FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Systém pro konsolidaci virtuálních počítačů
Student:	Bc. Michal Polák
Vedoucí:	Ing. Jan Fesl, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Datová centra umožňují na svých infrastrukturách spouštění velkého množství virtuálních strojů, které fyzicky běží na virtualizační uzlech. Vlivem postupného zapínání a vypínání virtuálních strojů dochází k tomu, že rozmístění těchto strojů dle určitého kritéria (např. minimální celková spotřeba el. energie, maximální výpočetní výkon infrastruktury či rovnoměrné zatížení virt. uzlů) není optimální.
Problém konsolidace virtuálních strojů je obecně NP-těžký a pro jeho řešení lze využít obecných existujících algoritmů pro nalezení globálního optima.
V rámci rešeršní části této diplomové práce prostudujte běžně známé algoritmy pro konsolidaci virtuálních počítačů s ohledem na minimalizaci zatížení používané síťové infrastruktury, vyberte popř. navrhněte vhodné algoritmy a mezi sebou je experimentálně porovnejte. Pro nejlepší nalezené řešení vytvořte implementaci v jazyce C++ či Python umožňující její reálné nasazení v prostředí systému OpenNebula. Implementaci otestujte a vyhodnoťte.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 24. února 2022 v Praze.

Diplomová práce

SYSTÉM PRO KONSOLIDACI VIRTUÁLNÍCH POČÍTAČŮ

Bc. Michal Polák

Fakulta informačních technologií
Katedra počítačových systémů
Vedoucí: Ing. Jan Fesl, Ph.D.
5. ledna 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Seznam pseudokódů

Chtěl bych touto cestou poděkovat své rodině a blízkým za jejich dlouhodobě trvající podporu a důvěru. Mé díky dále patří i spolužákům a přátelům, se kterými jsem mohl sdílet strasti i radosti studia. Ohromné díky patří vedoucímu této práce, panu Ing. Janu Feslovi, Ph.D., za jeho neuvěřitelnou trpělivost, ochotu diskutovat mé nápady a snahu dovést mě dál, než k této závěrečné práci. Poslední poděkování bych pak rád směřoval ke svým „gumovým kachničkám“, za všechny jimi vnuknuté i zamítnuté nápady.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 5. ledna 2023 .....

Abstrakt

V této práci se zabývám problémem konsolidace virtuálních strojů se zaměřením na optimalizaci zatížení síťové infrastruktury. Tento problém definuji a uvádím několik možných postupů řešení. Ty pak experimentálně otestuji na náhodně vygenerovaných datech a vyhodnotím.

Klíčová slova konsolidace virtuálních strojů, problém plnění přihrádek, virtualizace, SCIP, genetický algoritmus

Abstract

This thesis is dedicated to the problem of the virtual machines consolidation with a focus on optimizing the load of the network infrastructure. I define this problem and present several possible solutions. Finally those methods are experimentally tested on the randomly generated data for the purpose of evaluation.

Keywords virtual machine consolidation, bin packing problem, virtualization, SCIP, genetic algorithm

Seznam zkratek

SLA	Service Level Agreement
VU	Virtualizační uzel
SP	Síťový prvek
VS	Virtuální stroj
LV	Logická vzdálenost
SAT	Problém splnitelnosti logické formule
PBO	Pseudo boolean optimalizace
API	Aplikační rozhraní

Úvod

Jedním z významných pokroků v oblasti informatiky a technologii počítačů byl bezesporu vynález virtualizace. Tato technika, umožňující rozdělení fyzických prostředků jednoho fyzického stroje mezi více operačních systémů či logických kontejnerů, umožnila vznik modelu poskytování výpočetní či datové infrastuktury jako služeb. V návaznosti na to začalo vznikat znatelné množství tzv. datacenter, které tyto služby poskytují.

Mezi největší výhody tohoto modelu z hlediska zákazníků je možnost „outsourcingu“ (svěření problému externímu dodavateli těchto služeb) správy fyzické infrastuktury. Dalším z potenciálních důvodů využití služeb datacenter je škálovatelnost. Uživatel služeb datacentra může podle potřeby navýšit či snížit pronajaté množství například výpočetního výkonu, pokud se domnívá, že ho bude či nebude potřebovat. V poslední řadě bych rád zmínil výhodu spolehlivosti. Datacentra jsou často budována tak, aby předešla možným výpadkům jejich služeb, dostupnosti či ztrátě dat. Zároveň jsou často vybavena nejmodernějšími technologiemi, což například u bezpečnostních prvků může být velká výhoda. Menší společnosti či subjekty často nedisponují finančními prostředky, aby si mohli tyto technologie dovolit. Využití služeb datacentra pak může být nejen technicky lepší, ale v konečném důsledku i levnější alternativou.

Z pozice datacentra však vznikají různé dílčí problémy, které je třeba řešit. Například jedním ze způsobů konkurenčního boje jsou tzv. **SLA** neboli *Service level agreement*. Jedná se o dohodu mezi datacentrem a zákazníkem, která mimo jiné vymezuje rozsah dostupnosti poskytovaných služeb. Často tato dohoda obsahuje i maximální dobu nedostupnosti služeb ročně a při jejím nedodržení se datacentrum může vystavit smluvním pokutám. Zároveň existuje nezávislá organizace zabývající se hodnocením datacenter[1]. Pro dosažení nejvyšších standardů si může datacentrum dovolit maximálně nižší desítky minut nedostupnosti ročně. (Nedostupnost může vzniknout nejen výpadkem či závadou, ale také přetížením infrastruktury datacentra.) S tím souvisí další z dílčích problémů datacenter: efektivní rozložení zátěže. Asi největším provozním nákladem datacenter je spotřeba elektrického proudu, která se skládá ze dvou hlavních částí: samotná spotřeba hardware a spotřeba podpůrných zařízení např. chlazení. Rozložení zátěže tak může mít výrazný vliv na celkovou spotřebu a tím pádem i provozní náklady. Při dlouhodobé zátěži pak nelze zanedbat ani opotřebení hardware.

Zde je třeba zmínit ještě jeden klíčový faktor a sice proměnlivost zátěže datacenter. Jedná se o nutný důsledek již zmíněné škálovatelnosti z pohledu klientů. Při prvním spuštění je datacentrum bez jakékoliv zátěže. Ta vzniká průběžně, postupným vytvářením virtuálních instancí na požadavky jednotlivých zákazníků. Nárůst této zátěže lze však předpokládat jen obtížně. Další komplikací je ale zánik či případné změny nároků na výpočetní zdroje jednotlivých virtuálních instancí. Pokud například zákazník sníží množství požadované operační paměti, může potenciálně dojít ke vzniku neoptimálního využití celkového výkonu. Naopak v případě požadavku na navýšení pronajatých prostředků může nastat situace, kdy požadované prostředky nejsou

[1] Uptime Institute

na aktuálním fyzickém stroji dostupné a tudíž je potřeba virtuální instanci přemístit (tzv. migrace). Důsledkem těchto vlivů jsou však neustálé změny ve využití dostupných prostředků datacentra.

Podle mého názoru se jedná o velmi rozsáhlou problematiku. Její jednotlivé části jsou předmětem mnoha výzkumů a případná řešení mohou snadno nalézt uplatnění v praxi. Oblast poskytování služeb datacenter je na vzestupu a domnívám se, že tento trend bude pokračovat i v nejbližší budoucnosti. V rámci této práce jsem tedy navázal na svůj předešlý výzkum [1] a pokusil se ho rozšířit.

Kapitola 1

Cíl práce

Jak jsem již zmínil, tato práce rozšiřuje můj předešlý výzkum dané problematiky [1]. Zaměřuji se zde na konkrétní optimalizační kritérium problematiky konsolidace virtuálních strojů. Definuji vlastní optimalizační kritérium a navrhuji několik možných metod jeho řešení. Některé z daných metod jsem sám implementoval, v jiném případě jsem využil již existujícího a osvědčeného řešení, určeného k hledání a optimalizaci matematických problémů.

Hlavními částmi této práce pak jsou:

- formalizace problému
- generování testovacích dat
- popis jednotlivých metod řešení
- experimentální zhodnocení
- implementace umožňující reálné otestování

Osobně bych pak rád touto prací posunul svůj výzkum oblasti konsolidace virtuálních strojů. Domnívám se však, že potenciální celkový rozsah tohoto výzkumu přesahuje rozsah této práce.

Kapitola 2

Nastínění problematiky a definice problému

2.1 Úvod do terminologie

Virtualizační infrastruktura datacentra se může skládat z mnoha prvků, nejen samotný virtualizační hardware, ale i přidružená síťová zařízení a další. Pro účely této práce však není nutné detailní dělení a tudíž nám postačí zařízení rozdělit pouze do 2 kategorií:

- Virtualizační uzel
- Síťový prvkek

Virtualizační uzel (**VU**) je fyzický stroj (nejčastěji výkonný server), na kterém běží tzv. hypervizor neboli software zajišťující virtualizaci. Ten má určitý omezený počet zdrojů (například výpočetních jader procesoru, či množství operační paměti), které může přidělit (na něm umístěným) virtuálním instancím. Virtualizované mohou teoreticky být i jednotlivé části síťové infrastruktury, pro účely této práce je však nemusíme brát v úvahu (více v 2.4).

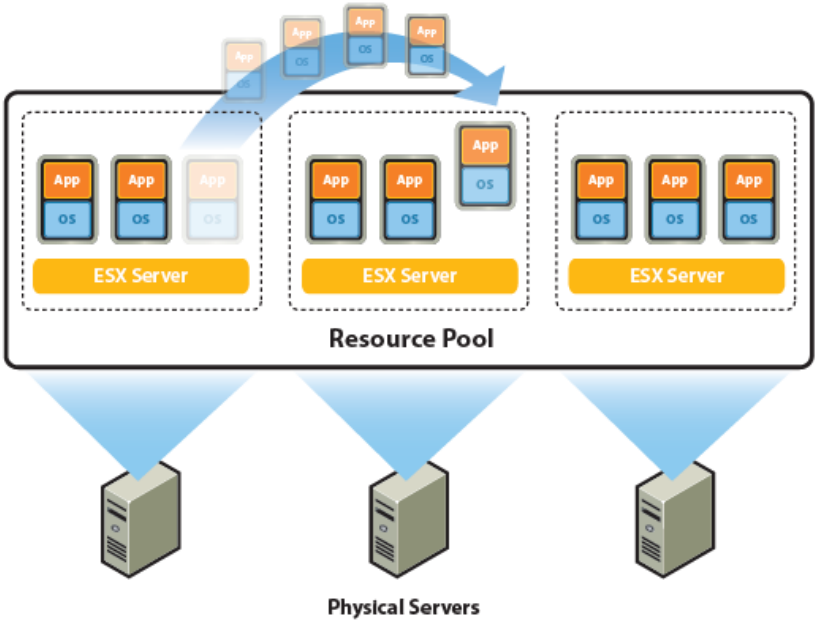
Síťový prvek (**SP**) je fyzický či virtuální stroj (například router, switch či firewall), který mimo jiné zajišťuje konektivitu mezi virtualizačními uzly, či jejich připojení k internetu.

Virtuální stroj (**VS**) je logický stroj, který běží pomocí hypervizoru na nějakém virtualizačním uzlu. Má konkrétní nároky na množství zdrojů. Může se nacházet v různých stavech, například ho lze pozastavit v rámci migrace.

Hypervizor je software zajišťující virtualizaci. Stará se o přidělování fyzických prostředků virtualizačního uzlu a správu virtuálních strojů (například spouštění, zastavování a migrace). Existují různé druhy hypervizorů i virtualizačních technologií, jejich rozdíly a výhody či nevýhody jsou však pro tuto práci nepodstatné.

Migrace je proces přemístění virtuálního stroje z původního virtualizačního uzlu na jiný. Existují různé způsoby migrace běžících i pozastavených. Přesun probíhá po přidružené síťové infrastruktuře, čímž nevyhnutelně dojde k jejímu (alespoň částečnému) zatížení. Množství dat, přenesené po síti se může lišit v závislosti na typu migrace i množství zdrojů migrovaného virtuálního stroje. Typicky však největší část přenesených dat tvoří operační paměť či data diskového úložiště.

Vizuální znázornění migrace mezi dvěma virtualizačními uzly reprezentuje obrázek 2.1.



Obrázek 2.1 Nákres migrace virtuálního stroje. Zdroj: [2]

2.2 Techniky optimalizace rozmístění virtuálních strojů

Obecně existují dva přístupy rozmístění virtuálních strojů nezávisle na kritériu, podle kterého chceme virtualizační infrastrukturu „optimalizovat“: Rozmisťování (anglicky *Virtual machine placement*) a konsolidace (anglicky *Virtual machine consolidation*).

2.2.1 Rozmisťování virtuálních strojů

Jedná se o jednodušší z těchto technik. Při vytvoření nové instance virtuálního stroje dojde za pomocí algoritmu či heuristiky k výběru (domněle) optimálního umístění, které je pak danou instancí zaplněno. Dále se však umístění tohoto virtuálního stroje nebere v potaz. Pokud tedy vlivem zániků a vzniků ostatních virtuálních strojů toto umístění přestane být vhodným, tato metoda to neřeší. Důsledkem je, že vzniká jev velmi podobný fragmentaci dat či systému souborů, kdy nerovnoměrné rozmístění může vést k plýtvání dostupnými zdroji. V extrémním případě pak může dojít k situaci, kdy při pokusu o vytvoření jiného virtuálního stroje nejsou dostupné dostatečné prostředky na žádném z virtualizačních uzlů, při přeuspořádání ostatních virtualních strojů již dostupné být mohou.

Někdy bývá tento postup aplikován nejen při vytvoření virtuálního stroje, ale i při jeho migraci (například v [3]). V takovém případě migrace slouží jako nástroj k uvolnění výpočetních prostředků na virtualizačním uzlu a dochází k hledání alternativního optimálního umístění migrovaného virtuálního stroje.

2.2.2 Konsolidace virtuálních strojů

Alternativní technikou je konsolidace virtuálních strojů. Pokud využiji již dřívější přirovnání k fragmentaci souborového systému, konsolidace lze přirovnat k procesu defragmentace. Tento proces může být zahájen při hledání místa pro vytvoření nového virtuálního stroje, ale i například při pokusu o optimalizaci zatížení celé virtualizační infrastruktury (například za účelem vypnutí

některých virtualizačních uzlů kvůli spotřebě energie). Tento proces se pak pomocí algoritmů či heuristik snaží nalézt optimální umístění všech virtuálních strojů naráz napříč virtualizační infrastrukturou. Nevýhodou tohoto přístupu však je, že může vést k velkému množství migrací a tudíž způsobit výrazně vyšší zatížení přidružené síťové infrastruktury či v extrémním případě narušení SLA.

Tento text se dále zabývá pouze konsolidací.

2.3 Definice problému konsolidace virtuálních strojů

Problém konsolidace virtuálních strojů tudíž spočívá v nalezení takového rozmístění virtuálních strojů na virtualizační uzly, aby nedošlo k překročení jejich dostupných prostředků. Problém může být navíc rozšířen o tzv. optimalizační kritérium. To představuje pomyslné zlepšení aktuálního stavu, kterého se konsolidace snaží dosáhnout. Více se optimalizačnímu kritériu věnuje samostatná část 2.4.

