Systém pro přiřazování směn zaměstnancům

TECHNICKÁ ZPRÁVA

Bäumelt Zdeněk, Šůcha Přemysl, Hanzálek Zdeněk

12.12.2013

Obsah

1 Úvod

Systém pro přiřazování směn je webový nástroj, který slouží pro tvorbu rozvrhů na pracovištích ve směnných provozech. Po nahrání vstupu řešeného problému v požadovaném formátu je uživateli vrácen automaticky vytvořený rozvrh směn. Rozvrhy jsou tvořeny pomocí knihovny algoritmů pro rozvrhování lidských zdrojů. Algoritmy řešící tvorbu rozvrhu se dynamicky mění na základě rozhodnutí hyperheuristiky, která je v knihovně nadstavbou nad ostatními algoritmy. Rozvrh je tak vylepšován z několika různých pohledů, které jsou zastoupeny různými algoritmy. Knihovna kombinuje použití algoritmů optimální a heuristických, které jsou vhodnější pro tvorbu rozvrhů pro velké instance problémů. Algoritmy během tvorby rozvrhu uvažují omezení daná Zákoníkem práce ČR a dále omezení daná kolektivní smlouvou. Přínos tohoto systému je dvojí - je zkrácena doba potřebná na vytvoření vlastního rozvrhu, který byl dříve tvořen ručně. Dále jsou rozvrhy vytvářeny efektivnějším způsobem než v případě ruční tvorby, např. dochází ke snižování přesčasů proplácených zaměstnavatelem. Oba tyto přínosy tak vedou na finanční úspory společnosti, která tento systém využívá.

Tato technická zpráva popisuje zejména jádro systému pro přiřazování směn zaměstnancům, a to je knihovna algoritmů pro automatické přiřazování směn. V sekci 2 je shrnuta rešerše stávajících algoritmů. Na základě této rešerše byly vybrány algoritmy, které budou součástí knihovny pro automatické přiřazování směn. V následujících sekcích jsou pak posány vybrané algoritmy. V sekci 3 je detailně popsána Scatter Search hyperheuristika. V následující sekci je uvedena skupina algoritmů, které mohou být používány jako zásuvné moduly dle potřeby. V sekci 5 je popsán algoritmus Adaptive Variable Neighborhood Search, který je založen na metodách lokálního prohledávání stavového prostoru. V sekci 6 je popsán algoritmus, který vznikl rozšířením algoritmu Adaptive Neighborhood Search.

2 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.

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 také 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.

2.1 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.

2.2 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.

2.2.1 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.

2.2.2 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 prů-

chodu 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 různých 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.

2.2.3 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 různých 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.

2.2.4 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.

2.2.5 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.

2.2.6 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 reálného prostředí možné použít ani hyper-heuristiku založenou na SAT heuristikách.

2.2.7 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.

2.2.8 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.

2.2.9 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.

3 Scatter Search hyper–heuristika

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. 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.

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.

3.1 Návrh algoritmu

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

3.2 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.

Algoritmus 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

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

Algoritmus 2 Pseudokód Scatter Search hyper-heuristiky (pokračování)

7. Nalezení nejvhodnějšího rozvrhu

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

3.3 Objektový model algoritmu

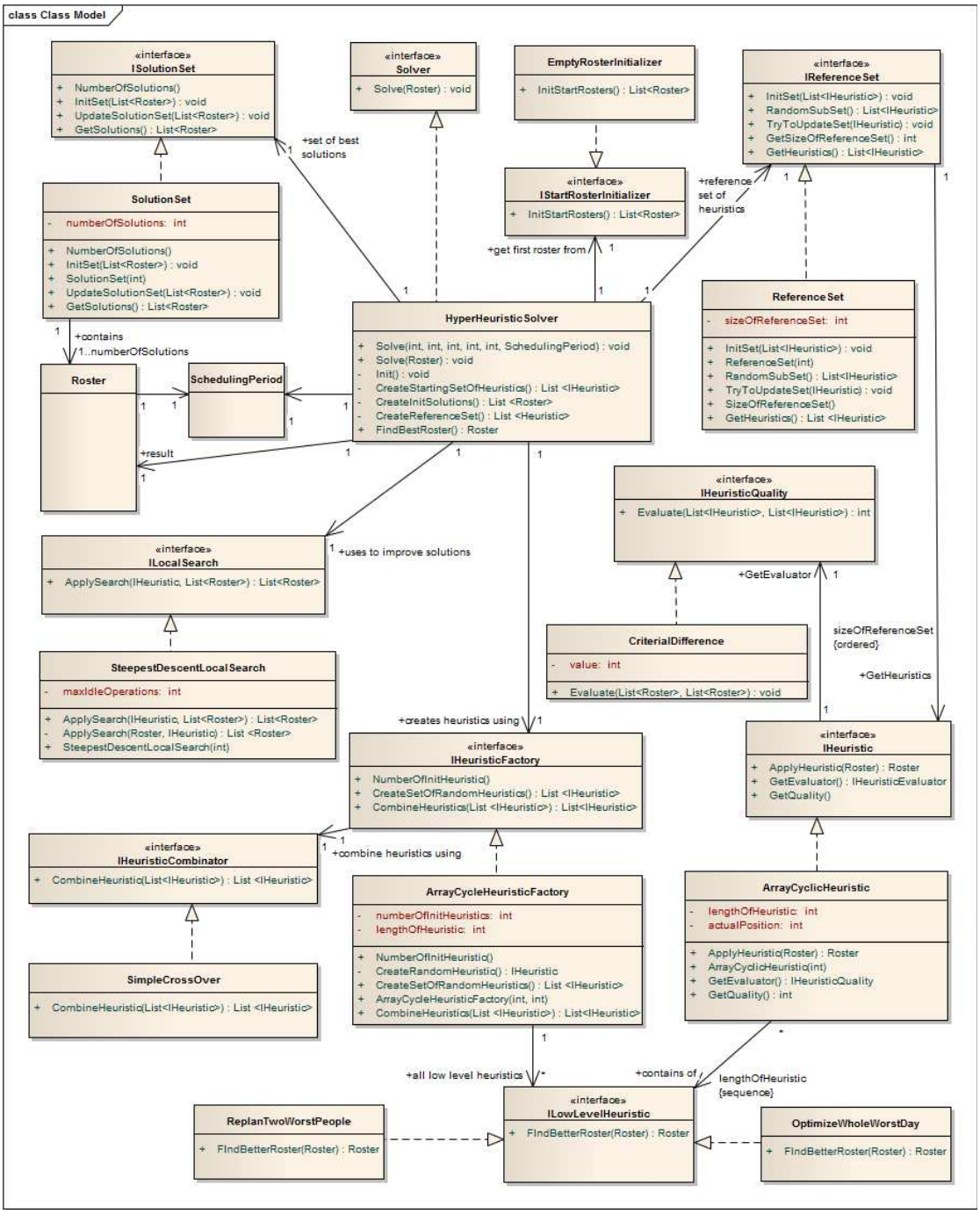
Objektový model algoritmu popisuje soustavu tříd a rozhraní vycházející z podoby Scatter Search algoritmu. Během návrhu struktury byl kladen důraz především na dosažení co největší modifikovatelnosti a rozšiřitelnosti algoritmu. Model zobrazuje objektový návrh na úrovni detailu odpovídající popisu algoritmu, 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 zmíněná 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 Nízkoúrovňové heuristiky

Knihovna algoritmů obsahuje základní stavební bloky, které mohou být využity modulárně. Jedná se o tzv. nízkoúrovňové heuristiky, které jsou popsány v této sekci. Tyto heuristiky 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.

