VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ŘAZENÍ MANIPULACÍ PRO MOŘICÍ LINKY

ORDERING OF JOBS FOR PICKLING LINES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MICHAL PLŠEK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ONDŘEJ KANICH, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Plšek Michal**
Program: Informační technologie
Název: **Řazení manipulací pro mořící linky**
Ordering of Jobs for Pickling Lines
Kategorie: Umělá inteligence

Zadání:

1. Prostudujte literaturu týkající se optimalizace procesů a umělé inteligence. Seznamte se s principem fungování mořících linek, soustřeďte se na jejich automatizované řízení.
2. Navrhněte aplikaci pro optimální seřazení manipulací transportní jednotky tak, aby byly dodrženy vstupní podmínky (tj. definovaná receptura a časy expozic) a zároveň byla produkční kapacita co nejvyšší. Berte v úvahu také parametry transportní jednotky (rychlost, zrychlení). Aplikace umožní navržené řešení vizualizovat, včetně parametrů.
3. Navrženou aplikaci z předchozího bodu implementujte.
4. Proveďte otestování Vaší aplikace na několika zvolených případech. Na základě výsledků nalezněte omezující (vstupní) podmínku. Opakovaným využitím aplikace prokažte zvýšení výkonu linky.
5. Dosažené výsledky shrňte a diskutujte. Uveďte možná rozšíření Vašeho řešení.

Literatura:

- Ertel, W.: *Introduction to Artificial Intelligence*, Springer, 2017, p. 316, ISBN 978-3-319-58487-4.
- Simeonovová, I., Knoflíček R.: *Application of computer simulation to increase the production rate of pickling line*, MM Science Journal, 2014, p. 3, DOI 10.17973/MMSJ.2014_03_201402.
- Simeonovová, I., Simeonov S.: *Production rate optimization by using simulation approach for designing of acid pickling lines*, Scientific proceedings IX. international congress "machines, technology, materials" 2012, 2012, p. 5, ISSN 1310-3946.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Kanich Ondřej, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Tato práce řeší problém plánování pohybů jednoho nebo více jeřábů při přesunu produktů mezi vanami mořicí linky. Harmonogramy zpracování jednotlivých produktů ve vanách jsou vytvořeny pomocí modifikované Shifting bottleneck heuristiky, zabraňující střetu produktů ve vanách. Pro prohledávání celého prostoru řešení je použit genetický algoritmus NSGA-II. Nad optimalizační procesem je postavena webová aplikace, která umožňuje správu a prohlížení produktů, částí mořicí linky, historie úloh a konfiguračních parametrů úlohy. Aplikace u úloh menšího rozsahu dosahuje zefektivnění až 30-45 % ve srovnání s naivními harmonogramy operací namáčení. Výsledkem této práce je funkční aplikace, na jejímž základě je možné postavit optimalizační aplikaci v jazyce C++ použitelnou k řešení obdobného problému o větším rozsahu.

Abstract

This work resolves the scheduling problem of multiple hoists transporting products between chemicals baths of pickling line. Harmonograms of products are calculated by modified Shifting bottleneck heuristic, which prevents product conflicts inside baths. Genetic algorithm NSGA-II is used for solution-space search. Web application built over the optimization process allows user to manage/edit products, hoists, baths, configuration parameters and optimization results. Applying proposed heuristic to smaller optimization tasks boosts production effectivity up to 30-45 % (comparing to naive harmonograms). The result of this work is application on the basis of which full-fledged C++ application might be programmed. Then it might be used for solving larger-scale problems.

Klíčová slova

mořicí linka, job shop, problém HSP, shifting bottleneck heuristika, NSGA-II, disjunktivní graf, EDD pravidlo, branch and bound, tornado, networkx, pymoo, python3

Keywords

pickling line, job shop, HSP problem, shifting bottleneck heuristic, NSGA-II, disjunctive graph, EDD rule, branch and bound, tornado, networkx, pymoo, python3

Citace

PLŠEK, Michal. *Řazení manipulací pro mořicí linky.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ondřej Kanich, Ph.D.

Řazení manipulací pro mořicí linky

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Kanicha, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Michal Plšek
10. května 2021

Poděkování

Rád bych poděkoval Ing. Ondřeji Kanichovi Ph.D. za velmi věcné zásahy do textové části práce.

Obsah

Kapitola 1

Úvod

Při řazení manipulací u mořicí linky se setkáváme s optimalizačním problémem: je třeba stanovit pořadí transportních operací jeřábů, které přenášejí produkty mezi vanami s kyselinou. V tomto vysoce rizikovém prostředí (kde se pracuje s vysoce koncentrovanými kyselinami) je klíčové dodržet časové intervaly, v rámci kterých musí být produkt vyjmut z vany ve které je namočen, a následně transportován do další vany dle receptury. Protože ocelové produkty korodují mezi namáčeními na vzduchu, není možné odložit pokračování zpracování produktu na neurčito, nýbrž co nejrychleji zahájit jeho další namáčení. Je tedy nutné stanovit takovou posloupnost transportních operací, která umožní nejen zkrácení intervalu kdy je produkt na vzduchu, ale zároveň manipulace seřadí takovým způsobem, aby byla propustnost výrobní linky co nejvyšší. Z hlediska složitosti se jedná o velice obtížný problém, které je nutné řešit sofistikovanými algoritmy. Protože značné množství plánovacích úloh se dotýká výrobního průmyslu, optimalizace výrobních procesů se dlouhodobě dostává do popředí zájmu subjektů, které chtějí snížit náklady na výrobu. Pro řešení složitých plánovacích problémů vzniklo velké množství algoritmů, většinou založených na tzv. meta-heuristických postupech. V tomto případě se jedná o problém ze specifické množiny plánovacích problémů, na něž se dá aplikovat Shifting bottleneck heuristika - která je součástí i zde.

Cílem této práce je navrhnout a implementovat aplikaci, která umožní stanovit pořadí transportních operací pro provedení efektivní výroby. Zároveň umožňuje pomocí uživatelského rozhraní spravovat parametry jeřábů, van a produktů; u produktů umožňuje nastavení alternativních van pro provedení namáčení. Také umožňuje vizualizaci výrobního procesu na mořicí lince.

Struktura práce je následující: sekce 2 obsahuje teoretický úvod do problematiky a současné způsoby řešení. Sekce 3 uvádí různé možnosti řešení tohoto problému a vybírá konkrétní metody. Sekce 4 tyto metody modifikuje dle potřeb zadání této práce a stanovuje finální postup řešení. Sekce 5 obsahuje implementační detaily metod popsaných ve čtvrté sekci. Sekce 6 obsahuje detaily provedených měření a porovnává jejich výsledky a navrhuje seznam možných rozšíření.

Kapitola 2

Plánování u mořicích linek

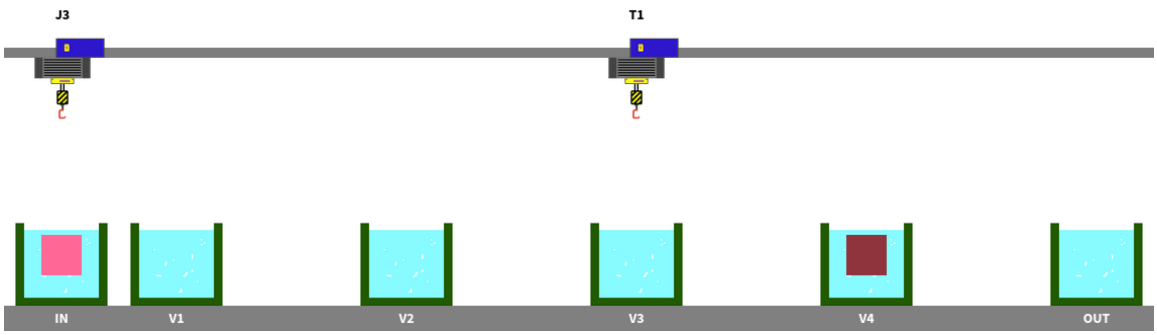
Tato sekce obsahuje teoretické základy nutné k vyřešení zadané optimalizační úlohy. Definuje problém HSP, popisuje princip fungování mořicí linky a rozděluje plánovací modely a časové harmonogramy. Dále stanovuje složitost plánovacích problémů.

2.1 Problém HSP

Zadaná úloha je problémem HSP (*hoist scheduling problem*). Dle klasifikace [3] problém spočívá v plánování pohybů jednoho nebo více jeřábů, které přemisťují produkty mezi stroji (zpravidla vanami), na kterých probíhá zpracování produktu. Může se např. jednat o žárové zinkování (namáčení ocelového produktu do roztaveného zinku) nebo v případě mořicí linky o namáčení produktů do roztoků kyselin.

Proces je v [3] popsán následovně: zdroji určenými ke zpracování produktů jsou nádrže obsahující chemické roztoky. Jednotlivý produkt musí být namočen v daných nádržích ve stanoveném pořadí a po stanovený časový interval. Probíhající namáčení produktu v lázni nesmí být přerušeno (produkt by mohl degradovat). Délka namáčení produktu v konkrétní lázni je předem stanovena a závisí zejména na: celkové ploše produktu, tloušťce stěny produktu, teplotě roztoku a jeho koncentraci.

Nedodržení času lázně může např. způsobit, že produkt nebude dostatečně očištěn, překročení času může způsobit jeho poškození. Součástí linky je vstupní a výstupní místo (případně vana), ze/do kterého jsou produkty přepravovány pomocí transportních jeřábů. Schéma mořicí linky je znázorněno na obrázku 2.1.



Obrázek 2.1: Ilustrace mořicí linky - vstupní pozice je vana nejvíce vlevo, výstupní pozice je vana nejvíce vpravo.

Produkty jsou před zahájením manipulace připevněny na transportní háky, které jsou od nich odděleny až po dokončení zpracování (jsou tedy spolu s produkty namáčeny v lázních). Transportní operaci jeřábu tvoří několik akcí:

1. přejezd jeřábu na umístění, kde je produkt namočen do lázně
2. zachycení háku a zdvih nad hladinu
3. zastavení pro účely okapu (aby se zabránilo kontaminaci dalších nádrží chemickým roztokem
4. zdvih do horní polohy
5. přejezd nad cílovou nádrž
6. ponoření produktu do lázně a vyháknutí.

Jeřáb při vykonávání transportní operace nezastavuje, pokud to není nutné (okap kyseliny nebo krátké čekání na uvolnění vany). Mohlo by dojít k oxidaci produktu na vzduchu.

Zadaný problém se dá definovat jako tzv. flexibilní job shop (popsán níže) s charakteristickými znaky:

- minimální a maximální časy zpracování produktu
- blokování operací - zákaz čekání mezi zpracováním produktu na navazujících strojích (neexistence vstupních a výstupních bufferů strojů)
- zákaz preempce u provádění operací.

Při řešení optimalizačních problémů využívajících notaci job shop vzniklo za poslední roky velké množství vědeckých prací [20]. Klasifikace HSP [3] udává, že existující notace job shop problému není dostačující pro reprezentaci všech případů, které mohou při řešení HSP problému nastat. Přesto je možné zadaný problém reprezentovat jako job shop: konkrétními příklady jsou práce [30] a [29].

2.1.1 Mořicí linky

Dle [15] se jedná o chemické zpracování produktů, při kterém dochází k odstraňování nečistot (skvrny, rez nebo anorganická kontaminace) u kovů. Tyto nečistoty mohou vznikat působením vzduchu, u nových produktů válcovaných za tepla pak mohou vznikat na povrchu oxidy železa. Právě mořením dochází k odleptání této oxidické vrstvy, která má vliv na kvalitu materiálu a nátěru. Pro své dobré vlastnosti je často využívána kyselina chlorovodíková (HCl 10 - 25 %), ale je možné použít i kombinaci kyselin [28]. Teplota lázně má nejčastěji 40 - 70°C. Po dokončení lázně v kyselině je povrch neutralizován v alkalickém roztoku uhličitanů.

Součástí zpracování produktu na mořicí lince může být aplikace tzv. taveniny [28]: jedná se o namáčení v lázni, jejíž obsah je tvořen NaOH (hydroxidem sodným) v kombinaci s oxidační nebo redukční složkou. Tato lázeň díky svému složení usnadňuje domoření v kyselinách. Nicméně proces namáčení produktu v tavenině se principiálně nijak neodlišuje od namáčení v ostatních vanách.

Mořicí linka může využívat automatizované jeřáby, které vykonávají automatické transportní operace pomocí instrukcí, které jsou nahrávány přímo do řídící jednotky jeřábu.

2.2 Plánovací modely

Plánování spočívá v alokaci úloh zdrojům (výrobním a transportním) takovým způsobem, že dochází k jejich optimálnímu zpracování, vedoucímu k vysoké vysoké výrobní efektivitě. Modely se dají rozlišit na deterministické a stochastické. U stochastických modelů platí, že obsahují určitou náhodnost: před zahájením simulace na takovém modelu by měla být známa pravděpodobnostní rozdělení, na základě kterých se odehrávají události (vstup produktu do systému, porucha stroje, časy zpracování produktu dané pravděpodobnostním rozdělením apod). U deterministických modelů jsou tyto údaje stanoveny hodnotami jejich parametrů: vznik události tedy není závislý na pravděpodobnosti (předpokládáme ideální podmínky, např. že nebude docházet ke zpoždění u transportu produktů). Stochastické modely díky tomu poskytují lepší znázornění reality.

Protože v rámci zadané úlohy je prováděna optimalizace průmyslového procesu spoléhajícího na přesné dodržování expozičních časů, byl zvolen deterministický model. V případě nedodržení časového plánu mohou produkty degradovat na vzduchu. Pokud dojde k poruše transportního zařízení, řešením je vygenerování nového časového plánu (v případě existence náhradního transportního zařízení), který naváže na nedokončené produkty jež je možné „zachránit“.

2.2.1 Celkové zařazení

Pinedo [21] rozdělil plánovací modely následujícím způsobem:

- Deterministické modely (popsány v následující sekci)
  - Modely s jedním strojem
  - Pokročilé modely s jedním strojem
  - Modely s paralelními stroji
  - Flow shop
  - Job shop
  - Open shop
- Stochastické modely
  - Modely s jedním strojem
  - Modely s jedním strojem a daty dokončení
  - Modely s paralelními stroji
  - Flow shop
  - Job shop
  - Open shop

2.2.2 Deterministické plánování

Problém plánování je optimalizační úlohou o obtížnosti NP [18]. Pracuje s konečnou množinou strojů, na kterých má proběhnout zpracování konečné množiny úloh (jobů) s předdefinovanými výrobními postupy. Joby jsou tvořenými sekvencí jednotlivých výrobních operací (sekvence se také nazývají *receptury*).

Plánovací notace

Množina jobů je označována J, s počtem jobů n. Množina strojů je označována M, s počtem strojů m. Operace O je definována dvojicí (i, j), kde i reprezentuje konkrétní stroj j konkrétní job (tedy O_{ij}). Dle Pineda [21] jsou s jobem spojovány následující parametry.

Datum zahájení zpracování jobu (*Release date*) – časový údaj reprezentující okamžik, kdy je job možné začít zpracovávat.
Datum dokončení jobu (*Due date*) – časový údaj reprezentující, kdy má být job dokončen (tj. připraven k odevzdání zákazníkovi).
Čas zpracování (*Processing time*) – Jedná se o časovou konstantu, určující počet časových jednotek nutných ke zpracování operace jobu na daném stroji. V případě tohoto řešení se jedná o dvojici konstant ($P_{ij}+$ a P_{ij}-), označující minimální a maximální čas zpracování.
Váha – Jedná se o konstantu udávající prioritu zpracování tohoto jobu nebo jeho důležitost vůči ostatním jobům.

Dle Pineda je [21] lze problém plánování popsat pomocí trojice (α, β, γ). První z prvků (α) popisuje typ plánovacího problému a může nabývat následujících hodnot:

Jeden stroj (*Single machine*) – Zpracování jobů probíhá na jednom stroji. Notace: $\alpha = 1$
Paralelní identické stroje (*Identical machines in paralell*) – V systému se nachází m paralelních identických strojů a joby, které obsahují jedinou operaci (kterou je možné provést na kterémkoliv ze strojů nebo na některém z podmnožiny strojů M_j). Notace: $\alpha = P_m$
Paralelní stroje s odlišnými rychlostmi zpracování (*Machines in paralell with different speeds*) – Na rozdíl od předchozího typu se stroje liší odlišnou délkou zpracování operace. Rychlost stroje i je označena konstantou v_i. Notace: $\alpha = Q_m$
Nesouvisející paralelní stroje (*Unrelated machines in paralell*) – Jedná se o zobecnění předchozího typu, ale rychlosti zpracování konkrétních jobů na konkrétních strojích se mohou lišit. Notace: $\alpha = R_m$
Flow shop – jedná se o verzi, ve které figuruje m strojů, které zpracovávají n jobů, které jsou tvořeny identický posloupnostmi operací (mají stejnou recepturu). Joby obvykle po dokončení zpracování na některém stroji vytvářejí fronty jobů, čekajících na zpracování na dalším stroji. Notace: $\alpha = F_m$
Flexibilní Flow shop (*Flexible flow shop*) – v tomto případě se jedná o zobecnění předchozího problému a verzí pracujících s paralelními stroji. Každý operace jobu je zpracována na jednom z podmnožiny strojů M_j. Notace: $\alpha = FF_c$
Job shop – V tomto případě mohou mít joby odlišné receptury (posloupnost operací prováděných na konkrétních strojích). Notace: $\alpha = J_m$
Flexibilní job shop (*Flexible job shop*) – Jedná se o zobecnění předchozího případu a verzí pracujících s paralelními stroji. Stejně jako v případě flexibilního flow shopu mohou být operace zpracovány na jednom z dané podmnožiny strojů. Notace: $\alpha = FJ_c$
Open shop – Každá operace je zpracována na některém z množiny strojů, ale některé časy zpracování mohou být nulové. Notace: $\alpha = O_m$

Druhý z trojice prvků popisující problém plánování (prvek β) může dle Pineda [21] nabývat jedné nebo více z následujících vlastností:

Čas zahájení (*Release dates*) – hodnota nastavuje nejmenší čas zahájení zpracování jobu. Pokud je tato hodnota nastavena, není možné do doby stanovené touto hodnotou zahájit zpracování jobu j. Notace: $\beta = r_j$
Preempce (*Preemptions*) – pokud je preempce při plánování povolena, zahájené zpracování operace jobu na stroji může být přerušeno a může být zahájeno zpracování jiné operace. Při navrácení původní operace ke zpracování na stroji, u kterého došlo k preempci, je možné pokračovat ve zpracování tam, kde bylo přerušeno (není třeba znovu zahajovat zpracování operace). Notace: $\beta = prmp$
Precedenční pravidla (*Precedence constraints*) – precedenční pravidla určují, že jeden nebo více jobů musí být dokončeny před zahájením zpracování jiného jobu. Pinedo rozlišuje několik typů precedenčních vlastností:

- u jobů s nanejvýše jedním následovníkem a jedním předchůdcem se jedná o tzv. *chains* (řetězce)
- u jobů s nanejvýše jedním následovníkem se jedná o tzv. *intree*
- u jobů s nanejvýše jedním předchůdcem se jedná o tzv. *outtree*.