2.3.1 Prostředky virtualizačních uzlů

V první řadě je třeba definovat prostředky, které hypervizor přiděluje jednotlivým virtuálním strojům. Těmi může být prakticky jakýkoliv nárok virtuálního stroje, například: počet jader procesoru, velikost operační paměti, velikost diskového prostoru, počet síťových rozhraní, dostupné grafické procesory apod. V praxi jsou přípustné prostředky definovány fyzickou realizací virtualizačních uzlů, která navíc musí být podporována hypervizorem. Ne nutně však musí existovat vztah 1 ku 1. Například pokud není nutné dosáhnout maximální propustnosti síťového rozhraní virtuálního stroje, může víc virtuálních síťových rozhraní být hypervizorem „mapováno“ na jediné rozhraní fyzické (opět v tomto případě záleží hlavně na typu použité virtualizace a hypervizoru). Jako další příklad bych rád uvedl různé možnosti realizace diskového prostoru. Jednodušší alternativa v tomto případě je, že každý z virtualizačních uzlů má vlastní diskový prostor, který poté přiděluje hypervizor. Tato metoda má výhodu rychlejšího přístupu virtuálního stroje k jeho datům. V případě migrace se však zvyšuje objem dat přenesených mezi virtualizačními uzly (a v případě tzv. *offline* migrace[1] i doba do opětovného spuštění virtuálního stroje). Oproti tomu existují technologie, že virtualizační uzly sdílí úložný prostor (většinou separátní fyzické zařízení, připojené přes vysokorychlostní linky. V takovém případě při migraci není nutné přesunout diskový prostor virtuálního stroje. Toto řešení ale pravděpodobně bude mít vliv na rychlost přístupu k takto uloženým datům.

Z tohoto důvodu jsem se tedy rozhodl brát v úvahu pouze nutné prostředky, které z principů mě známých nemohou a nebo z praktických důvodů nebývají sdíleny mezi virtualizačními uzly: počet jader procesoru a velikost operační paměti.

Obecně pak lze přidat další prostředky virtuálních strojů a tudíž i další *omezující kritéria* (viz níže). V takovém případě by s patřičnými úpravami řešení navrhovaná v této práci měla být aplikovatelná obdobným způsobem.

Zároveň z důvodu jednoduchosti, tato práce předpokládá homogenitu virtualizačních uzlů, tj. předpokládá, že všechny **VU** mají dostupné stejné celkové množství prostředků. Tento předpoklad v praxi nemusí být úplně neobvyklý. Datová centra často nakupují hardware ve velkém počtu totožných kusů. To mimo jiné usnadňuje automatizaci instalací potřebného softwarového vybavení (například ovladačů hardware) či se tím eliminují možné problémy s kompatibilitou různých implementací a architektur. Zároveň se tím zmenšuje množství naskladněného náhradního hardware za účelem urychlení reakce na případné závady. Opět při provedení patřičných úprav, by veškeré metody měly být použitelné.

[1] migrace vypnutého virtuálního stroje

V poslední řadě, tato práce předpokládá, že každý **VS** je umístěn pouze na jednom **VU** a tudíž neprobíhá žádná migrace. Tento předpoklad nemá žádný vliv na obecnost, jelikož v případě probíhajících migrací či vytváření **VS** lze vždy nejdříve počkat na dokončení těchto procesů.

▶ **Definice 2.1** (Omezující kritéria)**.** *Nechť u je **VU** a S^u je množina všech **VS** umístěných na **VU** u. Dále uznačme u_{CPU} jako množství fyzických jader procesoru dostupné **VU** u a ekvivalentně u_{RAM} jako množství operační paměti. Obdobně označme s_{CPU} respektive s_{RAM} jako požadované množství jader procesoru respektive operační paměti **VS** s. Pak omezujícími kritérii jsou:*

$$\forall u : u_{CPU} \geq \sum_{s \in S^u} s_{CPU}$$

$$\forall u : u_{RAM} \geq \sum_{s \in S^u} s_{RAM}$$

$$\forall s : (s \in S^u) \wedge (s \in S^v) \Rightarrow (u = v)$$

*Aneb žádný **VU** nemůže přidělit **VS** více prostředků, než má k dispozici, a žádný **VS** není na více **VU** zároveň.*

2.3.2 Podobnost problému s problémem plnění přihrádek

Čtenáři, který se orientuje v oblasti teoreické informatiky možná neunikla jistá podobnost se známým problémem označovaným jako „plnění přihrádek“ (anglicky *Bin packing problem*). Efektivně se jedná o jeho vícerozměrnou variantu, v tomto případě 2. (Každý typ prostředků přidělovaných virtualním strojům přidává vlastní dimenzi problému.)

Tento problém (v klasické i dvoudimenzionální variantě) je NP-těžký respektive NP-úplný [4, 5]. Obecně pro dostatečně velké instance nelze nalézt exaktní řešení v polynomiálním čase. Tudíž prakticky není možné nalézt toto řešení dostatečně rychle pro jeho využití.

Pro úplnost tohoto textu však uvádím i přímý důkaz NP-úplnosti problému konsolidace virtuálních strojů definovaném v této kapitole.

2.4 Optimalizační kritéria

Cílem konsolidace virtuálních strojů pak nejčastěji bývá optimalizace využití prostředků virtualizační struktury. Mnoho výzkumných publikací se zabývá různými kritérii a algoritmy hledání jejich optimálních stavů.

2.4.1 Související práce a dosavadní výzkum

Jak již bylo v této práci nastíněno, typickou motivací ke konsolidaci virtuálních strojů je úspora spotřeby elektrického proudu. Obsáhlý výzkum tohoto tématu udělal A. Beloglazov, který v [6] navrhl hned několik modelů a algoritmů za účelem energetické úspory datacentra pomocí migrací a vypínání či zapínání virtualizačních uzlů, dle aktuálního zatížení. Za zmínku stojí, že jeho práce zvažuje i negativní vliv na výkon datacentra a snaží se udržet „rovnováhu“ těchto dvou kritérií. Z dalších výzkumů zabývající se snížením energetické náročnosti stojí za zmínku například [7], kteří berou v úvahu nejen energetické nároky jednotlivých virtualizačních uzlů ale i případné dopady na **SLA**. Zajímavý je i výzkum [8] zabývající se heuristikami, které mimo jiné zvažují i tepelné emise hardware virtualizačních uzlů. Řešení navrhující [9] naopak bere v potaz i redundanci prostředků za účelem zvýšení odolnosti datacentra vůči závadám.

Je možné dohledat i mnoho dalších vědeckých publikací, zabývajících se různými kombanicemi kritérií optimalizace, to však není cílem této práce. Ta je zaměřena především na optimalizaci

zatížení přidružené síťové infrastuktury datacentra. Z podobných výzkumů stojí za zmínku například [10] zabývající se rovnoměrným rozložením zatížení síťových linek či [11] a [12], kteří se zabývají optimalizací síťového provozu pomocí využití fyzické vzdálenosti virtualizačních uzlů či clusterizací virtuálních strojů do skupin a jejich rozmístění do tzv. racků[2] (v tomto pořadí).

2.4.2 Návrh optimalizačního kritéria

V předcházejícím výzkumu [1] jsem navrhl optimalizační kritérium uvažující nejen fyzickou vzdálenost virtualizačních uzlů, ale i množství dat vyměnené v komunikaci virtuálních strojů. Toto kritérium má za cíl minimalizovat celkovou zátěž přidružené síťové infrastruktury a zároveň snížit dobu odezvy (a tudíž i dobu potřebnou k doručení dat) mezi virtuálními stroji, které spolu komunikují nejvíce. Tohoto kritéria se nadále držím i v této práci.

Pro účely tohoto kritéria je nutné nejdříve definovat pojem logická vzdálenost (nadále označovaná také jako **LV**). V kontextu této práce je tímto pojmem myšlen počet síťových prvků mezi dvěma virtualizačními uzly, kterými prochází síťová komunikace mezi dvojicemi virtuálních strojů umístěných na těchto **VU**. Logická vzdálenost přímo ovlivňuje dobu doručení dat mezi dvěma virtuálními stroji. Každý síťový prvek (fyzický i virtualizovaný) musí totiž příchozí data (například ve formě rámce či paketu [3]) vyhodnotit na základě příslušné hlavičky a rozhodnout o jejich případném poslání dále. Ačkoliv výrobci síťových prvků neustále zvyšují rychlost a propustnost svých zařízení, toto zpracování z principu trvá nějaký čas. Rozdíl, pokud paket projde jeden nebo několik desítek routerů (navíc potenciálně používajících různé protokoly, tunelovací technologie - například VPN či bezpečnostní nastavení) pak může být velmi znatelný. Navíc, pokud si dva virtuální stroje dlouhodobě vyměnují větší množství dat, zmenšení logické vzdálenosti těchto **VS** může mít za následek snížení celkové zátěže jednotlivých síťových prvků (například zmenšením množství překladů adres mezi jednotlivými segmenty sítě).

▶ **Definice 2.2** (Optimalizační kritérium)**.** *Nechť S je množina všech **VS** a $dist(s, s')$ je **LV** s, $s' \in S$. Dále nechť $comm(s, s')$ označuje množství libovolných jednotek síťové komunikace odeslaných mezi s, $s' \in S$. Pak optimalizační kritérium definujeme jako:*

$$\sum_{s,s' \in S} (dist(s, s') * comm(s, s'))$$

Konsolidace zvažovaná v této práci se pak zabývá nalezením takového rozmístění virtuálních strojů na virtualizačních uzlech, že splňuje všechna omezující kritéria a zároveň má minimální hodnotu kritéria optimalizačního.

Rád bych ještě upozornil na obecnost *comm*() v předchozí definici. Takto zavedená hodnota může v praxi reprezentovat téměř jakoukoliv metriku dat přenesených po síti (rámce, pakety, byty, gigabyty, i jiné). Tuto formulaci jsem zvolil úmyslně, jelikož kvůli důvodům popsaných v 2.4.2.1 nebylo možné zvolit, která (případně které) z jednotka by mohla být v praxi optimálním kandidátem.

2.4.2.1 Testovací data

Významným problémem konsolidace virtuálních strojů se zaměřením na optimalizaci zatížení síťové infrastruktury je podle mého názoru nedostatek informací. V době psaní této práce mi není známá jediná studie či datová sada obsahující volně dostupné informace o rozložení komunikace mezi virtuálními stroji v rámci datacentra. Zároveň se mi v průběhu mého výzkumu nepodařilo tato data získat. Z tohoto důvodu bylo za účelem experimentálního ověření nutné data náhodně vygenerovat. Jelikož se však nedomnívám, že komunikace mezi virtuálními stroji je náhodná, formuloval jsem v další kapitole několik myšlenek o tom, jak by mohla tato data vypadat.

[2] „skříň“ sloužící k ukládání serverů, které jsou nejčastější formou virtualizačních uzlů

[3] uvažujeme kontext 2. a 3. vrstvy ISO/OSI modelu

Kapitola 3
Generátor datových sad

V této kapitole popisuji jak a proč se domnívám, že vypadá struktura komunikace mezi virtuálními stroji. Dále pak popisuji mnou implementovaný generátor náhodných dat, odpovídající této teorii.

3.1 Vlastnosti virtuálních strojů a virtualizačních uzlů

Jak již bylo dříve zmíněno, tato práce předpokládá homogenní virtualizační uzly. Dalším předpokladem je „standardizace“ jejich výpočetních prostředků. Vevětšině případů moderní procesory obsahují mocninný počet jader o základu 2, případně součet dvou těchto mocnin. (V závislosti určení procesoru, pro osobní počítače tyto hodnoty bývají nejčastěji 4 a 8 výpočetních jader, pro využití datacenter pak existují procesory o 16, 24 přes 128 až po 192 jader.[1]) Obdobně moduly operační paměti RAM se vyrábějí v mocninách o základu 2 (typicky od 4 do 256 GB).

Ačkoliv většina hypervizorů či operačních systémů umožňuje vytvořit virtuální stroj s atypickým množstvím prostředků (například 9 výpočetních jader či 3 GB operační paměti RAM), tuto možnost jsem v rámci návrhu generování dat zavrhl. Atypické virtuální stroje budou dle mého názoru spíše méně častý jev. Datacentrum se navíc může v rámci pokusu o usnadnění efektivního zaplnění zdrojů jednotlivých virtualizačních uzlů omezit pouze na podporu typických **VS**.

3.2 Vlastnosti komunikace virtuálních strojů

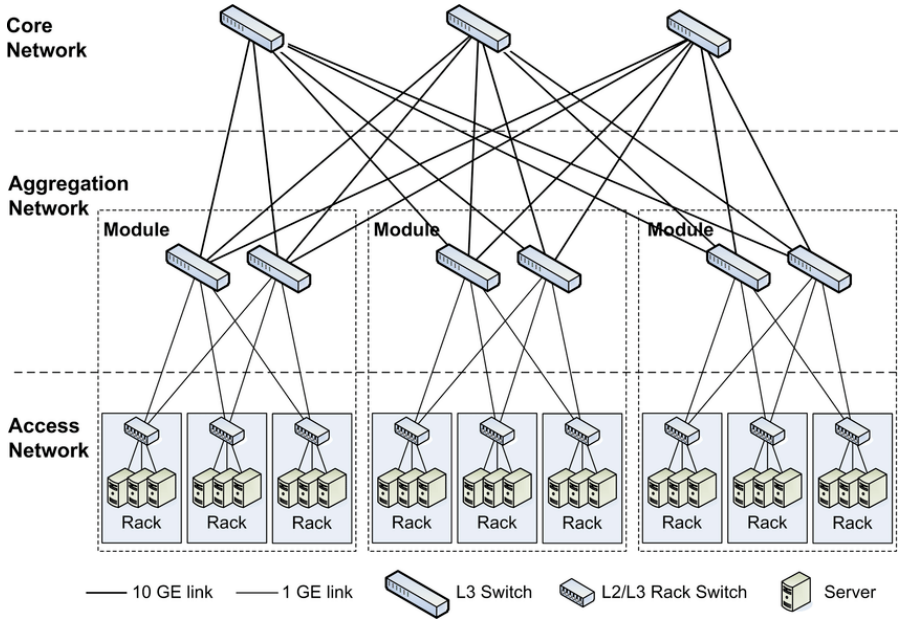
Ohledně samotné síťové komunikace uvnitř datacentra, domnívám se, že existují 2 typy situací:

- Běžná komunikace
- Komunikace v rámci clusterů

3.2.1 Běžná komunikace

Jedná se o relativně malé množství síťové komunikace mezi dvojicí **VS**. Avšak dochází k ní prakticky vždy. Praktickým příkladem této situace by mohl být ARP protokol, který při počátečním dotazu využívá broadcastu na linkové vrstvě ISO/OSI. Rámec s tímto dotazem tudíž dorazí ke každému virtuálnímu stroji v rámci jednoho segmentu sítě. Dalším příkladem by mohl být *discover* paket protokolu DHCP.

[1] viz. například řada procesorů EPYC společnosti AMD



Obrázek 3.1 Nákres typické třívrstvé topologie sítě. Zdroj: [13]

3.2.2 Komunikace v rámci clusterů

Dalším jevem v komunikaci mezi dvojicí ale i větší skupinou **VS** je komunikace tzv. clusterů. Cluster je v tomto případě skupina virtuálních strojů, které mezi sebou (přímo či přes jednoho z nich jako prostředníka) komunikují znatelně více a častěji. Jako praktický příklad bych uvedl klienty a server či databázi libovolné aplikace, či členy společné Active Directory společnosti Microsoft. Další z variant může být proxy server, který slouží k přístupu z venčí na ostatní členy clusteru. Domnívám se, že výskyt tohoto jevu nebude ojedinělý, avšak běžná komunikace by měla převažovat.

3.3 Vlastnosti síťové infrastruktury datacentra

Finální problém struktury dat je samotná topologie síťové infrastruktury datacentra. Teoreticky může být naprosto libovolná. Cílem datacenter však bude určitá míra redundance (kvůli spolehlivosti a případnému vyvažování zátěže). Na druhou stranu proti tomuto kritériu však jde pořizovací cena této infrastruktury. V neprospěch velké redundance i nízká míra zatížení jednotlivých síťových prvků. V praxi se tedy bude hledat určitý kompromis.