4.1 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.

5 Adaptive Neghborhood Search algoritmus

Algoritmus Adaptive Neighborhood Search (ANS) byl vybrán jakou součást algoritmu díky výborným výsledkům na soutěži Nurse Rostering Competition 2010 [21]. V článku [22], který popisuje základní myšlenky, je však několik klíčových prvků nedostatečně či neurčitě specifikovaných. V sekci Návrh algoritmu je popsána koncepce algoritmu založená na základních myšlenkách [22]. Další sekce pak obsahují popis jednotlivých úprav a vylepšení algoritmu pro zvýšení efektivity algoritmu.

Pseudokód celého algoritmu je uveden v Algoritmu 3. Jeho důležité části jsou dále přesněji popsány včetně pseudokódů jednotlivých metod.

Algoritmus 3 Pseudokód Adaptive Neighborhood Search algoritmu pro řešení NRP

```
1:  Vstup: Instance NRP I
2:  Výstup: Nalezený rozpis směn X*
3:  X^0 = getInitialRoster(I)   // viz. Algoritmus 4
4:  X* = X^0; dl = 0; iter = 0; adapIter = 0; adapF = max()
5:  REPEAT
6:    X = X^0; X' = X^0
7:    WHILE !localSearchEndReached() DO
8:      mv = getTypeOfMove()   // viz. kapitola 5.2
9:      M(X) = generateFeasibleMoves(X, mv)
10:     X = selectNeighborhoodSolution(X, mv, dl)   // viz. Algoritmus 5
11:     IF (f(X) < f(X')) THEN
12:       X' = X   //X' reprezentuje nejlepší řešení nalezené LS
13:     END IF
14:     dl = updateParameter(dl, ++iter, f(X))   // viz. Algoritmus 6
15:   END WHILE
16:   IF f(X) < f(X*) THEN
17:     X* = X
18:   END IF
19:   X^0 = X OR X^0 = X*   // viz. kapitola 5.8
20:   dl = 1
21: UNTIL stopCriterionReached()
```

Na začátku algoritmu je vygenerováno počáteční proveditelné řešení, které je následně optimalizováno pomocí přizpůsobivého výběru mezi třemi strategiemi prohledávání okolí. Výběr strategie je prováděn na základě aktuální hodnoty parametru, udávajícího úroveň diverzifikace. Prohledávání okolí je ukončeno poté, co nedojde během několika iterací k vylepšení řešení. Následuje nové spuštění agoritmu s tím rozdílem, že jako počáteční řešení je použito nejlepší doposud nalezené a úroveň diverzifikace je nastavena na maximální hodnotu. Ukončovacím kritériem pro celý algoritmus je počet iterací lokálního prohledávání, který byl při testování nastaven na 20 000.

5.1 Získání počátečního řešení

Vygenerování počátečního řešení popisuje Algoritmus 4, který generuje splnitelné počáteční řešení - splňuje tvrdé podmínky. Podmínka maximálně jedné směny za den u každého zaměstnance je splněna automaticky díky datové reprezentaci. Uvažujeme tedy jen podmínku požadovaného pokrytí jednotlivých směn v jednotlivé dny.

Použitá heuristika začíná s prázdným rozvrhem. Pro každý den D ze všech plánovaných dnů se náhodně vybere směna H, která má vyšší požadavek na pokrytí *sc(D, H)* nežli 0. Poté se náhodně vybere zaměstnanec, který na směnu H může být přiřazen. Provede se přiřazení a požadavek pro danou směnu H, v daný den D (*sc(D, H)*) je snížen o 1.

Náhodný výběr zaměstnance se opakuje dokud v daném dnu nejsou splněny všechny požadavky na pokrytí směn. Následně se pokračuje dalším dnem. Algoritmus neuvažuje žádnou z měkkých podmínek. Autoři na základě experimentů tvrdí, že počáteční rozpis směn má malý dopad na následnou výkonost ANS algoritmu.

5.2 Způsob přechodu k sousednímu řešení

Pro jakékoli řešení X je možné získat množinu jeho sousedních řešení pomocí přechodu *mv*, kde *mv* je přechod z množiny všech aplikovatelných přechodů *M(X)*. Sousedství řešení X poté označujeme jako *N(X)* $= \{X \oplus mv \mid mv \in M(X)\}$.V algoritmu se používají dva následující přechody:

Přesun jedné směny k jinému zaměstnanci ($mv_1(d, s_1, s_2)$) Aplikací kroku mv_1 získáme sousedství, kde směna přiřazená k zaměstnanci s_1 v den d je přiřazena zaměstnanci s_2 v den d, tedy $x_{s_2,d} = x_{s_1,d}$ a $x_{s_1,d} = -1$, kde $x_{s_1,d}$ reprezentuje identifikátor směny a pokud zaměstnanec nemá přiřazenu žádnou směnu, pak je hodnota x rovna -1. Množina sousedních řešení $M_1(X)$ obsahuje všechny výstupy

$$mv_1(d, s_1, s_2) \mid \forall d \in D, x_{s_1,d} \neq -1 \wedge x_{s_2,d} = -1$$

Prohození dvou směn přiřazených páru zaměstnanců ($mv_2(d, s_1, s_2)$) Aplikací kroku mv_2 získáme sousedství, kde směna přiřazená k zaměstnanci s_1 v den d je přiřazena zaměstnanci s_2v den d, tedy $x_{s_2,d} = x_{s_1,d}$ a naopak, tedy $x_{s_1,d} = x_{s_2,d}$. Směny, které mají daní zaměstnanci by z principu neměly být stejné a také nepracujeme

Algoritmus 4 Generování počátečního řešení

Výstup: realizovatelný rozpis směn

```
FOREACH DAY D FROM DAYS
  WHILE (getRandomUncoveredH(D) != NULL) DO
    assignRandomNurse(H)
    sc(D, H) --
  END WHILE
END FOREACH
```

se zaměstnanci, kteří v daný den nemají přiřazenu směnu ($x_{s,d} = -1$). Množina sousedních řešení $M_2(X)$ obsahuje všechny výstupy

$$mv_2(d, s_1, s_2) \mid \forall d \in D, x_{s_1,d}, x_{s_2,d} \neq -1 \wedge x_{s_1,d} \neq x_{s_2,d}$$

Výběr mezi těmito dvěma přechody se provádí na základě pravděpodobnosti q, která určuje pravděpodobnost použití sousedství $M_1(X)$. Sousedství $M_2(X)$ je použito s pravděpodobností *(1-q)*. V článku je hodnota $q = 1 - \varphi \cdot dens$, kde $\varphi = 0.4$ a hodnota *dens* reprezentuje hustotu dané instance problému

$$dens = \frac{\sum_{d \in D} \sum_{h \in H} sc(d, h)}{S \cdot D} \cdot 100\,\%$$

která je určena podílem sumy požadavků na všechny směny přes všechny dny a součinu počtu zaměstnanců S a počtu dnů rozvrhovaného období D. Výběr se tedy provede na základě náhodně vygenerovaného čísla r v rozsahu $< 0, 1 >$, a pokud $r < q$, tak je použito sousedství $M_1(X)$, v ostatních případech je použito sousedství $M_2(X)$.