Notace: $\beta = prec$
Sekvenčně závislé zahajovací časy (*Sequence dependent setup times*) – Jedná se o časovou konstantu, která stanovuje časové rozpětí mezi joby j a k, tj. po ukončení jobu j a před zahájením jobu k. Délka časového rozpětí může být závislá na konkrétním stroji (v takovém případě notace s_{ijk} mezi joby j a k) nebo se může jednat o čas nutný pro přípravu pro provádění nebo úklid po provedeném jobu (notace s_{0k} a s_{k0} u jobu k). Notace: $\beta = s_{jk}$
Skupiny jobů (*Job families*) – Joby mohou být zařazeny do skupin (skupina může obsahovat joby, které vyžadují jinou recepturu). Při zpracování jobů k a l ze stejné skupiny na stroji m není nutné započítat zahajovací čas. Notace: *fmls*.
Hromadné zpracování (*Batch processing*) – Stroj dokáže naráz zpracovat určité množství operací q. V případě rozdílných časů zpracování produktů je celkový čas hromadného zpracování stejný jako délka zpracování operace s nejdelší délkou zpracování. Notace: $\beta = batch(q)$
Poruchy (*Breakdowns*) – Jedná se o výskyt stavu kdy není stroj dostupný, a nedokáže tudíž zpracovávat operace. Notace: $\beta = brkdwn$
Způsobilost stroje ke zpracování operace (*Machine eligibility restrictions*) – Jedná se o parametr určující podmnožinu strojů M, které jsou schopny zpracovávat job j, resp. konkrétní operaci z jeho receptury. Notace: $\beta = M_j$
Permutace (*Permutation)* – tato vlastnost popisuje, že fronty před každým strojem (tzv. buffery) se řídí při výběru následující operace strategií FIFO (First In – First Out). Notace: $\beta = prmu$
Blokování (*Blocking*) – tato vlastnost značí, že může dojít k blokování operací, a to v případě, že jsou omezeny velikosti bufferů před a za stroji (tj. front produktů čekajících na zpracování na stroji, a na stroji dokončených produktů). Tím pádem dochází k situaci, kdy je stroj obsazen operací, která již byla dokončena a blokuje zahájení operace jiného jobu. Notace: $\beta = block$
No-wait – vlastnost značí, že u posloupných operací k a l jednoho jobu nemůže dokončená operace k čekat na uvolnění stroje m, potřebného pro zpracování následující operace l. Notace: $\beta = nwt$

Recirkulace (*Recirculation*) – k recirkulaci dochází v případě, kdy je operace jobu na jednom stroji provedena více než jedenkrát za sebou. Notace: $\beta = rcrc$

Třetí z trojice prvků popisující problém plánování (prvek γ) definuje jeden nebo více cílů, které je třeba minimalizovat. Cílem je tzv. objektivní funkce. Výstupní hodnota této funkce závisí na časovém harmonogramu provedení všech operací na strojích. Čas dokončení jobu j je označován symbolem C_j, čas dokončení operace jobu j na stroji i jako C_{ij}. V terminologii plánování se obecně pracuje s pojmy *lateness* (u jobu j značíme L_j) a *tardiness* (T_j), které určují nakumulované zpoždění u probíhajících výrobních procesů. Jsou definovány v rovnicích 2.1 a 2.2.

$$Lj = Cj - dj \tag{2.1}$$

$$Tj = max(Cj, -dj, 0) \tag{2.2}$$

Prvek dj označuje *due date* - časový údaj reprezentující, kdy má být job dokončen (tj. připraven k odevzdání zákazníkovi). Pinedo [21] také pracuje s konceptem tzv. *unit penalty*, což je funkce s předpisem v rovnici 2.3.

$$Uj = 1 \text{ pokud } Cj > dj, \text{ jinak } 0 \tag{2.3}$$

Všechny tři funkce L_j, T_j a U_j jsou použitelné pro stanovení některého z cílů (hodnota γ): nejmenší překročení časového plánu apod. Některými z často využívaných objektivních funkcí (cílů), které bývají následně minimalizovány, jsou:

- Délka výrobního procesu (*Makespan*) – Délka výrobního procesu je definována jako $max(C_1, C_2 .. C_n)$, tj. čas dokončení poslední operace nejpozději dokončeného jobu. Notace: $\gamma = C_{max}$
- Maximální zpoždění (*Maximum lateness*) – je definováno jako $max(L_1, L_2 .. L_n)$ a určuje největší překročení stanoveného času dokončení jobu. Notace: $\gamma = L_{max}$
- Celkový vážený čas dokončení (*Total weighted completion time*) – Jedná se o celkový součet vážených časů dokončení – tento údaj je použitelný např. pro stanovení nákladů na zásobu produktů (vázáno s testovaným časovým harmonogramem). Notace: $\gamma = \sum(w_j \cdot C_j)$
- Další funkce nad rámec této úlohy (celkové vážené zpoždění – *total weighted tardiness*, vážený počet zpožděných jobů – *weighted number of tardy jobs* atd.)

2.2.3 Časové harmonogramy

Dle Pineda [21] existuje několik základních tříd časových harmonogramů, které jsou výsledkem aplikace konkrétních pravidel pro plánování zpracování operací jobů na strojích. Jedná se o:

- Časový plán bez prodlení (*Non-delay*) – jedná se o třídu definovanou tak, že žádný ze strojů nesmí být nevytížen v případě, kdy některá operace čeká na zpracování. Ovšem i při aplikaci pravidel této třídy časových plánů může dojít k prodloužení celkového výrobního času (obzvlášť u nepreemptivního plánování). Preemptivní verze EDD pravidla (popsaného níže) generuje takovýto časový harmonogram.

- Aktivní časový plán – jedná se o třídu nepreemptivních časových plánů. Je definována tak, že v rámci aktivního časového plánu není možné vykonstruovat lepší plán změnou v pořadí zpracování operací na strojích (lepším plánem se rozumí takový plán, kde alespoň jedna z operací skončí dříve a žádná operace neskončí později). Tato třída časových plánů musí zároveň splňovat podmínku semi-aktivních časových plánů.
- Semi-aktivní časový plán – jedná se o třídu nepreemptivních časových plánů, kdy žádná z operací nemůže skončit dříve, pokud nedojde ke změně pořadí zpracování na některém ze strojů.

Pokud je objektivní funkce regulérní (tj. její hodnota je lineární v závislosti na časové náročnosti daného časového plánu), optimální časový plán se nachází v množině aktivních časových plánů. Pro nepreemptivní úlohy hledáme řešení pouze v množině aktivních a semi-aktivních časových plánů. Vennův diagram na obrázku 2.2 reprezentuje vztahy tříd.

Obrázek 2.2: Vzájemné vztahy tříd časových plánů. Upraveno z [21].

2.3 Složitost plánovacích problémů

Pro stanovení složitosti deterministického plánování je třeba definovat pojem *Turingův stroj*. Jedná se o teoretický model počítače s procesorovou jednotkou, programem a nekonečnou páskou pro zápis zápis mezivýsledků algoritmu. Existují dva typy Turingových strojů: deterministické a nedeterministické. V případě deterministického Turingova stroje navazuje na každý krok jediná následující akce definovaná v programu pomocí pravidel přechodové funkce. V případě nedeterministického Turingova stroje na krok může navazovat více než jedna následující akce.

Při řešení plánovacích problémů rozlišujeme mezi optimalizačními problémy a rozhodovacími problémy [21]. V případě rozhodovacích problémů je výsledkem výstup ve formátu pravda/nepravda. Tyto dva druhy problému jsou úzce spjaty: v případě, že existuje algoritmus pro řešení optimalizačního problému v polynomiálním čase, existuje také algoritmus pro řešení rozhodovacího problému v polynomiálním čase. Polynomiální časová složitost je definována jako $O(n^k)$, kde k je konstanta a n definuje velikost vstupních dat.

Příkladem propojení optimalizačních a rozhodovacích problémů může být optimalizace úlohy $FJ_c \mid\mid C_{max}$ (minimalizace výrobního času u problému flexibilní job shop). Jedná se o optimalizační problém; jako na rozhodovací problém se dá na úlohu nahlížet tak, zda existuje časový plán s výrobním časem nižším než je hodnota t.

2.3.1 Třídy problémů P a NP

Třída P obsahuje všechny rozhodovací problémy, které jsou řešitelné za pomoci Turingova stroje, a to v polynomiálním čase.

Třída NP zahrnuje všechny rozhodovací problémy, pro které může být navržené řešení (v případě této úlohy se jedná o vygenerovaný časový plán) ověřeno za pomocí Turingova stroje v polynomiálním čase. Dle [9] se jedná o problémy, u kterých můžeme za předpokladu existence časového plánu ověřit jejich platnost. Problémy v této třídě jsou řešitelné v polynomiálním čase za použití nedeterministického Turingova stroje. Třída P je podmnožinou třídy NP.

2.3.2 NP-úplné a NP-těžké problémy

Vezmeme-li v úvahu problém $FJ_c \mid\mid C_{max}$ pro počet strojů strojů $m \geq 2$ a počtu operací $= 3$ u každého jobu, tak dle [10] se i u takto redukovaného problému jedná o NP-těžkou složitost. Pro určení NP-těžké složitosti problému musíme nejprve definovat pojem polynomiální redukce.

Dle [11] je polynomiální redukce rozhodovacího problému *p'* na rozhodovací problém *p* funkcí v rovnici 2.4.

$$f : Dp' \longrightarrow Dp \tag{2.4}$$

D_p označuje množinu všech instancí problému *p*, instance problému je dle [11] vstupními daty jakékoliv procedury použité pro nalezení řešení tohoto problému. Funkce *f* musí splňovat následující kritéria:

- Hodnota funkce musí být dosažitelná v polynomiálním čase.
- Pro všechny instance $I \in Dp'$ existuje řešení k *I* pouze v případě, že existuje řešení k $f(I) \in Dp$.

Pokud tedy existuje redukce problému *p'* na problém *p* v polynomiálním čase, zapisujeme redukci jako $p' \propto p$.

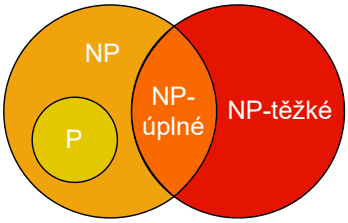
Třída problémů o složitosti NP-úplné je definována následovně: pokud je problém *p* NP-úplný tak platí, že $p \in$ NP a $q \propto p$ pro všechna *q* o složitosti NP. Jinak řečeno je rozhodovací problém *p* NP-úplný za předpokladu, že jakékoliv řešení problému *p* dokážeme ověřit v polynomiálním čase a všechny ostatní problémy třídy NP jsou polynomiálně redukovatelné na P. Jedná se výlučně o rozhodovací problémy (ne optimalizační).

Pro problémy o složitosti NP-těžké platí, že jsou alespoň tak složité jako nejsložitější problémy množiny NP. Jedná se o třídu problémů, které nemusí být rozhodovacími problémy. Třída je definována následovně: problém *p* je NP-těžký v případě, že pro všechny problémy *q* o složitosti NP platí, že $q \propto p$.

Složitostní třída NP-úplné tedy obsahuje nejsložitější problémy třídy NP. Třída NP-těžké obsahuje problémy alespoň tak složité jako nejsložitější problémy třídy NP. Vennův diagram na obrázku 2.3 reprezentuje složitost jednotlivých tříd P, NP, NP-úplné a NP-těžké. Tmavší barva značí vyšší složitost třídy.

2.4 Složitost plánovacích problémů

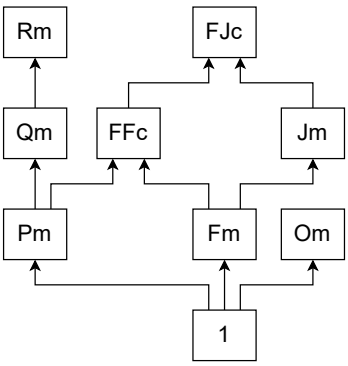
Mezi základní problémy, použitelné pro dokazování obtížnosti NP-těžké, náleží například:



Obrázek 2.3: Vennův graf složitosti NP problémů.

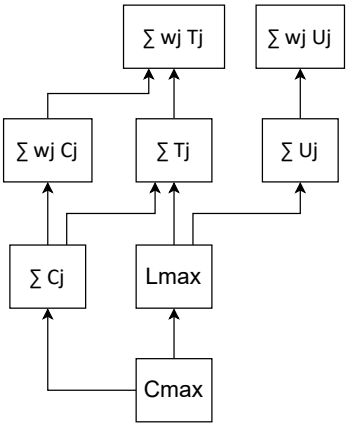
- Problém splnitelnosti booleovské formule – Je zadána množina proměnných a množina klauzulí definovaných nad těmito proměnnými. Stanovte hodnoty proměnných tak, aby každá z klauzulí byla platná.
- Hamiltonovská kružnice – Pro zadaný graf $G = (N, A)$ kde N je množina uzlů grafu a A množina spojnic mezi uzly, určete zda existuje smyčka spojující všechny uzly grafu. Tato smyčka prochází každým uzlem právě jednou.
- Problém kliky - Pro zadaný graf $G = (N, A)$, kde N je množina uzlů grafu a A množina spojnic mezi uzly, určete zda existuje klika o velikosti x.

Co se týče složitosti diskrétních plánovacích úloh, kromě [10] např. [24] dokázal, že i u jednoduchých verzí úloh typu job shop se jedná o NP-těžkou složitost. Pinedo [21] uvádí přehledné diagramy, reprezentující zvyšující se složitost v závislosti na typu systému: na obrázcích 2.4, 2.5 a 2.6.

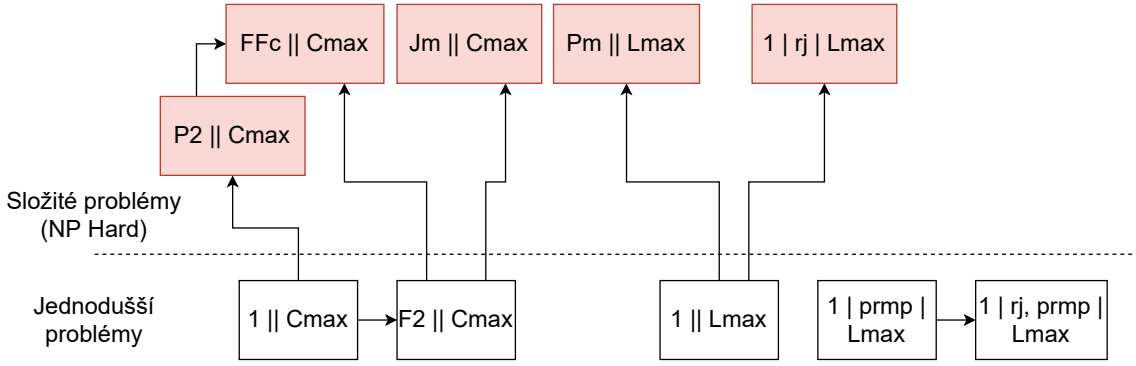


Obrázek 2.4: Narůstající složitost problému dle typu problému. Upraveno z [21].

Dopad třídy složitosti problému na jeho řešení je následující: v situaci, kdy hledáme řešení NP problému, je efektivnější ověřit navrhované řešení (tj. časový plán provádění operací jobu a transportních operací) vygenerované inteligentním způsobem (například genetickým algoritmem) v polynomiálním čase, než přímé nalezení sekvence splňující optimální podmínky.



Obrázek 2.5: Narůstající složitost problému dle objektivní funkce. Upraveno z [21].



Obrázek 2.6: Hierarchie složitosti vybraných problémů. Upraveno z [21].

Kapitola 3

Optimalizační metody

Tato kapitola popisuje obecné způsoby řešení problému flexibilní job shop. Obsahuje úvod do metod matematického programování a dále heuristiky a meta-heuristiky.

Zadáním úlohy je optimalizační problém o NP obtížnosti, a to ve formátu dle notace z předchozí sekce: FJ_c / *block, nwt* / C_{max}. V rámci tohoto problému hledáme aktivní nebo semi-aktivní časový plán všech operací jobů. Existují tři hlavní způsoby řešení, kterými je možné dosáhnout výsledku optimalizačního problému: heuristika, meta-heuristika a matematické programování.

3.1 Přístupy k řešení optimalizačních problémů

Tato sekce popisuje teoretické základy řešení optimalizačních problémů pomocí matematického programování, heuristiky a meta-heuristiky.

3.1.1 Matematické programování

Oblíbenou metodou pro získávání dobrých výsledků optimalizace je použití matematického programování programování, a to v častém spojení s prohledávací metodou *branch and bound* ([12] a [6]). Základním matematickým programem je tzv. lineární program. Objektivní funkce f je formulována v 3.1.

$$f = c_1 \cdot x_1 + c_2 \cdot x_2 \; .. + c_n \cdot x_n \tag{3.1}$$

Cílem programu je minimalizace tzv. nákladového vektoru, reprezentovaného hodnotami $c_1 \; .. \; c_n$. Rozhodovací proměnné jsou označeny jako $x_1 \; .. \; x_n$. Pokud nejsou některé z rozhodovacích proměnných diskrétní, tak program označujeme jako MIP (*mixed integer programming*). Pokud objektivní funkce neobsahuje žádný člen, který by byl nelineární, označujeme program jako MILP (*mixed integer linear programming*). Pokud objektivní funkce naopak obsahuje nelineární (kvadratický) člen, označujeme program jako MINLP (*mixed integer nonlinear programming*).