Z tohoto důvodu jsem zvolil tzv. třívrstvou síťovou architekturu. Ta se skládá ze 3 vrstev, jejichž síťové prvky nejsou propojeny s prvky stejné vrstvy, ale s prvky vrstvy „o jedna“ nižší respektive vyšší. Jednotlivé linky mezi vrstvami jsou typicky zdvojené kvůli redundanci. Virtualizační uzly jsou připojeny na síťové uzly nejnižší (tzv. přístupové) vrstvy. Typickou třívrstvou síťovou architekturu zobrazuje obrázek 3.1.

Za účelem obecnosti implementační části této práce jsem však vytvořil formát vstupního souboru generátoru, který jednoduchým způsobem popisuje libovolnou síťovou topologii ve zjednodušeném pojetí (uvedeném na začátku sekce 2.1).

```
7 4 1

s1 s2 s3 n1 n2 n3 n4

s1      2  s2 s3
s2      3  s1 n1 n2
s3      3  s1 n3 n4
n1      1  s2
n2      1  s2
n3      1  s3
n4      1  s3
```

Obrázek 3.2 Ukázka struktury jednoduchého .TPG souboru

3.3.1 Formát .TPG

Vstupní soubor generátoru začíná řádkou obsahující trojicí čísel oddělených mezerou: celkový počet uzlů topologie (virtualizačních i síťových), počet virtualizačních uzlů a počet „zaplněných“ výpočetních uzlů. Zaplněné výpočetní uzly jsou takové, jejichž výpočetní prostředky jsou téměř vyčerpány virtuálními stroji. Tento parametr má velký vliv na obtížnost dané instance, ale to bude podrobněji rozebráno v kapitole 7.

Následuje mezerami oddělený seznam jmen všech uzlů topologie. Jmenná konvence je v tomto případě „sID“ pro síťový uzel a „nID“ pro uzel virtualizační. (ID myšleno celé číslo označující konkrétní uzel, například s8 či n2.)

V poslední řadě pak následují řádky obsahující seznamy sousedů, jejichž syntaxe je opět oddělená bílými znaky v následujícím pořadí: jméno uzlu, počet sousedů, jména sousedů oddělená mezerami.

Příklad jednoduchého souboru .TPG se 4 virtualizačními a 3 síťovými uzly zobrazuje obrázek 3.2.

3.4 Implementace generátoru

K implementaci generátoru testovacích instancí jsem zvolil programovací jazyk C++. Samotný generátor je z algoritmického hlediska velmi přímočarý a jednoduchý.

3.4.1 Argumenty generátoru

Generátor má 3 povinné vstupní argumenty v tomto pořadí:

1. soubor .TPG
2. pravděpodobnost zastavení zaplnění **VU**
3. pravděpodobnost vzniku clusteru mezi dvojicí **VS**

Pravděpodobnost zastavení zaplnění virtualizačních uzlů ovlivňuje obsazenost výpočetních prostředků **VU**, které nejsou zaplněné (viz 3.3.1). Pro takové uzly pak dochází při přidání každého dalšího **VS** k vygenerování rozhodnutí, zda je tento **VS** poslední na tomto **VU**. Tato pravděpodobnost je dána tímto argumentem, jehož hodnotou může být celé číslo v intervalu 0-100%.

Pravděpodobnost vzniku clusteru mezi dvojicí **VS** je vyhodnocena při generování komunikace mezi jednotlivými dvojcemi **VS**. Tento argument může být opět celé číslo v intervalu 0-100%.

Výpis kódu 3.1 Implementace Dijkstrova algoritmu v generátoru dat

```
void calcDistances(std::vector<node> & topology){
        for(auto & it : topology){
                std::queue<std::pair<std::reference_wrapper<node>,int>> q;
                for(auto & iit : topology){
                        iit.state = stateNotFound;
                }

                q.push(std::make_pair<std::reference_wrapper<node>,
                                                int>(it,0));
                it.state = stateOpened;

                while(!q.empty()){
                        for(auto & iit : q.front().first.get().neighbours){
                                if(iit.get().state == stateNotFound){
                                        iit.get().state = stateOpened;
                                        q.push(std::make_pair<
                                         std::reference_wrapper<node>,int>
                                         (iit.get(), q.front().second+1));
                                }
                        }

                        q.front().first.get().state = stateClosed;
                        it.distances.push_back(std::pair<
                                std::reference_wrapper<node>,int>
                                (q.front().first.get(), q.front().second));
                        q.pop();
                }

                if(topology.size() != it.distances.size()){
                        std::cerr << "Found unreachable node or "
                             << "switch... Funy..." << std::endl;
                        exit(-666);
                }
        }
}
```

3.4.2 Určení logických vzdáleností

K určení logických vzdáleností generátor používá jednoduchou sekvenční implementaci Dijkstrova algoritmu. Ten pro každý z uzlů topologie vytvoří frontu vrcholů pro postupné prohledávání sousedů. Tato fronta zpočátku obsahuje pouze výchozí uzel. Iterativně pak dochází k nacházení, výpočtu vzdálenosti a přidání do fronty neobjevených uzlů, které sousedí s uzlem, který je aktuálně na začátku fronty. Algoritmus iteruje, dokud fronta není prázdná. Exaktní formulace tohoto algoritmu není pro účely tohoto textu potřebná, lze jí však nalézt například v [14]. Konkrétní implementaci v generátoru zobrazuje výpis kódu 3.1.

Rád bych upozornil na implementaci chybového ukončení generátoru, pokud v topologii existuje uzel, který není dosažitelný z jiného. Důvod je ten, že v takováto situace by efektivně mohla znemožnit migraci v rámci datacentra a tudíž tento stav není považován za validní.

Výpis kódu 3.2 Generování virtuálních strojů

```
int cnt = 0;
int tmpRAM;
int tmpCPU;
while(cnt < numNodes){
        tmpRAM = 0;
        tmpCPU = 0;
        while(true){
                virtualMachine tmp;

                if( ((cnt < filledNodes)
                                && (tmp.numCores + tmpCPU <= maxCores)
                                && (tmp.RAMSize + tmpRAM <= maxRAM))
                                    ||
                        (((int)(RNG()%100 + 1) > stopProbability)
                                && (tmp.numCores + tmpCPU <= maxCores)
                                && (tmp.RAMSize + tmpRAM <= maxRAM)) ){
                        virtualMachines.push_back(tmp);
                        tmpCPU += tmp.numCores;
                        tmpRAM += tmp.RAMSize;
                }else{
                        virtualMachines.back().lastInNode = true;
                        break;
                }
        }
        ++cnt;
}
```

3.4.3 Generování virtuálních strojů a jejich komunikace

Ke generování náhodných čísel jsem použil prostředky *Standard Template Library* jazyka C++. Konkrétně generátor pseudonáhodných čísel *std::default_random_engine*. K jeho inicializaci využívám *std::random_device*, což by podle dokumentace na většině implementací měl být nedeterministický generátor náhodných čísel. Domnívám se, že takto inicializovaný pseudonáhodný generátor bohatě postačuje pro účely této práce.

Generováním prostředků jednotlivých **VS** probíhá postupné zapĺňování jednotlivých **VU**. Algoritmus postupuje nejprve za cílem vygenerování **VS** umístěných na jednotlivých zaplněných **VU**, dokud by další **VS** neporušila omezující kritéria. Tímto způsobem generátor postupně zaplní požadovaný počet **VU**. Pro zbylé **VU** pak postupně generuje **VS** s argumentem danou pravděpodobností vygenerování dalšího **VS** či dokud nedojde k jeho zaplnění. Tato část zdrojového kódu je zobrazena v 3.2.

Samotné generování komunikace je pak spíše formální záležitost. Pro každou dvojici virtuálních strojů je nejdřív s příslušnou pravděpodobností určeno, zda jsou vzájemně součástí clusteru. Poté je vygenerována náhodná hodnota mezi 0 a příslušnou maximální hranicí (dané hranice budou detailně diskutovány v kapitole 7). Komunikace je pro účely této práce symetrická, tudíž se do ní započítávají příchozí i odchozí data. Hodnota komunikace mezi **VS** 1 a 2 je tedy vždy totožná, jako mezi **VS** 2 a 1. Zároveň generátor vždy určí hodnotu komunikace **VS** se sebou samotným jako 0. V praxi tato komunikace vůbec neopustí operační systém daného **VS** a tudíž je pro konsolidaci nepodstatná.

```
#nodes: 4

#nodeCores: 16 #nodeRAM: 32

NodeDistances
NodeName: n1 Distances: n1-0 n2-1 n3-3 n4-3
NodeName: n2 Distances: n2-0 n1-1 n3-3 n4-3
NodeName: n3 Distances: n3-0 n4-1 n1-3 n2-3
NodeName: n4 Distances: n4-0 n3-1 n1-3 n2-3

#virtualMachines: 6
2-4 1-4
8-16
16-2
8-16 8-8

CommunicationMatrix:
0 58 148 124 101 161
58 0 69 135 230 285
148 69 0 4 162 288
124 135 4 0 205 216
101 230 162 205 0 107
161 285 288 216 107 0
```

Obrázek 3.3 Ukázka struktury jednoduchého .DTC souboru

3.4.4 Formát výstupu – soubor .DTC

Generátor instancí vypisuje na standardní výstup. Formát je ale totožný se vstupním formátem některých dále popsaných metod konsolidace, který jsem označil jako .DTC. Opět se jedná o jednoduchý textový formát, tentokrát však i s anglickými komentáři. Ten má následující tvar:

1. *#nodes:* <počet **VU**>
2. *#nodeCores:* <počet výpočetních jader **VU**> *#nodeRAM:* <množství operační paměti **VU**>
3. *NodeDistances*
   - *NodeName:* <označení **VU**> *Distances:* <označení **VU**>-vzdálenost…
   - *NodeName:* <označení **VU**> *Distances:* <označení **VU**>-vzdálenost…
   - …
4. *#virtualMachines:* <celkový počet **VS**>[2]
   - CPU-RAM CPU-RAM…
   - CPU-RAM CPU-RAM……
5. *CommunicationMatrix:*[3]
   - <komunikace s prvním **VS**> <komunikace s druhým **VS**>…
   - <komunikace s prvním **VS**> <komunikace s druhým **VS**>…
   - …

   Jednoduchý příklad formátu .DTC zobrazuje obrázek 3.3

[2] následovaný řádkovým seznamem – na prvním řádku jsou **VS** umístěny na prvním **VU**…
[3] následovaný řádkovým seznamem komunikace, první řádek obsahuje komunikací prvního **VS**, druhý řádek druhého **VS**…

Kapitola 4
Heuristické řešení

Jako první z možných způsobů konsolidace bych rád uvedl metodu hledání clusterů, která byla již navržena v rámci [1]. Jedná se o heuristiku, jejíž cílem je vyhledat shluky virtuálních strojů, které spolu komunikují více a migrací je přemístit buď na totožný nebo co nejbližší virtualizační uzel. V průběhu tohoto procesu však může nastat, že by došlo k zaplnění některého z virtualizačních uzlů. Algoritmus tento problém řeší postupnou migrací virtuálních strojů, které nejsou součástí clusteru a jejichž migrace pokud možno ovlivní zatížení síťové infrastruktury co nejméně. V ideálním případě by tato heuristika měla vést k malému počtu migrací, avšak je velmi nepravděpodobné, že by dosahovala optimálního vytížení síťové infrastruktury. Pseudokód tohoto algoritmu je vyobrazen v 1.

4.1 Vstupní parametry algoritmu

Velmi významnou otázkou při nasazení této heuristiky je: „Co považovat za clustr a co již ne?“ Z tohoto důvodu jsem navrhl následující vstupní parametry algoritmu:

- hranice vytvoření clusterů
- hranice zvětšování clusterů

Jedná se o celočíselné parametry, které určují množství a velikost vyhledaných clusterů. Podrobněji se těmto parametrům věnuji v 4.2.1.

4.2 implementace

K implementaci tohoto heuristického algoritmu jsem opět zvolil jazyk C++. Vstupními argumenty implementace jsou:

1. soubor .DTC
2. hranice vytvoření clusterů
3. hranice zvětšování clusterů

Implementace poté načte vstupní data simulovaného datacentra a uloží je do následujících struktur 4.1.

Pole *startingPlacement* pak obsahuje veškeré informace o původních virtuálních strojích. Pole *nodeDistances* obsahuje informace o vzdálenosti virtualizačních uzlů. Závěrem pole *startingFilling* obsahuje informace o zabraných zdrojích jednotlivých virtualizačních uzlů.

Třída *countingNode* je pomocná třída, kterou implementace využívá při určení migrací **VS**.

Výpis kódu 4.1 Datové struktury implementace heuristického algoritmu

```
class virtualMachine {
public:
        int cores, RAM, nodePlacement;
        std::vector <int> communication;

        virtualMachine(int c, int r, int n):cores(c), RAM(r),
                                        nodePlacement(n){
                communication.reserve(numVMs);
        }

        virtualMachine(const virtualMachine & vm){
                cores = vm.cores;
                RAM = vm.RAM;
                nodePlacement = vm.nodePlacement;
                communication = vm.communication;
        }
};

class node {
public:
        int cores;
        int RAM;

        node():cores(0),RAM(0){}

        node(const node & n){
                cores = n.cores;
                RAM = n.RAM;
        }
};

std::vector <node> startingFilling;
std::vector <std::vector <int>> nodeDistances;
std::vector <virtualMachine> startingPlacement;

class countingNode {
public:
        int cores, RAM;
        std::vector<int> clusterMember;
        std::vector<std::pair<int, double>> attemptedStandalone;
        std::vector<int> standalone;

        countingNode():cores(0),RAM(0){}
};
```

4.2.1 Hledání clusterů

Implementace nejprve určí hraniční množství komunikace, které ještě považuje za cluster. Toho dosáhne následujícím výpočtem:

$$L = max(comm) - \frac{max(comm) - \overline{comm}}{l}$$

Kde L je hledaná hranice, $max(comm)$ je maximum komunikace, která v datacentru proběhla mezi dvěma **VS**, $\overline{comm}$ je aritmetický průměr komunikace mezi všema **VS** a l je vstupní parametr hranice vytvoření clusterů.

Poté implementace páruje **VS**. Jakákoliv dvojice, která mezi sebou komunikuje více, než je hranice L je považována za cluster. Dále jsou jednotlivé dvojice, které sdílejí jeden **VS** spojeny do clusterů o více **VS**. V poslední řadě pak implementace projde komunikaci jednotlivých **VS** v clusteru a jejich komunikaci s jinými **VS**, které nejsou zařazeny v žádném z doposud vytvořených clusterů. Pokud by přidáním tohoto virtuálního stroje došlo k navýšení celkové komunikace uvnitř clusteru o hodnotu argumentu hranice zvětšování clusterů (v procentech), je tento **VS** přidán do daného clusteru.

4.2.2 Výběr cílových virtualizačních uzlů

Implementace následně určí ideální rozmístění clusterů na virtualizační uzly. Za ideální umístění je považován virtualizační uzel, na kterém se původně nacházel **VS** s největším množstvím operační paměti. (Migrace tohoto stroje by pravděpodobně byla nejnáročnější.) Poté dojde k vyhodnocení pro všechny členy clusteru, zda se již na tento „ideální uzel“ vejdou, či zda je nutné pro ně hledat alternativní umístění. To je v případě potřeby vyhledáno hlavně podle vzdálenosti od „ideálního uzlu“. Konkrétní funkce jsou zobrazeny ve výpisu kódu 4.2.

Je vhodné podotknout, že po provedení rozmístění clusterů se vnitřní stav implementace může nacházet v nevalidním stavu. To vzniká díky tomu, že rozmisťování clusterů nebere v úvahu **VS**, které nejsou součástí žádného clusteru.

