algoritmus pokusí přiřadit službu té sestře, která měla službu i v původním rozvrhu, pokud nejsou prolomena nějaká omezení či platnost rozvrhu.
2) Pokud se mu to nepodaří pokusí se službu přidělit jiné sestře, pro kterou by služba byla kompatibilní s předchozím i následujícím dnem.
3) Pokud se ani takovouto sestru nepovede nalézt, pokusí se algoritmus přiřadit službu sestře, pro kterou je služba kompatibilní aspoň s předešlým či následujícím dnem pokud nejsou prolomena žádná omezení.
4) V případě nepřiřazení, se algoritmus pokusí najít jakoukoli sestru, které by mohl úkol přiřadit.
5) Pokud se ani toto nepovede spustí se backtrackovací algoritmus, který hledá takovou pozici, kdy by bylo ještě možné službu přiřadit.
6) Pokud takovouto pozici nenalezne je rozvrh označen za nepřerozvrhnutelný.
7) V opačném případě službu na dané místo přiřadí a znovu pokračuje stejným postupem dokud nejsou všechny služby přiřazeny nebo není nalezena služba kterou není možno přiřadit.

Z daného rozvrhu je pak vypočtena užitková funkce.

Díky náhodnému seřazení úloh (případně i sester) může opětovné volání heuristické metody navrátit různý rozvrh v závislosti na tomto pořadí. Je tedy možné danou heuristickou metodu zavolat vícekrát, čímž se zvýší prohledávaný prostor a z výsledků vybrat ten nejlepší. Při opětovném volání a náhodném inicializování struktur se ale prochází prostor řešení nahodile. Autoři se tedy snaží najít takový postup, aby byly počáteční struktury(seznam úloh a sester) inicializovány tak, aby byla co největší pravděpodobnost nalezení optimálního rozvrhu. K tomuto je využito genetických algoritmů. V první inicializační fázi je první populační generace zinicializována náhodně. Každý jedinec se skládá ze dvou chromozomů. Prvním je uspořádaný seznam úloh a druhým uspořádaný seznam sester. Pro každého jedince v populaci je poté spočtena jeho užitková funkce pomocí výše uvedené heuristické metody, kdy je přeskočena první inicializační fáze a v metodě jsou použity chromozomy jedince. Po vyhodnocení přichází fáze selekce, kdy jsou vybrány jedinci pro křížení do další generace. Pravděpodobnost výběru jedince pro křížení je přímo úměrná jeho užitkové funkci. Pro následné křížení bylo využito více metod (dvě dvoubodové a jedna jednobodová), jelikož dané seznamy jsou formulovány jako permutace a není tedy možné využít jednoduchého křížení. Poté ve fázi mutace byli s určitou malou pravděpodobností v chromozomu přehozeny dvě služby/sestry v pořadí. V nové generaci se vždy objevil i dosud nejlepší jedinec z předchozích generací. Pro novou generaci byla opět vyhodnocena jejich užitková funkce. V případě, že nebylo možné vytvořit rozvrh, byla k užitkové funkci přičtena penalizace. Celý proces genetického vývoje byl zastaven po určitém počtu generací či pokud již po určitý počet generací nebylo zaregistrováno zlepšení. Jelikož poměrně často uvízl algoritmus v lokálním optimu aniž by nalezl jakékoli přípustné řešení, byla zavedena technika hybridace, kdy pokud již po n generací nebyl nalezen jedinec s přípustným řešením, byl v nové generaci skřížený jedinec nahrazen náhodně vygenerovaným jedincem (tím se algoritmus přiblížil metodě opětovného volání heuristické metody, která na rozdíl od genetického algoritmu nemá sklon k uváznutí v lokálním optimu).

Dle článku byla testována metoda opětovného spouštění genetického algoritmu a samotného heuristického algoritmu s ekvivalentní časovou dotací a za daných podmínek byl výsledek genetického algoritmu vždy lepší či ekvivalentní samotné heuristické metodě. Metoda opětovného spouštění samotného heuristického algoritmu je naprosto vhodná pro paralelizaci, kdy v každém vlákně může probíhat jeden běh tohoto algoritmu. Paměťová náročnost tohoto řešení je také skoro minimální možná. Však z článku vyplývá, že řešení

pomocí genetického algoritmu na testovacích datech dávalo v 10% případů lepší výsledky. Genetický algoritmus je však také možno paralelizovat.

Při použití CUDA frameworku by původní rozvrh byl uložen ve sdílené paměti. Jeden kernel by sloužil k vyhodnocování konstruktivní heuristiky. Autoři v článku doporučují v každé generaci kolem 400 jedinců. Každý jedinec by byl zastoupen jedním vláknem. Výrazný problém v zrychlení bude způsobovat backtrackovací algoritmus, který bude ve warpu vytvářet divergentní větve. Tohoto problému se však nedá v daném algoritmu zbavit. Chromozomy a výsledný rozvrh, který jedinec vytváří bude třeba uložit do globální paměti stejně jako výslednou hodnotu užitkové funkce. Poté by v kernelu nastala selekce a křížení. K tomu by byla využita metoda double bufferingu, kdy by byla alokována dvě místa pro chromozomy (pro aktuální a předešlou populaci) a mezi nimi by se přepínalo přepínačem ve sdílené paměti (jelikož by byl pro všechny jedince v danou iteraci stejný, nedocházelo by k divergentním větvím). Poté by nastala mutace a algoritmus by přešel do další iterace. Vše by se tedy dělo bez komunikace s CPU.

D. Solving a bi-objective nurse rerostering problem by using a utopic Pareto genetic heuristic[4]

Aktuální článek navazuje na článek [3] a dále rozvíjí možnosti genetických algoritmů. Aktuálně se zabývá optimalizací přes více flexibilních omezení [6], [5]. V článku jsou uvedena tato tři omezení:

- Určité sestry by neměly pracovat více nočních služeb za sebou
- Sestrám by měl být v dané periodě přiřazen předem stanovený počet služeb
- Nově přerozvržené služby by měly co nejvíce odpovídat původnímu rozvrhu

První omezení článek dále považuje za pevné omezení a k optimalizaci se snaží pohlížet z dvou zbývajících hledisek. Algoritmus nejdříve zinicializuje první populaci. Toto provede algoritmem z [3] (popsaným výše). Tento algoritmus najde populaci, která se blíží k optimu z hlediska třetího omezení. Dále tím samým algoritmem s relaxací na všechny pevné omezení s výjimkou omezení na přiřazení pouze jedné služby jedné sestře na den nalezne utopického jedince výběrem nejlepšího z poslední generace. Tento jedinec bude pravděpodobně zástupcem neplatného rozvrhu a bude mít optimistickou užitkovou funkci. Spojením utopického jedince a výše zmíněné populace vznikne první generace pro nový genetický algoritmus. Každý jedinec pak vyhodnotí své dvě užitkové funkce (užitková funkce pro druhé a třetí flexibilní omezení). Tyto dvě hodnoty pak vytvoří prostor, ve kterém se hledají postupně řady nedominantních jedinců, které jsou postupně ohodnocovány Pareto indexem od hodnoty 1. Výsledná užitková funkce je rovna 1/Pareto index. Utopický jedinec je vždy ohodnocen Pareto indexem 1. Takto ohodnocení jedinci jsou poté vybírány ke křížení s pravděpodobností přímo úměrnou jejich ohodnocení (metodou rulety). Ke křížení je využit PMX (viz. [13]) operátor. Následně je provedena mutace, která s pravděpodobností 0,1% vybere jedince a přehodí mu pořadí dvou služeb/sester v chromozomu. Tím je vytvořena nová generace. Do nové populace se také vždy dostává utopický jedinec a spolu s ním i dva nejlepší jedinci z předchozí generace. Tím je zabráněno zapomínání nejlepších řešení.