5.3 Výběr sousedního řešení

K výběru sousedního řešení se používají tři strategie, mezi kterými se algoritmus rozhoduje tak, aby dosahoval příhodného poměru zanořování a prozkoumávání okolí. Pseudokód výběru je uveden v Algoritmu 5 a jednotlivé strategie jsou popsány v následujících podsekcích.

Výběr strategie se v každé iteraci provádí na základě hodnoty úrovně diverzifikace *dl*. Rozsah hodnot $< 0, 1 >$, kterých může *dl* nabývat, je rozdělen do tří částí pomocí parametrů β_1 a β_2, pro které platí $0 < \beta_1 < \beta_2 < 1$. Pokud $dl \in < 0, \beta_1)$, pak je použito intenzivní prohledávání. Pokud $dl \in < \beta_1, \beta_2)$, pak je použito střední prohledávání a pokud $dl \in < \beta_2, 1 >$, tak algoritmus využije diverzifikační prohledávání. V článku jsou hodnoty empiricky zvoleny $\beta_1 = 0.3$ a $\beta_2 = 0.7$.

5.4 Intenzivní prohledávání

Během intenzivního prohledávání je využívána heuristika *Tabu search*, která si ukládá určený počet řešení, která nemohou být vybrána. V případě kroku, který převede směnu mezi dvěma zaměstnanci v den *d*, je do zakázaných kroků přidán ten, který by tuto směnu vrátil k původnímu zaměstnanci. U kroku, který prohodí směny zaměstnanců s_1 a s_2 v den *d*, je do zakázaných kroků přidán takový, který by znovu přiřadil směnu $x_{s_1,d}(x_{s_2,d})$ k zaměstnanci $s_1(s_2)$ v den *d*.

Velikost tabu listu TT, neboli počet kroků, které musí být vloženy, než bude daný krok z Tabu listu odstraněn, je určen součtem náhodné hodnoty od 1 do 3 a parametru algoritmu. $TT = tl + rand(3)$, kde *tl* je dáno počtem zaměstnanců S jako $tl = \lfloor 0.8 \cdot S \rfloor$. Vnitřní koeficient byl zvolen empiricky.

Algoritmus 5 Výběr sousedního řešení

Vstup: aktuální řešení X, použitý krok mv, úroveň diverzifikace dl
Výstup: vybraný přechod k sousednímu řešení aplikovaný na aktuální řešení

```
IF dl ∈< 0, β1) THEN   // viz.kapitola 5.4
  X' = getBestSolution(mv,X)
  IF isTabuAndNotBest(X') THEN
    X' = getBestSolutionExceptTabu(mv,X)
  END IF
END IF
IF dl ∈< β1, β2) THEN   //viz. kapitola 5.5
  S* = getRandomSubsetOfNurses(⌊|S| /2⌋)
  X' = getBestSolution(mv,X,S*)
  IF isTheMostRecent(X') AND rand(1) < 0.5 THEN
    X' = getSecondBestSolution(mv,X,S*)
  END IF
END IF
IF dl ∈< β2, 1 > THEN   // viz. kapitola 5.6
  S* =getRandomSubsetOfNurses(⌊|S| /2⌋)
  M'(X) = getPromisingSolutionsSubset(mv,X,S*)
  IF |M(X)| > 0 THEN
    X' = getRandomSolution(M'(X))
  ELSE
    X' = getRandomSolution(M(X))
  END IF
END IF
RETURN mv
```

V každé iteraci je tedy výběr omezen na kroky, které nejsou v tabu listu. Z těch se vybere krok, který přinese nejvýznamnější zlepšení ohodnocení rozvrhu. V případě více kroků se shodným nejlepším zlepšením se z nich provede náhodný výběr.

Krok z tabu listu je přesto možné použít, ale pouze v případě, že vede k lepšímu ohodnocení řešení, než je nejlepší doposud nalezené.

5.5 Střední prohledávání

Na rozdíl od intenzivního prohledávání, kde se vybírá nejlepší krok ze všech proveditelných kroků, střední prohledávání vybírá krok z množiny, která je omezená podmnožinou zaměstnanců. Její velikost je rovna polovině celkového počtu zaměstnanců a je náhodně vybrána v každé iteraci algoritmu. Uvažované kroky se následně vztahují pouze k těmto zaměstnancům.

Z omezeného okolí je poté vybrán nejlepší krok. Pokud je však vybraný krok nedávno použitý, pak s pravděpodobností 0.5 vybereme druhý nejlepší krok. Pro tento účel je nutné uchovávat jednotlivé použité kroky ve formátu obsahujícím daného zaměstnance,

den a typ směny, ke kterému je připojeno číslo iterace, ve které byl krok použit.

Přesněji, při každém odebrání směny h od zaměstnance s v den d se uloží aktuální číslo iterace lokálního prohledávání (proměnná *iter* v Algoritmu 3) do příslušného záznamu na základě hodnot *s, d, h*. Takto je v dalších iteracích algoritmu možné snadno identifikovat, zda je nejlepší vybraný krok nedávno použitým.

5.6 Diverzifikační prohledávání

Záměrem tohoto typu prohledávání je přesunout řešení z lokálního optima v případě, že je detekováno uváznutí. Stejně jako v předcházejícím případě je výběr prováděn pouze nad podmnožinou zaměstnanců, která je volena náhodně v každé iteraci algoritmu. Nad těmito zaměstnanci je vybrána podmnožina tzv. slibných kroků, do které patří ty kroky, které vylepšují hodnotu alespoň jednoho měkkého omezení. Pokud takovéto slibné kroky pro aktuální podmnožinu zaměstnanců existují, pak je z nich náhodně vybrán jeden, který bude proveden. Pokud nebyl nalezen ani jeden slibný krok, pak je náhodně vybrán jakýkoli krok nad podmnožinou vybraných zaměstnanců.

5.7 Přizpůsobení úrovně diverzifikace

ANS algoritmus využívá mechanizmus přizpůsobivé úrovně diverzifikace na základě historie prohledávání okolí. Z počátku je hodnota úrovně nastavena na 0, aby mohla být rychle vylepšována hodnotící funkce. Při prohledávání je kontrolován rozdíl indexu aktuální iterace *iter* a indexu poslední iterace *adapIter*, při které došlo k aktualizaci úrovně diversifikace. Jakmile prohledávání po daný počet iterací $\theta = \lfloor \frac{S \cdot H}{10} \rfloor$, kde S je počet zaměstnanců a H je počet směn, nevylepšilo hodnotu řešení, dochází k navyšování úrovně diverzifikace dokud prohledávání nepřekoná stagnaci. Oproti tomu je také kontrolováno vylepšování řešení porovnáním hodnoty aktuálního řešení *f(X')* a hodnoty řešení při poslední změně úrovně *adapIter*. Pokud dochází k vylepšování řešení, je úroveň diverzifikace postupně snižována. Pseudokód metody pro změnu parametru *dl* je uveden v Algoritmu 6. Použité hodnoty dělitelů 6 a 10 jsou, dle autorů článku [22], ideálně zvolené pro ANS algoritmus.

5.8 Mechanismus restartu s nejlepším řešením

K využití restartu dojde ve chvíli, kdy lokální prohledávání během daného počtu iterací nemohlo vylepšit kvalitu řešení. Počet iterací byl v článku různý, dle časových omezení INRC-2010 pro jednotlivé instance byl nastavován na hodnoty 500, 1000 a 10000. Hodnoty neomezují přímo počet iterací algoritmu lokálního prohledávání, ale určují maximální počet po sobě jdoucích iterací, ve kterých nedojde k vylepšení řešení. Tato kontrola je prováděna metodou *localSearchEndReached()*, která je v podmínce while cyklu v Algoritmu 3.