4.2.3 Případná migrace samostatných virtuálních strojů

Pokud nastala situace, že se nějaký ze samostatných virtuálních strojů „nevejde“, je nutná jeho migrace na jiný virtuální stroj. Tyto migrace má implementace provádí v pořadí podle poměru komunikace s **VS** na daném **VU** ku komunikace s **VS** mimo daný virtualizační uzel. (Uzel s nejnižším poměrem je migrován první.) Nový virtualizační uzel tohoto **VS** je ten, kde lze **VS** umístit a zároveň má nejmenší množství dostupných volných prostředků.

Zde je však nutné podotknout, že může teoreticky nastat situace, kdy pro nějaký virtuální stroj již neexistuje virtualizační uzel, kam by ho šlo umístit. (To nastává díky tomu, že algoritmus nebere v úvahu využití prostředků **VU** tak, aby měl jistotu, že umístí všechny **VS**.) V takovém případě algoritmus selhal, ohlásí kritickou chybu a ukončí se.

4.3 Složitost algoritmu

Algoritmus této heuristiky lze jednoduše popsat pomocí pseudokódu 1. Výpočet maxima a průměru komunikace je kvadratický k počtu virtuálních strojů. Vyhledání počátečních clusterů je opět kvadratické k počtu virtuálních strojů. Spojení překrývajících se clusterů je lineární k počtu clusterů a tudíž lze shora omezit počtem virtuálních strojů. (Clusterů z principu nemůže být více než VS.) Proces výběru migrací pro clustery má složitost počet clusterů (který lze opět odhadnout shora) vynásobený počtem **VU**. V závěru případná migrace samostatných **VS** je shora

Výpis kódu 4.2 Implementace umístění clusterů heuristiky

```
int nearestGoodNode(int idealNode, int toPlace,
                std::vector<countingNode> & compute){

        if(compute[idealNode].cores + startingPlacement[toPlace].cores <=
                                    nodeCores && compute[idealNode].RAM +
                                    startingPlacement[toPlace].RAM
                                                    <= nodeRAM ){
                return idealNode;
        }else{
                int tmp = -1;
                for(int i = 0; i < numNodes; ++i){
                        if(compute[i].cores + startingPlacement[toPlace].
                                cores <= nodeCores && compute[i].RAM
                                    + startingPlacement[toPlace].RAM
                                    <= nodeRAM){
                                if(tmp == -1){
                                        tmp = i;
                                }else{
                                        if(nodeDistances[idealNode][i] <
                                        nodeDistances[idealNode][tmp])
                                                tmp = i;
                                }
                        }
                }

                if(tmp == -1){
                        std::cerr << "Fatal error" << std::endl;
                        exit(666);
                }
                return tmp;
        }
}

void findPlaceCluster(cluster & cls, std::vector<countingNode> & compute,
                                            std::vector <int> & solution){
        int maxOnNode = startingPlacement[cls.VMs[0]].nodePlacement;
        int maxRAM = nodeRAM - startingFilling[maxOnNode].RAM;
        for(auto & it : cls.VMs){
                if(nodeRAM - startingFilling[startingPlacement[it].
                                nodePlacement].RAM > maxRAM){
                        maxRAM = nodeRAM -
                startingFilling[startingPlacement[it].nodePlacement].RAM;
                        maxOnNode = startingPlacement[it].nodePlacement;
                }
        }
        for(auto & it : cls.VMs){
                int n = nearestGoodNode(maxOnNode, it, compute);
                compute[n].clusterMember.push_back(it);
                compute[n].cores += startingPlacement[it].cores;
                compute[n].RAM += startingPlacement[it].RAM;
                solution[it] = -1;
        }
}
```

omezitelná počtem **VS** krát počet možných umístění tudíž počet **VU**. To celkově dává následující složitost:

$$\mathcal{O}(n^2 + n^2 + m + m * n + m * n) = \mathcal{O}(n^2 + m * n)$$

Kde n je počet **VS** a m počet **VU**. To lze efektivně považovat za složitost $\mathcal{O}(n^2)$ za předpokladu, že $n > m$, jelikož oba počty jsou přirozená čísla. Variantu, že by v datacentru bylo víc aktivních **VU** než **VS** můžeme v tomto případě zanedbat. (Alternativně lze prázdné **VU** považovat za vyplé a tudíž je nebrat při výpočtu v úvahu.)

Algoritmus 1 Pseudokód heuristického algoritmu

```
1:  L = max(Comm) − (max(Comm) − prumer(Comm))/l
2:  clustery ← najdiDvojiceVS(Comm, L)
3:  spojPrekryvajiciSeClustery(clustery)
4:  for i in clustery do
5:      for j in VS & j not in clustery do
6:          if 1.1 * Comm(i) ≤ Comm(i + j) then
7:              pridej j do i
8:          end if
9:      end for
10: end for
11: for i in clustery do
12:     migruj(i, najdiVhodnyUzel(i))
13: end for
14: for i in VU do
15:     while preplneny(i) do
16:         j VS v i s nejnizsi Comm(uvnitr i)/Comm(vne i)
17:         migruj j na uzel s dostatkem volnych zdroju
18:     end while
19: end for
```

Kapitola 5

Řešení celočíselným programováním

Dalším způsobem řešení, již navrhnutým v rámci [1], je definovat problém jako tzv. celočíselné programování. Jedná se o způsob vyjádření problému pomocí celočíselných neznámých. Takto formulovaný problém lze pak řešit různými způsoby, většinou v závislosti na nějaké účelové funkci[1]. Existují řešiče či algoritmy přímo určené k řešení těchto problémů [15]. V rámci svého výzkumu jsem se však rozhodl použít trochu rozdílný přístup, jelikož se domnívám, že čisté celočíselné programování není vhodné pro formulaci optimalizačního kritéria formulovaného v této práci. Zvolil jsem tedy použití jednu z evolučních výpočetních technik – genetický algoritmus.

5.1 Genetický algoritmus

Podle [16] se jedná o jednu z nejznámějších evolučních výpočetních technik. Hlavní myšlenka algoritmu je založena na simulaci evoluce a principu přirozeného výběru popsaných Charlesem Darwinem. Jedná se o randomizovaný algoritmus.

5.1.1 Terminologie

V této sekci bych rád představil základní terminologii používanou genetickými algoritmy. Účelem je, aby případný neznalý čtenář získal alespoň základní povědomí o významu jednotlivých pojmů. Jejich případné detailní vysvětlení pak následuje dále v této kapitole.

- Gen – základní prvek, jediná hodnota, většinou binární (ale ne nutně)
- Chromozom – posloupnost genů, která tvoří jednu instanci řešení daného problému
- Generace – množina chromozomů, v průběhu algoritmu dochází k její obměně, bývá označována také jako populace
- Vhodnost – hodnota přiřazovaná jednotlivým chromozomům pomocí speciální vhodnostní funkce[2], určuje kvalitu řešení, které je reprezentováno daným chromozomem
- Křížení – způsob kombinace dvou chromozomů za účelem vytvoření nového

[1] anglicky *objective function*
[2] anglicky *fitness function*

- Selekce – způsob výběru chromozomů ke křížení
- Mutace – většinou náhodná změna v novém chromozomu (potomek křížení)
- Elitismus – technika sloužící k zajištění postupné optimalizace vhodnosti napříč generacemi

5.2 Implementace

K implemenaci jsem opět zvolil jazyk C++. Náhodný generátor využívá tato implementace stejný jako implementace generátoru dat (viz 3.4.3).

5.2.1 Vstupní argumenty

Implementace genetického algoritmu pro řešení konsolidace virtuálních strojů přímá následující vstupní argumenty:

1. soubor .DTC
2. počet chromozomů v generaci
3. počet generací
4. počet elitních chromozomů
5. maximální počet opakování elitních chromozomů
6. pravděpodobnost křížení
7. pravděpodobnost mutace

S výjimkou prvního jsou všechno argumenty celočíselné. Pravděpodobnosti křížení a mutace mohou nabývat hodnot $\langle 0, 100 \rangle$. Zbylé pak mohou být libovolné přirozené číslo, za předpokladu, že počet elitních chromozomů je nižší než počet chromozomů v generaci a maximáln počet opakování elitních chromozomů je nižší než počet generací.

5.2.2 Reprezentace dat

K uložení stavu datacentra včetně počátečního rozmístění virtuálních strojů jsem využil datové struktury zobrazené v 5.1. Pole *startingPlacement* obsahuje veškeré informace o virtuálních strojích, včetně jejich počátečního umístění. Pole *nodeDistances* pak opět udržuje informace o vzdálenosti jednotivých virtualizačních uzlů. Data uložená v těchto strukturách se však dále v průběhu algoritmu nemění a jsou využita pouze k výpočtu vhodnosti či závěrečnému vyhodnocení migrací.

5.2.2.1 Reprezentace generace

Jednotlivé generace jsou pak jednoduše reprezentovány pomocí dvojrozměrného celočíselného pole. Každý „řádek“ reprezentuje jeden chromozom. Ten se skládá z tolika čísel – genů, kolik je v řešené instanci virtuálních strojů. Každé prvek pak nabývá hodnoty indexu virtualizačního uzlu, kde je potenciálně daný virtuální stroj umístěn.

Výpis kódu 5.1 Datové struktury implementace genetického algoritmu

```
class virtualMachine {
public:
        int cores, RAM, nodePlacement;
        std::vector <int> communication;

        virtualMachine(int c, int r, int n):cores(c), RAM(r),
                                            nodePlacement(n){
                communication.reserve(numVMs);
        }

        virtualMachine(const virtualMachine & vm){
                cores = vm.cores;
                RAM = vm.RAM;
                nodePlacement = vm.nodePlacement;
                communication = vm.communication;
        }
};

std::vector <virtualMachine> startingPlacement;
std::vector <std::vector <int>> nodeDistances;
```

5.2.3 Výpočet vhodnosti

Vhodnost je v tomto kontextu celé číslo. Implementace ho chromozomům přiřazuje dvojím způsobem podle toho, zda daný chromozom splňuje omezující kritéria problému či nikoliv. Pokud je chromozom z hlediska problému konsolidace validní, je vhodnost tohoto chromozomu vypočtena jako exaktní hodnota optimalizačního kritéria. V opačném případě je vhodnost nastavena na speciální hodnotu (výrazně vyšší než jsou přípustné hodnoty optimalizačního kritéria), která je ale snížena o počet **VU**, které omezující kritéria neporušily.

Výsledkem jsou hodnoty vhodnosti, které jasně odlišují validní a nevalidní chromozomy. V rámci validních chromozomů pak vhodnost rovnou vypovídá o optimalizačním kritériu. V případě nevalidních chromozomů jsou pak rozlišeny chromozomy, ve kterých jsou omezující kritéria porušeny méně. K ohodnocení kvality řešení reprezentovaného daných chromozomem pak stačí jednoduché porovnání vhodnosti s ostatními chromozomy. (Čím nižší hodnota vhodnosti, tím lepší řešení chromozom reprezentuje.)

5.2.4 Křížení

Tato implementace používá tzv. bodové křížení. Oba rodičovské chromozomy jsou v náhodném místě rozděleny na dvě části (mohou být různě velké). Výsledný potomek pak přebírá jednu část genů od obou rodičů. Konkrétní implementace je vyobrazena v 5.3.

5.2.5 Selekce

Samotná volba selekce velmi ovlivňuje chování genetického algoritmu. Určuje tzv. selekční tlak, který vyjadřuje jakousi „míru preference“ výběru vhodnějšího chromozomu jako rodiče ke křížení. Vyšší selekční tlak pak urychluje rychlost konvergence algoritmu k optimu. Zároveň však zvyšuje riziko uváznutí v optimu lokálním či riziko ztráty diverzity populace. V rámci implementace popsané v této práci jsem použil selekci škálováním. Pravděpodobnost výběru jedince je přeškálována číselnou posloupností $\frac{k(k+1)}{2}$, kde k je pořadí prvku při jejich seřazení podle

Výpis kódu 5.2 Implementace vhodnostní funkce genetického algoritmu

```
int getNumOfNonoverloadedNodes(std::vector<int> & chromozome){
        std::vector<node> cluster;

        for(int i = 0; i < numNodes; ++i){
                cluster.emplace_back(0,0);
        }

        for(int i = 0; i < numVMs; ++i){
                cluster[chromozome[i]].cores += startingPlacement[i].cores;
                cluster[chromozome[i]].RAM += startingPlacement[i].RAM;
        }

        int tmp = numNodes;
        for(int i = 0; i < numNodes; ++i){
                if(cluster[i].cores > nodeCores || cluster[i].RAM > nodeRAM){
                        --tmp;
                }
        }
        return tmp;
}

long long int getCommSum(std::vector<int> & chromozome){
        long long int tmp = 0;
        for(int i = 0; i < numVMs; ++i){
                for(int j = 0; j < numVMs; ++j){
                        tmp += (long long int)(nodeDistances[chromozome[i]]
                [chromozome[j]] * startingPlacement[i].communication[j]);
                }
        }
        return tmp/2;   //every packet counted 2 times
}

void calculateFitness(std::vector<long long int> & fit,
                        std::vector<std::vector<int>> & pop){
        int numOfNonoverloadedNodes;
        for(int i = 0; i < populationSize; ++i){
                numOfNonoverloadedNodes = getNumOfNonoverloadedNodes(pop[i]);

                if(numOfNonoverloadedNodes == numNodes){
                        fit[i] = getCommSum(pop[i]);
                }else{
                        fit[i] = std::numeric_limits<long long int>::max()
                        - (long long int)numOfNonoverloadedNodes;
                }
        }
}
```

Výpis kódu 5.3 Implementace křížení genetického algoritmu

```
void crossover (std::vector <int> & parent1, std::vector <int> & parent2,
                                             std::vector <int> & child){
        int split = RNG()%numVMs;

        for(int i = 0; i < numVMs; ++i){
                if(i < split){
                        child[i] = parent1[i];
                }else{
                        child[i] = parent2[i];
                }
        }
}
```

Výpis kódu 5.4 Implementace selekce genetického algoritmu

```
int selection (std::vector<std::pair<int, long long int>> & sortedFit){

        int rand = (RNG() % selectMax) + 1, index = 1, i = 1;

        for(; i < rand; ++index){
                i += index + 1;
        }

        return sortedFit[index-1].first;
}
```

vhodnosti. (Vhodnější chromozomy mají vyšší pravděpodobost selekce.) Zdrojový kód selekce je zobrazen v 5.4.

5.2.6 Mutace

Mutace je hlavní způsob zvýšení diverzity populace. Tato technika slouží ke snížení rizika uváznutí algoritmu v lokálním optimu. Tato implementace využívá jednoduchou mutaci, kdy dojde ke změně náhodného genu vybraného chromozomu na náhodnou hodnotu.

V předcházejícím výzkumu [1] jsem experimentoval i výrazně drastičtějším přístupem, který s danou pravděpodobností mohl změnit libovolný počet genů v chromozomu. Tato varianta se však ukázala jako nepoužitelná a tudíž jí v této práci dále nezmiňuji.

5.2.7 Elitismus

Poslední důležitou součástí genetického algoritmu je elitismus. Jedná se o způsob zachování nejlepších jedinců při přechodu mezi generacemi. To má za následek, že nedochází k „zapomenutí“ již nalezených kandidátů na optimální řešení a tudíž případné nezhoršení výsledků s přibývajícími generacemi.

Další výhodou elitismu je možnost dřívějšího ukončení algoritmu. Při každém vyhodnocení generace dojde k porovnání nových nejvhodnějších chromozomů s původními. Pokud jsou tyto chromozomy stejné již několik generací, mohl algoritmus již nalézt globální optimum, nebo uváznout v optimu lokálním. V obou případech tedy dává smysl výpočet přerušit.