Algoritmus je podobný tomu z článku [3] a tedy i tento algoritmus by mohl být vhodný pro paralelizaci. Na implementaci je obtížnější než-li předchozí algoritmus, jelikož je třeba implementovat více genetických algoritmů stejně jako více konstruktivních heuristik. V případě více-hleďiskové optimalizace je však toto řešení, v korespondenci s článkem, výhodnější než vážení hledisek pro vytvoření jediné užitkové funkce pro minimalizaci.

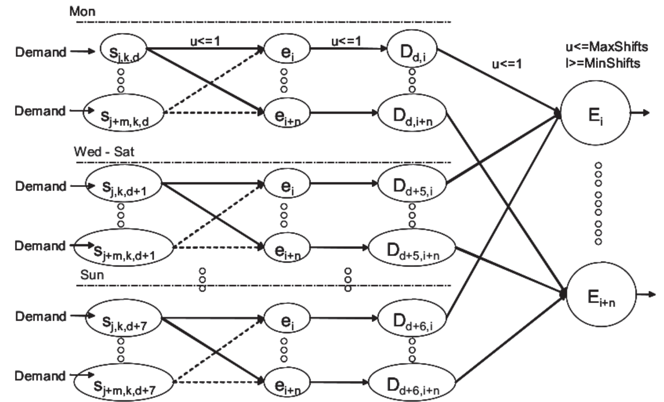
E. An evolutionary approach for the nurse rerostering problem[7]

Autoři vytvářejí opět genetický algoritmus pro řešení NRRP problému, vycházejí však z genetických algoritmů, které jsou osvědčeny pro rozvrhování. Omezení rozdělují na požadavky obsazení (minimální počet sester, které jsou potřeba na obsazení dané služby a daný den), s časem související omezení (sestře může být přiřazena v jeden den maximálně jedna služba, minimální čas pro odpočinek sester...) a omezení spojená se změnou rozvrhu (v den kdy sestra nemůže nastoupit do služby nemůže být sestře přiřazena služba). Algoritmus se za daných podmínek snaží minimalizovat tři hlediska: počet změn v rozvrhu (penalizace za odebrání a za přiřazení služby), antipreference sester (index určující ke každé službě každý den, jak nerada by sestra danou službu měla), nespravedlivost (rozdíl počtu služeb, které má sestra, oproti průměrnému počtu).

Algoritmus pracuje na principu evolučního/genetického algoritmu. Na rozdíl od principu, který používá Maz a Pato [3], [4] zde chromozom není tvořen náhodně seřazenou posloupností služeb/sester, ale přímo rozvrhem (nejčastější forma chromozomu v úloze rozvrhování). Ve fázi inicializace je vytvořena množina jedinců pomocí konstruktivního heuristického algoritmu. Z jedinců je vytvořena pareto množina a z té je poté vytvořena první populace pro genetický algoritmus. Při selekci jsou náhodně vybírány (s ohledem na jejich fitness funkce) dvojce jedinců. Při selekci je také použita metoda pro rozprostřený výběr, aby pro křížení nebyli vybráni dva velice si blízcí jedinci. Ke křížení je využit složitý algoritmus, který pro svojí práci využívá heuristický kombinační algoritmus, tabu search algoritmus a algoritmus pro řešení minimálního toku v sítích. Při křížení bere v ohled všechna hlediska nadepsaná výše. Po křížení nastává mutace, která je v případě, že k mutaci nastane, řešena přiřazením náhodného rozvrhu sestře a následným řešením minimálního toku v sítích. Po fázi mutace je jedinec přiveden do fáze vylepšení, kde pomocí lokálního prohledávání se algoritmus snaží rozvrh ještě vylepšit. K tomuto jsou použity tři algoritmy. První se snaží najít lokální optimum v rozvrhu jedné sestry, bere-li rozvrhy ostatních sester za fixní. Druhý se snaží najít lokální optimum v daný den a rozvrhy v ostatní dny považuje za fixní a poslední algoritmus hledá lokální optimum v dané sekci rozvrhu (prohazuje služby mezi sestrami ve vymezeném

časovém období). Po provedení všech těchto fází je vytvořena nová populace. Novou populaci opět tvoří jen jedinci, kteří v ní tvoří pareto množinu.

Algoritmus dle článku je univerzálnější než-li algoritmus autorů Moz a Pato [4]. Velký rozdíl ve výkonnosti algoritmu se projevuje v případě určení minimálního počtu souvislých dní, kdy musí sestry být ve službě. Za toto však algoritmus platí svojí složitostí a robustností, jelikož již ve svém základu obsahuje algoritmy pro minimální cenu toku, tabu search a další. Z pohledu paralelizace je tento algoritmus možné použít, avšak paměťová náročnost algoritmu není v článku uvedena a díky robustnosti se dá předpokládat, že bude neúnosně velká.



Obr. 3. Návrh sítě dle [8]

F. An optimal network-based approach to scheduling and re-rostering continuous heterogeneous workforces [8]

Autor tohoto článku se nezaměřuje pouze na problém přerozvrhování služeb sester, ale spíše na generický problém přiřazování služeb. Z tohoto pohledu tedy vzniká formulace daného problému na bázi problému přiřazování modelovaného pomocí problému minimalizace ceny toku v sítích (v článku maximalizace preferencí přiřazení). Článek je rozdělen do tří hlavních sekcí, kdy v první je uveden základní model a jeho využití pro rozvrhování směn technického dozoru v počítačových laboratořích. Návrh sítě je zaznamenán na Obr 3, kde $S_{j,k,d}$ znamená službu j, která vyžaduje zaměstnance ze skupiny k a je v den d. e_i označuje zaměstnance i, $D_{d,i}$ je celkový počet služeb zaměstnance i ve dni d a E_i vyjadřuje počet služeb zaměstnance i za celý týden. V této síti jsou pak označovány hrany minimálním/maximálním tokem a cenou toku. Cena toku mezi uzly s a e odpovídá preferenci přiřazení dané služby danému zaměstnanci/sestře. Pokud zaměstnanec nemůže službu vykonávat hrana mezi ním a službou neexistuje (má nulový maximální tok). Ve všech ostatních případech mají hrany mezi s a e nulový minimální a jedničkový maximální tok. Součet toku přes všechny hrany mezi danou službou a všemi zaměstnanci musí odpovídat minimálnímu počtu požadovaných zaměstnanců na danou službu. Maximální tok mezi e a D odpovídá počtu služeb, které může zaměstnanec v jeden den vykonat. Tento parametr se hodí v případě proměnných délek služeb, aby zaměstnanec odpracoval za den přes všechny služby maximálně daný počet hodin. V případě rozvrhování/přerozvrhování sester je tento parametr nadbytečný. Minimální/maximální tok mezi $D_{d,i}$ a E_i odpovídá minimálnímu/maximálnímu počtu služeb, které má daný zaměstnanec vykonat za týden. Daná síť na obrázku je tedy navržena pro rozvrhování pro období jednoho týdne. Je však možné ji jednoduše rozšířit do delší časový interval a případně další funkcionalitu (maximální počet směn o víkendu) atd. Autor pro rozvrhování jako cenu/preferenci volil součet preference zaměstnance vůči dané službě v součtu s preferencí zaměstnance (udané managementem firmy/nemocnice). Hlavní rozdíl mezi algoritmem pro rozvrhování a přerozvrhování spočívá v nastavení hodnot v preferencích (ze kterých je poté počítána užitková funkce). Pro přerozvrhování je k dané preferenci (vázané ke vztahu mezi zaměstnancem a službou) ještě připočítávána konstantní hodnota M v případě, že tato služba byla v původním rozvrhu danému zaměstnanci přidělena. Tím při maximalizaci ceny/preferencí algoritmus upřednostňuje rozvržení služeb, které již byly v původním rozvrhu použity.