3.1.2 Disjunktivní programování

Jedná se o matematický zápis disjunktivních formulací, které musí hledané optimální řešení splňovat. Tyto formulace jsou úzce spojeny s reprezentací řešení pomocí disjunktivního grafu (viz sekce 3.2). Prohledávání prostoru řešení může být vyřešeno například metodou *branch and bound* [21].

3.1.3 Meta-heuristika a heuristika

Dle [26] rozlišujeme heuristické a meta-heuristické postupy následujícím způsobem:

- heuristické postupy jsou algoritmy, které jsou navrženy k řešení konkrétního problému - nelze je generalizovat a nejsou aplikovatelné na jiné optimalizační problémy
- meta-heuristické postupy jsou oproti heuristickým generalizovatelné; pomocí těchto algoritmů lze řešit libovolné optimalizační problémy (můžeme je použít jako stavební blok při řešení specifického optimalizačního problému).

Meta-heuristické algoritmy jsou často stavebním kamenem algoritmů pro řešení složitých optimalizačních problémů. Může se jednat o inteligentní algoritmy založené na chování kolonií včel, mravenců nebo letících hejn ptáků. V neposlední řadě se jedná o tzv. genetické algoritmy. Dle [16] je genetický algoritmus efektivním prostředkem k řešení optimalizačních problémů, což dokazuje i množství na něm postavených prací [2]. Existuje mnoho zdrojů shrnujících momentální trendy v řešení problémů různých typů optimalizačních problémů, např. [12]. Vyčerpávající souhrnný přehled aplikace genetických algoritmů je uveden v [2]. Obsahuje údaje o přehledu řešení problému flexibilní job shop s počtem publikací > 2, shrnuté v tabulce 3.1.

Tabulka 3.1: Přehled řešení problému flexibilní job shop s počtem publikací > 2.

Algoritmus	Počet publikací	Zastoupení (%)
GA	70	54,69
GA + tabu search	13	10,16
NSGA-II	11	8,59
GA + lokální vyhledávání	10	7,81
GA + heuristika	5	3,91
GA + simulované žíhání	5	3,91
GA + var. neighb. search	5	3,91
Adaptivní GA	3	2,34
GA + PSO	3	2,34
GA v kombinaci s učením	3	2,34

Z těchto dat a výsledků uvedených prací založených na genetických algoritmech vyplývá, že se jedná o efektivní způsob řešení NP obtížných optimalizačních úloh. Dle [4] se problém job shopu řadí mezi nejobtížnější NP problémy, jehož instance z roku 1963 (10 jobů, 10 strojů) nebyla vyřešena více než čtvrt století. Bylo na ni aplikováno velké množství tehdejších dostupných algoritmů. Právě kvůli ní byly vyvíjeny a zdokonalovávány efektivní aproximační metody. Jednou z nich je Shifting bottleneck heuristika [1], založená na reprezentaci optimalizačního problému pomocí disjunktivních grafů.

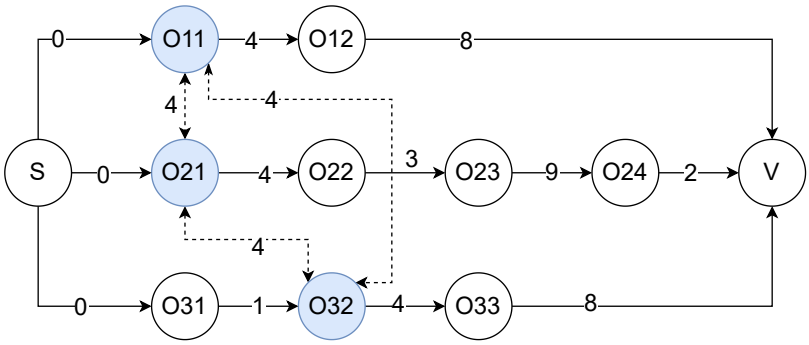
3.2 Reprezentace problému pomocí disjunktivních grafů

Efektivní znázorněním optimalizačního problému job shop je dle [21] disjunktivní graf. Tato metoda znázornění je velice populární, např. v [5] byla modifikována z hlediska výpočetní náročnosti. Základem tohoto znázornění je orientovaný graf G, který obsahuje konečnou množinu uzlů N (uzel reprezentuje operaci jobu na konkrétním stroji) a dvě množiny spojnic, A a B. Spojnice z množiny A reprezentují přechody mezi operacemi stejného jobu.

Množina B obsahuje spojnice mezi operacemi, které mají být zpracovány na stejném stroji, ale patří odlišným jobům. Tento druh spojnic propojuje operace oběma směry (u operací x,y vede jedna spojnice směrem od x k y a druhá od y k x). Spojnice z množiny A jsou označovány jako konjunktivní, z množiny B jako disjunktivní. V grafu se dále nacházejí vstupní uzel S a výstupní uzel V, které jsou jednosměrně propojeny na první (poslední) operace každého jednotlivého jobu.

Principem tohoto postupu je, že disjunktivní spojnice vytváří kliky pro každý stroj v grafu. Všechny uzly (operace) v rámci jedné kliky jsou zpracovávány na stejném stroji. Hodnota konjunktivních a disjunktivních spojnic vycházejících z konkrétního uzlu je dána dobou zpracování této operace. Délka spojnic mezi uzlem vstupu a prvními operacemi jobů je 0.

Na obrázku 3.1 jsou v disjunktivním grafu disjunktivní spojnice mezi operacemi různých jobů prováděnými na stroji M_1 vyznačeny přerušovanou čarou, konjunktivní spojnice pak plnou čarou. Klika stroje M_1 (B_1) v grafu je tedy (O_{11} - O_{31} - O_{21}).



Obrázek 3.1: Disjunktivní graf s konjunktivními a disjunktivními spojnicemi. Modře jsou označeny uzly disjunktivní kliky.

Dle [1] je směrovaný graf D je získán z nesměrovaného disjunktivního grafu G (na obrázku 3.1) odstraněním disjunktivních spojnic, tedy $D = (N, A)$. Výběr S_k z B_k tvoří právě jeden člen z každého páru disjunktivních spojnic (oboustranná disjunktivní spojnice = jeden pár jednosměrných spojnic) E_k. Výběr S_k je acyklický pokud neobsahuje směrovaný cyklus.

Jakýkoliv acyklický výběr S_k reprezentuje unikátní posloupnost operací zpracovávaných na stroji k. Kompletní výběr S tvoří množina selekcí S_k pro všechny stroje. Pokud dojde ke stanovení výběru S, tj. nahrazení disjunktivních spojnic za konjunktivní, vznikne směrovaný graf $D_s = (N, A \cup S)$. Kompletní výběr S je acyklický pokud je směrovaný graf D_s acyklický. Také platí, že pokud je kompletní výběr S acyklický, každý výběr S_k je acyklický. Každé S definuje rodinu časových plánů a každý časový plán náleží do jediné rodiny.

Celkový výrobní čas časového plánu optimálního pro dané S je roven nejdelší cestě v D_s. Problém je tedy pomocí disjunktivních grafů definován jako nalezení acyklického kompletního výběru S, v rámci kterého je minimalizována nejdelší cesta v grafu D_s.

3.3 Shifting bottleneck heuristika

Heuristickou metodu navrhl Adams a kol. v roce 1988 [1]. Jde o sofistikovaný iterativní algoritmus, který předpokládá, že při zpracovávání konečného množství jobů na konečném množství strojů se bude některý ze strojů chovat jako úzké hrdlo, které zvyšuje celkový čas

zpracování. V této sekci je popsána obecná verze algoritmu pro řešení diskrétní optimalizační úlohy FJ_c // C_{max}. Množina strojů je označena jako M. Pro hledání nejdelší cesty v grafu používám Bellman-Fordův algoritmus [23]. Algoritmus heuristiky je následující:

1. Nastavení množiny strojů M_0 na prázdnou množinu. Sestrojení grafu G obsahujícího pouze konjunktivní spojnice. C_{max} je udáno nejdelší cestou v grafu.
2. Pro každý ze strojů i z množiny M – M_0 je řešen NP problém $1 \mid r_j \mid L_{max}$ (jeden stroj, počítá se s časy zahájení, objektivní funkcí je největší zpoždění - řešení popsáno níže).
3. Na stroji k s nejvyšší hodnotou $L_{max}(i)$ je naplánována nalezená nejlepší sekvence operací, která je výstupem problému $1 \mid r_j \mid L_{max}$. Hodnota $L_{max}(k) = max(L_{max}(i))$, $i \in \{M - M_0\}$. Do grafu G jsou vloženy disjunktivní spojnice operací prováděných na stroji k, stroj k je přidán do množiny M_0.
4. Pro každý stroj w z množiny $\{M_0$ - $k\}$ dojde k:
   (a) smazání disjunktivních spojnic mezi operacemi na tomto stroji
   (b) dojde k řešení problému $1 \mid r_j \mid L_{max}$, kde jsou časy zahájení a ukončení operací určeny pomocí nejdelší cesty v grafu
   (c) nalezení sekvence operací, která minimalizuje objektivní funkci $L_{max}(w)$
   (d) vložení nových disjunktivních spojnic do grafu G
5. Pokud $M_0 \neq M$, dojde k navrácení na krok 2.

Pro každou iteraci algoritmu platí, že podmnožinu M_0 tvoří stroje, jejich disjunktivní spojnice již byly stanoveny (sekvence provádění operací na tomto stroji již byla stanovena). V kroku 2 je určeno, který ze strojů z množiny $\{M - M_0\}$ bude zahrnut do množiny M_0. Kritériem je aby se jednalo o stroj, který v rámci své činnosti vytváří největší zpoždění. Dle Pineda [21] vyjadřuje hodnota L_{max} kritičnost každého stroje (nakolik je stroj úzkým hrdlem - tzv. *bottleneck*). Následně je vybrán stroj k s nejvyšší hodnotou L_{max} (pokud je více strojů s touto hodnotou, je vybrán první z nich). Pro tento stroj je použita vygenerovaná optimální sekvence operací (podproblém $1 \mid r_j \mid L_{max}$). Hodnota C_{max} (nejdelší cesta v grafu) reprezentující celkový výrobní čas se dá nyní popsat rovnicí 3.2.

$$C_{max}(M_0 \cup k) \geq C_{max}(M_0) + L_{max}(k) \tag{3.2}$$

V kroku 4 dochází ke snaze snížit výrobní čas u naplánovaných strojů z množiny M_0. Dokud se v množině M nacházejí stroje nezařazené do M_0, iteruje se od kroku 2.

3.3.1 Podproblém 1 | rj | Lmax

Pro řešení tohoto podproblému (minimalizace zpoždění na jednom stroji s časy zahájení) Shifting bottleneck heuristiky je nejprve nutné v rámci grafu (ve kterém se v danou chvíli nachází pouze konjunktivní spojnice a disjunktivní spojnice strojů z množiny M_0) vypočítat pro každou tímto strojem zpracovávanou operaci následující údaje:

- p_{ij} – počet časových jednotek nutných k zpracování operace jobu j na stroji i

- d_{ij} – vzorec: $C_{max}(M_0)$ + p_{ij} - *nejdelší cesta od operace jobu* j *prováděné na stroji* i *k výstupnímu uzlu*
- r_{ij} – nejdelší cesta od vstupního uzlu k uzlu operace jobu j prováděné na stroji i.

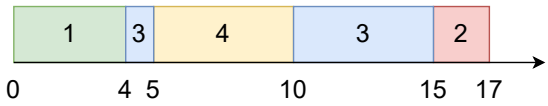
3.3.2 EDD pravidlo

Jedná se o plánovací pravidlo (EDD = *earliest due date*) určující, že v každém bodu časové osy jsou na daném stroji přednostně plánovány operace jobu s nejnižší hodnotou parametru d_{ij}. Pravidlo bere v potaz startovní časy operací jobů (parametr r_{ij}), což ilustruje následující příklad. Tabulka 3.2 obsahuje parametry operací jobů, které mají být provedeny na stroji M_1.

Tabulka 3.2: Parametry operací jobů pro ukázku aplikace EDD pravidla.

job	r_{1j}	p_{1j}	d_{1j}
1	0	4	8
2	1	2	12
3	3	6	11
4	5	5	10

Protože jsou vzaty v úvahu startovní časy operací a byla použita preemptivní verze EDD pravidla, výsledný časový harmonogram na obrázku 3.2 má následující podobu.



Obrázek 3.2: Diagram ilustrující posloupnost operací vygenerovanou za pomocí preemptivního EDD pravidla.

3.3.3 Metoda branch and bound

Jedním z relativně snadných způsobů řešení problému *1 | r_j | L_{max}* je metoda *branch and bound*. Tento problém je NP-těžký - dá se převést na problém se třemi oddíly [21]. Protože se jedná o podproblém v rámci Shifting bottleneck heuristiky [13], je zde důležitá výpočetní efektivita.

Řešení tvoří výchozí uzel, ze kterého se v dalších úrovních směrem dolů vytváří jednotlivá řešení. Uzel tvoří posloupnost operací jobů různých produktů na konkrétním stroji, v této posloupnosti jsou pozice na zpracování obsazeny nebo neobsazeny již naplánovanými operacemi. Ve výchozím uzlu jsou všechny pozice neobsazeny. Pozic je v uzlu tolik, kolik existuje operací ke zpracování na jednom stroji. Z tohoto výchozího uzlu vedou větve k n uzlům nižší úrovně, ve kterých je první pozice obsazena některou z operací. V každé následující úrovni je *n-1* uzlů, větvemi navazujících na uzel vyšší úrovně.

Existují omezující podmínky (*bounds*), pomocí kterých se provádí „ořezávání“ nepotřebných větví (*branches*). Jednou z nich je omezující podmínka 3.3, dále označovaná jako *kritérium C*.

$$r_a < min(max(t, r_b) + p_b), b \in J \tag{3.3}$$

Notace: a je testovaná operace jobu na konkrétním stroji, r_a čas začátku operace k, J je množina operací jobů které nejsou zatím naplánovány, t značí čas kdy jsou již dokončeny všechny naplánované operace na tomto stroji a p_b je čas trvání operace b. Minimum je vypočítáno ze všech operací z množiny J - $\{a\}$.

Dle [19] lze při řešení tohoto problému použít buď enumerativní metody nebo aplikovat aproximační algoritmus. V [25] a [21] je popsáno řešení $1 \mid r_j \mid L_{max}$, při kterém dochází k využití preemptivního EDD pravidla. Toto pravidlo je optimální pro problém $1 \mid r_j, prmp \mid L_{max}$ - je možné podle něj stanovit nízká omezení (*lower bounds*) u nepreemptivní úlohy. Tyto hodnoty jsou užitečné pro stanovení větve, kterou se má pokračovat při vyhledávání. Postup je následující:

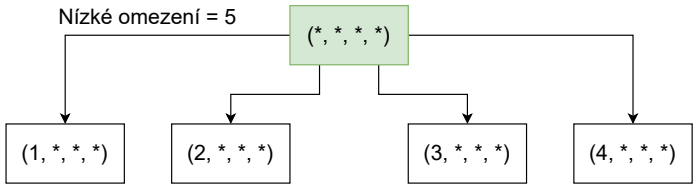
1. Pro výchozí uzel je vypočítáno nízké omezení a jsou vygenerovány podřazené uzly.
2. Pro uzly je vypočítáno nízké omezení. Toto nízké omezení je zároveň platné pro všechny další podřazené uzly (u žádného nebude nižší).
3. Větev je možné uříznout, pokud:
   (a) u uzlu dojde k porušení kritéria C (pouze u uzlů druhé úrovně a nižších)
   (b) pokud je nízké omezení uzlu vyšší než u nalezeného nepreemptivního řešení.
4. Pro pokračování je vybrán vygenerovaný uzel s nejnižší hodnotou.
5. Algoritmus můžeme zastavit ve chvíli, kdy není možné nalézt nepreemptivní řešení splňující všechna omezení, které by zároveň mělo hodnotu nižší než nejlepší nalezené nepreemptivní řešení.

Postup je prezentován na následujícím příkladu. Mějme tabulku 3.2 reprezentující problém $1 \mid r_j \mid L_{max}$, pořadí žádné operace zatím není stanoveno. Aplikací preemptivního EDD pravidla dojdeme k výchozímu časovému harmonogramu na obrázku 3.2.

Pro stanovení hodnoty L_j (zpoždění operace j) je použita rovnice 3.4.

$$L_j = \text{čas dokončení operace j} - d_j \tag{3.4}$$

Platí tedy, že L_1 = -4, L_2 = 5, L_3 = 4, L_4 = 0, tedy $L_{max} = max$(-4, 5, 4, 0) = 5. Protože se jedná o relaxaci (použití preemptivní verze EDD pravidla), je reálná hodnota $L_{max} \geq 5$. Jedná se o kandidátní řešení uzlu první úrovně (počátečního uzlu) a nízké omezení platné pro podřazené uzly (druhé úrovně a níž). Následně vytvoříme uzly druhé úrovně (viz obrázek 3.3).



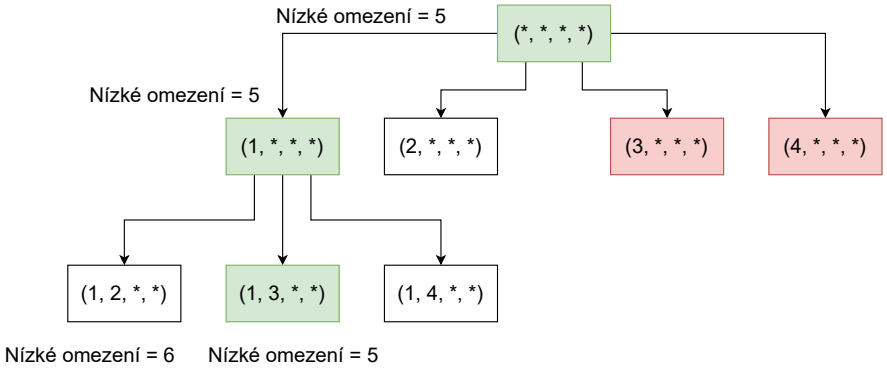
Obrázek 3.3: Metoda branch and bound - vygenerované uzly druhé úrovně.

Pro uzel počínající jobem 1 platí: L_1 = -4, L_2 = 5, L_3 = 4, L_4 = 0. Tedy L_{max} = 5 ($L_{max} \geq 5$).

Pro uzel počínající jobem 2 platí: L_1 = -1, L_2 = -9, L_3 = 7, L_4 = 2. Tedy L_{max} = 7 ($L_{max} \geq 7$).