5.2.8 Samotný algoritmus

Samotný algoritmus pak postupnou iterací vytváří nové generace pomocí výše popsaných metod. Každou generaci vyhodnotí pomocí vhodnostní funkce a uloží elitní chromozomy. To opakuje, dokud nedojde k maximálnímu opakování elitních chromozomů, nebo vytvoření poslední generace. Nejvhodnější dostupné řešení je pak označeno za výsledek. Konkrétní popis algoritmu je zobrazen pomocí pseudokódu 2.

Algoritmus 2 Pseudokód implementovaného genetického algoritmu

```
1:  generace ← RNG()
2:  elitni ← pocatecniKonfigurace
3:  elitniOpakovani ← 0
4:  pocetGeneraci ← 0
5:  while pocetGeneraci ≤ maxGeneraci and elitniOpakovani ≤ maxElitniOpakovani do
6:      vypoctiVhodnost(generace)
7:      if elitni = najdiNejvhodnejsi(generace) then
8:          ++elitniOpakovani
9:      else
10:         elitni ← najdiNejvhodnejsi(generace)
11:         elitniOpakovani ← 0
12:     end if
13:     novaGenerace ← elitni
14:     for i dokud novaGenerace neni plna do
15:         if pravdepodobnostKrizeni(RNG()) then
16:             novaGenerace[i] ← krizeni(generace, skalovaneRNG())
17:         else
18:             novaGenerace[i] ← generace[i]
19:         end if
20:         if pravdepodobnostMutace(RNG()) then
21:             novaGenerace[i] ← mutace(novaGenerace[i])
22:         end if
23:     end for
24:     ++pocetGeneraci
25:     generace ← novaGenerace
26: end while
27: Return najdiNejvhodnejsi(generace)
```

5.3 Složitost algoritmu

Předpokládejme, že:

- G je velikost generace
- G_{max} je maximální počet generací
- h je vhodnostní funkce
- P_m je pravděpodobnost mutace
- P_c je pravděpodobnost křížení
- m je funkce mutace

- c je funkce křížení

Genetický algoritmus popsaný v této kapitole provede maximálně G_{max} iterací. V každé iteraci dojde k porovnání a uložení elitních chromozomů. To je však konstatní oproti počtu elitních chromozomů. Dále v každé iteraci dojde pro každý chromozom k jeho ohodnocení vhodnostní funkcí a z danými pravděpodobnostmi k jeho mutaci a křížení. Složitost mutace je v tomto případně konstantní. Složitost křížení je lineární ku G. Finálně složitost vhodnostní funkce je kvadratická ku G. Celková složitost implementace tudíž je:

$$\mathcal{O}(G_{max} * G * (\mathcal{O}(h) + P_m * \mathcal{O}(m) + P_c * \mathcal{O}(c)) = \mathcal{O}(G_{max} * G * (G^2 + G) = \mathcal{O}(G_{max} * G^3)$$

Pro vypovídající srovnání s heuristickým algoritmem z kapitoly 4 je však potřeba dodat, že složitosti jsou vyjádřeny v závislosti na rozdílném parametru. Parametr G reprezentuje počet chromozomů, z nichž každý obsahuje n prvků (počet VS). Pro efektivní srovnání časových složitostí obou algoritmů lze například složitost genetického algoritmu vyjádřit jako $\mathcal{O}(G_{max} * (G * n)^3)$

Kapitola 6
Řešení za pomoci řešiče

Dále v této práci navrhuji nový postup řešení problému konsolidace. Jak jsem již v této práci zmínil, problém konsolidace je ekvivalentní problému plnění přihrádek, který je NP-úplný. Jako takový by tudíž měl jít převést na jiný (stejně či více obtížný k řešení). Čtenář by se v tuto chvíli mohl pozastavit nad tím, jaký by takový převod měl smysl. Důvodem je ten, že různé výpočetní problémy byly již předmětem mnoha výzkumů. Existují řešiče – programy určené k hledání řešení daných instancí těchto problémů. Myšlenka této metody tudíž v převodu problému konsolidace na problém (či formulaci), pro kterou existuje efektivní řešič a ten pak využít.

6.1 Problém splnitelnosti

Rád bych v tomto místě poděkoval prof. RNDr. Pavlu Surynkovi, Ph.D., který mi velmi pomohl objasněním převodu problému plnění přihrádek na problém splnitelnosti. Tato část textu by bez jeho pomoci pravděpodobně nevznikla.

Asi nejznámnějším NP-úplným [17] problémem je takzvaný problém splnitelnosti (známý také jako *SAT*). Jeho rozhodovací varianta spočívá v určení, zda je daná logická formule splnitelná či nikoliv.

Pokusím se zde alespoň nastínit část myšlenky převodu omezujících kritérií na SAT. Předpokládejme $\forall x \in \{0, 1\}$ logické proměnné. Každý virtualizační uzel je pak reprezentován logickými proměnnými v počtu odpovídajícímu počtu jeho výpočetních jader krát množství jeho operační paměti krát celkový počet virtuálních strojů (označme je pro účely této myšlenky popořadě A, B, C). Vhodná je v tomto případě vizuální představa jako „kvádr o těchto rozměrech“. V takovém případě reprezentují rozměry A a B jednotlivé výpočetní prostředky daného **VU**. RozměrC pak představuje, který **VS** tyto zdroje využívá. Jelikož nesmí dojít k využití libovolného zdroje více virtuálními stroji, musí platit:

$$\forall a \in A, \forall b \in B : \bigwedge_{c,c\prime \in C}^{c \neq c\prime} (\neg x_{abc} \vee \neg x_{abc\prime})$$

Obdobným způsobem je nutné zkonstruovat formuli omezující počet prostředků každého **VS** na přesně požadovaný počet (ve všech dimenzí). Zároveň je nutné formulovat pro všechny **VS** exklusivitu umístění tj následující formule platí maximálně pro jeden „kvádr“:

$$\forall a \in A, \forall b \in B : \bigvee_{c \in C} x_{abc\prime}$$

```
max: 3 x1 + 2 x2 - 40 ~x3;
x1 + x2 = 2;
3 x1 - 2 x2 - 2 x3 <= 1;
```

Obrázek 6.1 Ukázka jednoduchého souboru formátu .opb

Případný převod optimalizačního kritéria pak je také možný [18], avšak existují jiné formulace a problémy, které umožňují výrazně snazší formulaci a použití řešičů.

6.2 Pseudo boolean optimalizace

Výrazně vhodnější variantou je z mého pohledu pseudo boolean optimalizace. Tato alternativa umožňuje snazší formulaci omezujících kritérií. Zároveň však tento formát umožňuje přirozený popis optimalizačního kritéria pomocí tzv. účelové funkce.

Vědecká komunita pravidelně pořádá soutěže pseudo boolean řešičů [19], tudíž i řešiče určené pro tento problém se neustále zdokonalují. Zajímavostí je, že některé z nich k pseudo boolean optimalizaci přistupují pomocí jejího převodu na SAT [18].

Zvolil jsem tedy tuto variantu jako nejvhodnější.

6.2.1 Volba řešiče

Rád bych v tomto místě ještě zmínil jednu problematiku. Samotná volba řešiče může být komplikovaná. V rámci tvorby této práce jsem například dlouho experimentoval s řešičem *opdb*, dostupným z [20]. Ten se však i pro potřeby tohoto výzkumu ukázal jako výpočetní silou nedostačující, jelikož i pro extrémně malé instance problému konsolidace byl výpočet extrémně pomalý.

Domnívám se, že různé řešiče implementují různé algoritmy a heuristiky, z nichž ne všechny mohou být vhodné pro velmi specifickou formulaci uvedenou níže (6.5).

6.3 Formát .opb

Exaktní popis formátu určeného pro řešiče PBO lze nalézt na [21]. V této práci uvedu vágnější avšak snáze srozumitelný popis. Problém PBO se tedy skládá z:

1. účelové funkce
2. omezení

Účelová funkce je vždy uvedena jako první. Skládá se z označení cíle (například *max* jako maximalizuj) a za dvojtečkou samotné celočíselné funkce boolovských proměnných.

Zatímco účelová funkce je vždy jen jedna, omezeních může problém obsahovat libovolné množství. Omezení pak mohou představovat celočíselné rovnice nebo nerovnice boolovských proměnných. Pouze takové kombinace proměnných, pro které jsou splněna všechna uvedená omezení, jsou přípustné jako vzory účelové funkce.

Účelová funkce i jednotlivá omezení jsou odděleny středníkem.

Přípustné operace v účelové funkci i jejích omezeních jsou pouze aritmetické sčítání, odčítání, násobení a negace logické proměnné. Sčítání a odčítání jsou značeny klasicky pomocí + a -, násobení jednoduchým zřetězením (místo „$a * b$“ postačí „$a\ b$“) a negace pomocí ~.

Jednoduchý případ takového formátu je uveden na obrázku 6.1.

Existují však i jiné, více či méně podobné formáty PBO. Například .pbo již využívá symbol násobení a místo využití oddělovače považuje každý řádek za samostatné omezení.

Zároveň je třeba podotknout, že ačkoliv je formát .opb velmi obecný, velké množství řešičů a jejich soutěží se zaměřuje pouze na lineární či kvadratickou variantu tohoto problému. V takovém případě jsou pak na funkci i její omezení kladeny odpovídající nároky.

6.4 Kvadratická pseudo boolean optimalizace

Dále v této práci vyjadřuji optimalizační kritérium konsolidace jako kvadratickou celočíselnou funkci boolovkých proměnných. Z tohoto důvodu se domnívám, že je vhodné zmínit o této kategorii něco víc.

Minimalizace či maximalizace hodnoty kvadratické pseudo boolean funkce je NP-úplný problém. [22]

6.4.1 Linearizace celočíselných pseudo boolean funkcí

Jedním ze způsobů řešení tohoto podtypu je jeho převod na lineární alternativu pomocí tzv. procesu linearizace. V minulosti již bylo navrhnuto několik způsobů například [23].

V tomto textu pro ilustraci uvádím známou a široce aplikovatelnou techniku linearizace uvedenou v [24]:

▶ **Definice 6.1** (Jednoduchá linearizace)**.** *Mějme celočíselnou funkci boolovských proměnných. Tato funkce lze vyjádřit pomocí lineární funkce stejného typu tak, že:*

1. *každý kvadratický člen $x_i x_j$ nahradíme členem $y_i j$*
2. *pro každý z takto nově vytvořených členů přidáme následující omezení:*
   - $y_i j \leq x_i$
   - $y_i j \leq x_j$
   - $y_i j \geq x_i + x_j - 1$

6.5 Implementace převodu souboru .DTC na .opb

Pro převod již popsaného problému konsolidace na OPB je nutné nejdříve vyjádřit umístění konkrétního **VS** na konkrétním **VU** pomocí booleanovských proměnných. Uvažujme tedy opět S) množinu všech **VS** a U množinu všech **VU**. Pak definujeme všechny boolovské proměnné x_{su}, kde $s \in S$ a $u \in U$, jako:

- 1 – v případě že **VS** s se nachází na **VU** u
- 0 – jinak

Uvažujme pak s_{RAM} jako nároky s na operační paměť u_{RAM} dostupnou operační paměť v a obdobně pro CPU. Omezující kritéria lze pak formulovat pomocí omezení PBO následujícím způsobem:

$$\forall s \in S : \sum_{u \in U} x_{su} = 1$$

$$\forall u \in U : \sum_{s \in S} (x_{su} * s_{RAM}) <= u_{RAM}$$

$$\forall u \in U : \sum_{s \in S} (x_{su} * s_{CPU}) <= u_{CPU}$$

```
min: 93205*x6*x10 + 186410*x6*x11 + 93205*x7*x9 + 93205*x7*x11
+ 186410*x8*x9 + 93205*x8*x10 + 39296*x0*x7 + 78592*x0*x8 +
39296*x1*x6 + 39296*x1*x8 + 78592*x2*x6 + 39296*x2*x7 +
13419*x3*x10 + 26838*x3*x11 + 13419*x4*x9 +
13419*x4*x11 + 26838*x5*x9 + 13419*x5*x10 +
99*x0*x4 + 198*x0*x5 + 99*x1*x3 + 99*x1*x5
+ 198*x2*x3 + 99*x2*x4 + 80*x0*x10 + 160*x0*x11
+ 80*x1*x9 + 80*x1*x11 + 160*x2*x9 + 80*x2*x10;
1*x0 + 1*x1 + 1*x2 = 1;
1*x3 + 1*x4 + 1*x5 = 1;
1*x6 + 1*x7 + 1*x8 = 1;
1*x9 + 1*x10 + 1*x11 = 1;
1*x0 + 3*x3 + 1*x6 + 3*x9 <= 4;
2*x0 + 4*x3 + 2*x6 + 4*x9 <= 16;
1*x1 + 3*x4 + 1*x7 + 3*x10 <= 4;
2*x1 + 4*x4 + 2*x7 + 4*x10 <= 16;
1*x2 + 3*x5 + 1*x8 + 3*x11 <= 4;
2*x2 + 4*x5 + 2*x8 + 4*x11 <= 16;
```

Obrázek 6.2 Ukázka malé instance problému konsolidace ve formátu .opb

Dále nechť $dist(s, s')$ je **LV** s, $s' \in S$ a $comm(s, s')$ označuje množství libovolných jednotek síťové komunikace odeslaných mezi s, $s' \in S$. Optimalizační kritérium lze vyjádřit jako následující účelová funkce:

$$\forall s, s' \in S, \forall u \in U : min : \sum (comm(s, s') * dist(s, s') * x_{su} * x_{s'u})$$

Samotná implementace převodu je pak velmi jednoduchá. Čistý převod formátu .DTC na .opb se efektivně skládá pouze z přepočtu indexů logických proměnných. Ukázku převedené instance o 4 **VS** a 3 **VU**, kde účelová funkce obsahuje pouze kvadratické členy s nenulovou konstantou zobrazuje obrázek 6.2.

Má implementace však neslouží pouze jako převod. Kvůli zajištění netriviální spočetnosti instancí generovaných pro účely kapitoly 7 je počet **VU** roven 50 a počet **VS** sahající k 1200. Testované instance problému by tudíž kompletním PBO popisu dosáhly až $\sum_{i=1}^{1200}((i-1)*50*49) = 1762530000$ rozdílných kvadratických členů účelové funkce s nenulovou konstantou. Takový soubor .opb má sám o sobě několik GB a ani v této práci použitý výkonný řešič není schopen instanci takýchto rozměrů řešit v čase, který bych považoval za rozumný či snesitelný.

Z tohoto důvodu implementace umí problém redukovat na menší problémy podle následujících vstupních argumentů:

1. soubor .DTC
2. limit počtu komunikujících dvojic **VS** – pokud není stanoven příliš vysoko, výsledná instance PBO bude brát v potaz pouze takové **VS**, které jsou mezi tolika nejvíce komunikujícími páry **VS**
3. limit vzdálenosti – pokud není stanoven příliš vysoko, výsledná instance připouští migraci **VS** pouze na takové **VU**, které jsou vzdáleny do tohoto limitu od jejich původního umístění
4. fixované **VS** – nepovinný argument jména souboru, obsahující proměnné celkové instance PBO, které reprezetují takové **VS**, jejichž umístění je pevně dáno (například z předchozího řešení dílčí instance) – tyto **VS** již nejsou obsaženy ve výstupu, který bere pouze v potaz jejich nároky na **VU**

Touto implementací jsem umožnil rozdělení celkové instance problému na několik menších dílčích problémů. Ty popisují jen omezenou část celkového problému, jsou však řešitelné ve dle mého názoru rozumném časovém intervalu.

Problémem tohoto přístupu je však možnost vytvoření neřešitelné instance vlivem omezení vzdálenosti migrací v kombinaci s obsazením dostupných **VU** s již fisovanými **VS**. Zároveň je třeba podotknout, že sloučením řešení těchto dílčích instancí pravděpodobně nevznikne optimální řešení instance původní. Jedná se pouze o způsob, jak se mu pokusit přiblížit rychleji, který však nemusí být úspěšný.