Výše popsaná formulace poskytuje poměrně flexibilní framework pro rozvrhování, není však možné do něj zakomponovat podmínky pro vzájemné vyloučení dvou služeb (noční služba následovaná ranní). Pro tyto účely jsou vytvořené rozšiřující podmínky a daná síť je tedy formulována pomocí LP úlohy. S přidanými omezeními se již nejedná o čistou úlohu minimalizace ceny toku a tedy nemusí ani platit teorém o celočíselnosti řešení. Ve většině případů však výsledný tok celočíselný je. Při testování bylo zjištěno, že u velkých úloh sice není výsledný tok celočíselný, ale jen malý zlomek proměnných tuto celočíselnost porušuje. Autor pro řešení těchto úloh doporučuje CPLEX interior-point metodu, která vyřeší LP úlohu. V případě neceločíselnosti proměnných se nepoužije metoda větví a mezí, ale pouze Gomoryho řezů, která v relativně krátké době nalezne již celočíselné řešení.

Dle prováděných testů je algoritmus sám o sobě velice rychlý a většinu praktických úloh vyřeší do sekundy. V případě rozvrhování sester však nijak neřeší problémy s on-off-on vzorem služeb a ani se spravedlivostí rozdělení služeb (tento problém je obcházen přiřazením preferencí od managementu). Bereme li v úvahu autorovo doporučení používat k řešení CPLEX inter-point metodu s následným řešením neceločíselnosti pomocí Gomoryho řezů, je algoritmus spíše předurčen k sekvenčnímu výpočtu a nezdá se být vhodným pro paralelní zpracování. Po odstranění omezení, kvůli kterým je třeba využít LP formulaci, je možné využít paralelní algoritmy pro nejlevnější toky v sítích(Cost-Scaling algoritmus na GPU);

G. Bipartite Graph Matching Computation on GPU[9]

V článku autoři popisují přístup k řešení přiřazovacího problému v bipartitních grafech při výpočtech na GPU. Pro práci použily jako výchozí Maďarský algoritmus a Aukční algoritmus. Maďarský algoritmus byl ve výsledku použit jen pro srovnání v testovací fázi, jelikož vykazuje prvky sekvenčního chování a není proto vhodný pro paralelní zpracování na GPU. K tomuto zpracování byl tedy využit Aukční algoritmus. Bereme-li dvě části bipartitního grafu, první označíme jako osoby a druhé jako objekty. V aukčním

algoritmu se snaží osoby podávat nabídky za dané objekty. Ta nabídka která má nejvyšší příhoz (v případě článku se jedná o hodnotu rozdílu hodnoty (rozdíl benefitu osoby z objektu a jeho ceny) mezi nejlepším a druhým nejlepším objektem o který má osoba zájem) dostane objekt přidělen. Toto se děje ve dvou fázích. V první fázi (přihazování) podávají osoby své nabídky za objekty. V druhé fázi (přiřazování) objekty zkontrolují své příhozy a přiřadí se osobě s nejlepší nabídkou. Při přiřazení objekt zvýší svojí cenu dle nejvyšší nabídky. Algoritmus přechází mezi těmito dvěma fázemi do té doby, dokud nemají všechny osoby přiřazen nějaká objekt.

Při převádění daného algoritmu pro paralelní zpracování na GPU, jsou vytvořeny dva kernely - jedno pro první přihazovací fázi a druhé pro přiřazovací fázi. V inicializační fázi jsou vytvořeny proudy pro vstup, výstup i dočasné proměnné. Matice vah reprezentující graf je uložena jako 2D proud konstant. Ostatní proudy (ceny objektů, indexy objektů přiřazených osobám, indexy osob přiřazených objektům, nabídky osob a jejich cíle) jsou jednorozměrné. Po inicializaci se periodicky opakují oba kernely. V přihazovacím kernelu je každé osobě přiřazeno jedno vlákno a každá osoba pokud dosud nemá přiřazen objekt dá nabídku. Po té je spuštěn druhý kernel, ve kterém je každý objekt zastoupen jedním vláknem. Zde si objekty vybírají svého následujícího vlastníka podle jejích příhozu a upravuje svojí cenu a případně se uvolní od dosavadního vlastníka. Poté je řízení vráceno zpět CPU s hodnotou počtu přiřazených objektů. Pokud je počet přiřazených objektů roven počtu osob algoritmus končí. V opačném případě přechází algoritmus do další iterace.

V porovnání se sekvenčním provedením vykazuje algoritmus v průměru osminásobné zrychlení. Pro svůj výpočet potřebuje jen malý objem dat, který může být celý uložen v paměti zařízení (pro rozumně velké problémy). V každé iteraci je však řízení předáváno zpět CPU pro kontrolu ukončovací podmínky. Zpět je však posílána jen jediná hodnota a tím je minimalizováno zpoždění komunikace mezi CPU a GPU.

H. Parallel implementation of flow and matching algorithms[10]

Článek je rozdělen do dvou základních částí. První část se zaobírá řešením problému hledání maximálního toku v síti a druhá hledáním nejlevnějšího toku.

K řešení hledání maximálního toku se všeobecně používá Ford-Fulkersonům, Edmonds-Karpův a Goldbergův algoritmus. K paralelnímu zpracování je dle autorů nejpříznivější Goldbergův push-relabel algoritmus, jehož paralelní implementace je známa od roku 1992. Algoritmus pracuje na principu vtlačování toku do sítě tak, že tok může být vtlačen pouze do hran, které ještě mají volnou kapacitu a vedou do uzlů, které mají o jedna nižší výšku (hodnota přiřazená každému uzlu vyjadřující potenciál při "přelévání" toku - viz [10]) než-li aktuální uzel. V případě, že do uzlu přitéká více toku, než-li může z uzlu odtéci (není možno jeho přebytečný tok vtlačit do okolní sítě) je uzel přeznačen a je mu nastavena taková výška, aby v příští iteraci již existoval uzel, do kterého bude moci tok přesunout. Tak může uzel změnit výšku tak, aby se jeho přebytečný tok vrátil zpět do zdroje. Algoritmus iteruje do té doby, dokud je v síti nějaký aktivní uzel (uzel, který má vyšší přítok než-li odtok) kromě zdroje a cíle toku. Pro zrychlení daného algoritmu použily autoři ještě dvě heuristiky. Heuristika Global Relabelingu využívá BFS prohledávání směrem od zdroje toku, kdy každému uzlu přiřadí výšku rovnu vyšší hodnotě z vzdálenosti od cíle toku či jeho aktuální vzdálenosti. Tím zamezíme velkému počtu relabel operací, jelikož zvýšíme uzlu jeho výšku na minimální možnou aby se vůbec jeho tok mohl dostat do cíle. Heuristika Gap relabelingu odstraňuje ze sítě uzly, ze kterých se nikdy nebude moci dostat tok do cíle (nebudou moci nikdy uspokojit výškové omezení). K implementaci autoři použili Hongovu implementaci bez použití zámků. Tato implementace však vyžaduje atomické operace (pro sčítání a odčítání). Na rozdíl od základní verze tato verze při vtlačování toku nebere v úvahu residuální hrany s uzlem o jedna nižší výšky, ale residuální hranu s nejnižším uzlem. V použití na CUDA frameworku, inicializace a heuristiky jsou spouštěny na CPU zatímco push-relabel operace jsou spuštěny na GPU. Po určitém počtu iterací je řízení z GPU vráceno CPU, které provede pomocné heuristiky a poté vrátí řízení zpět GPU. Algoritmus končí ve chvíli, kdy všechen tok vytékající ze zdroje přitéká do cíle.