Restart spočívá v opětovném spuštění lokálního prohledávání s počátečním řešením shodným s nejlepším doposud nalezeným řešením *(X*)*, nebo nejlepším řešením z posledního běhu lokálního prohledávání *(X)*. Každé z těchto řešení má stejnou pravděpo-

Algoritmus 6 Přizpůsobení úrovně diverzifikace

Vstup: aktuální *dl*, číslo aktuální iterace *iter*, ohodnocení aktuálního řešení *f(X')*
Výstup: aktualizovaná hodnota *dl*

```
IF iter - adapIter > θ THEN
   dl = dl + (1 - dl)/6
   adapF = f(X'); adapIter = iter;
ELSE
   IF f(X') < adapF THEN
      dl = dl - dl/10
      adapF = f(X'); adapIter = iter;
   END IF
END IF
RETURN dl
```

dobnost zvolení. Následně je při restartu nastavena úroveň diverzifikace $dl = 1$. To zajistí provedení několika diverzifikačních kroků v prvních iteracích nového prohledávání.

6 Rozšíření algoritmu Adaptive Neghborhood Search

Po implementaci algoritmu tak, jak je popsán v článku [22], nebyla kvalita výsledků nijak vysoká a u složitějších instancí byly také problémy s velmi dlouhou dobou běhu algoritmu, která byla potřeba ke zlepšení výsledků. Po testování a analýze výsledků byly označeny hlavní příčiny problémů:

- velmi časté využití nedeterministických kroků v algoritmu
- absence využití rozvrhovací logiky
- náhodný výběr přechodu z obrovského prostoru řešení
- téměř nulová šance na provedení na sebe navazujících prohození (například prohození dvou po sobě jdoucích směn u jednoho zaměstnance - takzvané prohození bloků)

Za účelem eliminace těchto problémů byly navržena následující rozšíření algoritmu z předchozí sekce.

6.1 Kontrola schopností zaměstnance

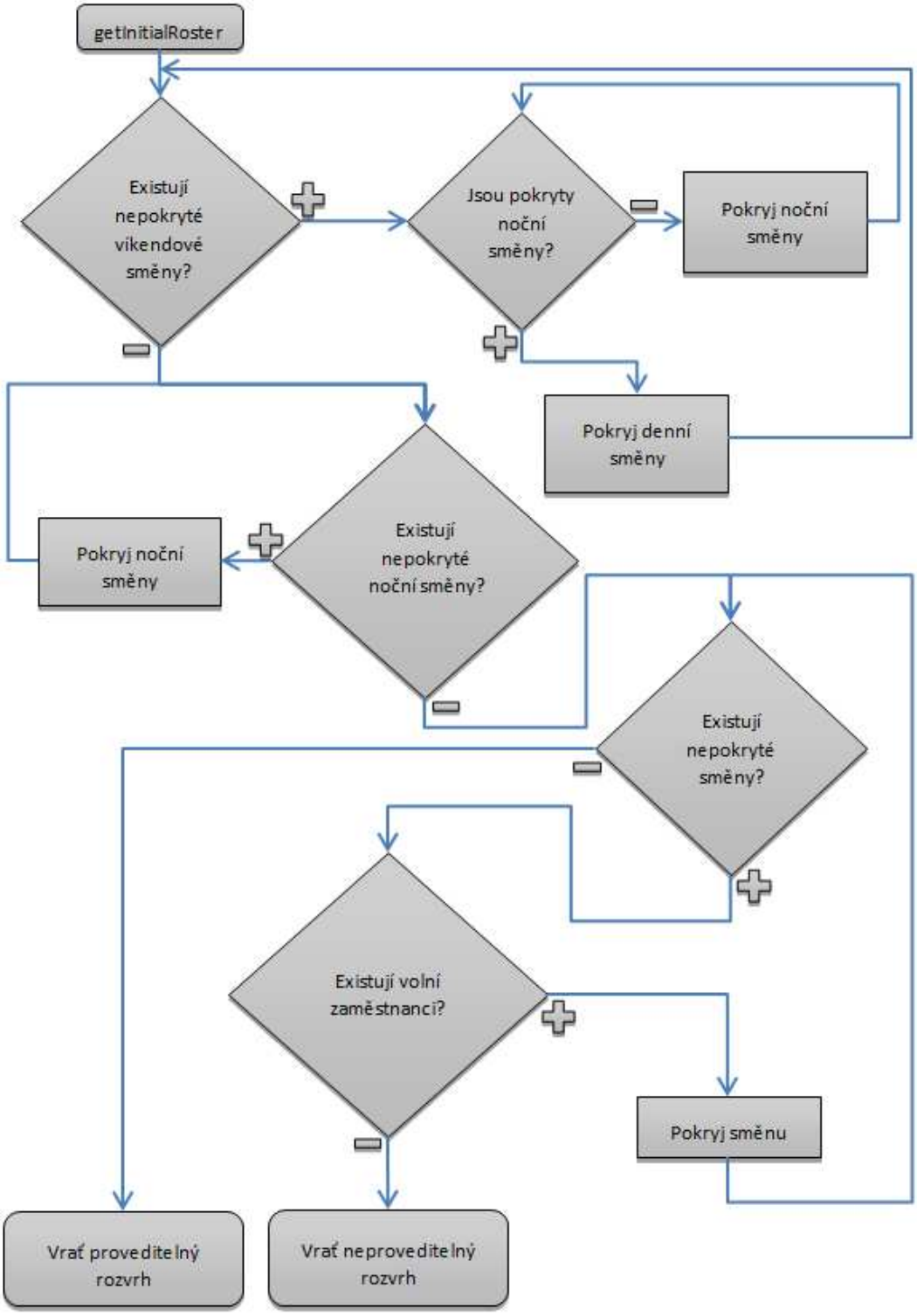
Velmi vážným problémem, který se však objevil až při testování instancí ze soutěže INRC [21], byla absence kontroly schopností zaměstnance, které udávají, že může na vybrané směně pracovat. Toto způsobovalo porušení tvrdých omezení a tedy vznik neproveditelného rozvrhu. U vybraných instancí z university v Nottinghamu [20] se problém neprojevil, jelikož u všech zaměstnanců využívají právě jednu úroveň schopností.

V rámci řešení bylo potřeba jednak zajistit vytvoření korektního rozvrhu na počátku algoritmu (tak aby neporušoval žádná tvrdá omezení) a poté upravit operace s rozvrhem tak, aby nedošlo k nepovolenému přiřazení směny k zaměstnanci bez potřebných schopností. Na obě tato místa byla tedy přidána kontrola ověřující požadovanou dovednost zaměstnance.

Přidání kontroly dovedností do metody pro generování sousedních řešení na řádku 9 v Algoritmu 3, mělo také vliv na snížení časové náročnosti algoritmu. Stalo se tak díky tomu, že před modifikací se do sousedství generovala prohození každého zaměstnance se všemi ostatními (pokud nebereme v úvahu další omezující podmínky), kdežto po úpravě se zaměňují směny pouze v rámci skupiny zaměstnanců, kteří mají schopnost potřebnou pro danou směnu.