6.6 Řešič SCIP

Pro účely této práce se nakonec projevila jako vhodná optimalizační sada SCIP [25, 26]. Jedná se o komplexní nekomerční řešič, který obsahuje různé algoritmy a heuristiky k předzpracování i řešení různých optimalizačních problémů. Podporuje i řešení lineární a kvadratické pseudo boolean optimalizace.

Tento řešič má 3 režimy použití:

- integrace do jiné aplikace pomocí jeho aplikačního rozhraní
- spuštění v interaktivním režimu
- ovládání pomocí argumentů při spuštění

Pro účely testování pro tuto práci jsem se rozhodl použít interaktivní režim, jelikož umožňuje snadný export řešení v kombinaci s nastavením časového limitu běhu. Ostatní parametry běhu jsem ponechal v defaultních hodnotách, které pro měření zde uvedená postačují. Domnívám se, že studium optimálního nastavení pro problém konsolidace virtuálních strojů bude zdlouhavé a rád bych se mu věnoval v budoucím výzkumu.

6.7 Použití řešiče SCIP na v této práci formulovaný problém konsolidace

Pro automatické testování jsem pak použil možnost spuštění vších příkazů interaktivního módu jako argument při spuštění řešiče.

Skript v jazyce bash použitý k testování pak vypadal zhruba takto 6.1. Instance, ne kterých je test proveden jsou určeny argumenty skriptu. Poté uživatel zadá jméno pro dočasné soubory pro aktuální test. Následně probíhá vygenerování a řešení jednotlivých dílčích instancí, dokud je možné další vytvářet, nebo dokud řešič nenarazí na neřešitelnou instanci. Po vyřešení dílčí instance dojde k uložení řešení, odstranění interních proměnných řešiče případnému vygenerování další dílčí instance.

6.7.1 Ukázky výstupu řešiče SCIP

V této sekci bych rád zběžně ukázal jevy, se kterými jsem se v průběhu používání řešiče SCIP setkal.

Obrázek 6.3 ukazuje předzpracování načteného problému. K tomu dochází vždy před pokusem o jeho vyřešení. V rámci tohoto procesu dochází k reformulaci problému (například pomocí slučnování omezení PBO, které jsou také kategorizovány). Domnívám se, že v této fázi je též rozhodnut počáteční algoritmus, který řešič použije při řešení.

Další obrázek 6.4 ukazuje postupně nalezená řešení. Za povšimnutí zde stojí obzvlášť sloupce *dualbound, primalbound* a *gap*. Ty reprezentují heuristický či vypočtený odhad optimálního řešení a jak moc se mu aktuální řešení blíží.

SCIP občas v průběhu řešení vyhodnotí, že aktuálně používaný algoritmus již nemusí být dále efektivný. V takovém případě běh přeruší, problém přeformuluje a pokusí se k jeho dořešení jiným algoritmem. Toto chování demonstruje obrázek 6.5

Výpis kódu 6.1 Testovací skript pro řešič SCIP

```
#!/bin/bash
read f
for i in "$@"; do
ret=1
echo "  $i Started"
touch ./result/$i
while [[ $ret -eq 1 ]]; do
./convert.out ./instance/$i.DTC 100 3 ./result/$i >./$f.opb
ret=$?
scip -c \
"read ./$f.opb set limits time 600 optimize write solution $f.sol quit" \
>>./output/$i
cat $f.sol >>./output/$i
t=$(cat $f.sol | fgrep infeasible | wc -l)
if [[ $t -ne 0 ]]; then
echo "infeasible found!!!"
break
fi
cat $f.sol | tail -n +3 | fgrep -v andresultant | \
awk '{print $1}' >>./result/$i
echo "iter done"
done
echo "  $i Done"
done
```

```
read problem <./a.opb>
============

original problem has 3250 variables (815 bin, 0 int, 2435 impl, 0 cont) and 2698 constraints
limits/time = 600

presolving:
(round 1, fast)       0 del vars, 40 del conss, 0 add conss, 0 chg bounds, 25 chg sides, 50 chg coeffs, 0 upgd conss, 0 impls, 5033 clqs
(round 2, exhaustive) 0 del vars, 40 del conss, 0 add conss, 0 chg bounds, 25 chg sides, 50 chg coeffs, 223 upgd conss, 0 impls, 5033 clqs
   (0.3s) probing: 1000/3250 (30.8%) - 0 fixings, 0 aggregations, 27615 implications, 0 bound changes
   (0.3s) probing: 1001/3250 (30.8%) - 0 fixings, 0 aggregations, 27641 implications, 0 bound changes
   (0.3s) probing aborted: 1000/1000 successive useless probings
   (0.3s) symmetry computation started: requiring (bin +, int -, cont +), (fixed: bin -, int +, cont -)
   (0.3s) symmetry computation finished: 64 generators found (max: 1500, log10 of symmetry group size: 33.6)
(round 3, exhaustive) 0 del vars, 40 del conss, 8 add conss, 0 chg bounds, 25 chg sides, 50 chg coeffs, 223 upgd conss, 0 impls, 9505 clqs
presolving (4 rounds: 4 fast, 3 medium, 3 exhaustive):
 0 deleted vars, 40 deleted constraints, 8 added constraints, 0 tightened bounds, 0 added holes, 25 changed sides, 51 changed coefficients
 0 implications, 9505 cliques
presolved problem has 3250 variables (815 bin, 0 int, 2435 impl, 0 cont) and 2666 constraints
     60 constraints of type <knapsack>
    163 constraints of type <setppc>
   2435 constraints of type <and>
      8 constraints of type <orbitope>
transformed objective value is always integral (scale: 1)
Presolving Time: 0.33
```

Obrázek 6.3 Ukázka předzpracování problému řešičem SCIP

```
 time | node  | left  |LP iter|LP it/n|mem/heur|mdpt |vars |cons |rows |cuts |sepa|confs|strbr|  dualbound   | primalbound  |  gap   | compl.
p 0.4s|     1 |     0 |     0 |     - |  clique|   0 |3250 |2666 |5093 |   0 |  0 |   0 |   0 | 0.000000e+00 | 3.722160e+05 |    Inf | unknown
p 0.4s|     1 |     0 |     0 |     - | vbounds|   0 |3250 |2668 |5093 |   0 |  0 |   2 |   0 | 0.000000e+00 | 3.662500e+05 |    Inf | unknown
  0.5s|     1 |     0 |   427 |     - |    59M |   0 |3250 |2693 |5105 |   0 |  0 |  27 |   0 | 0.000000e+00 | 3.662500e+05 |    Inf | unknown
L 1.0s|     1 |     0 |   453 |     - |undercov|   0 |3250 |2695 |5105 |   0 |  0 |  29 |   0 | 0.000000e+00 | 3.603500e+05 |    Inf | unknown
  1.1s|     1 |     0 |   652 |     - |    61M |   0 |3250 |2695 |5180 |  75 |  1 |  29 |   0 | 2.966000e+03 | 3.603500e+05 |  Large | unknown
  1.3s|     1 |     0 |   873 |     - |    61M |   0 |3250 |2692 |5195 | 143 |  2 |  30 |   0 | 1.586300e+04 | 3.603500e+05 |2171.64%| unknown
  1.4s|     1 |     0 |  1068 |     - |    62M |   0 |3250 |2677 |5208 | 198 |  3 |  31 |   0 | 2.597375e+04 | 3.603500e+05 |1287.36%| unknown
  1.5s|     1 |     0 |  1195 |     - |    62M |   0 |3250 |2677 |5259 | 249 |  4 |  31 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  1.7s|     1 |     0 |  1252 |     - |    63M |   0 |3250 |2677 |5284 | 274 |  5 |  31 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  1.8s|     1 |     0 |  1427 |     - |    64M |   0 |3250 |2677 |5363 | 353 |  6 |  31 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  1.9s|     1 |     0 |  1581 |     - |    65M |   0 |3250 |2677 |5443 | 433 |  7 |  31 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.1s|     1 |     0 |  1860 |     - |    65M |   0 |3250 |2678 |5483 | 513 |  8 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.2s|     1 |     0 |  2003 |     - |    66M |   0 |3250 |2678 |5548 | 578 |  9 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.4s|     1 |     0 |  2145 |     - |    67M |   0 |3250 |2678 |5614 | 644 | 10 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.4s|     1 |     0 |  2200 |     - |    68M |   0 |3250 |2678 |5649 | 679 | 11 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
 time | node  | left  |LP iter|LP it/n|mem/heur|mdpt |vars |cons |rows |cuts |sepa|confs|strbr|  dualbound   | primalbound  |  gap   | compl.
  2.4s|     1 |     0 |  2215 |     - |    68M |   0 |3250 |2678 |5658 | 688 | 12 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.4s|     1 |     0 |  2327 |     - |    69M |   0 |3250 |2678 |5556 | 755 | 13 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.4s|     1 |     0 |  2327 |     - |    69M |   0 |3250 |2678 |5556 | 755 | 13 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.5s|     1 |     0 |  2391 |     - |    69M |   0 |3250 |2678 |5603 | 802 | 14 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.6s|     1 |     0 |  2408 |     - |    69M |   0 |3250 |2606 |5603 | 802 | 14 |  32 |   0 | 2.966000e+04 | 3.603500e+05 |1114.94%| unknown
  2.6s|     1 |     0 |  2408 |     - |    69M |   0 |3250 |2606 |5603 | 802 | 14 |  32 |   0 | 3.962000e+04 | 3.603500e+05 | 809.52%| unknown
  2.6s|     1 |     0 |  2421 |     - |    70M |   0 |3250 |2606 |5614 | 813 | 15 |  32 |   0 | 3.962000e+04 | 3.603500e+05 | 809.52%| unknown
  2.6s|     1 |     0 |  2456 |     - |    71M |   0 |3250 |2606 |5638 | 837 | 16 |  32 |   0 | 3.962000e+04 | 3.603500e+05 | 809.52%| unknown
```

Obrázek 6.4 Ukázka postupného nacházení řešení problému řešičem SCIP

```
  3.8s|     1 |     2 |  4030 |     - |    75M |   0 |3250 |2606 |5349 | 945 | 25 |  32 |  10 | 3.962000e+04 | 3.603500e+05 | 809.52%| unknown
(run 1, node 1) restarting after 95 global fixings of integer variables

(restart) converted 371 cuts from the global cut pool into linear constraints

presolving:
(round 1, fast)       115 del vars, 23 del conss, 0 add conss, 0 chg bounds, 0 chg sides, 20 chg coeffs, 0 upgd conss, 0 impls, 8980 clqs
(round 2, medium)     116 del vars, 26 del conss, 0 add conss, 0 chg bounds, 0 chg sides, 20 chg coeffs, 0 upgd conss, 0 impls, 8978 clqs
(round 3, exhaustive) 120 del vars, 32 del conss, 5 add conss, 0 chg bounds, 0 chg sides, 20 chg coeffs, 0 upgd conss, 0 impls, 8950 clqs
(round 4, exhaustive) 120 del vars, 33 del conss, 5 add conss, 0 chg bounds, 1 chg sides, 20 chg coeffs, 368 upgd conss, 0 impls, 8950 clqs
(round 5, medium)     120 del vars, 33 del conss, 6 add conss, 0 chg bounds, 3 chg sides, 23 chg coeffs, 368 upgd conss, 0 impls, 8950 clqs
(round 6, exhaustive) 120 del vars, 35 del conss, 6 add conss, 0 chg bounds, 3 chg sides, 97 chg coeffs, 368 upgd conss, 0 impls, 8950 clqs
presolving (7 rounds: 7 fast, 6 medium, 4 exhaustive):
 120 deleted vars, 35 deleted constraints, 6 added constraints, 0 tightened bounds, 0 added holes, 3 changed sides, 106 changed coefficients
 0 implications, 8950 cliques
presolved problem has 3130 variables (800 bin, 0 int, 2330 impl, 0 cont) and 2948 constraints
    344 constraints of type <knapsack>
    165 constraints of type <setppc>
   2330 constraints of type <and>
      5 constraints of type <orbitope>
    104 constraints of type <logicor>
transformed objective value is always integral (scale: 1)
Presolving Time: 0.36
transformed 3/5 original solutions to the transformed problem space

 time | node  | left  |LP iter|LP it/n|mem/heur|mdpt |vars |cons |rows |cuts |sepa|confs|strbr|  dualbound   | primalbound  |  gap   | compl.
  3.9s|     1 |     0 |  4465 |     - |    75M |   0 |3130 |2948 |5252 |   0 |  0 |  32 |  10 | 3.962000e+04 | 3.603500e+05 | 809.52%| unknown
  4.0s|     1 |     0 |  4665 |     - |    77M |   0 |3130 |2948 |5336 |  84 |  1 |  32 |  10 | 3.962000e+04 | 3.603500e+05 | 809.52%| unknown
  4.2s|     1 |     0 |  4923 |     - |    77M |   0 |3130 |2946 |5370 | 158 |  2 |  33 |  10 | 4.061000e+04 | 3.603500e+05 | 787.34%| unknown
```

Obrázek 6.5 Ukázka změny algoritmu řešičem SCIP v průběhu řešení

```
  599s| 57800 | 57801 |  1451k|  24.6 |   599M |  45 |3045 |3909 |4883 |  86k|  1 |  14k|3375 | 1.622550e+05 | 3.544100e+05 | 118.43%| unknown

SCIP Status        : solving was interrupted [time limit reached]
Solving Time (sec) : 600.00
Solving Nodes      : 57880 (total of 58930 nodes in 3 runs)
Primal Bound       : +3.54410000000000e+05 (200 solutions)
Dual Bound         : +1.62255000000000e+05
Gap                : 118.43 %


written solution information to file <a.sol>


solution status: time limit reached
objective value:                                354410
x15679                                               1   (obj:0)
x4906                                                1   (obj:0)
x15074                                               1   (obj:0)
x13324                                               1   (obj:0)
x18584                                               1   (obj:0)
x10019                                               1   (obj:0)
```

Obrázek 6.6 Ukázka dosažení časového limitu řešičem SCIP

```
read problem <./a.opb>
============

original problem has 2555 variables (135 bin, 0 int, 2420 impl, 0 cont) and 2527 constraints
limits/time = 600

presolving:
presolving (1 rounds: 1 fast, 0 medium, 0 exhaustive):
 1546 deleted vars, 65 deleted constraints, 0 added constraints, 60 tightened bounds, 0 added holes, 40 changed sides, 11 changed coefficients
 0 implications, 2063 cliques
presolving detected infeasibility
Presolving Time: 0.03

SCIP Status        : problem is solved [infeasible]
Solving Time (sec) : 0.05
Solving Nodes      : 0
Primal Bound       : +1.00000000000000e+20 (0 solutions)
Dual Bound         : +1.00000000000000e+20
Gap                : 0.00 %


written solution information to file <a.sol>


solution status: infeasible
no solution available
```

Obrázek 6.7 Ukázka nalezení neřešitelné instance řešičem SCIP

Jelikož v některých případech se projevila i optimalizace vygenerovaných dílčích instancí jako časově náročná. Z tohoto důvodu při svých měřeních nastavil časový limit na 10 minut. V případě, že časový limit vyprší, je běh ukončen a za řešení se považuje dosud nejoptimálnější nalezené. Toto lze vidět na obrázku 6.6.

Další obrázek 6.7 ukazuje stav, kdy řešič vyhodnotil, že daná instance problému není řešitelná. K tomuto jevu může dojít v průběhu předzpracování i řešení daného problému.