V implementaci na CUDA je pro každý uzel vytvořena datová struktura node, která obsahuje položky excess, height, toSourceOrSink (pointer na hranu vedoucí přímo k zdroji či cíli) a firstOutgoing (pointer na první hranu vycházející z uzlu). Pro reprezentaci hran je v implementaci struktura adj, které obsahuje položky vertex (uzel, do kterého hrana vede), flow (pointer na hodnotu zbývající kapacity), mate (pointer na opačně orientovanou hranu), next (pointer na následující hranu). Z GPU jsou do CPU zasílány informace o přebytcích toku v uzlech, výšce uzlů a zbývajících kapacitách. Zpět do GPU jsou z CPU v průběhu algoritmu zasílány jen informace o výškách (které mění heuristika).

Druhá část se zabývá řešením váženého přiřazovacího problému. Autoři tento problém převádějí na problém nejlevnějšího maximálního toku, který poté řeší pomocí cost scaling algoritmu. Algoritmus ke svému průběhu využívá ceny uzlů, které jsou při inicializaci všechny nastaveny na 0. Ve svém průběhu hledá ϵ-optimální řešení. Toto řešení se snaží cyklicky zlepšit. K tomuto účelu využívá metodu Refine. Metoda Refine zlepšuje ϵ-optimální řešení o koeficient α. Nejprve saturuje všechny přípustné hrany, čímž se ale z toku stane pseudotok a s tím, že některé uzly se stanou aktivními a některé budou mít záporný nadbytek toku. Pomocí operací push a relabel se z pseudotoku opět stane ϵ-optimální tok. Push operace je volána na uzly, které jsou aktivní a mají přípustnou hranu s nenaplněnou kapacitou. Operace Relabel je volána na uzly, kteří takovou hranu nemají a je jím tímto změněna cena. Metoda Refine je volána do té doby, dokud ϵ není menší než $1/n$.

Pro daný algoritmus jsou dále využity dvě heuristiky: Price Updating a Arc Fixing. V Price Updating heuristice je spočtena vzdálenost uzlů od nejbližšího uzlu s kladným úbytkem toku (záporným přebytkem) a od jeho ceny je poté odečten ϵ-násobek této vzdálenosti. Arc Fixing heuristika

vynechává z grafu hrany, jejichž redukovaná cena je větší než $2n\epsilon$.

Při vytváření paralelní verze tohoto algoritmu je zde popsaný algoritmus spojen s push-relabel algoritme bez zámků z předešlé části. Zde má opět každý uzel vlastní vlákno. Toto vlákno vykonává Refine kód do té doby, dokud se nezmění daný pseudotok na tok (dokud daný uzel má kladný nadbytek toku). Kvůli omezení GPU, které při velkých úlohách může nahlásit launch time out, je zde omezení na daný počet cyklů, po kterých se předá řízení zpět CPU a to ho bez změny okamžitě zase vrátí GPU. Pro každé vlákno je potřeba čtyř lokálních proměnných: e' pro uložení aktuálního nadbytku toku, y' pro uložení uzlu do kterého vede hrana s nejnižší redukovanou cenou a min_cp' a tmp_cp' pro nalezení tohoto uzlu. Zbývající proměnné jsou sdílené mezi vlákny.

Autor v článku neprovádí experimenty, které by poukazovaly na efektivnost paralelní implementace v ohledu k sekvenční implementaci. Řešení má relativně nízké nároky na paměť a celý výpočet probíhá na GPU. Pouze v případě přiblížení se časové hranici pro timeout je řízení předáno CPU, které po prvním předání řízení jednou zavolá výše popsané heuristiky.

I. GPU Implementation of the Branch and Bound method for knapsack problems [11]

Autoři v tomto článku představují hybridní implementaci metody větví a mezí specializovanou na řešení problému batohu na GPU. K prohledávání stavového prostoru pomocí větví a mezí využívají prohledávací strategii do šířky. V algoritmu se předpokládá předseřazení objektů v sestupném pořadí podle poměru ceny ku váze.

Algoritmus staví BFS strom řešení od objektu s nejvyšším poměrem ceny ku váze, kde ke každému uzlu je přiřazena pětice - váha ω, která je počítána jako součet vah objektů dosud přiřazených do batohu a všech následujících objektů v seřazeném, které v součtu nepřevýší váhu batohu; cena p, počítána podobně jako předchozí hodnota, ale z cen objektů; slacková proměnná s, které určuje index prvního objektu v seřazeném seznamu, který se již nevejde do batohu; U horní Dantzigova mez a L dolní mez. V každé úrovni se strom dělí na maximálně dva podstromy. Tato situace nastává pokud je index nově přiřazovaného objektu menší než slacková proměnná s jeho rodiče. Pokud tomu tak je vytvoří se k uzlu dva potomci - jeden pro přiřazení objektu do batohu a druhý, který naopak určuje, že do batohu nebude objekt zařazen (pro tento uzel je třeba přepočítat pětici). Uzel může mít jednoho potomka v případě, že index nově přidávaného objektu je roven slackové proměnné rodiče. Tento potomek pak určuje, že nebude nový objekt do batohu přidán (a hlouběji ve stromu ani žádný jiný). Uzel také nemusí mít žádného potomka a to v případě, že je prořezán - jeho horní mez U je nižší než nejlepší dosud známá dolní mez.

Samotný algoritmus je pak rozdělen do následujících části:

- CPU:
  - Výpočet metody větví a mezí pokud je počet přípustných uzlů malý
  - Přesun seznamu uzlů do paměti GPU
  - Spouštění jader na GPU
  - Nalezení tabulky označených uzlů
  - Přiřazení substituovaných adres v tabulce označených uzlů a její přesun do paměti GPU
- GPU:
  - Kernel pro větvení
  - Kernel pro výpočet mezí
  - Kernel pro výpočet nejlepší dolní meze
  - Kernel pro označování uzlů k prořezání a spojení seznamu uzlů po prořezání

Všechny objekty jsou uloženy v paměti textur a kapacita batohu je uložena v paměti konstant.

V kernelu větvení jsou z každého uzlu vytvořeny dva nové. Při implementaci je vytvořen pouze jeden nový uzel. Druhý uzel (díky své podobnosti s aktuálním) přepíše ten stávající. Jak již bylo psáno, pokud je index nového uzlu menší než s stávajícího, je vytvořen nový uzel, pro který je vypočteno ω, p a s a ponechán stávající (jelikož nový uzel bude mít stejnou pětici jako stávající). Je-li index roven s je také vytvořen nový uzel a je mu inkrementována hodnota s, zatímco stávající je označen za nepřípustný (U nastaveno na 0).

Kernel pro výpočet mezí se dělí do tří částí. První slouží k výpočtu nových ω, p, s a U. Jelikož je výpočet členitý, jsou zde na minimum omezeny přístupy to globální paměti uložením potřebných informací do registrů. Po vypočtení horní a dolní meze jsou jejich hodnoty uloženy do globální paměti pomocí atomických operátorů.

Kernel pro výpočet nejlepší dolní meze provádí redukci mezi dolními mezemi všech uzlů na aktuální hladině.

Kernel pro označení uzlů k prořezání vytvoří tabulku binárních hodnot, kde každý uzel je označen k prořezání (hodnota 0) pokud jeho horní mez U je nižší než nejlepší dosud známá dolní mez (již není možné rozváděním tohoto uzlu získat lepší řešení než takové, které je při nejhorším nejlepší z jiného uzlu). Výsledná tabulka je poté přenesena zpět na CPU.