6.2 Tvorba počátečního rozvrhu

Autoři výchozího algoritmu udávají, že kvalita počátečního rozvrhu má zanedbatelný vliv na výsledky algoritmu. I přes tento fakt bylo věnováno úsilí pro vývoj algoritmu pro nalezení počátečního rozvrhu. Vývojový diagram metody vracející počáteční rozvrh je na obrázku 2.



Obrázek 2: Vývojový diagram tvorby počátečního řešení

Při vytváření rozvrhu se nejprve přiřazují noční víkendové směny. Zaměstnanec, kterému bude směna přiřazena, se vybírá náhodně z podmnožiny všech zaměstnanců, která je určena pro pokrytí nočních víkendových směn. Výběr zaměstnanců do této podmnožiny záleží na dni, ve kterém je přiřazovaná směna. Pokud je to sobota, pak jsou přiřazováni zaměstnanci, kteří mají den předtím, tedy v pátek, volno (nemají přiřazenou žádnou směnu). Pokud je přiřazována nedělní noční směna, pak podmnožinu tvoří zaměstnanci, kteří mají o den dříve, tedy v sobotu, přiřazenu noční směnu. Vzhledem k tomu, že přiřazování provádíme od začátku rozvrhovaného období do konce, není potřeba kontrolovat, jaká směna navazuje, jelikož následující den mají všichni zaměstnanci volno. Pokud by došlo k situaci, že podmnožina zaměstnanců bude prázdná, pak by se vybíral zaměstnanec pro přiřazení z podmnožiny zaměstnanců pro pokrytí víkendových směn.

Výběr zaměstnanců pro pokrytí víkendových směn se opět rozděluje podle přiřazovaného dne. Pokud se jedná o sobotu, tak není požadavek na předchozí den specifikován - zaměstnanec může mít jak volný, tak pracovní pátek. Pokud budeme přiřazovat neděli, jsou vybráni pouze zaměstnanci s pracovní sobotou.

Po obsazení všech nočních víkendových směn se přiřazují ostatní směny, které připadají na víkend. Podmnožina zaměstnanců pro tyto směny byla již popsána. Pokud by nastala situace, že bude podmnožina prázdná, vybere se náhodně zaměstnanec z těch, kteří mají v daný den volno.

Dalším krokem po rozdělení víkendových směn je přiřazení směn nočních. Opět se postupuje od začátku rozvrhovaného období a zaměstnanec pro směnu se vybírá z podmnožiny zaměstnanců, kteří jsou vhodnými kandidáty na noční směnu. Tito zaměstnanci, kterým má být přiřazena noční směna v den *d*, musí splňovat jednu z následujících podmínek:

- v den *d-1* mají přiřazenu noční směnu
- v den *d-1* a zároveň v den *d-2* mají přiřazenu noční směnu
- v den *d-1* a zároveň v den *d-2* nemají přiřazenu žádnou směnu

Pokud by podmnožina takto vybraných zaměstnanců byla prázdná, přiřadí se ke směně náhodně vybraný zaměstnanec, který má volný den.

Po přiřazení všech nočních směn se pokrývají zbývající požadavky, do kterých jsou již zaměstnanci přiřazováni náhodně, tak jak tomu bylo v původním algoritmu u celého rozvrhu.

6.3 Bloky směn

Původní verze algoritmu vytvářela sousední řešení pomocí dvou typů prohození - Single shift a Shift pair, které byly popsány výše. V obou těchto případech se vždy jednalo o prohození směn v rámci stejného dne. V rozvrhu se však mohou objevit situace, které jsou pomocí těchto jednoduchých prohození velmi obtížně řešitelné. Jednu z nich například ukazuje obrázek 3, na kterém je část rozvhu, obsahující naplánované denní a noční směny dvou zaměstnanců v 5 dnech.

Jak je z obrázku 3 patrné, prohození vyznačených bloků o velikosti 3 by mělo přinést snížení penalizace. Pokud však budeme mít k dispozici původní verzi algoritmu, pak je docílení požadovaného prohození velmi nepravděpodobné. Příčinou je to, že provedení kteréhokoli dílčího prohození penalizaci zvýší:

- prohození u 10. dne způsobí vznik 2 nových volných dnů, okolo kterých budou dny pracovní - to je často relativně hodně penalizovaný pattern.
- prohození u 11. dne způsobí jednak vznik jednoho osamoceného volného dne, ale zejména také kombinaci N-D, která býva často vysoce penalizována. V reálném světě by se jednalo o porušení Zákoníku práce [19], jelikož mezi těmito směnami je pauza pouhé 2 hodiny, kdyžto minimální přestávka daná zákonem je 12 hodin, případně 8 ve vypsaných výjimečných případech.
- prohození u 12. dne je opět proti Zákoníku práce [19] a také pravděpodobně způsobí navýšení penalizace z důvodu velkého počtu směn u prvního zaměstnance a nízkého počtu směn u zaměstnance druhého.

Vzhledem ke zvýšení penalizace po těchto prohozeních není možné, aby byla provedena v rámci intenzivního nebo středního prohledávání. Řešení se zhoršujícím ohodnocením může být akceptováno pouze v rámci diverzifikačního prohledávání a dojde k tomu pouze v případě, že neexistuje žádné sousední řešení, které by vylepšovalo alespoň jedno měkké omezení. Když navíc vezmeme v úváhu velikost instance, tak například už u instancí s počtem zaměstnanců blížícím se 10, které rozvrhují 4týdenní období, je šance na prohození zmíněných dnů mizivá.

Pro řešení této a jí podobných situací byl algoritmus rozšířen o práci s bloky směn. První verze implementace práce s bloky umožňovala využívat bloky o velikosti v zadaných mezích při generování všech tří druhů sousedství. Nejefektivnější rozsah velikosti bloků byl vybrán na základě analýzy instancí a její výběr byl následně ověřen také testováním. Nejlepších výsledků dosahoval algoristmus s bloky o velikosti od 1 do 4 dnů.

Úprava s sebou přinesla také nevýhodu v podobě vysokého nárůstu prostoru řešení. Pro zjednodušení uvedu počet všech možných prohození při maximalní velikosti bloku 1 u instance GPost:

$$S = d \cdot \frac{n!}{k!(n-k)!} = 28 \cdot \frac{8!}{2!(8-2)!} = 784$$

Obrázek 3: Ukázka ideální aplikace blokového prohození směn

kde S je počet sousedních řešení, d je počet dní v rozvrhovaném období, n je počet zaměstnanců a k je hodnota udávající prohození vždy mezi dvěma zaměstnanci. Pokud použijeme bloky, jejichž velikost bude od 1 do 4, pak celkový počet možných prohození, odpovídající počtu sousedních řešení, bude

$$S = \sum_{i \in \langle 0,4)} (d-i) \cdot \frac{n!}{k!(n-k)!} = 28 \cdot 28 + 27 \cdot 28 + 26 \cdot 28 + 25 \cdot 28 = 2968$$

Velikost sousedství tedy roste téměř lineárně s navýšením velikosti bloků. Jelikož se pro každé sousední řešení musí spočítat penalizace, což je časově nejnáročnější operace v algoritmu, roste lineárně s velikostí bloků také čas běhu algoritmu potřebný k provedení zadaného počtu iterací. Jelikož ale testy potvrdily velký přínos bloků ve snížení penalizací u všech testovaných instancí, bylo potřeba časovou náročnost vyřešit a zároveň bloky zachovat.