Pro uzel počínající jobem 3 platí: L_1 = 5, L_2 = 8, L_3 = -2, L_4 = 8. Tedy L_{max} = 8 ($L_{max} \geq 8$).

Poslední z uzlů ovšem porušuje kritérium C. Totéž platí pro uzel (4,*,*,*). Obdobně postupujeme u generování uzlu počínajícího operací jobu 1, tj. vytvoříme z něj větve na uzly obsahující posloupnosti (1,2,**), (1,3,**), (1,4,*,*) a vypočítáme hodnotu L_{max}. U uzlu s posloupností (1,3,*,*) byla díky preemptivnímu pravidlu EDD stanovena sekvence (1,3,4,2) s $L_{max} = 5$ ($L_{max} \geq 5$) splňující všechna omezení a nízké omezení uzlu (2,*,*,*) je vyšší než u tohoto kandidátního řešení: jedná se o optimální řešení. Finální strom vedoucí k řešení je znázorněn na obrázku 3.4.



Obrázek 3.4: Metoda branch and bound - strom vedoucí k optimálnímu řešení.

3.4 Genetické algoritmy

Jedná se o třídu meta-heuristických algoritmů, která je postavena na biologickém konceptu evoluce. Jedním z důležitých předpokladů evoluce je, že živočišné druhy jsou schopny se ve svém prostředí adaptovat. Dle [16] jsou v rámci celkového zařazení jsou u výpočetní inteligence (*computational intelligence*) rozlišovány tři hlavní větve: neuronové sítě, fuzzy logika a genetické algoritmy.

Základní genetický algoritmus je založen na množině kandidátních řešení (populaci), která reprezentují řešení optimalizačního problému. Reprezentace jedinců může nabývat různých podob, mohou být vyjádřeni např. pomocí bitových řetězců. Reprezentace řešení se nazývá chromozom. Dle [16] ilustruje genetický algoritmus následující pseudokód:

1. inicializace populace
2. pro každého jedince v populaci je provedeno:
   (a) křížení
   (b) mutace
   (c) mapování fenotypu
   (d) výpočet hodnoty fitness funkce
3. selekce nové populace

4. pokud není splněna ukončovací podmínka, přechází se na krok 2.

Jednotlivé kroky jsou podrobněji rozepsány v následujících sekcích.

3.4.1 Křížení a mutace

Křížení je genetický operátor, který umožňuje vznik nových jedinců kombinací různých chromozomů. Je možné jej provést mezi dvěma a více jedinci. Křížení různých jedinců lze provést na n zvolených (předem daných nebo náhodně určených) místech. Výběr konkrétních jedinců pro křížení může být proveden selektivním algoritmem (viz níže) nebo mohou být jedinci vybráni náhodně.

Aplikace operátoru mutace má za následek, že u vybraného jedince dojde k náhodným změnám v chromozomu. Aby tento operátor mohl být využíván, existuje několik podmínek:

- Prostor řešení by neměl být příliš omezen omezovacími podmínkami (každý bod v prostoru řešení by v optimálním případě měl být dostupný z libovolného dalšího bodu).
- Operátor mutace by měl být nezaujatý (neměl by prohledávat prostor řešení pouze určitým směrem) - ovšem za předpokladu, že tomu tak není schválně, např. při cíleném prohledávání neprozkoumaných částí prostoru.
- Operátor mutace by měl být škálovatelný. Pro splnění této podmínky je možné použít např. Gaussovo pravděpodobnostní rozložení, podle kterého je určeno, zda má dojít k mutaci. Dá se také použít pravděpodobnost $1/n$, kde n je délka chromozomu (počet prvků). Pravděpodobnost mutace se dá vypočítat u každého prvku chromozomu.

3.4.2 Mapování fenotypu a výpočet fitness funkce

Při mapování fenotypu (po provedení mutací a křížení) dochází k převedení reprezentace řešení (genotyp) na vlastní řešení (fenotyp). Pokud je genotypem konkrétní řešení, je možné mapování přeskočit.

Fenotyp řešení je ohodnocen tzv. fitness funkcí. Tato funkce udává, nakolik konkrétní řešení splňuje optimalizační problém. Fitness funkce musí spravedlivě ohodnotit kvalitu řešení. V případě, že existuje více než jeden cíl (např. minimalizace objektivních funkcí, které jsou zároveň zatíženy omezeními), je nutné zvážit kritéria, podle kterých bude hodnoceno řešení splňující pouze jeden z cílů. Protože se v případě této úlohy jedná o minimalizaci hodnot objektivních funkcí, cílem je co nejnižší hodnota fitness u jednotlivých jedinců.

3.4.3 Selekce a ukončovací podmínka

Z nově vygenerovaných jedinců jsou vybráni do nové populace ti, kteří dosahují nejlepších fitness hodnot. Je možné nejlepší jedince přenést nepozměněné do další generace (elitismus), vybírat do další populace pouze nově vygenerované jedince (comma selekce) nebo kombinaci původních a nově vygenerovaných jedinců (plus selekce). Comma selekce sice nezahrnuje nejlepší jedince z původní populace, zato však dokáže spolehlivě překonat lokální minimum.

Selekce může být použita nejen pro výběr jedinců vytvářejících novou populaci, ale i pro výběr jedinců určených ke křížení (pro tento výběr mohou být žádoucí odlišná kritéria). Příkladem selektivních algoritmů, které mohou být využívány operátory selekce, jsou:

- Roulette selekce - výběr jedinců probíhá pomocí rovnoměrného pravděpodobnostního rozdělení, kde pravděpodobnost výběru spočívá na fitness hodnotě jedince. Metoda

se dá jednoduše popsat tak, že u kola rulety jsou jednotlivé sloty proporcionálně rozděleny mezi jednotlivá řešení podle své fitness hodnoty. Kulička následně vybere jedno z řešení.

- Turnajová selekce - je zvolena hodnota k, která určuje počet vybraných jedinců z původní populace. Následně je náhodně vybráno k vybraných jedinců a je porovnána jejich fitness hodnota. Pokud platí, že $k = 2$, turnajová selekce je označována jako binární turnaj. V jednom turnaji může být vybrán nejlepší jedinec nebo více jedinců.

Operátory selekce jsou vzhledem ke svému vztahu k multiobjektivní optimalizaci popsány níže.

Častým způsobem pro ukončení běhu genetického algoritmu je vygenerování předdefinovaného počtu generací. Algoritmus je také možné zastavit v případě, že po daný počet generací nebo časových jednotek nedochází ke zlepšení výsledného řešení. K tomu může dojít např. z důvodu, že algoritmus osciluje v lokálním optimu.

3.4.4 Multiobjektivní optimalizace

Pokud dochází k vyhodnocení více než jedné objektivní funkce, jedním ze způsobu řešení je stanovit váhy každé z nich. Výsledná funkce f s váhou w zahrnující dvě objektivní funkce je popsána rovnicí 3.5.

$$f = f_1 \cdot w + f_2 \cdot (1 - w) \tag{3.5}$$

Konstanta $w \in <0,1>$, f_1 a f_2 jsou objektivní funkce. Požadavkem na vyhodnocování nicméně může být aby nebyla dominantní ani jedna z objektivních funkcí. Multiobjektivní optimalizace se projevuje u operátorů selekce. Tyto operátory provádějí zpravidla dva základní kroky:

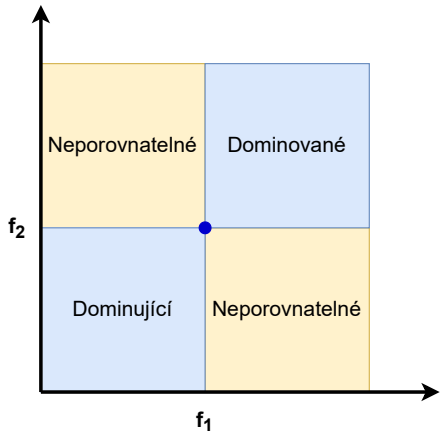
1. nedominantní třídění
2. optimalizace sekundárního kritéria (toto kritérium je závislé na konkrétním algoritmu, např. u algoritmu *NSGA-II* se jedná o výpočet tzv. *crowding distance* vzdálenosti a následně provedení selekce ve formě binárního turnaje v závislosti na hodnotách fitness a crowding distance).

Nedominantní třídění

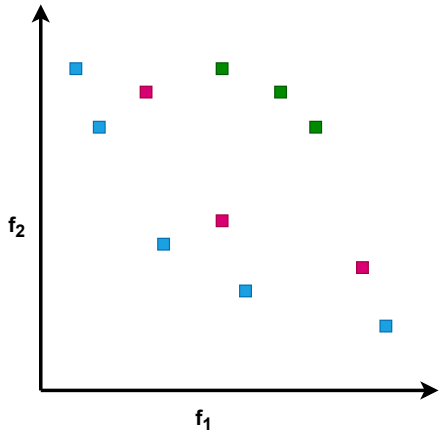
K provedení tohoto kroku je nutné vysvětlit pojem Paretovo optimum. Řešení splňuje toto optimum za předpokladu, že není dominováno jiným řešením v prostoru řešení. Vezměme v potaz dvě objektivní funkce, f_1 a f_2. Každé řešení dělí prostor řešení na čtyři kvadranty, znázorněné na obrázku 3.5.

Levý dolní kvadrant na obrázku obsahuje řešení, která jsou lepší než testované řešení (uprostřed). Pravý horní kvadrant obsahuje řešení horší než testované řešení. Zbylé dva kvadranty obsahují řešení, která jsou s testovaným neporovnatelná. Množina řešení v celém prostoru, která mají prázdný levý dolní kvadrant (tedy nejsou horší než žádná jiná řešení) se nazývá Paretovou sadou. Tato množina vytváří v celém prostoru řešení tzv. Paretovu hranici.

Nedominantní třídění seřadí všechna řešení dle jejich sad a přidělí jim pozici (*rank*). Řazení ilustruje obrázek 3.6.



Obrázek 3.5: Diagram znázorňující rozdělený prostor řešení vůči bodu uprostřed v rámci Paretova optima (f_1, f_2 reprezentují objektivní funkce). Upraveno z [16].



Obrázek 3.6: Řešení seřazená dle Paretových setů vytvářejí Paretovy hranice, rozdílné barvy určují jejich hodnocení (f1, f2 reprezentují objektivní funkce). Upraveno z [16].

Stejně barevná řešení nejvíce vlevo vytvářející Paretovu hranici (3.6) mají nejvyšší hodnocení 1, další pak hodnocení 2 atd. Efektivní algoritmus nedominantního řazení je podrobně rozebrán v [8].

Optimalizace sekundárního kritéria

Jedná se o druhý krok prováděný operátorem selekce. U algoritmu NSGA-II [8] je při provádění tohoto kroku nejprve vypočítána hodnota *crowding distance* každého řešení. Hodnota určuje, jaká je vzdálenost tohoto řešení k sousedním řešením stejného hodnocení. Algoritmus pro výpočet této hodnoty je následující:

- Pro každou hranici F_i (Paretův set se stejným hodnocením, n označuje počet řešení) je provedeno:
  - vzdálenost d (crowding distance) každého jedince je nastavena na 0
  - Pro každou objektivní funkci m je provedeno:

* řešení v rámci F_i jsou seřazena na základě hodnoty objektivní funkce m (výsledné seřazení je reprezentováno jako S)
* V rámci S jsou okrajovým hodnotám nastaveny nekonečné vzdálenosti, tj.: $d_0 = d_n = \infty$, $d_0 \in S$, $d_n \in S$
* Pro každé $k \in <2, n\text{-}1>$:
  - $d_k = d_k + (M_{k+1} - M_{k-1})/(M_{max} - M_{min})$, kde M_{max} a M_{min} reprezentují maximální a minimální hodnotu objektivní funkce m, M_{k-1} reprezentuje hodnotu objektivní funkce u řešení v pořadí *k-1* (obdobně M_{k+1})

Jakmile je hodnota crowding distance stanovena u každého řešení, může dojít k selekci jedinců. Selekce je u NSGA-II prováděna pomocí binárního turnaje, kde je primárním faktorem hodnocení jedince. Pokud dojde ke střetu dvou jedinců o stejném hodnocení, sekundárním faktorem je hodnota crowding distance. Algoritmus takto zajišťuje dostatečnou pestrost vybraných řešení. Pokud počet řešení s nejvyšším hodnocením (1) není dostatečný, jsou do cílové populace vybírána také řešení s nižším hodnocením (2, 3..).

Kapitola 4

Modifikace metod a návrh řešení

Následující sekce rozšiřuje předchozí sekci o modifikace vybraných metod. Popisuje modifikované jednoduché a úplné disjunktivní grafy, metodu pro stanovení nejdelší cesty v grafu, modifikovanou Shifting bottleneck heuristiku, zvolený genetický algoritmus a jeho operátory. Popisuje celkovou vytvořenou heuristiku skládající se z těchto částí a v závěru obsahuje návrh aplikace pro řešení optimalizačního problému.

4.1 Výběr metod a notace

Z výsledku srovnání efektivity meta-heuristických metod a matematického programování [17] vyplývá, že postup založený na meta-heuristice (genetický algoritmus) ve vysokém počtu případů došel k téměř optimálnímu řešení v lepším čase. Matematické programování sice garantuje kvalitu výsledných řešení, avšak jedním z jeho omezení (i při použití algoritmu branch and bound) je vysoká časová náročnost v případě rozsáhlých problémů.

Zatímco meta-heuristika v podobě genetického algoritmu nabízí ověřený způsob rychlého prohledání prostoru řešení a postupnou optimalizaci získaných sekvencí, Shifting bottleneck heuristika zajišťuje optimalizaci sekvence a efektivní způsob ověření, nakolik je zvolená sekvence řešením daného optimalizačního problému.

Protože při plánování úloh v chemickém průmyslu může dojít k situaci, kdy bude nutné vygenerovat přijatelný časový harmonogram většího množství jobů v řádu desítek minut, je strategické využít místo matematického programování meta-heuristický postup (genetický algoritmus). Ten je možné doplnit o Shifting bottleneck heuristiku pro vylepšení výsledku.

Dle [21] jsou genetické algoritmy dobrou volbou pro optimalizační problémy s jednou nebo více objektivními funkcemi, protože se opírají o populaci jedinců (řešení). Řešení nacházející se v Paretově optimu (případně stanovené váhy jednotlivých objektivních funkcí) přímo ovlivňují složení následující generace.

Tabulka 4.1 obsahuje notaci optimalizačního problému. Řešením úlohy je minimalizace hodnoty C_{max} (čas výroby) určená rovnicí 4.1.

$$C_{max} = \sum_{i=1}^{n} max(C_i) \tag{4.1}$$

Řešení úlohy je částečně inspirováno modifikovanou Shifting bottleneck heuristikou uvedenou v [29].

Tabulka 4.1: Parametry operací jobů pro ukázku aplikace EDD pravidla.

J_i	job i ($i \in <1, n>$)
O_i	počet jednotlivých operací jobu J_i ($i \in <1, n>$)
OP_{ij}	operace s číslem pořadí j jobu J_i ($j \in <1, O_i>$)
P_{ij}-	minimální čas zpracování operace OP_{ij}
P_{ij}+	maximální čas zpracování operace OP_{ij}
C_i	čas dokončení poslední operace jobu J_i
R_h	transportní zdroj ($h \in R$)
$TPij$	transportní operace mezi operacemi OP_{ij} a OP_{ij+1}
TR_{ij}	transportní zdroje, které mohou provést transport TP_{ij}
M	množina strojů (van)
R	množina transportních zdrojů (jeřábů)
ET_{kl}	čas prázdného přejezdu transportního zdroje mezi stroji M_k a M_l (k,$l \in M$)
LT_{kl}	čas transportu produktu mezi stroji M_k a M_l (k,$l \in M$)

4.2 Modifikace disjunktivního grafu

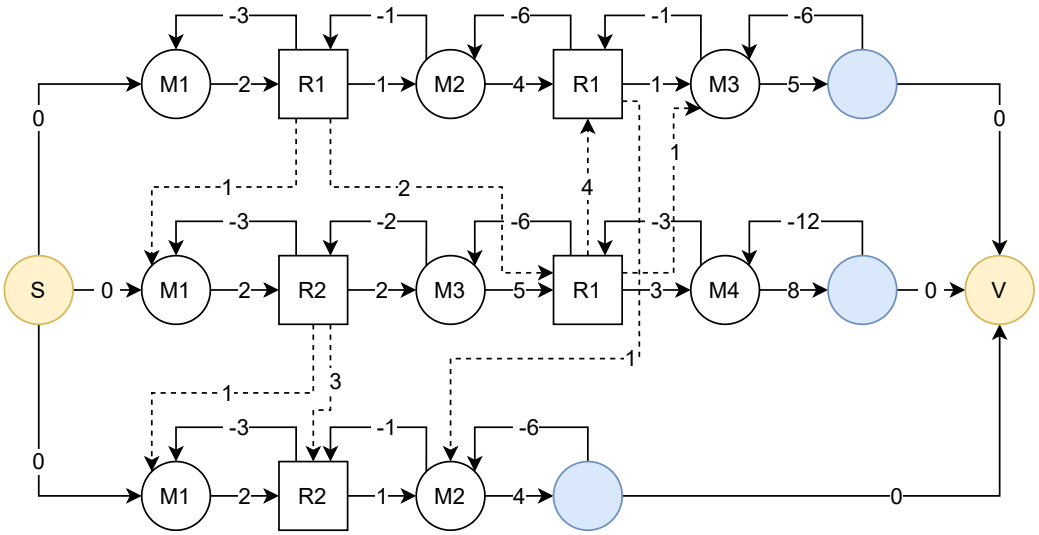
Na rozdíl od disjunktivního grafu původní nemodifikované Shifting bottleneck heuristiky tento graf nereprezentuje problém přiřazení operací jobů konkrétním strojům. Modifikovaný disjunktivní graf je vygenerován poté, co dojde ke stanovení sekvencí, v jakých mají být joby zpracovány - obousměrné disjunktivní spojnice jsou nahrazeny jednosměrnými (známe pořadí zpracování).

Modifikovaný graf obsahuje navíc uzly reprezentující transportní operace. Uzly reprezentující transportní operace v [29] se dají dohledat už v [14]. V následujícím popisu grafu se používá notace z předchozí sekce. Graf obsahuje prvky v 4.2.