Kernel pro spojení uzlů po prořezání nahrazuje uzly, které jsou označeny k prořezání, uzly od konce seznamu tak, aby aktivní uzly byli v souvislém poli. Při každém nahrazení se zmenší počet aktivních uzlů o jedna. Tím je na konci jasné, kolik uzlů je aktivních a zbytek pole se ořízne a dále se s ním již nepracuje. Tím je vytvořena množina přípustných uzlů pro další větvení.

Na CPU je poté implementován algoritmus větví a mezí, který je využit v případě, že je malý počet aktivních uzlů (dle doporučení počet uzlů menší než 192) a paralelní zpracování by bylo pomalejší než zpracování na CPU. Dále je na CPU implementován algoritmus, který po navrácení tabulky označených uzlů k prořezání určí adresy uzlů, které je mají v poli nahradit. Tyto adresy jsou poté výchozí parametr pro poslední popsaný kernel.

J. Parallelization of Binary and Real-Coded Genetic Algorithms on CUDA[12]

Článek se věnuje implementaci generického genetického algoritmu na framework CUDA a udává doporučení pro implementaci jednotlivých části genetického algoritmu

(GA). Autoři implementovali verzi algoritmu pro jedince reprezentované reálným vektorem a binárním vektorem.

Důležitým faktorem z pohledu efektivnosti GA je dle autorů generátor náhodných čísel, jelikož je jeho použití velice časté a proto robustní generátory výrazně zpomalují chod GA. Jako vhodná metoda se jeví inicializace generátoru náhodnou hodnotou, kterou na počátku vygeneruje robustní generátor a následné použití Park-Millerova generátoru, který následující pseudonáhodnou hodnotu odvozuje jednoduchým matematickým vzorcem z hodnoty aktuální.

Autoři dále zmiňují možnost využití tzv. ostrovů, kde je vytvořeno několik autonomních ostrovů, kde každý ostrov nezávisle na ostatních má vlastní evoluci. Každý ostrov pak může dokonvergovat k jinému lokálnímu minimu. V této základní podobě se jedná o paralelní verzi vícenásobného volání genetického algoritmu. Algoritmus je však rozšířen o migraci mezi ostrovy, kdy z jednoho ostrovu se na jiný může přepravit určitý maximální počet jedinců (v praktickém použití se jedná a jedince s nejlepší užitkovou funkcí). Díky tomuto mechanizmu se ostrovy navzájem kladně ovlivňují a je částečně omezena konvergence k jednomu lokálnímu optimu.

II. Specifikace problému

Ve zdravotnických centrech (nemocnicích) musí být poskytována zdravotní péče 24 hodin denně. V našem případě budeme předpokládat, že tato péče je zprostředkovávána zdravotními sestrami. Pro zajištění této 24 hodinové péče je rozdělena pracovní doba do 3 směn - ranní, odpolední, noční. Každá taková směna musí být zajištěna určitým minimálním počtem služeb, přičemž jedna služba může být přiřazena pouze jedné sestře. Ze zákona může jedna sestra vykonávat pouze jednu službu denně. Kromě tohoto omezení jsou zakázány i další kombinace služeb, které by porušovaly zákoník práce. V našem případě se jedná například o následování noční směny směnou ranní. Kdyby k tomuto došlo, sloužila by jedna sestra souvisle 16 hodin, což je nepřípustné. Při rozvrhování je třeba vycházet z požadavků sester, které mohou mít v daném rozvrhovaném období dovolenou a tedy je nepřípustné těmto sestrám ve stanovené dny přiřazovat služby. Kromě daných omezení je třeba rozlišovat klasifikaci sester, jelikož služba může mít požadavky na klasifikaci sestry, který ji má vykonávat.

Při přerozvrhování vycházíme z již známého rozvrhu, který ale je třeba upravit z důvodů vnějšího narušení - např. při nutné změně služby pro danou sestru či absenci pro nemoc. V tu chvíli je třeba vytvořit nový rozvrh od aktuálního dne až po konec rozvrhovaného období. Při vytváření nového rozvrhu se snažíme vytvořit takový rozvrh, který by byl co nejpodobnější tomu původnímu (minimalizovat počet změn), jelikož každá změna v rozvrhu může negativně zasahovat do osobního života sester, které již mohou mít na dané období v daný den změny jiné plány. Při přerozvrhování je navíc třeba brát v úvahu stejně jako při rozvrhování právní předpisy a požadavky sester, které neumožní v daný den dané sestře změnu. Náš problém je omezen i maximálním počtem změn a tedy nový rozvrh oproti starému se může lišit maximálně v 16-cti službách.

III. Návrh modelu

A. *Paměťový model*

Pro správnou funkci algoritmu je třeba mít v paměti uloženy objekty, se kterými budeme pracovat. Za základní objekty budeme považovat:

- Setříděný seznam služeb (L_s)
- *Setříděný seznam zdravotních sester* (L_n)
- Původní/výchozí rozvrh (R_0)
- Binární matice klasifikace sester K_n
- Pole změn vůči výchozímu rozvrhu (C_r)
- Maska pokrytí aktuálního rozvrhu (M_r)
- Nejlepší dosud nalezený rozvrh (R^*)
- Užitková funkce R^* ($z(R^*)$)
- *Setříděný seznam služeb pro* R^*
- Zámek pro přístup k R^*

objekty používané v rozšíření základního algoritmu jsou nadepsány kurzívou

V následujícím textu předpokládáme pro analýzu paměťových nároků maximální možnou mez při přerozvrhování počet:

- sester - $2^5 = 32$
- dní - $2^5 = 32$
- typů služeb - $2^2 = 4$
- změn v rozvrhu = $2^4 = 16$

Z důvodů univerzálnosti bereme možné i maximální obsazení rozvrhu - 1024 služeb.

Tabulka I ukazuje v přehledu velikosti jednotlivých struktur, které jsou třeba pro běh algoritmu. Minimální velikost struktur, které je možné dosáhnout v případě, že by byla veškerá data uložená v binárním poli a z tohoto pole v případě použití parsována vyjadřuje parametr Velikost binární representace. Velikost typové representace naopak určuje velikost těchto objektů v případě, že by jednotlivé položky byli uloženy jako standardní typy (int, short...). Tato velikost je ve většině případů větší. Za tuto cenu však získáváme navýšení výkonu, jelikož data není třeba parseovat a jsou vždy zarovnaná. Velikosti struktur jsou vztaženy k jednomu běhu heuristiky. V paralelním zpracování bude tedy třeba vytvořit strukturu o udávané velikosti pro každé vlákno.

TABULKA I
TABULKA VELIKOSTÍ STRUKTUR

Objekt	Velikost	
	binární reprezentace	**typové reprezentace**
Směna	2b	2B
Služba	10b	2B
Seznam L_s	1280B	2kB
Pole C_r	320b	64B
Maska M_r	128B	2kB
Rozvrh R_0	256B	2kB
Matice K_n	4kB	64kB
Rozvrh R^*	320b	64B

Službou je myšlen identifikátor, který přesně popisuje jednu určitou službu z původního rozvrhu, kterou je třeba rozvrhnout i do nového rozvrhu. Službu lze tedy jednoznačně identifikovat

sestrou, která tuto službu vykonávala v původním rozvrhu a dnem ve kterém ji vykonávala. Směna označuje jeden typ služby. Směna tedy může být ranní, odpolední, noční či volná směna.

Jako sdílené objekty, které jsou pro všechny vlákna totožné, můžeme brát výchozí rozvrh R_0, matici K_n a struktury k uložení nejlepšího dosud nalezeného rozvrhu.