Poslední obrázek 6.8 pak reprezentuje variantu, kdy řešič nalezl řešení dosahující dříve vypočteného či odhadnutého optima. V takovém případě není nutné pokračovat v hledání dalších řešení.

```
  339s| 51300 |    18 |  1758k|  33.1 |   238M |  36 |1281 |1450 |2945 |  96k|  0 |2509 |3936 | 1.565218e+05 | 1.599760e+05 |   2.21%|  99.39%

SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 339.69
Solving Nodes      : 51342 (total of 52918 nodes in 2 runs)
Primal Bound       : +1.59976000000000e+05 (92 solutions)
Dual Bound         : +1.59976000000000e+05
Gap                : 0.00 %

written solution information to file <a.sol>


solution status: optimal solution found
objective value:                                159976
x13020                                               1   (obj:0)
x16728                                               1   (obj:0)
x18930                                               1   (obj:0)
x8064                                                1   (obj:0)
```

Obrázek 6.8 Ukázka nalezení optimálního řešení problému řešičem SCIP

Kapitola 7

Experimentální vyhodnocení

7.0.1 Testovací data

Za účelem otestování dat jsem vytvořil několik typů datových sad. Jednotlivé typy mají rozdílné vlastnosti a tudíž by se pro ně jednotlivé algoritmy mohly chovat rozdílně. Tyto datové sady jsem pojmenoval:

- genLow
- random
  - light
  - medium
  - hard
- typed
  - light
  - medium
  - hard

Každá datová sada pak obsahuje 160 náhodně vygenerovaných instancí s příslušnými parametry.

7.0.1.1 Společné vlastnosti vygenerovaných instancí

Vygenerované instance mají vždy 50 virtualizačních uzlů a mezi 250 a 1200 virtuálními stroji. Zároveň pravděpodobnost vytvoření clusteru mezi dvěma virtuálními stroji byla ve všech případech 5%. Závěrem ve všech generovaných instancích bylo 40 z 50 virtualizačních uzlů plně zaplněno virtuálními stroji. Důvodem je snaha předejít triviálně řešitelným instancím, kdy by omezující kritéria neměla vliv na testované metody.

V poslední řadě pak všechny instance byly vygenerovány jako simulace standardí třívrstvé topologie (viz. 3.1). Tato vlastnost by však neměla mít vliv na kvalitu výsledků, jelikož ani jedna z v této práci poopsaných metod neklade na síťovou infrastrukturu žádné předpoklady.

7.0.1.2 Dataset genLow

Definující vlastností této datové sady je nízká komunikace v rámci clusterů. Teoretická maximální komunikace mezi 2 virtuálními stroji uvnitř clusteru nikdy nepřekročí pětinásobek teoretické maximální komunikace mezi 2 libovolnými virtuálními stroji. V tomto datasetu by dle mého názoru měla heuristika z kapitoly 4 být prakticky nepoužitelná. Zbylé dvě metody by tudíž měly vykazovat lepší výsledky.

7.0.1.3 Dataset random

Jedná se o nejobecnější datovou sadu. Virtuální stroje jsou zde vygenerovány s různorodými (nikoliv však čistě náhodnými) parametry. Vyšší variabilita jednotlivých **VS** by tak měla zvyšovat náročnost hledání potenciálních umístění **VS** na **VU**. Očekával bych, že v tomto případě bude nejlepší výsledky vykazovat metoda za použití řešiče SCIP.

7.0.1.4 Dataset typed

Tato varianta je alternativou k typu *random*. Všechny virtuální stroje jsou téměř homogenní, tj. existuje pouze velmi nízký počet typových **VS**. (Například velikost operační paměti virtuálních strojů mohla být jen 4, 8 nebo 16 apod.) To by mělo umožnit větší množství migrací a tudíž i snazší umístění jednolivých **VS** na **VU**. Domnívám se, že zde má heuristický algoritmus z 4 největší šanci konkurovat zbylým metodám.

7.0.1.5 Typy datasetů podle zaplnění virtualizačních uzlů

Datasety *typed* a *random* se pak dále dělí podle zaplnění zbylých virtualizačních uzlů. V průměry by měly být vytíženy z následujících procentech své kapacity:

- light – 30%
- medium – 55%
- hard – 65%

Tento parametr má opět vliv hlavně na splnění omezujících kritérií v průběhu řešení.

7.1 Testování jednotlivých metod a jejich parametry

Heuristika popsaná v 4 vyžaduje jako vstupní argumenty hodnoty určující hranice tvorby a rozšiřování clusterů. V rámci své práce na této problematice se mi nepodařilo určit obecnou optimální hodnotu těchto parametrů. V rámci testování tudíž tuto metodu pouštím s více hodnotami těchto argumentů a za výsledek považuji ten nejoptimálnější.

Genetický algoritmus v 5.1 je oproti tomu ze své podstaty ovlivněný náhodou a pravděpodobností. Abych tento vliv kompenzoval, uvažuju u této metody nejlepší ze 3 nezávislých běhů.

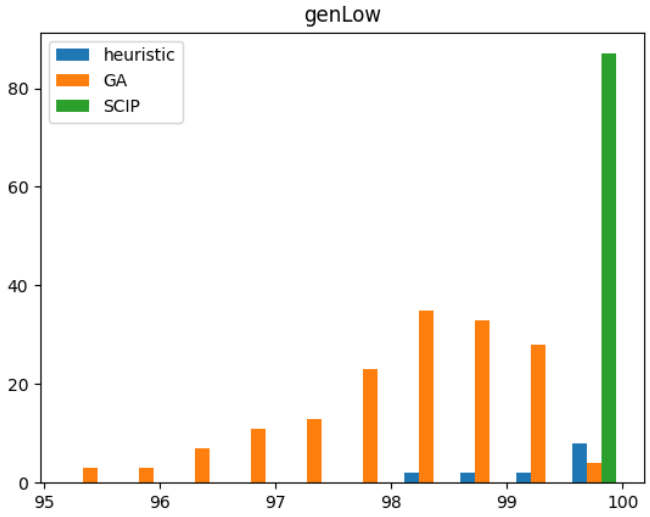
Řešení pomocí optimalizační sady SCIP pak disponovalo znatelně větším množstvím času výpočtu.

7.1.1 Selhání algoritmů

Všechny implementované metody pak mohly selhat dvěma způsoby. Buď nenalezly takové rozmístění virtuálních uzlů,které by splňovalo omezující kritéria, nebo navržené řešení bylo v kontextu optimalizačního kritéria horší, než původní stav. V této práci mezi těmito dvěma chybami v rámci provedených testů nerozlišuji.

Algoritmus	Dosažená procentuální část optimalizačního kritéria v původním stavu	Počet selhání algoritmu	Množství migrované operační paměti
Heuristika	99.3%	146	5814 GB
Genetický algoritmus	98.2%	0	1355 GB
metoda pomocí SCIP	99.9%	73	12752 GB

Tabulka 7.1 Tabulka úspěšnosti pro sadu genLow



Obrázek 7.1 Histogram zlepšení optimalizačního kritéria pro sadu genLow

7.1.1.1 Porovnání efektivity migrací

Jako další kritérium k porovnání jednotlivých metod jsem využil efektivitu migrací. Za předpokladů nastíněných o jednotlivých parametrech virtuálních strojů lze usoudit, že v případě migrace po síťové infrastruktuře bude mít nejvetší podíl migrace operační paměti.

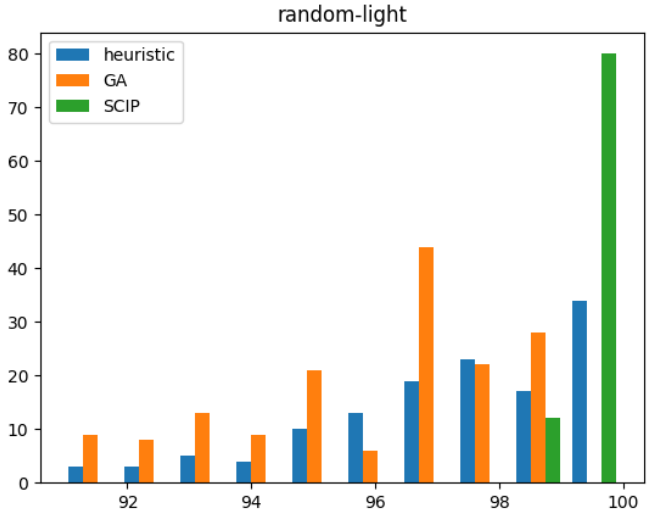
Pro kompletnost vyhodnocení tudíž porovnávám i celkový objem operační paměti, který je potřeba k dosažení navrhovaného stavu.

7.2 Výsledky testů

Následující histogramy a tabulky shrnují výsledky provedených měření. Pro každý dataset je uvedena tabulka obsahující průměrnou hodnotu dosaženého optimalizačního kritéria vzhledem k původnímu stavu, počet testovacích instancí, kde metoda selhala a průměrné množství migrované operační paměti. Samotný histogram rozložení jednotlivých hodnot optimalizačního kritéria v vzhledem k původním stavům může posloužit mimo jiné ke znázornění stability kvality výsledků dané metody.

Algoritmus	Dosažená procentuální část optimalizačního kritéria v původním stavu	Počet selhání algoritmu	Množství migrované operační paměti
Heuristika	97.3%	29	8288 GB
Genetický algoritmus	96%	0	1617 GB
metoda pomocí SCIP	99.5%	68	12488 GB

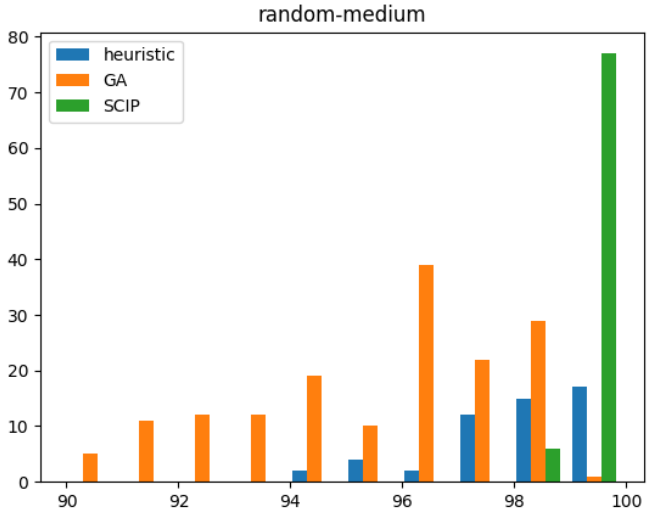
Tabulka 7.2 Tabulka úspěšnosti pro sadu random-light



Obrázek 7.2 Histogram zlepšení optimalizačního kritéria pro sadu random-light

Algoritmus	Dosažená procentuální část optimalizačního kritéria v původním stavu	Počet selhání algoritmu	Množství migrované operační paměti
Heuristika	98%	108	6439 GB
Genetický algoritmus	95.6%	0	1635 GB
metoda pomocí SCIP	99.5%	77	12543 GB

Tabulka 7.3 Tabulka úspěšnosti pro sadu random-medium



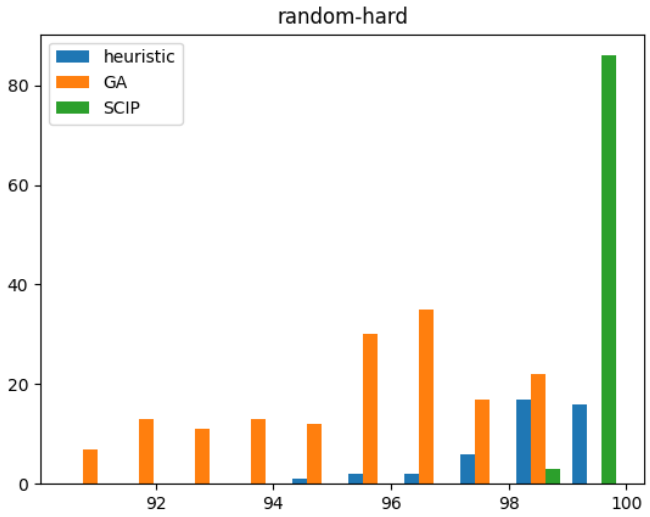
Obrázek 7.3 Histogram zlepšení optimalizačního kritéria pro sadu random-medium

Algoritmus	Dosažená procentuální část optimalizačního kritéria v původním stavu	Počet selhání algoritmu	Množství migrované operační paměti
Heuristika	98.4%	116	6209 GB
Genetický algoritmus	95.5%	0	1674 GB
metoda pomocí SCIP	99.5%	71	12851 GB

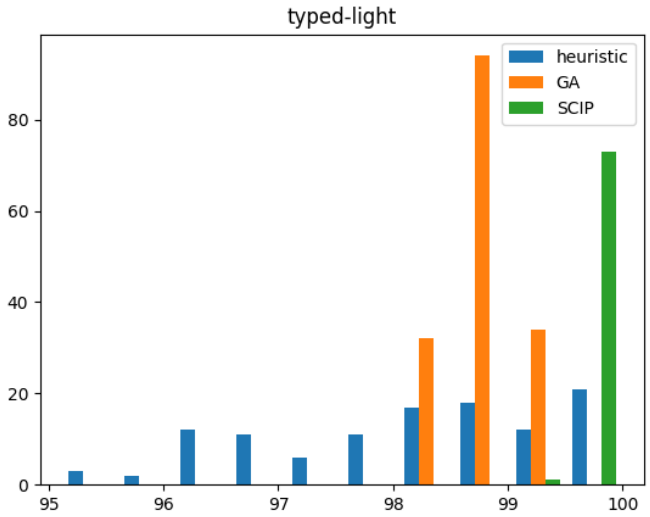
Tabulka 7.4 Tabulka úspěšnosti pro sadu random-hard

Algoritmus	Dosažená procentuální část optimalizačního kritéria v původním stavu	Počet selhání algoritmu	Množství migrované operační paměti
Heuristika	98.2%	47	3900 GB
Genetický algoritmus	98.8%	0	579 GB
metoda pomocí SCIP	99.7%	86	5782 GB

Tabulka 7.5 Tabulka úspěšnosti pro sadu typed-light



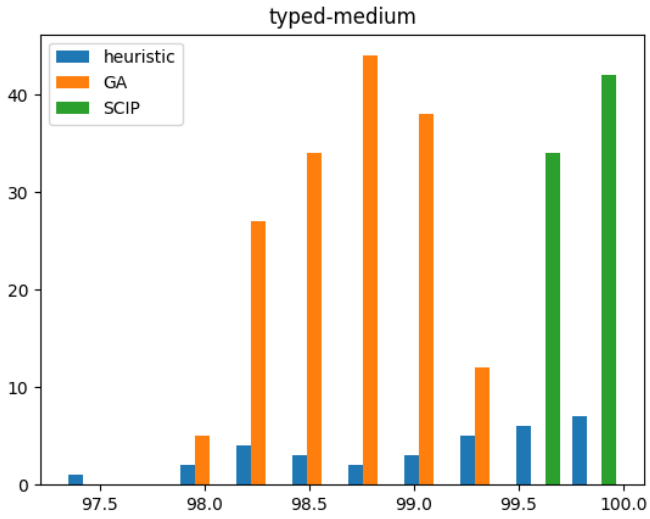
Obrázek 7.4 Histogram zlepšení optimalizačního kritéria pro sadu random-hard



Obrázek 7.5 Histogram zlepšení optimalizačního kritéria pro sadu typed-light

Algoritmus	Dosažená procentuální část optimalizačního kritéria v původním stavu	Počet selhání algoritmu	Množství migrované operační paměti
Heuristika	99%	127	3397 GB
Genetický algoritmus	98.7%	0	556 GB
metoda pomocí SCIP	99.7%	84	5867 GB

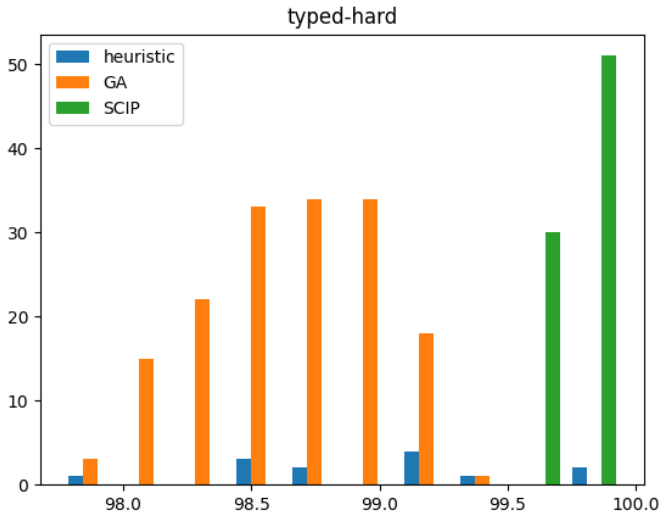
Tabulka 7.6 Histogram zlepšení optimalizačního kritéria pro sadu typed-medium



Obrázek 7.6 Histogram zlepšení optimalizačního kritéria pro sadu typed-medium

Algoritmus	Dosažená procentuální část optimalizačního kritéria v původním stavu	Počet selhání algoritmu	Množství migrované operační paměti
Heuristika	98.9%	147	3460 GB
Genetický algoritmus	98.6%	0	547 GB
metoda pomocí SCIP	99.75%	79	5989 GB

Tabulka 7.7 Histogram zlepšení optimalizačního kritéria pro sadu typed-hard



Obrázek 7.7 Histogram zlepšení optimalizačního kritéria pro sadu typed-hard

7.2.1 Interpretace výsledků

7.2.1.1 Heuristika založená na hledání a sdružování clusterů

Tato metoda jeví pozitivní znaky. Dle očekávání si metoda vedla velmi špatně v datasetu *genLow*. Pro datasety *random* a *typed* již jeví mnohem lepší výsledky. Pro velmi vytížená datacentra (instance *medium* a *hard* sice dochází k relativně častému selhání algoritmu. Pokud má však heuristika dostatek místa k migracím (instance *it*), začíná být více než srovnatelná s ostatními metodami. Zároveň jeví znaky rovnoměrného rozdělení zlepšení optimalizačního kritéria. To vše za řádově rychlejšího běhu, oproti oběma alternativám.