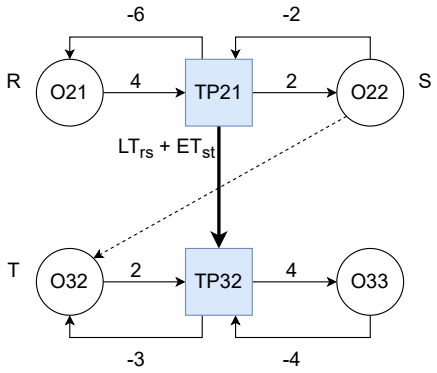
$$G = (V_m \cup V_t \cup C \cup D_m \cup D_t) \tag{4.2}$$

Prvek V_m reprezentuje uzly operací prováděných na strojích, prázdné koncové uzly jobu (každá poslední operace jobu prováděná na některém stroji ukazuje na vlastní uzel tohoto typu; čas zpracování jobu na tomto uzlu je 0 - na obrázku 4.1 označené modře), výchozí uzel S a koncový uzel V (na V odkazuje každý z modrých koncových uzlů jobu). V_t označuje množinu uzlů reprezentujících transportní operace na některém jeřábu. C označuje množinu konjunktivních spojnic, které oboustranně spojují jednotlivé operace jobu. Spojnice s pozitivní hodnotou směřují od operace OP_{ij} k následujícímu transportnímu uzlu (TP_{ij}): tyto hodnoty reprezentují hodnotu P_{ij}+. Negativní spojnice vedoucí opačně reprezentují hodnotu P_{ij}-. Spojnice vedoucí od transportního uzlu TP_{ij} k operačnímu uzlu OP_{ij+1} reprezentuje transportní čas LT_{kl} ($k = j$, $l = j$+1) mezi dvě operacemi jobu. Obrácená spojnice (od následující operace k předchozímu transportnímu uzlu) nabývá hodnoty -LT_{kl}. Negativní spojnice v modifikovaném disjunktivním grafu jsou dle [31] nástrojem jak v disjunktivním grafu definovat minimální a maximální časy zpracování operací.

D_t označuje množinu disjunktivních spojnic, které propojují transportní operace. Propojení konkrétních transportních uzlů závisí na tom, jaká je na daném transportním zdroji stanovena sekvence operací. Spojnice jsou jednosměrné a reprezentují hodnotu $LT_{rs} + ET_{st}$ (transportní zdroj musí nejprve dokončit transport z r do s a poté přejet ke stroji t; prázdný přesun je naznačen přerušovanou čarou, viz 4.2).



Obrázek 4.1: Modifikovaný disjunktivní graf s konjunktivními a orientovanými disjunktivními spojnicemi. Zahrnuje dva produkty se třemi operacemi namáčení a jeden se dvěma operacemi. Transportní operace jsou označeny hranatě, operace namáčení kulatě.

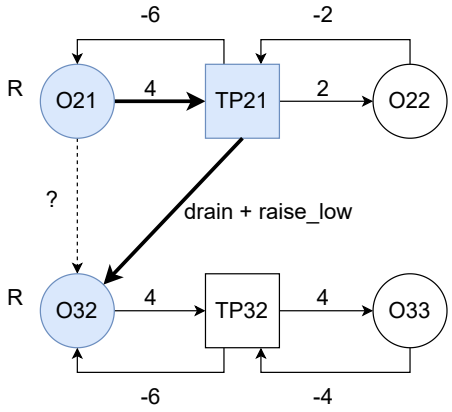


Obrázek 4.2: Disjunktivní spojnice (tučně) mezi transportními uzly modifikovaného disjunktivního grafu.

D_m reprezentuje množinu disjunktivních spojnic propojujících uzly operací prováděných na stejném stroji. Tyto uzly jsou propojeny pomocí transportního uzlu první z operací (kvůli minimálním a maximálním časům zpracování by jinak nebylo možné reprezentovat hodnotu P_{ij}-, jak je naznačeno přerušovanou čarou na obrázku 4.2). Na rozdíl od [29] jsem hodnotu této spojnice určil jako čas nutný k zdvihu produktu do nízké pozice a okapání produktu nad vanou (*raise* + *drain*): nedochází tím ke střetu dvou produktů v jedné vaně (viz obrázek 4.3).

Takovýto disjunktivní graf s orientovanými spojnicemi reprezentuje validní řešení, pokud neobsahuje žádné pozitivní smyčky (dají se nalézt pomocí Bellman-Fordova algoritmu). Příkladem je graf na obrázku 4.1 reprezentující sekvence jobů (J_1, J_2, J_3) na strojích M_1, M_2, M_3, M_4 a transportních zdrojích R_1, R_2:

J_1: $M_1 \longrightarrow R_1 \longrightarrow M_2 \longrightarrow R_1 \longrightarrow M_3$
J_2: $M_1 \longrightarrow R_2 \longrightarrow M_3 \longrightarrow R_1 \longrightarrow M_4$



Obrázek 4.3: Disjunktivní spojnice (tučně: *drain + raise_low*) mezi operacemi na stejném stroji modifikovaného disjunktivního grafu.

J_3: $M_1 \longrightarrow R_2 \longrightarrow M_2$

Naplánované sekvence na strojích a transportních zdrojích jsou následující. Sekvence jsou reprezentovány pomocí disjunktivních spojnic (přerušovaná čára) v grafu na obrázku 4.1 (kvůli zjednodušení nejsou jejich hodnoty přesné, ty je nutné dopočítat ze zadaných parametrů jeřábů a van).

M_1: $OP_{11} \longrightarrow OP_{21} \longrightarrow OP_{31}$
M_2: $OP_{12} \longrightarrow OP_{32}$
M_3: $OP_{22} \longrightarrow OP_{13}$
M_4: OP_{23}
R_1: $TP_{11} \longrightarrow TP_{22} \longrightarrow TP_{12}$
R_2: $TP_{21} \longrightarrow TP_{31}$

Zjednodušený graf G obsahuje pouze uzly reprezentující operace prováděné na strojích ($G = C \cup V_m \cup D_m$). Je využíván pro ověření, zda řešení neobsahuje pozitivní cyklus (aby nemusel být zbytečně konstruován úplný graf G'). jednosměrné disjunktivní spojnice D_m vedou přímo mezi uzly a jejich hodnota je čas zpracování operace na prvním z uzlů (jako na obrázku 3.1). Úplný graf G' (viz obrázek 4.1) obsahuje úplnou množinu uzlů a spojnic: $G' = (V_m \cup V_t \cup C \cup D_m \cup D_t)$.

Bellman-Fordův algoritmus

Algoritmus původně slouží k detekci negativních cyklů a nalezení nejkratší cesty v grafu.

Bellman-Fordův algoritmus [23] je specifický tím, že se dá použít pro výpočet nejdelší cesty u grafů obsahujících spojnice s negativními hodnotami a detekuje existenci pozitivních cyklů. Aby bylo dosaženo tohoto požadovaného chování, je třeba před jeho průběhem vynásobit spojnice grafu hodnotou -1, a po provedení výpočtu navrácenou hodnotu. Pokud disjunktivní graf G neobsahuje pozitivní cyklus, vrací algoritmus nejdelší cestu v grafu G (danou součtem hodnot spojnic mezi uzly v cestě). Existence pozitivního cyklu v grafu označuje řešení jako nepřijatelné pro řešení optimalizační úlohy [1].

4.3 Modifikovaná Shifting bottleneck heuristika

Heuristika je popsána v [29]. Od nemodifikované heuristiky se liší následovně:

V kroku 2 původní heuristiky jsou dále u řešení podproblému $1 \mid r_j \mid L_{max}$ časy dokončení operací d_{ij} vypočítány pomocí:

- maximálního času provádění operace ($P_{ij}+$), pokud se jedná o operaci probíhají na stroji
- transportního času LT_{kl} mezi stroji k a l, pokud se jedná o transportní operaci.

V kroku 4 je graf otestován na existenci pozitivních cyklů. Pokud nějaký obsahuje, dochází k opravě sekvence na stroji k:

1. disjunktivní spojnice operací na stroji k jsou smazány z grafu
2. jsou postupně vytvářeny nové sekvence záměnou dvou zaměnitelných operací prováděných na tomto stroji
3. pro každou vygenerovanou sekvenci je provedeno následující:
   (a) disjunktivní spojnice jsou přidány do grafu
   (b) nad grafem je proveden Bellman-Fordův algoritmus: pokud graf neobsahuje pozitivní cyklus, algoritmus pokračuje krokem 5 (dle původní heuristiky), jinak dojde ke smazání disjunktivních spojnic a je testována další vygenerovaná sekvence.

Protože počet možných řešení úlohy je kvůli existenci *no-wait* omezení snížen, tato modifikace slouží k širšímu prohledávání prostoru řešení.

4.4 Modifikovaný genetický algoritmus

Na základě požadovaných vlastností (multiobjektivní optimalizace) byl zvolen elitistický algoritmus NSGA-II využívající nedominantní třídění. Jeho úkolem je minimalizace objektivních funkcí *Storage* a C_{max} (čas výroby tohoto časového plánu).

Hodnota objektivní funkce *Storage* je definovaná jako suma časových jednotek, po které jsou produkty odloženy ve vstupních/výstupních bufferech strojů. Hodnotu objektivní funkce C_{max} určuje délka výrobního procesu.

Chromozom jedince tvoří posloupnost operací a transportních úloh na strojích a transportních zdrojích (obrázek 4.4).

TP11	TP12	TP13	TP21	TP22	O21	O11	O22	O12	O23	O13	O14
R1	R1	R2	R1	R2	M2	M1	M3	M3	M1	M2	M4

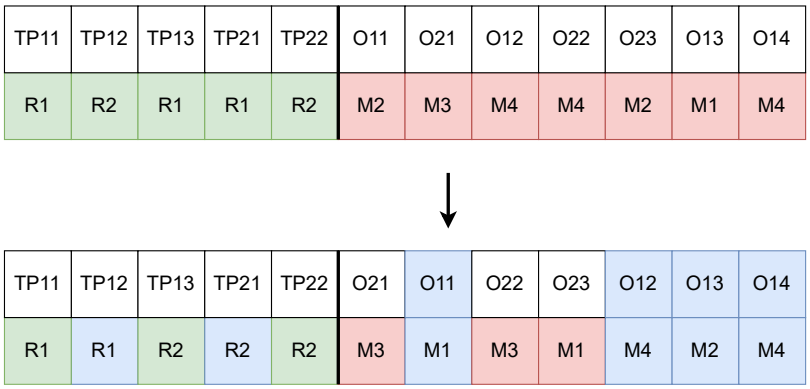
Obrázek 4.4: Detail chromozomu (TP - transportní operace, O - operace na strojích, R - transportní zdroje, M - stroje).

Velikost populace i počet generací jsou dány hodnotou konfiguračního parametru. Co se týče vztahu těchto hodnot, je vhodné řídit se závěry [27] (větší populace a méně generací).

Protože je prostor řešení velmi omezen a nalezení přijatelného řešení může zabrat velké množství času, podmínkou ukončení je vyčerpání počtu generací. V případě aktivovaného konfiguračního parametru nekonečného generování je deaktivována ukončovací podmínka vyčerpání nastaveného počtu generací.

Genetické operátory

V případě mutace je možné vybírat z alternativních strojů pro operaci zpracovávanou na konkrétním stroji. Také je možné náhodně zaměnit pořadí operací namáčení nebo transportu n jobů v rámci chromozomu. Míra mutace chromozomu je stanovena hodnotou konfiguračního parametru. Průběh mutace je znázorněn na obrázku 4.5.

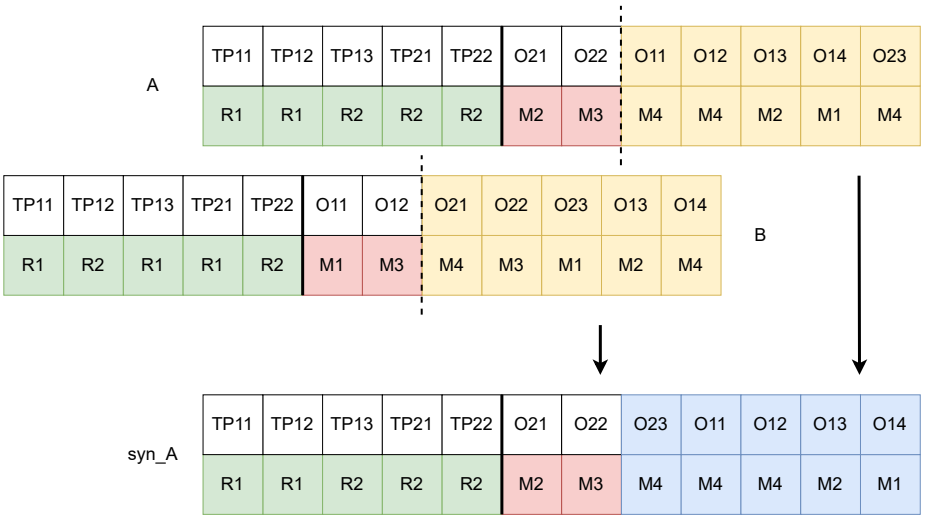


Obrázek 4.5: Operátor mutace (TP - transportní operace, O - operace na strojích, R - transportní zdroje, M - stroje).

Křížení (na obrázku 4.6) probíhá v náhodně zvoleném bodu m a vždy jsou zkříženi pouze dva jedinci. Výsledkem křížení jsou dva nově vzniklí jedinci, kteří v populaci nahradí svoje rodiče. Způsobem jakým jedince zkřížit je následující algoritmus:

1. Pomocí operátoru selekce jsou vybráni kandidáti ke křížení: A a B.
2. Jsou vytvořeny kopie rodičů - syn_A a syn_B.
3. Podle úspěšnosti potomka (je přenesena z rodiče a stanovena na základě sestavování disjunktivních grafů z jeho chromozomu) je určeno, jaká jeho část by měla být křížena. V případě sestavení úplného grafu G' jsou kříženy části nemáčení nebo transportů. V případě sestavení pouze jednoduchého grafu G jsou kříženy transporty. V případě nesestavení jednoduchého grafu G jsou kříženy části namáčení.
4. U posloupnosti operací namáčení nebo transportů je určen náhodný bod křížení m a je stanoveno, zda se má aplikovat levá nebo pravá část druhého rodiče.
5. Ze zvolené části rodiče je vytvořen seznam identifikátorů operací produktů. Ten je následně aplikován na tuto část chromozomu potomka.
6. Obdobně se postupuje i u druhého potomka.

Deduplikační operátor má za úkol vyřadit z populace jedince, kteří mají stejné konkrétní znaky jako jiný jedinec v současné populaci (totožné pořadí operací namáčení a transportů v rámci jedince).



Obrázek 4.6: Příklad křížení dvou jedinců - části namáčení v bodu m = 2 (TP - transportní operace, O - operace na strojích, R - transportní zdroje, M - stroje).

4.5 Finální heuristika

Jedná se o modifikovanou Shifting bottleneck heuristiku inspirovanou v [29], využívající k efektivnímu prohledávání prostoru genetický algoritmus NSGA-II. Postup je následující:

1. Po spuštění algoritmu jsou vygenerováni výchozí členové populace - sekvence operací - na odpovídajících strojích (vanách). Transportní operace mezi nimi jsou přiřazeny náhodným transportním zařízením (jeřábům).
2. Pro každou generaci je provedeno:
   (a) U každého člena populace je provedeno:
      i. Pro každý stroj je vytvořen časový plán operací.
      ii. Dojde k vytvoření jednoduchého disjunktivního grafu G, kde množina D_m obsahuje pouze disjunktivní spojnice prvního ze strojů (aby byly do jisté míry respektováno pořadí operací v rámci jedince, generované genetickými operátory). Na graf je aplikována modifikovaná Shifting bottleneck heuristika. Pokud se nepodaří najít řešení bez pozitivních cyklů, pokračuje se krokem *iv*.
      iii. Dojde k vytvoření úplného grafu G', kde D_m obsahuje disjunktivní spojnice operací na strojích získaných v předchozím kroku a $D_t = \emptyset$. Na graf je aplikována modifikovaná Shifting bottleneck heuristika (jsou plánovány pouze transportní operace).
      iv. Dojde k vyhodnocení původní sekvence tohoto člena populace - pokud se nepodařilo sestrojit graf G, hodnoty C_{max} a *Storage* jsou nastaveny na vysokou hodnotu a jsou vynásobeny konstantou 16 (aby došlo k odlišení jednotlivých sad řešení). Pokud se nepodařilo sestrojit graf G', jsou hodnoty C_{max} a *Storage* vypočítány z grafu G a vynásobeny konstantou 5. Pokud se podařilo najít řešení s hodnotou $Storage = 0$, je uloženo (pokud je stávající nejlepší řešení horší, tak je přepsáno).

(b) Dochází ke křížení, aplikaci deduplikačního operátoru a mutaci jedinců. Při nedostatečném počtu jedinců z důvodu deduplikace dochází ke generování nových jedinců opakovaným křížením.

3. Pokud má dojít k zastavení algoritmu na základě ukončovacích podmínek, vrátit nejlepší nalezené řešení s hodnotou $Storage = 0$. Jinak se pokračuje krokem 2.

4.6 Návrh aplikace

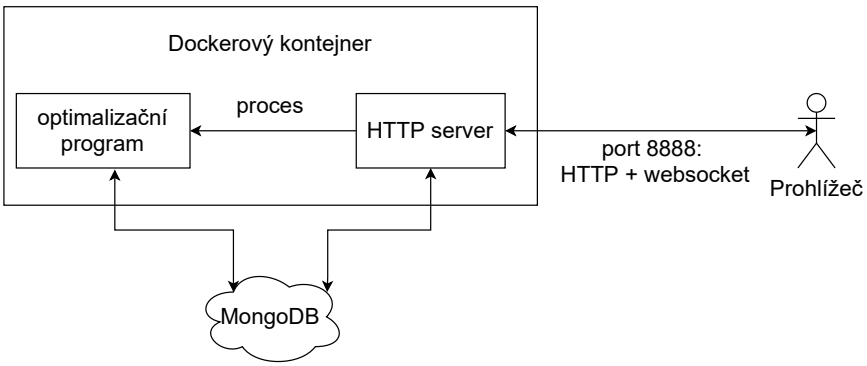
Tato sekce obsahuje návrh aplikace, která umožní vytvoření a vizualizaci časového plánu nad zadanými produkty, prohlížení výsledků proběhlých optimalizačních úloh, a uživatelské nastavení konfiguračních parametrů algoritmu.