Výchozí rozvrh R_0:
Výchozí rozvrh R_0 je konstantní po celou dobu běhu algoritmu a můžeme očekávat, že vlákna budou přistupovat k prostorově blízkým datům a tedy je vhodné jej uložit do paměti textur. Pro 32 dní a 32 sester se jedná o 1024 buněk jednorozměrného pole. Každá buňka v sobě obsahuje směnu, kterou daný zaměstnanec v daný čas měl vykonávat. Směna je zakódován následujícím způsobem:

- 0 = bez služby
- 1 = ranní směna
- 2 = odpolední směna
- 3 = noční směna

V případě 4 typů služeb se jedná o 2 bity na každou buňku. Pole je tedy možné uložit do $(1024 * 2)/8 = 256$ bajtů.

Binární matice K_n
Jelikož každá zdravotní sestra může mít, a v praxi často mívá, jinou klasifikaci/specializaci popřípadě služební postavení (hlavní sestra, sestra na dětském oddělení...) a nemůže tedy jakákoli sestra převzít službu jiné, je třeba toto omezení zanést i do přerozvrhovacího algoritmu. Z důvodů univerzálnosti je toto omezení implementováno binární maticí K_n, kdy ke každé sestře*službě je přiřazena binární hodnota určující, zda daná sestra je klasifikována danou službu sloužit. Binární matice K_n má pak v binární reprezentaci $(32 * 1024)/8 = 4096$ bajtů.

Struktury pro R^*:
Nejlepší dosud nalezený rozvrh R^* bude uložen v paměti ve struktuře totožné se strukturou pro pole změn vůči výchozímu rozvrhu C_r. Bude se jednat o kopii této struktury. Aby byl umožněn přístup k tomuto objektu všem vláknům ze všech bloků, je uložen v globální paměti. Velikost tohoto objektu je při minimalizaci paměťových prostředků rovna $16 * 10 * 2 = 320$ bitů, jelikož je třeba si pamatovat kterou službu (jedna z 1024) přiřadíme na rozdíl od původního rozvrhu R_0 na jakou novou pozici (jedna z 1024) pro 16 změn. Jelikož bude objekt uložen v globální paměti není třeba takto optimalizovat a zpomalovat výkon parsováním dat z bitového pole. Můžeme tedy jeden záznam uložit jako dvojici slov/celočíselných proměnných. Velikost se nám poté zvýší na $16 * 16 * 2 = 512$ bitů. Optimalizace by byla vhodná při přesunutí kopie tohoto objektu do sdílené paměti.

K rozvrhu R^* je do globální paměti dále uložená hodnota jeho užitkové funkce $z(R^*)$ jako jediná celočíselná hodnota. Pro možnosti využití heuristik je dále k R^* uložen i setříděný seznam služeb L_s, ze kterých byl rozvrh odvozen. Tento seznam obsahuje maximálně 1024 služeb. Jelikož index každé služby je maximálně 10-ti bitové číslo, zabírá tento objekt v globální paměti $(1024 * 10)/8 = 1280$ bajtů. Pro synchronizovaný přístup k těmto objektům je dále třeba zámek, který povolí v jednom okamžiku přístup k těmto datům pouze jednomu vláknu. Jedná se o logickou hodnotu udávající, zda již k datům nějaké vlákno přistupuje.

Mimo sdílených objektů potřebuje každé vlákno pro svojí práci následující vyhrazené objekty:

Setříděný seznam služeb L_s:
Setříděný seznam služeb L_s uchovává setříděné služby v pořadí, ve kterém se bude algoritmus následně pokoušet je přiřadit do nově vytvářeného rozvrhu. Pro stanovené rozměry problému zabere tento setříděný seznam minimálně 1280 bajtů (odvozeno výše). Z tohoto důvodu musí být umístěn v globální paměti. Jelikož k položkám není sekvenční přístup, není třeba použití bitové reprezentace.

Setříděný seznam sester L_n:
Setříděný seznam sester L_n uchovává pořadí sester, ve kterém je bude algoritmus následně prohledávat při hledání pozice pro přiřazení služby. Pro horní mez 32 sester je třeba alokovat minimálně $(32 * 5)/8 = 20$ bajtů. Jelikož je k tomuto objektu velice častý přístup, bylo by vhodné jej umístit do rychlé paměti (sdílené paměti).

Bitové pole masek obsazenosti rozvrhu M_r: Při rozvrhování je třeba v průběhu algoritmu vědět, která pole rozvrhu jsme již obsadily a naopak která jsou volná pro přiřazení nové služby. K tomuto účelu slouží bitové pole masek obsazenosti rozvrhu M_r. Každé pole ve výchozím rozvrhu je zde reprezentováno jedním bitem, který udává, zda je již pole přiřazeno či zda je volné k přiřazení. Velikost této struktury je tedy $1024/8 = 128$ bajtů. Struktura je uložena v globální paměti.

Pole změn vůči výchozímu rozvrhu C_r
Jelikož můžeme v maximálním případě v rozvrhu vykonat 16 změn, není vhodné ukládat pro každý běh vlastní rozvrh. Místo toho jsou uloženy jen změny v přiřazení služeb v aktuálním rozvrhu oproti výchozímu rozvrhu R_0 v poli změn vůči výchozímu rozvrhu C_r. Kombinací tohoto pole, bitového pole M_r a rozvrhu R_0 jsme schopni zkonstruovat aktuální rozvrh. Pole změn vůči výchozímu rozvrhu zabere pro 16 změn $16 * 10 * 2 = 320$ bitů.

IV. Základní algoritmus

A. CPU verze

CPU verze základního algoritmu pracuje plně sekvenčně. Algoritmus vychází z konstruktivní heuristiky, která je popsána v článcích [2], [3]. V hlavní metodě Main, jíž popisuje Algoritmus 1, je nejdříve ze vstupního souboru načten původní rozvrh R_0, binární matice klasifikace sester a požadované absence. Po načtení těchto dat je třeba inicializovat struktury, se kterými bude pracovat výkonná metoda běhu algoritmu (Algoritmus 2). Mezi tyto struktury patří L_s, M_r či C_r. Velikost těchto struktur je odvozena od parametrů, které jsou načteny ze vstupného souboru, a proto jsou alokovány až poté. Následně metoda Main spouští samotný běh algoritmu

(výkonnou metodu běhu algoritmu). Tato metoda najde rozvrh R^*. Nakonec je výsledný rozvrh uložen či vytištěn na obrazovku.

```
  Input: vstupní soubor
  Output: R*
1 begin
2 |  načtení vstupního souboru;
3 |  inicializace struktur;
4 |  spuštění běhu algoritmu;
5 |  uložení/vytištění výsledků;
6 end
```

Algoritmus 1: Hlavní metoda Main na CPU

Samotná výkonná metoda běhu algoritmu (Algoritmus 2) nejprve zinicializuje vstupní struktury - tj. naplní L_s seznamem služeb, které je potřeba rozvrhnout, zinicializuje pole M_r a C_r. V poli M_r ponechá přiřazené pouze služby o kterých již na začátku víme, jak budou ve výsledném rozvrhu vypadat - ty které mají být zmražené (zachovat se i do nového rozvrhu v nezměněné podobě) a služby, které má nahradit absence. V poli C_r budou zaznamenány jen služby, které má nahradit absence. Poté přejde algoritmus do hlavní smyčky. Každý průběh touto smyčkou představuje jeden pokus o přerozvrhnutí pomocí L_s. Na začátku každé iterace je třeba vymazat změny ve strukturách, které provedla předešlá iterace. Dále je třeba zamíchat seznam služeb L_s, aby nová iterace neprováděla přesně to samé jako předchozí. Následně je ukazatel aktuálně přiřazované služby přesunut na první pozici v L_s. Po této inicializaci následuje přiřazovací smyčka, ve které jsou služby jedna po druhé zařazovány do rozvrhu. Algoritmus se snaží službu přiřadit v následujících fázích:

1) Přiřazení služby na původní pozici, pokud toto přiřazení neprolamuje pevná omezení
2) Přiřazení služby sestře, která je pro danou službu klasifikovaná a pro kterou by přiřazení služby neznamenalo prolomení pevných ani pružných omezen
3) Přiřazení služby sestře, která je pro danou službu klasifikovaná a pro kterou by přiřazení služby neznamenalo prolomení pevných omezení a pružná omezení mohou být narušena službou v následující či předcházející den
4) Přiřazení služby sestře, která je pro danou službu klasifikovaná a pro kterou by přiřazení služby neznamenalo prolomení pevných omezení

Fáze jsou vykonávány sekvenčně a pokud se již podařilo službu přiřadit, jsou následující fáze přeskočeny a přechází se rovnou k přiřazování následující služby z L_s. V každé fázi je také testováno, nemá-li daná sestra v daný den již v novém rozvrhu přiřazenou jinou službu. Ve 2. až 4. fázi probíhá hledání sestry, které by bylo možné službu přiřadit v vzestupném pořadí jejich indexů.