Prvním krokem byla změna z definice rozsahu bloků na definici velikosti bloků, která se má použít při generování. Je tedy například možné generovat sousedství pouze za použití bloků směn o velikosti 2 a 4, místo nastavení rozsahu od 2 do 4, kde by byly navíc generovány bloky o velikosti 3.

Dále bylo testováno použití bloků pouze u některých druhů sousedství. Za účelem snížení velikosti prostoru řešení bylo porovnáno využití bloků ve všech typech sousedství, pouze ve středním prohledávání a nebo zároveň v intenzivním a diverzifikačním prohledávání.

Vzhledem k možnosti využít pouze zadané velikosti bloků a ne rozsahu velikostí bylo implementováno využití bloků na základě jejich dopadu na kvalitu řešení. Algoritmus má zadané velikosti bloků, které může používat, a na svém začátku je při generování sousedství využívá všechny. Vytváří si při tom statistiku, která obsahuje informace o tom, kolik zlepšujících řešení bylo nalezeno s každou velikostí bloku. Doba tohoto učení je nastavitelná a její varianty byly také testovány v rámci experimentů. Po dosažení zadaného počtu iterací (doba učení je specifikována jako část z celkového počtu iterací - např. třetina, polovina) se vyhodnotí výsledky tak, že se určí procentuální úspěšnost jednotlivých bloků (součet úspěšností je roven 100%). Na základě této úspěšnosti jsou bloky dále používány - nejprve se využívá nejúspěšnější blok a je využíván nejdéle. Počet iterací s tímto blokem je

$$I = Z \cdot B_1$$

kde I je počet iterací s daným blokem, Z je počet iterací zbývající do konce algoritmu ve chvíli, kdy bylo ukončeno učení, a B_1 je procentuální úspěšnost bloku. Následně se dle klesající úspešnosti střídají další velikosti bloků.

Finální stav implementace si kladl za cíl co nejvíce využívat nejefektivnější velikost bloku. Proto byla nastavena nejkratší doba učení (třetina celkového počtu iterací) a střídání bloků bylo změněno na jejich přidávání. Po ukončení učení je tedy používán pouze nejlepší blok. Jakmile proběhne počet iterací daných jeho úspěšností, je k němu přidán druhý nejlepší blok a jsou využívány oba. Například pokud bude celkový počet iterací algoritmu $Z = 150$, počet iterací pro učení nastaven na $U = 50$, velikosti bloků

od 1 do 4 a dosažené úspěšnosti bloků budou $B_1 = 20\%$, $B_2 = 15\%$, $B_3 = 30\%$, $B_4 = 40\%$, pak první bude blok o velikosti 4 a poběží 40 iterací, poté se k němu přidá blok o velikosti 3, po dalších 30 iteracích blok o velikosti 1 a na 15 posledních iterací i blok o velikosti 2. Poslední část algoritmu tedy opět využívá všechny velikosti bloků.

Velikost bloků se také projevila u dvou způsobů přechodu k sousednímu řešení, které algoritmus využívá a jsou popsány v kapitole 5.2.

U přesunu jedné směny k jinému zaměstnanci je podmínka, aby jeden ze zaměstnanců měl v den, kdy dochází k prohození, volno. U bloků o velikosti větší než 1 byla tato podmínka zachována pouze u prvního dne. Pokud by podmínka měla platit u všech dní v rámci bloku, omezila by radikálně sousedství daného rozvrhu, což by mělo špatný vliv na výsledky algoritmu. Pokud bude F reprezentovat volný den a S den, kdy má zaměstnanec přiřazenu směnu, pak je možné v v rámci tohoto přechodu provést prohození bloků *S-S-F-S* a *F-S-S-F*. Je tomu tak, jelikož první den je u jednoho zaměstnance pracovní (S) a u druhého volný (F). Nebylo by však možné prohodit například bloky *S-S-F-S* a *S-F-F-F*.

Podmínka druhého způsobu, který prohazuje dvě různé přiřazené směny, byla rozšířena i na bloky. Prohazují se tedy bloky směn mezi zaměstnanci, kteří mají v rámci každého dne v bloku směny odlišné. U bloku velikosti 1 se při použití tohoto způsobu přechodu nemohly vyskytovat volné dny, u větších bloků je to již možné. Podmínka je nebere v úvahu, nesmí však být na prvním dnu bloku. Pokud označíme denní směnu D, noční N a volný den F, může druhý způsob prohodit například bloky *D-D-F-D* a *N-N-N-N*. Není ale možné prohodit bloky *D-D-F-D* a *N-D-N-N* z důvodu shodných směn na druhé pozici, nebo bloky *D-D-F-D* a *F-N-N-N* z důvodu volného prvního dne u druhého zaměstnance.

6.4 Odstranění náhodných podmnožin zaměstnanců

Původní verze algoritmu pracovala v každém typu prohledávání s určitou skupinou zaměstnanců - při využití intenzivního prohledávání to byli všichni zaměstnanci, u středního a diverzifikačního prohledávání se jednalo o náhodně zvolenou polovinu zaměstnanců, která se vybírala vždy nově před každým generováním sousedství.

V modifikovaném algoritmu byl tento postup zachován pouze u diverzifikačního vyhledávání, kde se předpokládá potřeba dostat se z lokálního minima, pro což je náhodný výběr zaměstnanců dobrou volbou. Stejně tak velikost podmnožiny, která obsahuje polovinu všech zaměstnanců, je kompromisem mezi dostatečným výběrem a přiměřeným časem, potřebným pro ohodnocení jednotlivých sousedních řešení.

Střední prohledávání původně vybírá nejlepší nebo 2. nejlepší řešení ze sousedství, které je tvořeno přechody nad náhodně vybranou polovinou zaměstnanců. Jelikož se zaměřuje na výběr nejlepšího přechodu, byla podmnožina zaměstnanců upravena tak, aby obsahovala nejvíce penalizované zaměstnance. Počet zaměstnanců v podmnožině byl zachován na polovině z celkového počtu - výběr probíhá tak, že jsou všichni zaměstnanci seřazeni podle hodnoty své penalizace sestupně a do podmnožiny je zařazena první polovina seznamu.

Úkolem intenzivního prohledávání je vybrat ze sousedství takové řešení, které oproti současnému rozvrhu přinese co největší snížení penalizace. Z tohoto důvodu se sousedství tvoří z prohození mezi všemi zaměstnanci, což z něj dělá největší a z pohledu času algoritmu, stráveného ohodnocováním rozvrhů, také nejnáročnější sousedství. Lze však předpokládat, že největší pokles penalizace přinese prohození, ve kterém jako jeden ze zaměstnanců figuruje zaměstnanec, který má v aktuálním rozvrhu vysokou penalizaci. Na základě této myšlenky se pro generování sousedství využívá podmnožina, která tvoří třetinu nejvíce penalizovaných zaměstnanců z aktuálního rozvrhu. Podmnožina tedy může mít v každé iteraci jinou velikost, což je dáno tím, že se do ní nemohou dostat zaměstnanci s nulovou penalizací. Její maximální velikost je však rovna jedné třetině celkového počtu zaměstnanců. Můžeme proto říci, že tato úprava ušetří minimálně dvě třetiny času algoritmu, který se v původní variantě spotřeboval na ohodnocování sousedních řešení u intenzivního prohledávání.