Použitá technologie

- python3 - programovací jazyk pro vědecké výpočty a rychlý návrh aplikací
- javascript/HTML/CSS - technologie klientské části
- docker - kontejnerizace řešení, přenositelnost aplikace

Architektura řešení

Řešení je postaveno na architektuře klient-server. Jedná se o dockerový kontejner obsahující HTTP server, ke kterému je možné se připojit z prohlížeče (klientské webové rozhraní). Aplikace ukládá uživatelská data v cloudu (MongoDB Atlas). Koncept ilustruje obrázek 4.7.



Obrázek 4.7: Architektura řešení klient-server.

HTTP server přijímá požadavky, otevírá websocketové spojení pro komunikaci v reálném čase, a ukládá uživatelská data do cloudové databáze. Optimalizační program je spuštěn jako nový proces. Websocketové spojení mezi klientským prohlížečem a serverem umožní čekání na výsledek probíhající úlohy. Přihlášení k aplikaci je podmíněno znalostní výchozího uživatelského jména a hesla.

Parametry optimalizačního programu

V rámci uživatelského rozhraní je možné zejména přidávat/odebírat/modifikovat jeřáby (transportní zdroje) a vany (stroje). U každé vany je možné nastavit hodnoty následujících

konfiguračních parametrů (viz tabulka 4.2). U každého jeřábu je možné nastavit následující parametry (tabulka 4.3).

Tabulka 4.2: Konfigurační parametry mořicí vany.

Název parametru	Typ	Popis
Název vany	string	Popis vany.
Označení vany	string	Krátké označení vany do ganttova diagramu a animace (M_1, M_2..)
Position	int <0; n>	Celočíselná hodnota reprezentující pořadí vany, použitá pro výpočet vzdálenosti mezi vanami (vany vedle sebe = vzdálenost 1).
Exp_min	float	Minimální expozice ve vaně.
Exp_max	float	Maximální expozice ve vaně.
Drain	float	Počet časových jednotek nutných pro okap.
Dive	float	Počet časových jednotek nutných pro ponoření produktu do vany (z horní polohy jeřábu).

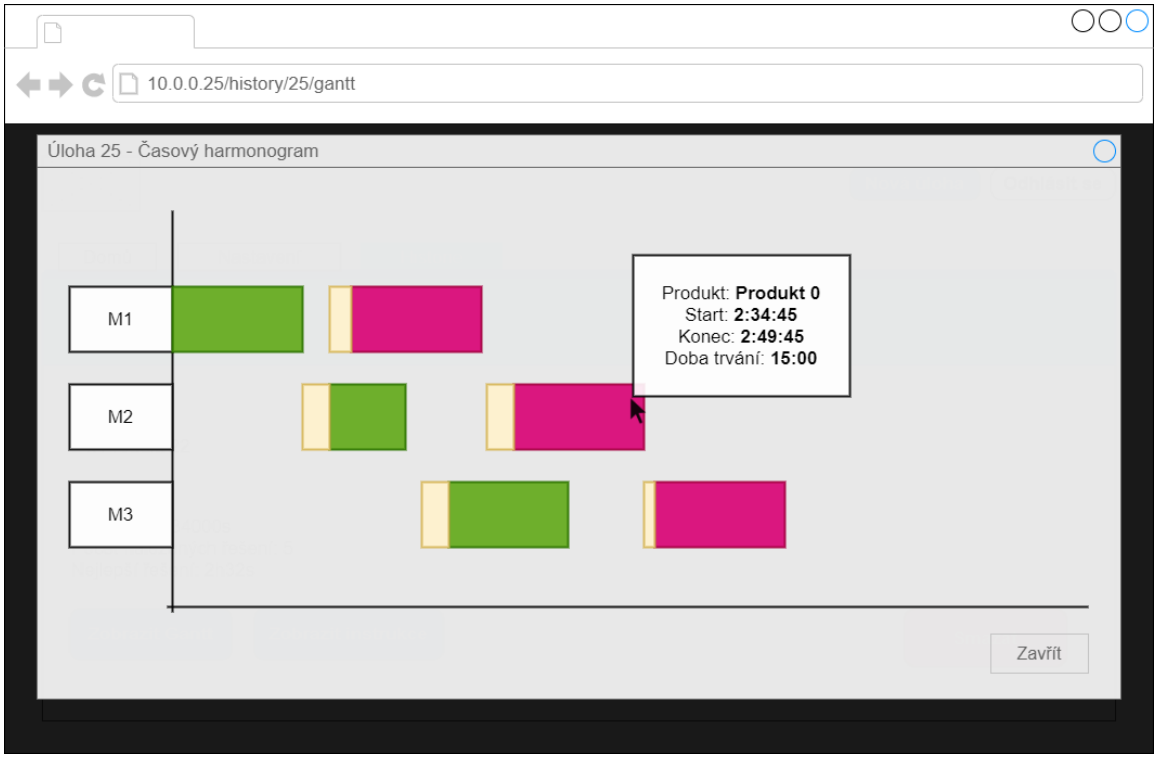
Je možné přidávat produkty a u každého definovat posloupnost operací, včetně volby alternativních van pro provedení namáčecí operace. Produkty se dají následně vybrat při vytváření nové optimalizační úlohy. Kvůli náročnosti úlohy je při vytváření úkolu možné specifikovat hodnotu, dle které se bude počet plánovaných produktů multiplikovat (při plánování 2 produktů A a 3 produktů B s multiplikátorem 4 bude výsledný počet produktů 8·A a 12·B). V nastavení aplikace je možné změnit hodnoty těchto konfiguračních parametrů, které změní chování NSGA-II (tabulka 4.4). Výchozí hodnoty těchto parametrů jsou uvedeny v příloze B.

V rámci aplikace je možné prohlížet již dokončené optimalizační úlohy, u kterých byl nalezen výsledek. Každý výsledek ukládá informace o délce trvání procesu, datu průběhu a nalezeném řešení. Součástí jsou abstraktní textové instrukce pro automatické řízení pohybu jeřábu, ganttův diagram reprezentující časový harmonogram řešení a animace vizualizující výrobu. Ganttův diagram graficky reprezentuje posloupnost namáčecích a transportních operací. Po kliknutí na každou z operací se zobrazí základní údaje operace (viz obrázek 4.8).

Při plánování úlohy dojde k výběru jeřábů a produktů, které mají být zpracovány. Je nastaven počet jednotlivých produktů a je také možné změnit hodnoty konfiguračních parametrů. Průběh úlohy je možné kdykoliv zrušit. Návrh aplikace počítá s tím, že jeřáby nejsou z hlediska jejich pohybu přiřazeny konkrétním vanám. Každý jeřáb obsluhuje vanu z vlastní kolejnice, nedojde ovšem ke střetu více produktů při namáčení produktů jedné vany nebo okapu nad vanou (tento problém je vyřešen modifikací disjunktivního grafu).

Tabulka 4.3: Konfigurační parametry jeřábu.

Název parametru	Typ	Popis
Název jeřábu	string	Popis jeřábu.
Označení jeřábu	string	Krátké označení jeřábu do ganttova diagramu a animace (R_1, R_2..)
Raise_low	float	Počet časových jednotek nutných pro zdvih do nízké polohy (od ponoření).
Raise_high	float	Počet časových jednotek nutných pro zdvih do vysoké polohy (z nízké polohy).
Loaded_move	float	Počet časových jednotek nutných pro přejezd tohoto jeřábu s naloženým produktem o jednu pozici.
Empty_move	float	Počet časových jednotek nutných pro prázdný přejezd tohoto jeřábu o jednu pozici.



Obrázek 4.8: Drátový model: ganttův diagram reprezentující nalezené řešení.

Tabulka 4.4: Konfigurační parametry aplikace.

Název parametru	Typ	Popis
Pravděpodobnost mutace změnou prvku	float <0; 1 - *pravděpodobnost mutace změnou pořadí*>	Pravděpodobnost, že bude prvek mutován změnou prvku za alternativní transportní zdroj nebo stroj.
Pravděpodobnost mutace změnou pořadí prvku	float <0;1>	Pravděpodobnost, že bude prvek mutován změnou pořadí operací v prvku.
Počet měněných operací namáčení při mutaci změnou prvku	int <0;*len(clen_['namaceni'])*>	Hodnota je použita pro výpočet pravděpodobnosti, že konkrétní operace bude mutována.
Počet měněných operací transportu při mutaci změnou prvku	int <0;*len(clen_['transporty'])*>	Hodnota je použita pro výpočet pravděpodobnosti, že konkrétní operace bude mutována.
Počet měněných jobů při mutaci pořadí prvku	int <0; počet jobů>	Počet jobů, u kterých bude změněno pořadí namáčecích nebo transportních operací v rámci chromozomu.
Počet generací	int <1; *n*>	Počet generací NSGA-II.
Populace	int <2; *n*>	Velikost populace NSGA-II.
Počet procesů	int <1; *n*>	Hodnota určující počet procesů poolu u paralelního zpracování populace.
Nekonečné generování	*True*/*False*	Pokud je hodnota *True*, je počet generací při zahájení řešení nahrazen za vysokou konstantu (1.000.000+).

Kapitola 5

Implementace aplikace

Tato sekce obsahuje popis implementace aplikace a algoritmů navržených v předchozí sekci. Obsahuje detaily implementace HTTP serveru, informace ke spuštění aplikace a provozu, strukturu HTML šablon a databáze. Dále popisuje implementaci procesu optimalizační úlohy v balíku *task* a komunikaci mezi jednotlivými moduly v tomto balíku.

5.1 Volba řešení a použité knihovny

Architektura klient-server je implementována dle návrhu v sekci 4.6. V implementaci jsou použity následující knihovny jazyka python3:

- *dnspython*[1], *tornado*[2] - implementace webového serveru
- *numpy*[3], *pymoo*[4] - knihovna pro provádění multiobjektivní optimalizace s možností paralelizace, obsahující implementaci genetického algoritmu NSGA-II
- *matplotlib*[5] - vykreslení ganttových grafů pro odladění
- *networkx*[6] - knihovna pro sestavení a analýzu grafů
- *bellmanford*[7] - implementace Bellman-Fordova algoritmu
- *pymongo*[8] - knihovna pro připojení k databázi MongoDB
- *cryptography*[9] - zabezpečení hesla v databázi
- *pathos*[10] - knihovna pro multiprocessing, dokáže serializovat více druhů objektů než vestavěný balík jazyka *multiprocessing*

Pro zápis logů do souboru *log.log* je využit vestavěný python3 balík *logging*. Logování slouží pouze pro testovací účely.

[1] https://www.dnspython.org/
[2] https://www.tornadoweb.org/en/stable/
[3] https://numpy.org/
[4] https://pymoo.org/
[5] https://matplotlib.org/
[6] https://networkx.org/
[7] https://pypi.org/project/bellmanford/
[8] https://pypi.org/project/pymongo/
[9] https://cryptography.io/en/latest/
[10] https://pyi.org/project/pathos/

5.2 Docker kontejner

Aplikace je koncipována pro spouštění na operačním systému na bázi Linuxu (testováno na Lubuntu 20.10). Pro jednodušší testování aplikace na jiných operačních systémech je aplikace umístěna do dockerového kontejneru. Zdrojový kód obsahuje soubor *Dockerfile*, který slouží k vytvoření obrazu s předinstalovanými prerekvizitami, připraveného ke spuštění jediným příkazem. Zdrojový kód nicméně obsahuje soubor *Requirements.txt* a soubor s pokyny k nativnímu spuštění *README.Md*.

Aplikaci se doporučuje spouštět z dockeru pouze v případě, že ji není možné zprovoznit nativně (možné snížení výkonu aplikace).

5.3 HTTP server

Server je implementován pomocí knihovny *tornado* jazyka python. Jedná se o velice výkonný neblokující webový framework, který podporuje protokol HTTP. Serverová část poskytuje otevřený TCP port 8888.

Implementace webového serveru se nachází v souboru *server.py*. Dochází zde k vytvoření připojení ke cloudové databázi MongoDB Atlas[11] a k předání tohoto připojení do instancí HTTP handlerů, jež odpovídají za zpracování různých HTTP požadavků. Dále dochází k nastavení XSRF zabezpečení u cookies [7], na jejichž základě je ověřováno uživatelovo přihlášení do aplikace. Toto ověření je implementováno přepsáním metody *get_current_user* třídy *RequestHandler* v modulu *handlers.custom*. Následně je možné v rámci handlerů využívat dekorátor *@authenticated*, který nepřihlášené uživatele směruje na adresu „/login“. Tuto validaci je možné využívat u všech handlerů - jejich metod *post* i *get*. Protože se jedná o jednoduché klientské rozhraní se staticky načítanými šablonami, nejsou pro aktualizaci a mazání záznamu použity zprávy typu UPDATE a DELETE (dle REST by měly být použity, formuláře HTML zatím tyto metody nepodporují) - je tato funkcionalita v handlerech zpracovávána metodou post.

Seznam handlerů je následující:

- *LogoutHandler* - metoda *get* provede smazání zabezpečené cookie a přesměruje uživatele na adresu „/“.
- *LoginHandler* - metoda *post* zpracuje data z přihlašovacího formuláře (email a heslo), případně nastaví zabezpečenou cookie a přesměruje uživatele na adresu „/“. Metoda *get* vrátí šablonu přihlašovací stránky.
- *IndexHandler* - načítá šablonu hlavní stránky.
- *ConfigHandler* - metoda *get* načítá šablonu stránky s konfiguračními parametry. Metoda *post* zpracovává data z formuláře s konfiguračními parametry.
- *JerabyHandler*, *VanyHandler* - metoda *get* načítá šablonu s detailem jeřábu, seznamem jeřábů nebo vytvořením nového jeřábu. Metoda *post* zpracovává data z formuláře - buď se jedná o vytvoření nového jeřábu, uložení změn u stávajícího jeřábu či smazání záznamu z databáze. Totéž platí pro handler *VanyHandler*. *VanyHandler* zároveň obsahuje metodu, která při smazání záznamu vany kaskádově smaže i dotčené produkty (ty jsou na vany pevně fixované).

[11] https://www.mongodb.com/cloud/atlas2

- *ProduktyHandler* - metoda *get* vrací šablonu s detailem produktu, seznamem produktů nebo přidáním nového produktu. Metoda *post* zpracovává data z formuláře - uložení údajů, vytvoření nového produktu nebo smazání produktu.

- *HistorieHandler* - metoda *get* vrací detail dokončené úlohy s nalezeným řešením. Dochází zde ke generování klikatelných časových úseků (k zobrazení detailů úseku je využita javascriptová knihovna *JqueryUI*[12] a knihovna *Popper.js*[13]), je vytvořena sada textových instrukcí pohybu jeřábů pro zobrazení v šabloně a ze záznamu řešení úlohy z databáze je vytvořen seznam instrukcí, které jsou v souboru šablony dynamicky převedeny na javascriptový kód. Výsledkem je spustitelná animace s využitím knihovny *GSAP*. Metoda *post* odpovídá za provedení smazání jednoho nebo více záznamů z databáze.

- *TaskHandler* - metoda *get* vrací šablonu s formulářem pro vytvoření nové úlohy. Součástí tohoto formuláře je výběr počtu produktů, které mají být zpracovány a konkrétních jeřábů odpovědných za transportní operace. Data z tohoto formuláře jsou zpracována metodou *post*, která vrací šablonu probíhající úlohy. Více informací o propojení zbytku serveru a balíku *task* (procesu optimalizační úlohy) obsahuje následující sekce.

Modul *handlers.custom* obsahuje mixin objekt, který je využíván metodou *get* v rámci handlerů s podobnou funkcionalitou (*VanyHandler*, *JerabyHandler*, *HistorieHandler* - načtení detailu záznamu z databáze).

Přihlášení do aplikace

Přihlášení probíhá zadáním e-mailové adresy a hesla. Pro případy, kdy server není spuštěn lokálně, je bezpečnost zajištěna pomocí TLS. Samopodepsaný certifikát a klíč jsou k dispozici v adresáři *ssl*.

Ověření hesla probíhá pomocí utility *Scrypt*, který je součástí knihovny *cryptography*. Jedná se o PKDF (derivační funkci podobnou *bcrypt*, která je navržena tak, aby znesnadnila útok hrubou silou na heslo, např. pomocí specializovaných HW zařízení). Heslo je v databázi uložené v podobě zakódovaných bajtů, spolu s heslem je zde uložena i sůl (náhodný vektor).

Přihlašovací údaje

https://localhost:8888/login
e-mail: *test@shophacker*
heslo: *xplsekpassword25*

Propojení serveru a balíku task

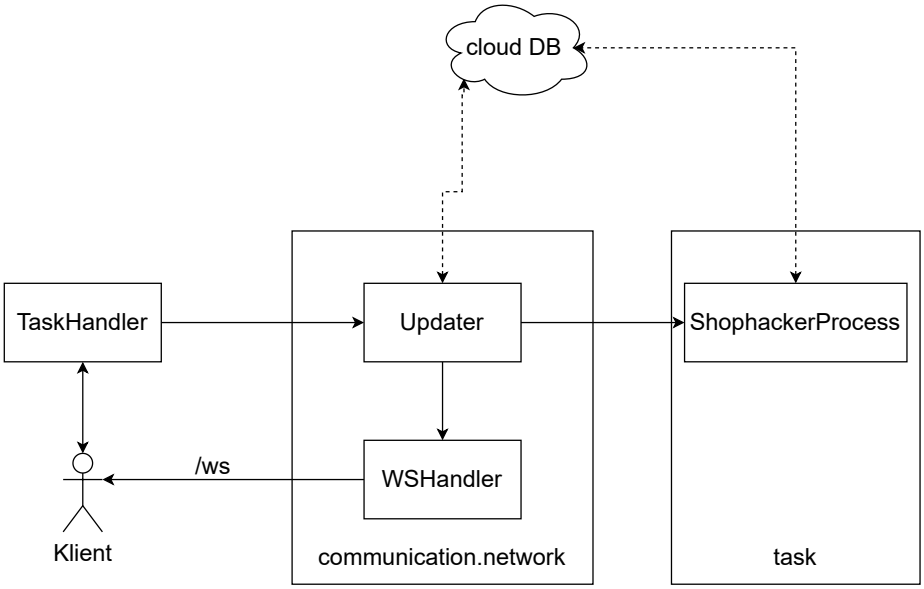
Před vrácením šablony založí *TaskHandler* instanci *WSHandler* - websocket handler, který se nachází v modulu *communication.network*. K tomuto zabezpečenému websocketu (wss://) na adrese „/ws“ se nyní může připojit klientský javascriptový kód obsažený v navrácené šabloně probíhající úlohy. Pomocí websocketu informuje server klienta o stavu zadaného úkolu. Aktualizace probíhá tak, že *TaskHandler* při zahájení úkolu zavolá funkci *startTask* třídy *Updater*, která se nachází v modulu *communication.network*. Pomocí funkce je zahájen

[12] https://jqueryui.com/
[13] https://popper.js.org/

proces *ShophackerProcess* (popsán níže). Z třídy *Updater* je po 5 sekundách volána funkce *update*. Ta nejprve kontroluje zda proces stále běží, a poté zda byl v databázi v rámci záznamu s konkrétním ID uložen výsledek. Pokud ano, je pomocí websocketu odeslána zpráva *„update“* o nejlepším výsledku klientovi, kde ji javascript zpracuje a zobrazí v šabloně probíhající úlohy. Pokud je proces stále aktivní, dojde k naplánování spuštění této funkce znovu za 5 sekund.