V případě, že se nepodaří službu přidělit ani v jedné z fází, zakončí se pokus vytváření rozvrhu z aktuálního seznamu L_s a přejde se do další iterace. Nenastane-li tento jev a podaří-li se vytvořit nový rozvrh, který splňuje tvrdá omezení, je tento nový rozvrh ohodnocen užitkovou funkci $z(R)$. Pokud je $z(R)$ menší než užitková funkce nejlepšího dosud nalezeného rozvrhu $z(R^*)$, je rozvrh R^* přepsán aktuálním rozvrhem a $z(R^*)$ je přepsána $z(R)$.

```
   Input: Ls, Mr, Cr, R0, Kn
   Output: R*
 1 inicializace vyhrazených objektů;
 2 while # běhu < cílový počet běhů do
 3 |  vyprázdni uložené změny;
 4 |  zamíchej Ls;
 5 |  přesuň ukazatel na 1. službu;
 6 |  while nejsou přiřazeny všechny služby do
 7 |  |  if lze službu přiřadit na původní místo then
 8 |  |  |  přiřaď službu na původní místo;
 9 |  |  end
10 |  |  else if ∃ sestra n splňující podmínky druhé fáze
   |  |  then
11 |  |  |  přiřaď službu sestře n;
12 |  |  end
13 |  |  else if ∃ sestra n splňující podmínky třetí fáze
   |  |  then
14 |  |  |  přiřaď službu sestře n;
15 |  |  end
16 |  |  else if ∃ sestra n splňující podmínky čtvrté fáze
   |  |  then
17 |  |  |  přiřaď službu sestře n;
18 |  |  end
19 |  |  else
20 |  |  |  nerozvrhnutelné pomocí aktuálního Ls;
21 |  |  |  break;
22 |  |  end
23 |  end
24 |  if byl vytvořen nový rozvrh then
25 |  |  vypočti z(R);
26 |  |  if z(R) < z(R*) then
27 |  |  |  přepiš R* aktuálním rozvrhem R
28 |  |  end
29 |  end
30 end
```

Algoritmus 2: Výkonná metoda běhu algoritmu na CPU

B. GPU verze

Metoda Main v GPU verzi (Algoritmus 3) stejně jako v CPU verzi musí nejdříve načíst vstupní data. Následně je alokována paměť pro výše popsané struktury. Na rozdíl od verze pro CPU je třeba alokovat pro každé vlákno vlastní vyhrazené struktury. Po alokaci jsou zinicializovány globální struktury - rozvrh R_O, matice K_n a užitková funkce $z(R^*)$. Poté je spuštěn kernel pro inicializaci vyhrazených struktur jednotlivých vláken. ****************Možná by šlo spojit inicializaci a výkonnou část do jednoho kernelu************. Po inicializaci všech struktur spouští CPU na GPU kernel pro výkonnou metodu běhu algoritmu. Nakonec je opět výsledný rozvrh R^* zaznamenán do souboru či vytištěn na obrazovku.

Na rozdíl od výkonné metody běhu algoritmu na CPU, kernel na GPU (Algoritmus 4) popisuje pouze jeden pokus

Input: vstupní soubor
Output: R^*
1 **begin**
2 načtení vstupního souboru;
3 alokace struktur;
4 inicializace globálních struktur;
5 spuštění kernelu inicializace struktur;
6 spuštění kernelu běhu algoritmu;
7 uložení/vytištění výsledků;
8 **end**

Algoritmus 3: Hlavní metoda Main na GPU

o vytvoření nového rozvrhu ze seznamu L_s. Následně je vypočtena užitková funkce rozvrhu, pokud byl úspěšně rozvržen. Pokud ne, je tato funkce rovna nekonečnu. Po rozvržení (případně zjištění, že je rozvrh neplatný) jsou všechna vlákna v bloku sesynchronizována. Poté je pomocí redukce zjištěn minimální nalezený rozvrh v daném bloku R^{*b}. Pokud je má tento rozvrh nižší užitkovou funkci než rozvrh R^*, je jím nahrazen. Díky tomu omezíme přístup k objektům R^*. V případě, že bychom redukci nepoužily by se mohlo stát že by více vláken chtělo přepsat R^*, ale uvízla by na zámku pro přístup k tomuto objektu. To by omezilo paralelismus a daná část by byla povahou spíše sekvenční.

V. Úpravy základního algoritmu

Základní algoritmus slouží jako základní kámen, na který je možné dále stavět. Sám osobě může mít nízkou schopnost nalezení platného rozvrhu a proto je vhodné jej dále upravovat. Je možné využít heuristiky pro prořezávání prostoru výsledků pro jeho urychlení či např. mechanizmy pro zvýšení pravděpodobnosti nalezení platného rozvrhu.

A. *Backtrackovací algoritmus*

Při přiřazování služeb dle seřazeného seznamu služeb můžeme dojít do chvíle, kdy již nenalezneme žádnou pozici, kam bychom mohli službu umístit, jelikož všechny možné pozice, které by neporušovaly žádné tvrdé omezení jsou již zabrány. Základní algoritmus v tuto chvíli končí a rozvrh označí za neplatný. Abychom omezily tento stav, můžeme vracet úpravy do chvíle, dokud se neuvolní nějaká pozice pro aktuální službu. Poté tuto službu umístit a dále se pokusit umístit tímto zpětným algoritmem odebrané služby a poté i služby dosud nepřiřazené.

Tato myšlenka je platná v případě sekvenčního provádění na CPU, ale v případě implementace na GPU by vlákno, které by aplikovalo tento algoritmus blokovalo ostatní vlákna ve warpu a tím nesmírně zpomalila celý běh. Z tohoto důvodu je třeba upravit tento algoritmus tak, aby byl aplikovatelný na GPU.

V upraveném algoritmu omezíme možnost posunutí služby, kterou není možné v aktuální iteraci nikam přiřadit pouze o jednu pozici vpřed. Tím je tedy zrušena i poslední změna, kterou aplikovala poslední přiřazená služba. V další iteraci se pokusíme službu znovu přiřadit, pokud se nám toto nepovede, opět ji posuneme o pozici v před. V této podobně by však často nastalo zacyklení algoritmu. Proto musíme omezit počet těchto posunů určitou maximální hodnotou.