6.5 Kontrola nepovolených kombinací směn

Jak už bylo zmíněno v sekci 6.3, existují kombinace směn, které se v rozvrhu nesmějí objevit, jinak by porušoval Zákoník práce [19]. Tyto kombinace zpravidla bývají u rozvrhovaných instancí definovány pomocí patternů a mají nastavenu relativně vysokou penalizaci v případě jejich výskytu. Výsledky testů však ukázaly, že se tyto kombinace i přesto v rozvrhu objevují. Algoritmus je pravděpodobně nedokázal eliminovat z důvodu vysoké míry náhodnosti výběru přechodu.

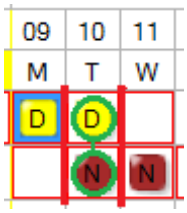
S cílem eliminace výskytu těchto patternů byla naimplementována kontrola nepovolených kombinací směn. Skládá se ze dvou částí, z inicializační a ze samotné kontroly vytvářených rozvrhů.

Inicializační část se provede na začátku celého algoritmu, ještě před spuštěním lokálního prohledávání. Vzájemně porovnává všechny typy směn (zajímají nás všechny variace o velikosti 2), které se v aktuální rozvrhované instanci vyskytují. Zaměřujeme se při tom na rozdíl mezi časem konce první směny a časem začátku druhé směny. Tento rozdíl musí být roven nebo větší než hodnota stanovená při konfiguraci algoritmu, v opačném případě je daná kombinace směn zařazena do seznamu nepovolených.

Tvorba seznamu nepovolených směn byla implementována dodatečně. Dříve se kontrolovala kombinace směn až při generování sousedního rozvrhu. Tato kontrola ale vedla k radikálnímu nárůstu časové náročnosti algoritmu, který byl způsoben častým porovnáváním hodnot typu DateTime.

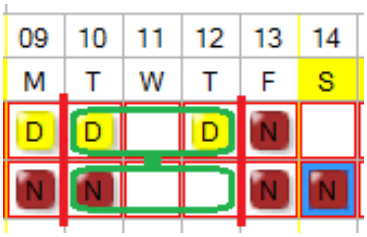
Samotná kontrola nepovolených směn probíhá při vytváření sousedních řešení. Při prohození směn v den D mezi zaměstnanci Z_1 a Z_2 se kontroluje u obou zaměstnanců kombinace se směnou v den *D-1* a v den *D+1*, jak je vidět na obrázku 4, kde jsou kontrolované kombinace směn odděleny tlustou červenou čárou. V tomto případě tedy dojde ke dvěma kontrolám. Maximálně může u jednoho prohození dojít ke 4 kontrolám navazujících směn.

Kontrola se provádí také při výměně větších bloků směn. Příklad ukazuje obrázek 5, kde dochází k prohození bloku o velikosti 3. Kontrola probíhá opět pouze u hraničních dnů, jak vyznačuje červená čára. Kontrolu kombinací směn uvnitř bloku není potřeba



Obrázek 4: Kontrola navazujících směn u bloku o velikosti 1

provádět, jelikož se kontrola prováděla jak při generování počátečního rozvrhu, tak při každém prohození. Ke vzniku nepovolených kombinací uvnitř bloku tedy nemohlo dojít.



Obrázek 5: Kontrola navazujících směn u bloku o velikosti 3

Pokud je při kontrole nalezena jedna nebo více nepovolených kombinací směn, pak není řešení, generované tímto prohozením, zařazeno do sousedství. To se o takto zahozená řešení zmenší a opět tím docílíme časové úspory při ohodnocování rozvrhů.

6.6 Odebrání nepenalizovaných zaměstnanců

Před generováním sousedství v každém ze tří druhů prohledávání, které algoritmus používá, dochází k omezení množiny zaměstnanců. Výběr zaměstnanců byl popsán v kapitole 6.4. Do algoritmu bylo přidáno ještě jedno filtrování zaměstnanců, které již není tak radikální a provádí se až těsně před přidáním nového rozvrhu do sousedství. Jedná se o vyřazení zaměstnanců s nulovou penalizací.

Předpokládaný cíl prohození směn nebo bloků směn mezi dvěma zaměstnanci, je snížení penalizace celého rozvrhu. Každé prohození však může ovlivnit pouze penalizace zaměstnanců, kteří se ho účastní. Provádíme-li prohození mezi zaměstnanci, kteří mají oba hodnotu penalizace rovnu nule, není možné, aby prohození vedlo k lepšímu řešení. Proto takovéto kombinace do sousedství nejsou vloženy a tím je opět ušetřen čas na ohodnocování rozvrhů.

Pokud má nulovou penalizaci pouze jeden ze zaměstnanců, prohození bude testováno. V této situaci může dojít k tomu, že součet penalizací obou zaměstnanců po prohození bude nižší, než původní penalizace jednoho zaměstnance, a tedy dojde k nalezení lepšího rozvrhu.

Tento typ filtrování zaměstnanců se nepoužívá u diverzifikačního prohledávání, jelikož by mohl diverzifikaci omezit.

6.7 Odebrání nepenalizovaných dnů

Další možnost, jak předejít zbytečnému ohodnocení rozvrhu, o kterém dopředu víme, že nenabízí lepší penalizaci, je filtrace prohození, které připadají na nepenalizované dny. Algoritmus sice nabízí u každého zaměstnance seznam porušených podmínek, který obsahuje jejich jednotlivé penalizace a dny v rozvrhovaném období, kdy došlo k porušení. Tento seznam je ale aktualizován pouze metodou rozvrhu *RecalculateAllPenaltiesAndUpdateViolations*, která je časově velmi náročná a není možné ji pouštět po provedení každého prohození.

Pro zjištění, zda je některý den u daného zaměstnance penalizovaný, byl algoritmus rozšířen o nový atribut ve třídě *Employee*. Jedná se o *PenalizedDay* - pole typu bool, jehož velikost je shodná s počtem dnů rozvrhovaného období. Hodnota *True* na indexu i říká, že daný zaměstnanec má v den s indexem i nějakou penalizaci. Není možné říci, že má přiřazenu směnu, za kterou je penalizován, jelikož penalizace může být také například za jeden samostatně stojící den volna. Toto pole se aktualizuje při volání metody *RecalculatePenalty* třídy *Employee*.

Před samotným provedením prohození se poté kontroluje, zda daný den je alespoň u jednoho ze zaměstnanců penalizovaný. Pokud tomu tak není, prohození nebude provedeno. U bloků je podmínkou pro provedení prohození výskyt alespoň jednoho penalizovaného dne v rámci bloku.

Tuto filtraci můžeme provádět díky znalosti toho, že v používaném algoritmuu se penalizace váže na všechny dny, které ji způsobují. Pro ujasnění uvedu příklad omezení maximálního počtu po sobě jdoucích pracovních dnů na 5. Pokud by zaměstnanec měl přiřazeny směny od pondělí až do neděle každý den, pak by také pole *PenalizedDay* obsahovalo hodnotu *True* pro každý den od pondělí až do neděle. Nikoliv pouze pro sobotu a neděli, jakožto 1. a 2. den, který překračuje stanovené omezení. Díky tomuto faktu víme, že prohození mezi dny, které u obou zaměstnanců nejsou penalizovány, nemůže přinést vylepšení rozvrhu.

Stejně jako předcházející omezení, tak i kontrola nepenalizovaných dnů není použita u diverzifikačního prohledávání.