V případě, že chce klient úlohu předčasně ukončit, klikne na tlačítko „Ukončit“ - při kliknutí na tlačítko je odeslán požadavek, který zpracovává metoda *TaskHandler.post*. Při tomto zpracování je zavolána funkce stop třídy *Updater*, která ukončí běžící proces s optimalizační úlohou, vloží čas ukončení do databáze k záznamu a uzavře websocket.

Pokud je úloha dokončena, *Updater* se pokusí pomocí websocketu poslat zprávu „finished“ klientovi. Pokud je klient dosažitelný, v šabloně probíhající úlohy se zobrazí výzva o dokončení úlohy. Komunikace balíku *task* a klienta pomocí modulu *communication.network* je znázorněna na obrázku 5.1. Podrobný přehled komunikace je znázorněn na obrázku 5.2.

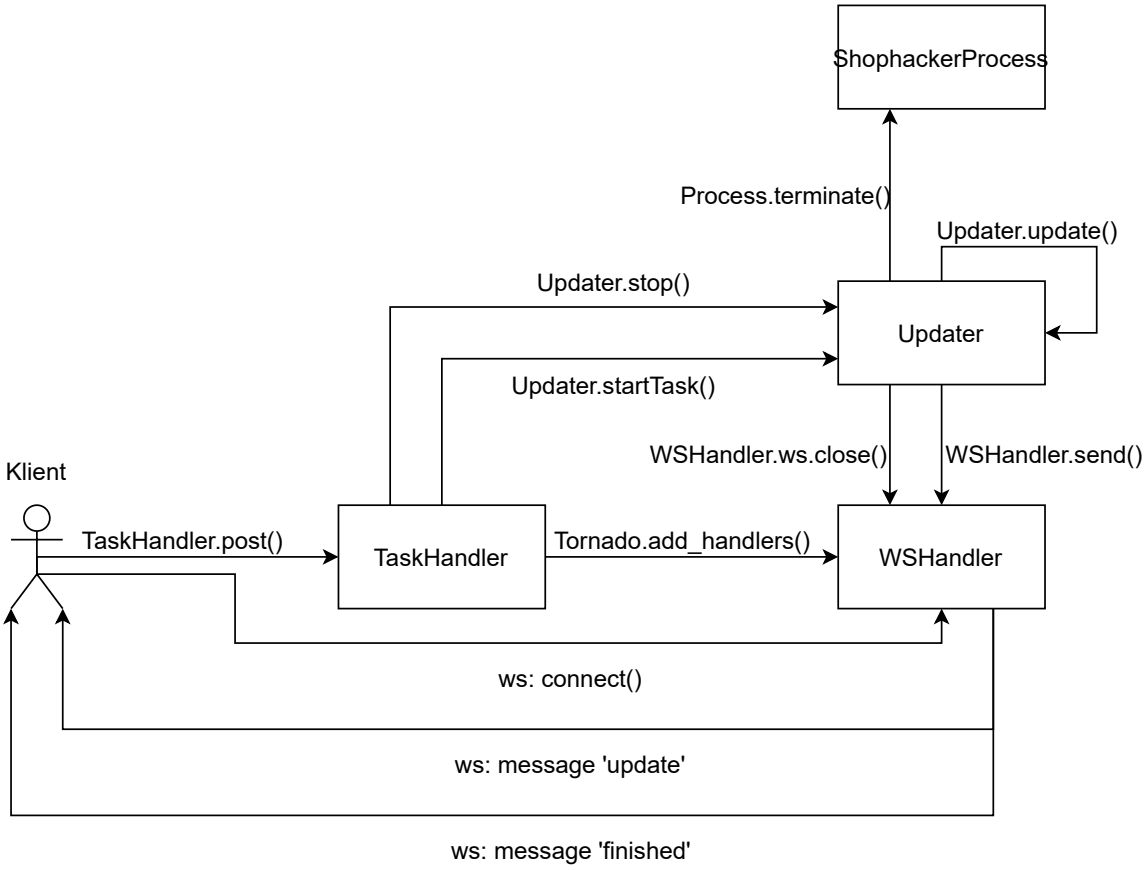


Obrázek 5.1: Ilustrace komunikace klienta a balíku task.

5.4 Šablony stránek

Soubory šablon se jsou umístěny v adresáři *templates*. V podadresáři *css*, *js* a *images* se nachází statické skripty, CSS stylopisy a obrázky pro vytvoření animace. *Tornado* umožňuje při vytváření šablony do souboru předat jako argument metody *render* proměnné, ze kterých se dá generovat HTML/CSS/JS kód přímo v souboru. Rovněž je možné používat konstrukce jazyka python - cykly, podmínky, je možné definovat nové proměnné, importovat moduly atd. Těchto vlastností je využito např. při generování dynamického javascriptového kódu animace.

Soubory šablon je možné skládat dohromady: výchozí šablona stránky je umístěna v souboru *base.html*, horní navigace je umístěna v souboru *nav.html* a jednotlivé stránky obsahu (přehledy záznamů, detaily záznamu..) v ostatních .html souborech. Tyto části se dynamicky skládají do výchozí šablony *base.html* pomocí značky *{% block %}*.



Obrázek 5.2: Komunikace komponent TaskHandler, WSHandler a Updater.

Formuláře v šablonách obsahují volání funkce *xsrf_form_html()*. Takto se vytváří formulářové pole *_xsrf* u každého formuláře typu POST, které zajišťuje autentizační vrstvu proti XSRF útoku.

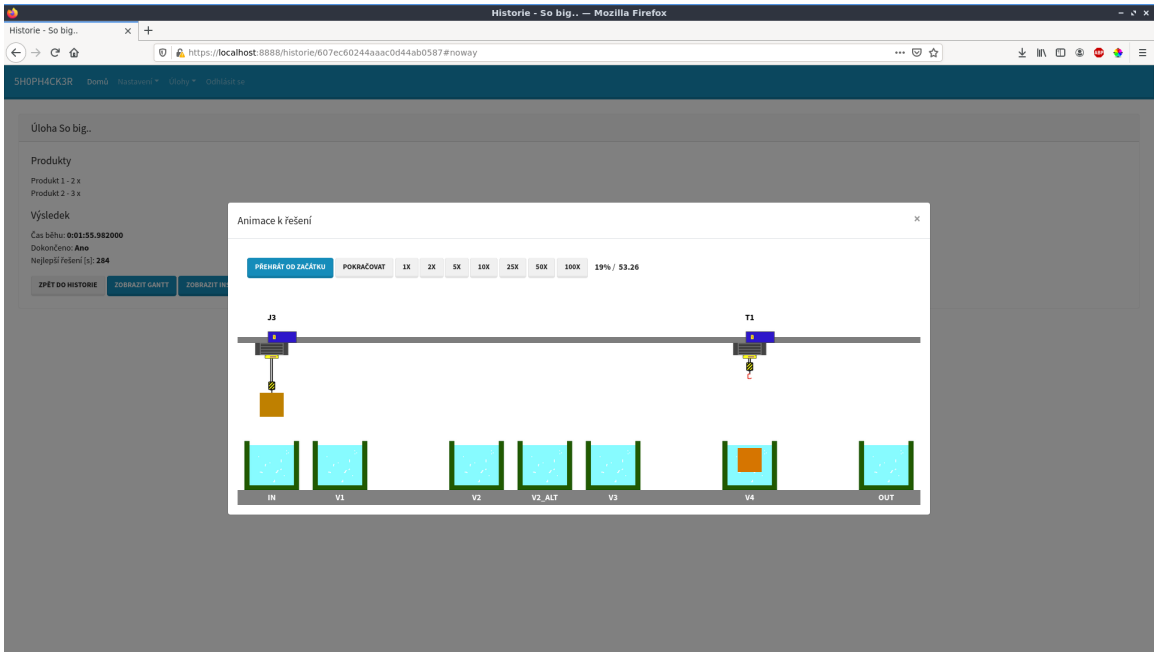
Šablony jsou postaveny na šablonovacím systému *Bootstrap*[14] - jedná se o knihovnu definující HTML/CSS/JS kód vizuálně a funkčně sladěných prvků uživatelského rozhraní (vyskakovací okna, responzivní bloky, sloupce apod).

Doporučené rozlišení
S ohledem na kvalitní zobrazení animace a ganttova diagramu je pro plnou funkčnost klientské části aplikace vyžadováno rozlišení šířky nejméně 1366px a moderní verze prohlížečů Firefox/Google Chrome.

5.5 Animace

Součástí reprezentace nalezeného řešení úlohy je animace, která simuluje transportní operace jeřábů v reálném čase. Součástí animace je možnost pozastavit přehrávání a zrychlit jej (až 100-násobně). Aby byly operace dostatečně viditelné, je koncipována na maximální počet 10 van. Probíhající animaci ukazuje obrázek 5.3.

[14] https://getbootstrap.com/



Obrázek 5.3: Probíhající animace namáčení.

Animace je vytvořena dynamicky vytvořena pomocí javascriptové knihovny *GSAP*[15] - nejprve je ze záznamu v databázi vytvořena posloupnost akcí které se mají provést. Tato posloupnost je následně přímo v šabloně přegenerována na javascriptový kód: do *GSAP TimelineMax* objektu je postupně vkládán seznam animačních akcí které se mají provést. Každá akce určuje, který prvek stránky má být animován, změnu pozice elementu, čas zahájení a délku trvání. Výstupem je právě sekvenční *TimelineMax* objekt, který se chová jako sekvenční nástroj umožňující provádět animaci do něj umístěných prvků. *TimelineMax* umožňuje spuštění, zrychlení/zpomalení animace, opakování přehrání - tyto funkce jsou navázány na stisknutí příslušného tlačítka. Na pravé straně je umístěn časovač (aktuální výrobní doba).

5.6 MongoDB

MongoDB je nejvíce používaná NoSQL databáze. Na rozdíl od relačních databází nepracuje s tabulkami, ale ukládá nestrukturovaná data do kolekcí. Jednou z vlastností MongoDB je upřednostňování vysoké dostupnosti dat před spolehlivostí transakce (SQL principy ACID). Podporuje horizontální škálovatelnost a ukládání různých typů dokumentů do jednotlivých kolekcí. Tyto kolekce ukládají dokumenty jako objekty, tudíž se dají z databáze snadno převést např. na slovníky nebo json objekty. Také proto bývá součástí při vývoji moderních webových aplikací. Zde je MongoDB použito zejména pro jednoduchost při vývoji, ve srovnání s relační databází jako MySQL.

Protože optimalizační úloha je spouštěna v samostatném procesu, je nutné založit pro tento proces novou instanci objektu *MongoClient*, který se stará o připojení ke cloudové databázi. Ke stejné situaci dochází i v balíku *task*, kde je výpočet hodnot jedinců populace

[15] https://greensock.com/gsap/

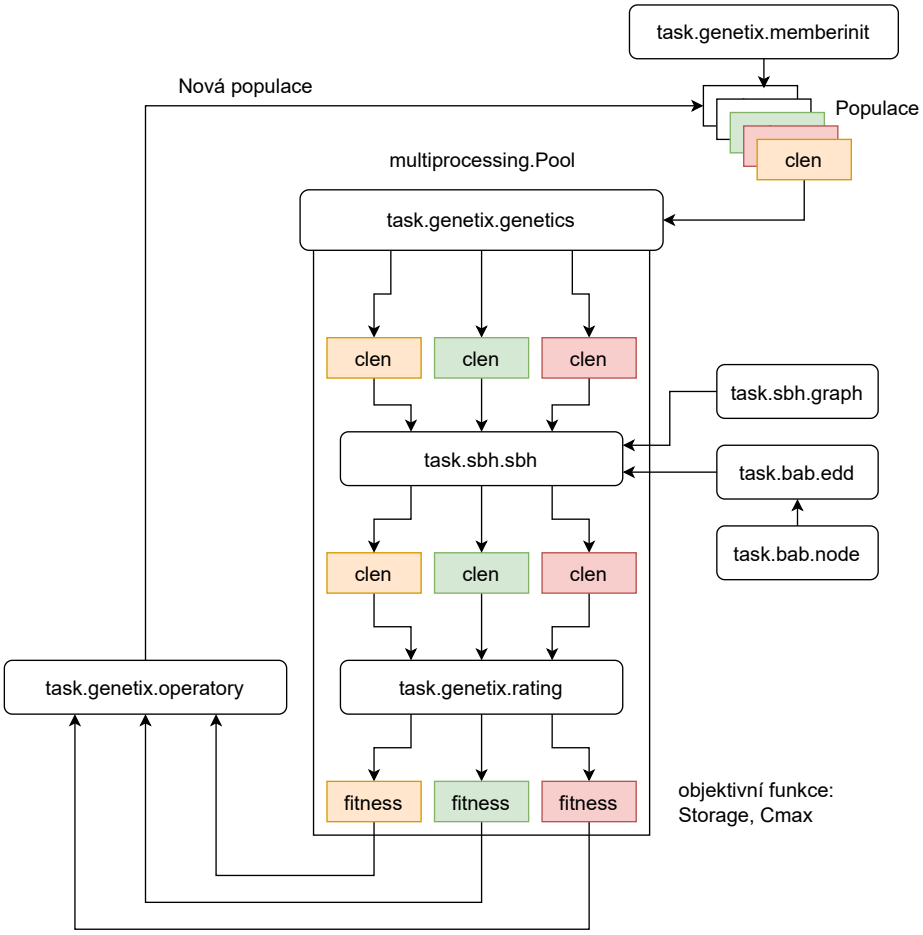
předán poolu procesů - zde je při nalezení řešení opět nutné vytvořit instanci v rámci konkrétního procesu.

Zabezpečení přenosu dat mezi databází v cloudu a lokálně spuštěným serverem je zajištěno pomocí TLS. Samotné přihlašovací údaje k databázi jsou umístěny do slovníku v souboru *credentials.py*.

Databáze je rozdělena do kolekcí uvedených v příloze 1. U každé tabulky v příloze je uvedena struktura uloženého dokumentu. Požadovaný formát parametrů lze nalézt výše v sekci 4.6.

5.7 Balík task

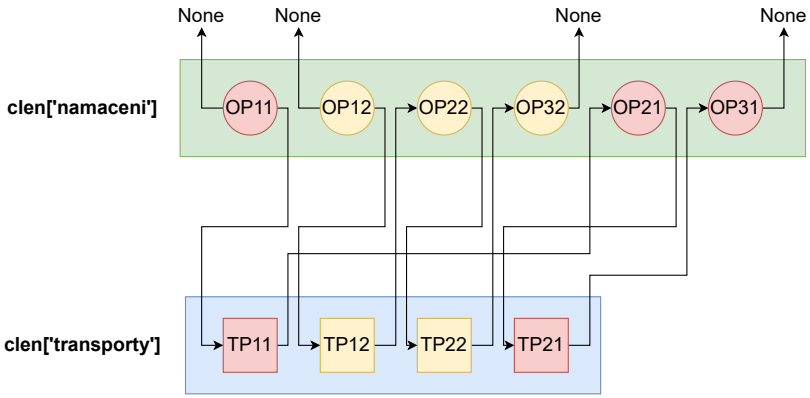
Balík implementuje popsanou heuristiku potřebnou k řešení optimalizační úlohy. Součástí jsou následující moduly. Celkový proces řešení úlohy (propojení modulů balíku *task*) je znázorněn na obrázku 5.4.



Obrázek 5.4: Schéma propojení modulů balíku task při řešení optimalizační úlohy.

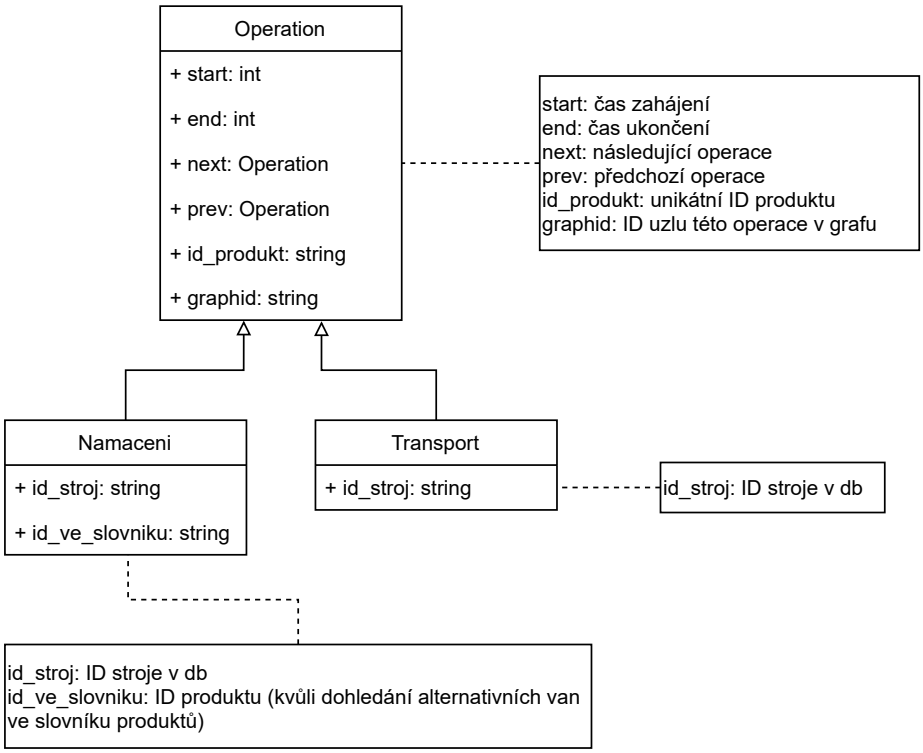
Modul **task.shophacker** obsahuje třídu reprezentující proces optimalizační úlohy. Je zde vytvořena instance algoritmu NSGA-II knihovny *pymoo* a předána do instance problému *Jobshop*, definovaného v modulu *task.genetix.genetics*. Po nepřerušovaném dokončení úlohy dojde k aktualizaci času jejího dokončení.