7.2.1.2 Celočíselné programování genetickým algoritmem

Nejlepší výsledky jeví právě tento přístup. Nejen že algoritmus stabilně dosahuje největší optimalizace, zároveň v žádném z měření neselhal. Z histogramů se pak jeví normální rozdělení dosažených optimalizací. Zároveň, navdory randomizované povaze algoritmu, dokonce vykazuje nejnižší množství migrované operační paměti. Jedná se tedy o velmi dobrou variantu.

7.2.1.3 Dílčí řešení řešičem SCIP

Výrazným překvapením pro mě byly výsledky částečných řešení pomocí řešiče pseudo boolean optimalizace. Tato metoda se projevila jako velmi stabilní, čímž ale její pozitivní stránky končí. Úspěšnost optimalizace zhruba v 50-60% případů v kombinaci s nízkou mírou optimalizace (zlepšení o maximálně 0.5%) a řádově vyšším množstvím migrované operační paměti tuto metodu v této formě dělá nepoužitelnou. To vše navzdory faktu, že tato metoda měla k dispozici výrazně více času k výpočtu, s průměrnou dobou běhu okolo 30 minut na instanci. Domnívám se, že se jedná o důsledek dělení problému na dílčí instance, které se však vzájemně neberou v potaz. Dochází zde tedy ke ztrátě informace a vypočtené hodnoty tudíž nemají takovou váhu.

Kapitola 8

Nasazení v systému OpenNebula

Pro praktické nasazení libovolné metody zmíněné v této práci je potřeba vyřešit několik dílčích problémů:

1. Sběr dat o rozmístění virtuálních strojů
2. Sběr informací o množství komunikace mezi virtuálními stroji
3. Určení logické vzdálenosti mezi virtualizačními uzly
4. Migraci virtuálních strojů

8.1 Sběr informací o množství komunikace mezi virtuálními stroji

Jedná se o asi nejobtížnější část, která vyžaduje sběr dat, ideálně na úrovni operačního systému virtuálních strojů či hypervizoru. V případě OpenNebuly by však za tímto účelem měl postačovat tento *Monitoring Driver* [27]. Nastává zde však problém identifikace komunikace uvnitř a mimo množinu virtuálních strojů.

8.2 Určení logické vzdálenosti mezi virtualizačními uzly

Vzhledem k tomu, že minimálně fyzická infrastruktura datacenter se mění spíš výjimečně, přichází v tomto směru v úvahu řešení pomocí konfiguračního souboru. Ten by pak teoreticky mohl jít aktualizovat například pomocí známé utility *traceroute*.

8.3 Sběr dat o rozmístění virtuálních strojů a jejich migrace

Za tímto účelem se dle mého názoru skvěle hodí knihovna *libvirt*. Jedná se o virtualizační aplikační rozhraní, které je kompatibilní s většinou používaných hypervizorů. Obsahuje i mimo jiné API přímo určené k zjišťování informací a migraci virtuálních strojů [28].

8.4 Praktická implementace

Bohužel, z časových důvodů [1] musím ponechat tuto kapitolu nedokončenou...

[1] odevzdání této práce

Kapitola 9
Závěr

V průběhu této práce jsem nastínil problematiku datacenter a s ní spojené problémy konsolidace a umisťování virtuálních strojů. Problém konsolidace jsem pak korektně zadefinoval včetně konkrétního optimalizačního kritéria. Dále jsem vytvořil generátor dat, který může být využit k simulaci libovolného datacentra a konsolidace v něm.

Následně jsem uvedl a implementoval tři možné způsoby problému konsolidace a jejich aplikaci na konkrétní variantu, zmíněnou v této práci. Objasnil jsem i principy a funkcionalitu těchto řešení.

Tyto metody jsem dále experimentálně porovnal na náhodně vygenerovaných instancích problému konsolidace a výsledky analyzoval. Jako suverénně nejvhodnější metoda se jeví celočíselný genetický algoritmus. Navzdory tomu bych se rád v budoucnosti vrátil k metodě řešení pomocí řešiče SCIP. Domnívám se, že tato metoda nedosáhla v této práci ani zdaleka svého potenciálu a stojí tudíž za další výzkum. Domnívám se, že i zmíněná metoda založená na hledání clusterů by mohla jít dál zdokonalit.

V závěru jsem pak bohužel nestihl implementovat nasazení genetického algoritmu do prostředí systému OpenNebula. Nastínil jsem tedy alespoň některé technologie, které považuji za vhodné k takovéto implementaci.

Většina cílů práce tudíž byla splněna s výjimkou praktického nasazení.

Rád bych se v budoucnosti k této problematice vrátil v rámci dalšího studia či výzkumu, jelikož mi přijde zajímavá a potřebnáv praxi.

Bibliografie

1. POLAK, Michal; FESL, Jan. Cluster-Oriented Virtual Machine Low Latency Consolidation Algorithm. In: *Proceedings of the 2022 8th International Conference on Computer Technology Applications*. Vienna, Austria: Association for Computing Machinery, 2022, s. 113–120. ICCTA '22. ISBN 9781450396226. Dostupné z DOI: 10.1145/3543712.3543743.
2. *VMware Infrastructure Architecture Overview* [online]. VMware, Inc., 2006 [cit. 2023-01-03]. Dostupné z: https://www.vmware.com/pdf/vi_architecture_wp.pdf.
3. MASDARI, Mohammad; NABAVI, Sayyid Shahab; AHMADI, Vafa. An overview of virtual machine placement schemes in cloud computing. *Journal of Network and Computer Applications*. 2016, roč. 66, s. 106–127. ISSN 1084-8045. Dostupné z DOI: https://doi.org/10.1016/j.jnca.2016.01.011.
4. LODI, Andrea; MARTELLO, Silvano; MONACI, Michele. Two-dimensional packing problems: A survey. *European Journal of Operational Research*. 2002, roč. 141, č. 2, s. 241–252. Dostupné z DOI: 10.1016/s0377-2217(02)00123-6.
5. JOHNSON, D. S.; DEMERS, A.; ULLMAN, J. D.; GAREY, M. R.; GRAHAM, R. L. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM Journal on Computing*. 1974, roč. 3, č. 4, s. 299–325. Dostupné z DOI: 10.1137/0203025.
6. BELOGLAZOV, A. *Energy-efficient management of virtual machines in data centers for cloud computing*. Department of Computing a Information Systems, The University of Melbourne, 2013. Dis. pr.
7. MOGES, Fikru Feleke; ABEBE, Surafel Lemma. Energy-aware VM placement algorithms for the OpenStack Neat Consolidation Framework. *Journal of Cloud Computing*. 2019, roč. 8, č. 1. Dostupné z DOI: 10.1186/s13677-019-0126-y.
8. YAVARI, Maede; GHAFFARPOUR RAHBAR, Akbar; FATHI, Mohammad Hadi. Temperature and energy-aware consolidation algorithms in cloud computing. *Journal of Cloud Computing*. 2019, roč. 8, č. 1. Dostupné z DOI: 10.1186/s13677-019-0136-9.
9. ZHOU, Biyu; WU, Jie; ZHANG, Fa; LIU, Zhiyong. Resource optimization for survivable embedding of virtual clusters in cloud data centers. *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. 2017. Dostupné z DOI: 10.1109/pccc.2017.8280436.
10. CHEN, Tao; GAO, Xiaofeng; CHEN, Guihai. Optimized virtual machine placement with traffic-aware balancing in data center networks. *Scientific Programming*. 2016, roč. 2016, s. 1–10. Dostupné z DOI: 10.1155/2016/3101658.
11. MENG, Xiaoqiao; PAPPAS, Vasileios; ZHANG, Li. Improving the scalability of data center networks with traffic-aware virtual machine placement. *2010 Proceedings IEEE INFOCOM*. 2010. Dostupné z DOI: 10.1109/infcom.2010.5461930.

12. YAPICIOGLU, Tevfik; OKTUG, Sema. A traffic-aware virtual machine placement method for Cloud Data Centers. *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. 2013. Dostupné z DOI: 10.1109/ucc.2013.62.
13. FERDAUS, Md Hasanul; MURSHED, Manzur; CALHEIROS, Rodrigo; BUYYA, Rajkumar. Network-Aware Virtual Machine Placement and Migration in Cloud Data Centers. In: 2015, s. 42–91. ISBN 9781466682139. Dostupné z DOI: 10.4018/978-1-4666-8213-9.ch002.
14. DIJKSTRA, Edsger W. A note on two problems in connexion with graphs. *Numerische mathematik*. 1959, roč. 1, č. 1, s. 269–271.
15. SHERALI, Hanif D.; DRISCOLL, Patrick J. Evolution and state-of-the-art in integer programming. *Journal of Computational and Applied Mathematics*. 2000, roč. 124, č. 1, s. 319–340. ISSN 0377-0427. Dostupné z DOI: https://doi.org/10.1016/S0377-0427(00)00431-3. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.
16. ZELINKA, Ivan; OPLATKOVÁ, Zuzana; ŠEDA, Miloš; OŠMERA, Pavel; VČELAŘ, František. *Evoluční Výpočetní techniky: Principy a aplikace*. BEN - technická literatura, 2009.
17. ZHOU, JianMing; LI, Yu. What is Cook's theorem? *ArXiv*. 2015, roč. abs/1501.01910.
18. EEN, Niklas; SÖRENSSON, Niklas. Translating Pseudo-Boolean Constraints into SAT. *JSAT*. 2006, roč. 2, s. 1–26. Dostupné z DOI: 10.3233/SAT190014.
19. *Pseudo-boolean competition 2012*. [B.r.]. Dostupné také z: https://www.cril.univ-artois.fr/PB12/.
20. BARTH, Peter. *OPBDP*. [B.r.]. Dostupné také z: https://sourceforge.net/projects/opbdp/.
21. ROUSSEL, Olivier; MANQUINHO, Vasco. *Input/output format and solver requirements for the competitions of ...* [B.r.]. Dostupné také z: https://www.cril.univ-artois.fr/PB10/format.pdf.
22. BILLIONNET, Alain; JAUMARD, Brigitte. A decomposition method for minimizing quadratic pseudo-Boolean functions. *Operations Research Letters*. 1989, roč. 8, č. 3, s. 161–163. ISSN 0167-6377. Dostupné z DOI: https://doi.org/10.1016/0167-6377(89)90043-6.
23. FURINI, Fabio; TRAVERSI, Emiliano. Extended Linear Formulation for Binary Quadratic Problems. In: 2013.
24. MALLACH, Sven. Compact Linearization for Binary Quadratic Problems subject to Assignment Constraints. *4OR*. 2018, roč. 16. Dostupné z DOI: 10.1007/s10288-017-0364-0.
25. BESTUZHEVA, Ksenia; BESANÇON, Mathieu; CHEN, Wei-Kun; CHMIELA, Antonia; DONKIEWICZ, Tim; DOORNMALEN, Jasper van; EIFLER, Leon; GAUL, Oliver; GAMRATH, Gerald; GLEIXNER, Ambros; GOTTWALD, Leona; GRACZYK, Christoph; HALBIG, Katrin; HOEN, Alexander; HOJNY, Christopher; HULST, Rolf van der; KOCH, Thorsten; LÜBBECKE, Marco; MAHER, Stephen J.; MATTER, Frederic; MÜHMER, Erik; MÜLLER, Benjamin; PFETSCH, Marc E.; REHFELDT, Daniel; SCHLEIN, Steffan; SCHLÖSSER, Franziska; SERRANO, Felipe; SHINANO, Yuji; SOFRANAC, Boro; TURNER, Mark; VIGERSKE, Stefan; WEGSCHEIDER, Fabian; WELLNER, Philipp; WENINGER, Dieter; WITZIG, Jakob. *The SCIP Optimization Suite 8.0*. 2021-12. Technical Report. Optimization Online. Dostupné také z: http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

26. BESTUZHEVA, Ksenia; BESANÇON, Mathieu; CHEN, Wei-Kun; CHMIELA, Antonia; DONKIEWICZ, Tim; DOORNMALEN, Jasper van; EIFLER, Leon; GAUL, Oliver; GAMRATH, Gerald; GLEIXNER, Ambros; GOTTWALD, Leona; GRACZYK, Christoph; HALBIG, Katrin; HOEN, Alexander; HOJNY, Christopher; HULST, Rolf van der; KOCH, Thorsten; LÜBBECKE, Marco; MAHER, Stephen J.; MATTER, Frederic; MÜHMER, Erik; MÜLLER, Benjamin; PFETSCH, Marc E.; REHFELDT, Daniel; SCHLEIN, Steffan; SCHLÖSSER, Franziska; SERRANO, Felipe; SHINANO, Yuji; SOFRANAC, Boro; TURNER, Mark; VIGERSKE, Stefan; WEGSCHEIDER, Fabian; WELLNER, Philipp; WENINGER, Dieter; WITZIG, Jakob. *The SCIP Optimization Suite 8.0*. 2021-12. ZIB-Report, 21-41. Zuse Institute Berlin. Dostupné také z: `http://nbn-resolving.de/urn:nbn:de:0297-zib-85309`.

27. *Monitoring driver*. [B.r.]. Dostupné také z: `https://docs.opennebula.io/6.4/integration_and_development/infrastructure_drivers_development/devel-im.html#`.

28. [B.r.]. Dostupné také z: `https://libvirt.org/docs/libvirt-appdev-guide-python/en-US/html/`.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `generator` ..... zdrojové kódy generátoru dat
    - `skripty` ..... ukázka testovacích skriptů
  - `prace` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `prace.pdf` ..... text práce ve formátu PDF
- `data`
  - `instance` ..... testovací instance
  - `vysledky` ..... výstup testů
  - `statistika` ..... zpracovaný výstup testů