Input: L_s, M_r, C_r, R_0, K_n
Output: R^*
1 zamíchej L_s;
2 přesuň ukazatel na 1. službu;
3 **while** *nejsou přiřazeny všechny služby* **do**
4 **if** *lze službu přiřadit na původní místo* **then**
5 přiřaď službu na původní místo;
6 **end**
7 **else if** $\exists$ *sestra n splňující podmínky druhé fáze* **then**
8 přiřaď službu sestře n;
9 **end**
10 **else if** $\exists$ *sestra n splňující podmínky třetí fáze* **then**
11 přiřaď službu sestře n;
12 **end**
13 **else if** $\exists$ *sestra n splňující podmínky čtvrté fáze* **then**
14 přiřaď službu sestře n;
15 **end**
16 **else**
17 nerozvrhnutelné pomocí aktuálního L_s;
18 break;
19 **end**
20 **end**
21 **if** *nebyl vytvořen nový rozvrh* **then**
22 $z(R) = \infty$;
23 **end**
24 **else**
25 vypočti $z(R)$;
26 **end**
27 synchronizace vláken;
28 redukce a nalezení rozvrhu s minimálním $z(R)$ v bloku;
29 **if** *(idVlákna == 0)&&($z(R^{*b} < z(R^*)$)* **then**
30 $R^* = R^{*b}$;
31 **end**

Algoritmus 4: Výkonný kernel běhu algoritmu na CPU

B. *Setříděný seznam sester*

V průchodu jedné iterace algoritmu a hledání místa, kam bychom mohli službu přiřadit procházíme sestry v nějakém předem určeném pořadí. Pokud bychom v každém běhu a v každé iteraci procházeli sestry ve stále stejném pořadí, docházelo by tím k permanentní preferenci určitých sester před jinými a tím by i nastával případ, kdy by určitá část sester měla permanentně více služeb než zbytek.

Tento problém je možné řešit přidáním rozdílného seznamu sester ke každému běhu algoritmu. Prohledávání poté bude probíhat v pořadí určeném tímto seznamem. Tím bude při každém běhu při prohledávání preferována jiná část sester a celkově budou moci být výsledné rozvrhy přes několik rozvrhovaných období spravedlivější. V případě opětovného volání běhu algoritmu dojde k širšímu prohledávání prostoru, jelikož bude běh inicializován nejen náhodně seřazeným seznamem služeb ale i sester.

Další možností je tento seznam na začátku každé iterace algoritmu náhodně zamíchat. Tím dosáhneme při rozvrhování nejvyšší spravedlivosti přiřazování služeb sestrám za cenu výkonu, který nám zabere míchání seznamu sester. Ve spolupráci s výše popsaným backtrackovacím algoritmem se

zvýší i pravděpodobnost nalezení platného rozvrhu, jelikož se sníží šance zacyklení, protože i když se dvakrát dostaneme do stejné podoby seřazeného seznamu služeb, může být výsledek přiřazení služeb rozdílný, jelikož se sestry pro přiřazení služby velice pravděpodobně budou procházet v jiném pořadí.

C. Cyklické volání s návratem

Původní algoritmus by byl plně homogenní a celý by fungoval na GPU a po kladném vyhodnocení ukončovací podmínky by předal výsledky CPU. Jelikož je tento algoritmus jen minimálně závislý na předchozím běhu je z praktického hlediska vhodné tento algoritmus upravit tak, aby bylo možné jej ukončit kdykoli v jeho průběhu a získat dosud nejlepší výsledek.

V tomto případě by tedy byl algoritmus ukončen na GPU po určitém počtu průběhů a poté by bylo řízení navráceno zpět CPU. CPU by vypsalo dosavadní nejlepší nalezený výsledek (jeho hodnotu užitkové funkce) a zkontrolovalo by, zda nepožaduje uživatel ukončení běhu. V tom případě by nejlepší dosud známý výsledek uložil a dále nepokračoval. V opačném případě by bylo řízení vráceno zpět GPU, které by dále hledalo lepší rozvrhy.

D. Ovlivnění nejlepším rozvrhem

Ve chvíli, kdy již dlouhou dobu nebylo nalezeno řešení, které by bylo lepší než nejlepší doposud známé řešení můžeme předpokládat, že optimum bude někde poblíž tohoto nejlepšího doposud známého optima.

Můžeme tedy místo náhodného prohledávání celého prostoru vycházet z tohoto dosud nejlepšího řešení a prohledávat jeho mutace. V případě náhodného procházení mícháme prohazováním celý seznam služeb. V případě této úpravy bychom zinicializovali seznam služeb kopií seřazeného seznamu služeb dosud nejlepšího řešení a ten bychom poté promíchávaly. Zužování prohledávaného prostoru a postupné přibližování se nejlepšímu známému řešení je zajištěno snižováním pravděpodobnosti promíchání (mutace) jednotlivých prvků.

E. Inverzní hledání

Základní algoritmus pracuje v několika fázích, kdy v každé fázi projde všechny sestry a pokusí se najít tu, které by v dané fázi mohl službu přiřadit. Při SIMD zpracování může často nastat situace, kdy skoro všechny vlákna ve warpu již svojí službu přiřadily a čekají na poslední vlákno, které stále vyhledává přípustnou pozici. Tím může být velký potenciál paralelního zpracování nevyužit.

Je možné se na problém koukat i jiným pohledem. Místo toho, abychom v každé fázi procházeli celý seznam sester, můžeme seznam sester projít pouze jednou a poznamenat si, která sestra splňuje požadavky které fáze. Pokud nalezneme sestru, která splňuje podmínky druhé fáze (přiřazení služby sestře, pro kterou předešlá i následující služba je z pohledu slabých omezení přípustná) není třeba další sestry prohledávat. Tento přístup sice pořeže prohledávání pouze v případě, kdy nějaká sestra splňuje podmínky první či druhé fáze základního algoritmu, ale sníží počet přístupů do globální paměti, jelikož data jedné sestry je třeba přečíst pouze jednou. Je-li tedy zásadním faktorem zpomalení doba přístupu do globální paměti, může tento přístup zrychlit zpracování.

LITERATURA

[1] Michael Vincent Chiaramonte: *Competitive Nurse Rostering and Rerostering*, A Dissertation Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy, 2008

[2] Margherita Moz, Margherita Vaz Pato: *An Integer Multicommodity Flow Model Applied to the Rerostering of Nurse Schedules*, Annals of Operations Research 119, 285–301, 2003

[3] Margherita Moz, Margherita Vaz Pato: *A genetic algorithm approach to a nurse rerostering problem*, Computers & Operations Research 34 (2007) 667 – 691

[4] Margherita Moz, Margherita Vaz Pato: *Solving a bi-objective nurse rerostering problem by using a utopic Pareto genetic heuristic*, J Heuristics (2008) 14: 359–374

[5] Eckart Zitzler and Lothar Thiele: *An Evolutionary Algorithm for Multiobjective Optimization, The Strength Pareto Approach*, TIK-report No.43, May 1998

[6] Eckart Zitzler, Marco Laumanns, and Stefan Bleuler: *A Tutorial on Evolutionary Multiobjective Optimization*

[7] Broos Maenhout, Mario Vanhoucke: *An evolutionary approach for the nurse rerostering problem*, Computers & Operations Research 38 (2011) 1400–1411

[8] Shane A. Knighton: *An optimal network-based approach to scheduling and re-rostering continuous heterogeneous workforces*, A Dissertation Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy, ARIZONA STATE UNIVERSITY, August 2005

[9] Cristina Nader Vasconcelos and Bodo Rosenhahn: *Bipartite Graph Matching Computation on GPU*, Leibniz Universitaet Hannover

[10] Agnieszka Łupińska: *Parallel implementation of flow and matching algorithms*, Jagiellonian University, Kraków

[11] Mohamed Esseghir Lalami, Didier El-Baz: *GPU Implementation of the Branch and Bound method for knapsack problems*,2012 IEEE 26th International and Distributed Processing Symposium Workshops & PhD

[12] Ramnik Arora, Rupesh Tulshyan, Kalyanmoy Deb: *Parallelization of Binary and Real-Coded Genetic Algorithms on CUDA*

[13] Darrell Whitley and Nam-Wook Yoo: *Modeling Simple Genetic Algorithms for Permutation Problem*, Computer Science Department, Colorado State University