6.8 Modifikace úrovně diverzifikace

ANS algoritmus používá přizpůsobení úrovně diverzifikace na základě aktuální hodnoty parametru *dl* - diversification level, který je detailněji popsán v kapitole 5.7. Ke zvyšování diverzifikace dochází ve chvíli, kdy počet iterací bez zlepšení překročí parametr θ, který se určuje z počtu směn a zaměstnanců u dané instance problému.

U testovacích instancí bylo zjištěno, že v některých případech je hodnota tohoto parametru příliš nízká a pro jiné instance naopak příliš vysoká. Hodnoty parametru pro testované instance ukazuje tabulka 1.

Instance	θ
Millar-1	2
GPost	2
SINTEF	13
Valouxis	5
WHPP	10

Tabulka 1: Hodnoty parametru θ u testovacích instancí

Ideální hodnota parametru byla v intervalu $< 5, 10 >$, proto byl parametr θ v algoritmu zaměněn za θ', který ke svému výpočtu využívá hodnotu původního parametru:

$$\theta' = min(max(5, \theta), 10)$$

Změna má za cíl lepší výsledky intenzivního prohledávání u menších instancí a snížení délky běhu algoritmu u větších instancí.

6.9 Úprava mechanizmu restartu

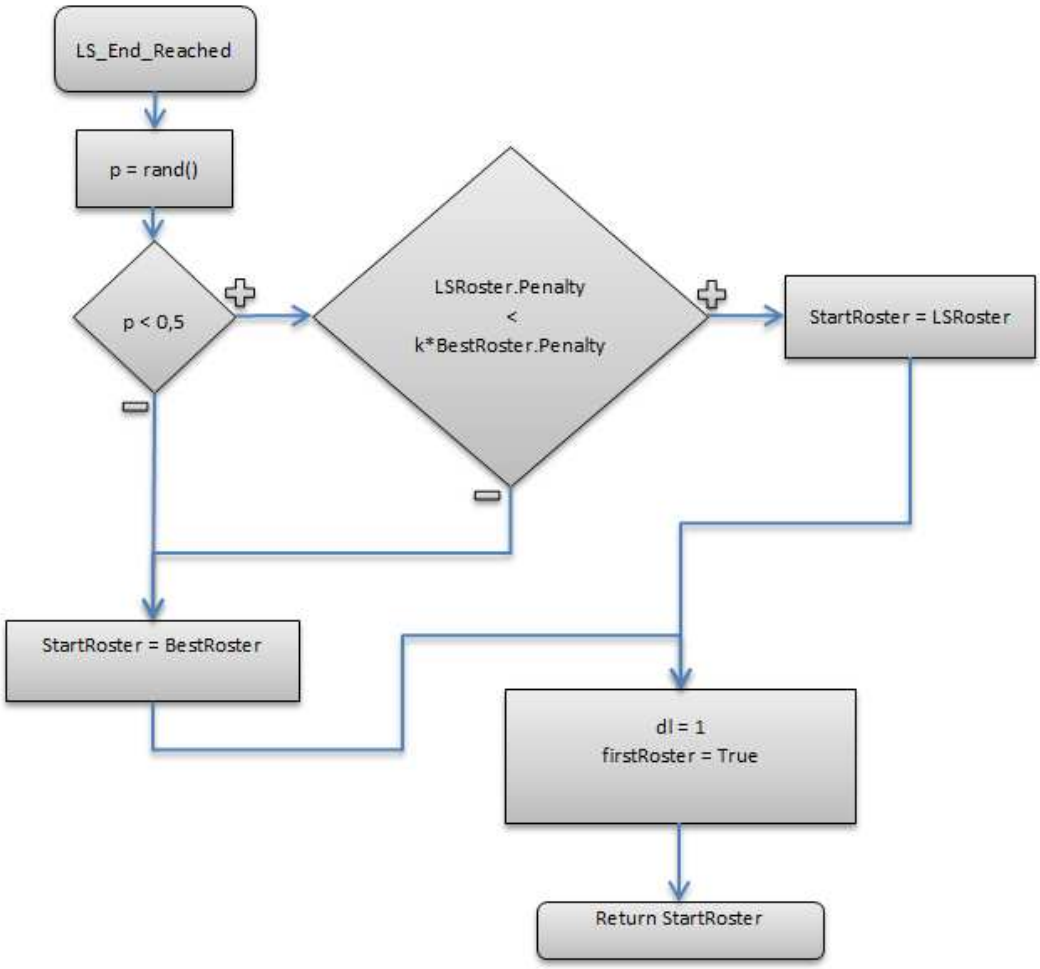
ANS algoritmus využívá mechanizmus restartu po daném počtu iterací lokálního prohledávání, během kterých nedošlo ke zlepšení penalizace. Při restartu se náhodně zvolil buď doposud nejlepší rozvrh, nebo nejlepší rozvrh z posledního běhu lokálního prohledávání. Ten byl nastaven jako počáteční a opět bylo spuštěno lokální prohledávání. Upravený postup algoritmu při restartu ukazuje vývojový diagram na obrázku 6.

Po ukončení lokálního prohledávání se se shodnou pravděpodobností rozhodujeme mezi výběrem celkově nějlepšího rozvrhu *BestRoster*, nebo nejlepšího rozvrhu z poslední běhu lokálního prohledávání *LSRoster*. Pokud je vybrán *LSRoster*, provede se porovnání jeho penalizace s *BestRoster* a v případě, že je jeho ohodnocení k-krát vyšší, bude jako nový počáteční rozvrh zvolen *BestRoster*. Tato kontrola zabraňuje opakovanému hledání zlepšení u rozvrhu, který je mnohonásobně horší a obsahuje nějaký závažný problém, který nebyl během jednoho běhu lokálního prohledávání odstraněn. Parametr k byl při testování nastaven na hodnotu 2.

Po výběru nového počátečního rozvrhu se stejně jako v původní verzi algoritmu nastaví level diverzifikace na nejvyšší hodnotu. Díky tomu bude na začátku LS využito diverzifikační prohledávání. Nezaručuje to však přijetí jakéhokoliv nalezeného rozvrhu. Pro zvýšení efektu diverzifikace byl přidán parametr *firstRoster*, který lokálnímu prohledávání říká, že aktuálně hledá první rozvrh a ten musí být přijat bez ohledu na jeho ohodnocení.

6.10 Shrnutí úprav algoritmu

Popsané úpravy byly navrhovány postupně na základě dosažených ohodnocení na testovacích instancích. Obecně je možné je rozdělit na úpravy, které měly hlavní cíl snížit časovou náročnost algoritmu, a úpravy, které byly zaměřeny na výrazné zlepšení



Obrázek 6: Vývojový diagram provedení restartu algoritmu

penalizace. Do druhé skupiny patří zejména úprava generování počátečního rozvrhu a také implementace práce s bloky. Délka běhu algoritmu a penalizace výsledného rozvrhu jsou však úzce propojeny a většina úprav měla vliv na obě tyto hodnoty.

7 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)

19. Zákoník práce, Zákon č. 262/2006 Sb (http://portal.gov.cz/app/zakony/zakon?q=262/2006)

20. Nottingham University, Employee Scheduling Benchmark Data Sets. (http://www.cs.nott.ac.uk/~tec/NRP/)

21. First International Nurse Rostering Competition, 2010. (http://www.kuleuven-kulak.be/nrpcompetition)

22. Zhipeng Lü, Jin-Kao Hao, Adaptive neighborhood search for nurse rostering, European Journal of Operational Research, 2012, 218, 3, Pages 865876 (http://www.sciencedirect.com/science/article/pii/S0377221711010939)