Modul **task.classes** obsahuje třídy, které dohromady tvoří chromozom jedince. Třídy *Transport* a *Namaceni* jsou odděleny z důvodu snazší identifikace při odlaďování a případném rozšiřování programu. Reprezentace chromozomu jedince (slovník s klíči *transporty* a *namaceni*, obsahující seřazený seznam instancí tříd *Transport* a *Namaceni*) je znázorněna na obrázku 5.5. Oba seznamy operací uložené pod klíči jsou seřazené v pořadí, v jakém budou operace probíhat. Atributy *prev* u prvních a *next* u posledních instancí třídy *Namaceni* mají hodnotu *None*.



Obrázek 5.5: Reprezentace chromozomu jedince.

Na obrázku 5.6 je vyobrazen diagram tříd modulu *task.classes*.



Obrázek 5.6: Třídy modulu task.classes.

Modul **task.genetix.memberinit** obsahuje implementaci třídy *Sampling* knihovny *pymoo*. Tato třída odpovídá za generování jedince populace úvodní populace. V rámci metody

_do dochází k vygenerování náhodného pořadí operací namáčení a transportu a přiřazení náhodných van a jeřábů.

Modul **task.genetix.genetics** obsahuje implementaci třídy *Problem* knihovny *pymoo*. Třída reprezentuje zadaný problém: V rámci metody *_evaluate* dochází k paralelnímu vyhodnocení kvality jedinců populace nad danou populací aplikací Shifting bottleneck heuristiky a validací řešení pomocí disjunktivních grafů. Toto hodnocení je umístěno do vnořené funkce *custom_eval*. Multiplikace výsledků optimalizace (pokud je žádoucí) je provedeno metodou *multiply*. Metoda *insert_to_db* odpovídá za vložení výsledku optimalizace do databáze.

Paralelizace vyhodnocování jedinců každé generace je dosaženo vytvořením instance *Pool* - poolu procesů knihovny *pathos*. Tato knihovna je použita, protože výchozí balík *multiprocessing* jazyka python nepodporuje serializaci funkce *custom_eval* v tomto modulu (pro provádění komunikace mezi procesy).

Modul **task.genetix.rating** obsahuje funkce *calculate_simple_storage* a *calculate_full_storage*, které slouží k výpočtu objektivní funkce *Storage* u daného jedince populace. Dále obsahuje funkce *simple_windowing* a *full_windowing*, které slouží k výpočtu časových oken každé operace (atributy *start* a *end* instance třídy *Namaceni* nebo *Transport*) po provedení Shifting bottleneck heuristiky.

Modul **task.genetix.operatory** obsahuje implementaci genetických operátorů knihovny *pymoo*: *Deduplikace*, *Krizeni* a *Mutace*. Třída *Deduplikace* implementuje deduplikační operátor. Metoda *is_equal* porovnává nově vygenerované členy populace a zkoumá, zda se již v populaci nenacházejí. Porovnání v tomto případě probíhá na základě pořadí identifikátorů jednotlivých zpracovávaných produktů, a to v rámci celého chromozomu (operací transportu i namáčení). Třída *Mutace* implementuje operátor mutace. Metoda *_do* provádí samotnou mutaci postupně nad každým členem populace. V závislosti na hodnotě konfiguračního parametru je proveden jeden z typů mutace. Metoda *mutace_order* provádí mutaci změnou pořadí: operace namáčení nebo transportu konkrétního jobu mění svoje pořadí v rámci chromozomu (počet jobů závisí na hodnotě konfiguračního parametru). Metoda *mutate_change* mění stroj, na kterém je zpracovávána konkrétní operace (namáčení nebo transport) - a to v závislosti na hodnotě konfiguračního parametru. Třída *Krizeni* implementuje operátor křížení. Metoda *_do* řídí samotné provádění křížení nad zvolenými členy populace (za výběr odpovídá knihovna pymoo, je použita výchozí pravděpodobnost křížení = 0.9). Metoda *crossover* implementuje samotné křížení zvolených jedinců A a B, a to na základě kvality jejich řešení. Postup křížení je popsán v sekci 4.4.

Modul **task.sbh.sbh** sdružuje funkce implementující Shifting bottleneck heuristiku a provádějící s ní spojené operace. Funkce *init_harmonogramy* vytváří seznamy reprezentující posloupnosti operací namáčení prováděných v konkrétních vanách a seznamy operací transportů prováděných na konkrétních jeřábech. Tyto seznamy jsou jedním ze vstupů Shifting bottleneck heuristiky. Funkce *fake_edd* přidá do jednoduchého disjunktivního grafu disjunktivní spojnice mezi operacemi naplánovanými na první z van. Funkce *sbh* je implementací heuristiky tak, jak je popsána v sekci 4.3. Je v ní volána funkce *branch_and_bound* z modulu *task.bab.edd*, které má za úkol nalézt nejlepší sekvenci operaci na jednom stroji aplikací EDD pravidla a metody branch and bound. Funkce *init_lb_ub* je využívána u reparačního kroku Shifting bottleneck heuristiky. Pomocí ní dochází k nalezení mezí v seznamu operací prováděných na jednom stroji, v rámci kterých je možné vyměnit tuto operaci s jinou.

Modul **task.sbh.graph** obsahuje všechny funkce, které jsou používány pro práci s disjunktivním grafem (jednoduchým i úplným). Funkce *bellman* využívá knihovnu *bellmanford*

k nalezení nejdelší cesty v disjunktivním grafu, případně vrací oznámení o pozitivním cyklu. Funkce *add_disj_from_simple* slouží k přidání disjunktivních spojnic mezi operacemi na stejném stroji, a to z jednoduchého do úplného disjunktivního grafu. Zároveň zde dochází k úpravě popsané v sekci 4.2 (viz obrázek 4.3). Funkce *update_clen_graf* vypočítá z jednoduchého nebo úplného disjunktivního grafu atributy *start* a *end* prováděných operací. Funkce *add_disjunctive* odpovídá za přidání disjunktivních spojnic do jednoduchého nebo úplného grafu, a to ze sekvence operací naplánovaných na jednom stroji aplikací EDD pravidla. Funkce *remove_disjunctive* odebírá disjunktivní spojnice konkrétního stroje z grafu. Funkce *sbh_reparation* se stará o provádění reparačního kroku u Shifting bottleneck heuristiky (pokud graf po přidání disjunktivních spojnic obsahuje pozitivní cyklus). Funkce *simple_graph* zkonstruuje jednoduchý disjunktivní graf za použití knihovny *networkx*, funkce *full_graph* úplný graf (oba bez disjunktivních spojnic).

Modul **task.bab.edd** obsahuje implementaci prohledávací metody branch and bound popsané v sekci 3.3. Funkce *branch_and_bound* postupně prohledává strom řešení, řídí generování poduzlů uzlu (uzly jsou instancemi *Node* z modulu *task.bab.node*) a ukládá nejlepší nalezené řešení. Funkce *crawl_up* provádí stoupání zpět na nadřazený kořenový uzel v případě, že dojde k vygenerování listové úrovně stromu.

Modul **task.bab.node** obsahuje implementaci třídy *Node*, která vytváří instanci uzlu stromu metody branch and bound. Součástí instance je implementace EDD pravidla pro vytvoření sekvence operací na jednom stroji. Instance *Node* obsahuje následující atributy:

- *Node.lmax* - hodnota L_{max} tohoto uzlu
- *Node.is_ok* - *True* pokud uzel není odříznutý, jinak *False*
- *Node.planned* - seznam ID operací, které jsou již naplánovány na stroji
- *Node.poddani* - seznam podřazených uzlů
- *Node.father* - odkaz na nadřazený uzel
- *Node.kriterium_A* - *True*, pokud sekvence operací uzlu není preemptivní, jinak *False*

Metoda *single_machine_planning* provede stanovení sekvence operací na daném stroji pomocí preemptivního EDD pravidla. Aby nedocházelo k ověřování zpracovávané úlohy v každém bodě časové osy, je použita technika, kdy dochází ke stanovení dalšího časového momentu, ve kterém má dojít ke změně zpracovávané operace. Toto chování implementuje metoda *find_next*. Metoda *node_test_c* slouží k ověření, zda uzel splňuje kritérium C. Pokud ne, je takový uzel označen k odříznutí nastavením atributu *is_ok* na hodnotu *False*. Metoda *calculate_pj* vypočítá čas zpracování na tomto stroji. Tato hodnota odpovídá hodnotě spojnice v disjunktivním grafu dle sekce 4.2.

Kapitola 6

Vyhodnocení výsledků

Tato sekce obsahuje detaily porovnání výsledků optimalizační úlohy s neoptimalizovanými výsledky a údaje o testovacím prostředí. Obsahuje specifikace testovacího hardware, popis testovacích dat, výsledky testování vedoucí k nalezení úzkého hrdla, porovnání získaných výsledků a na závěr několik návrhů na zlepšení výsledků optimalizace a vylepšení aplikace.

6.1 Provedené testy

Testování proběhlo na PC s operačním systémem Lubuntu 20.10, s osmijádrovým CPU AMD FX-8350 4GHz (cache 2MB), s pamětí RAM 16GB DDR3.

Celkový počet možných validních permutací c operací v namáčecí části chromozomu je dle [22] určen pomocí 6.1.

$$c = \frac{n!}{m_1! \cdot m_2! \cdot \ .. \ m_k!} \tag{6.1}$$

Hodnota n určuje celkový počet operací namáčení, které budou součástí chromozomu a m_x určuje počet namáčení operací v rámci konkrétního produktu x. Platí tedy, že $n = m_1 + m_2 + .. \ m_k$. Už pro úlohy relativně malého rozsahu se jedná o vysoká čísla, např. u 1 produktu s 6 a dvou produktů se 4 operacemi namáčení jde o 210210 potenciálních výsledků. Jejich počet je navíc umocněn permutacemi transportních operací.

Prostor řešení je velmi omezen díky existenci no-wait omezení a neexistenci vstupních a výstupních bufferů. V množině permutací se nachází pouze zlomek těch, které reprezentují řešení problému. Je tedy nutné otestovat velké množství řešení, než dojde k nalezení validního výsledku, a proto je potřeba identifikovat úzké hrdlo. Data v adresáři *tests/analyzy/test1* byla získána měřením časů průběhu funkcí a metod modulů *task.bab*, *task.sbh* a funkce *custom_eval* pro hodnocení člena populace z modulu *task.genetix.genetics*. Samotná časová data jsou lehce zkreslena zapisováním výsledků do logu. Jeden ze získaných výsledků je uveden v tabulce 6.1. Data jsou získána na šesti jádrech, jedná se o 2 produkty s 6 operacemi namáčení a 1 produkt se 4 operacemi. Byla otestována 1 populace o 50 členech.

Z dat lze snadno vyčíst, že vysoce vytěžovanou částí optimalizačního procesu jsou funkce pro práci nad disjunktivním grafem (*bellman*, *calculate_pj*) a funkce používané v rámci řešení podproblému $1 \mid r_j \mid L_{max}$ Shifting bottleneck heuristiky (*single_machine_planning*, *calculate_pj*). Časové výsledky funkce *custom_eval* stanovují, jak dlouho trval samotný proces vyhodnocení všech jedinců.

Tabulka 6.1: Měření 1: úzké hrdlo optimalizační úlohy.

Název funkce/metody	Počet volání	Celkový čas průběhu (s)
bellman	30141	21,174459070000093
calculate_pj	14059	0,06947610000000065
find_next	9895	0,033270860000000096
single_machine_planning	3018	20,796203979999994
node_test_c	1777	0,007619700000000006
add_disjunctive	1036	0,07346661999999998
remove_disjunctive	627	0,008317949999999998
crawl_up	605	0,0016783299999999994
branch_and_bound	408	20,97973208999999
sbh	83	26,87930442
sbh_reparation	53	5,076683440000002
init_lb_ub	53	0,00120215
custom_eval	50	27,213285100000007
fake_edd	50	0,03260589999999999
simple_graph	50	0,014702900000000005
init_harmonogramy	50	0,00880272
update_clen_graf	33	0,2680320599999999
full_graph	33	0,01802939
add_disj_from_simple	33	0,00214208

K ověření, že program správně pracuje s pamětí (resp. zdroje alokované pro jedince v rámci jedné populace se nepřenáší do další populace) byly použity dvě generace jedinců o četnosti 100. Výstup příkazu *free -s 1* byl přesměrován do souboru a výsledky jsou dostupné v adresáři *tests/analyzy/test2*. Bylo ověřeno, že garbage collector jazyka python po dokončení zpracování jedné populace postupně uvolňuje alokovanou paměť.

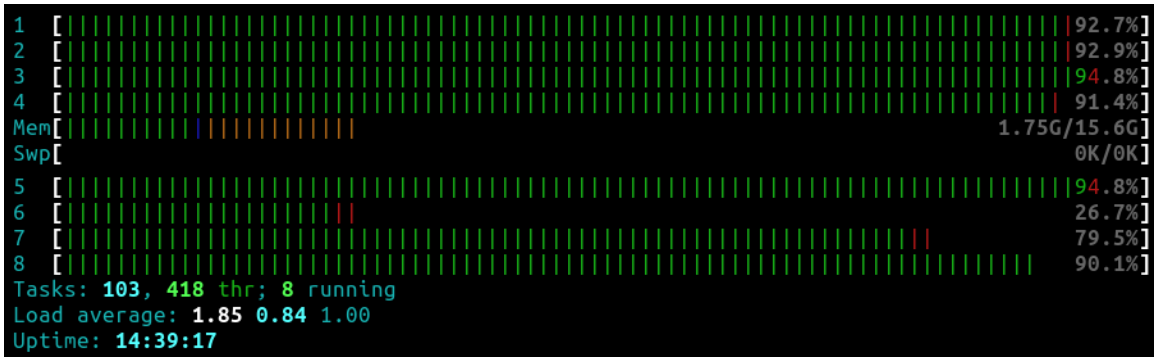
Rovněž byla otestována paralelizace úlohy. Dvakrát za sebou byl změřen čas provádění vyhodnocení 3 generací o velikosti 50 jedinců, a to postupně v 1, 3 a 6 paralelních procesech. Úplné výsledky jsou k dispozici v adresáři *tests/analyzy/test3* a shrnuty v tabulce 6.2.

Tabulka 6.2: Měření 1: testování paralelního zpracování na 3 generacích o 50 jedincích.

Počet procesů	Čas průběhu 1	Čas průběhu 2
1	4m44s	5m47s
3	1m47s	2m33s
6	58s	1m10s
7	1m17s	43s

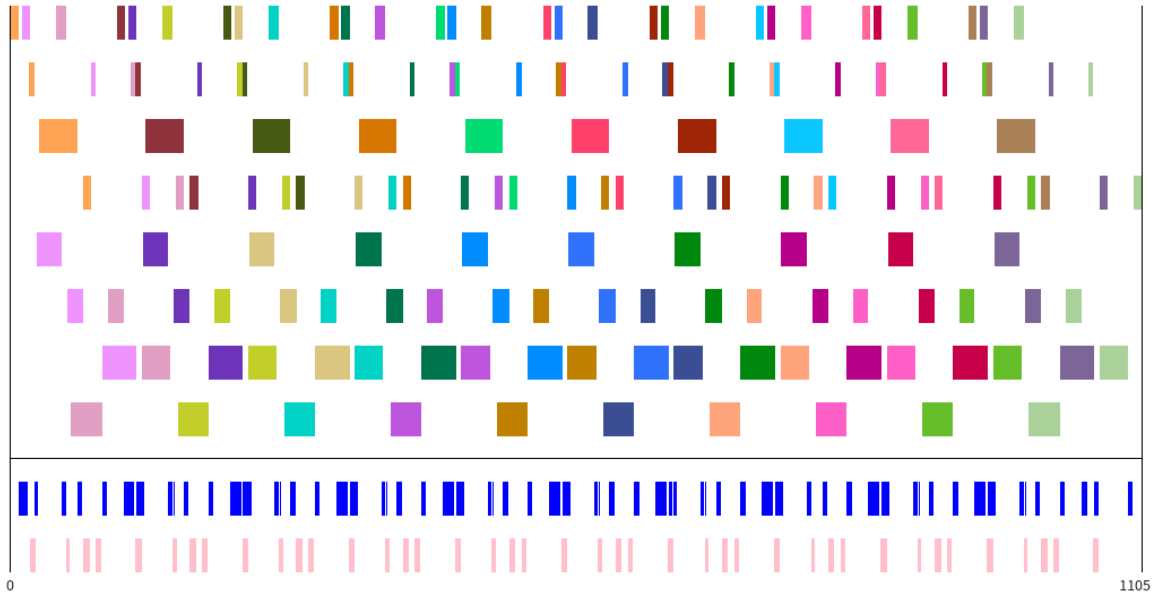
Výsledky ukazují, že paralelizace snižuje celkovou dobu vyhodnocování. Vytížení jader při vyhodnocování odpovídá použitému počtu procesů, viz obrázek 6.1. Mezi 6 a 7 procesy už rozdíl není natolik markantní, také závisí na složitosti vyhodnocování konkrétních jedinců (jak ukazuje rozdíl mezi 6 a 7 procesy v 1. provedeném měření).

Dále byl otestován multiplikátor výsledku optimalizace. Původní úloha s 2 stejnými produkty o 6 operacích namáčení a 1 produktem se 4 operacemi má řešení reprezentované na obrázku 6.3. Toto řešení má celkový výrobní čas $C_{max} = 154$ (řešení lze nalézt na stránce *Historie* ve webové aplikaci). V případě provedení úlohy s multiplikátorem 10 (ve



Obrázek 6.1: Část výstupu programu *htop* při provádění optimalizační úlohy paralelně v 7 procesech.

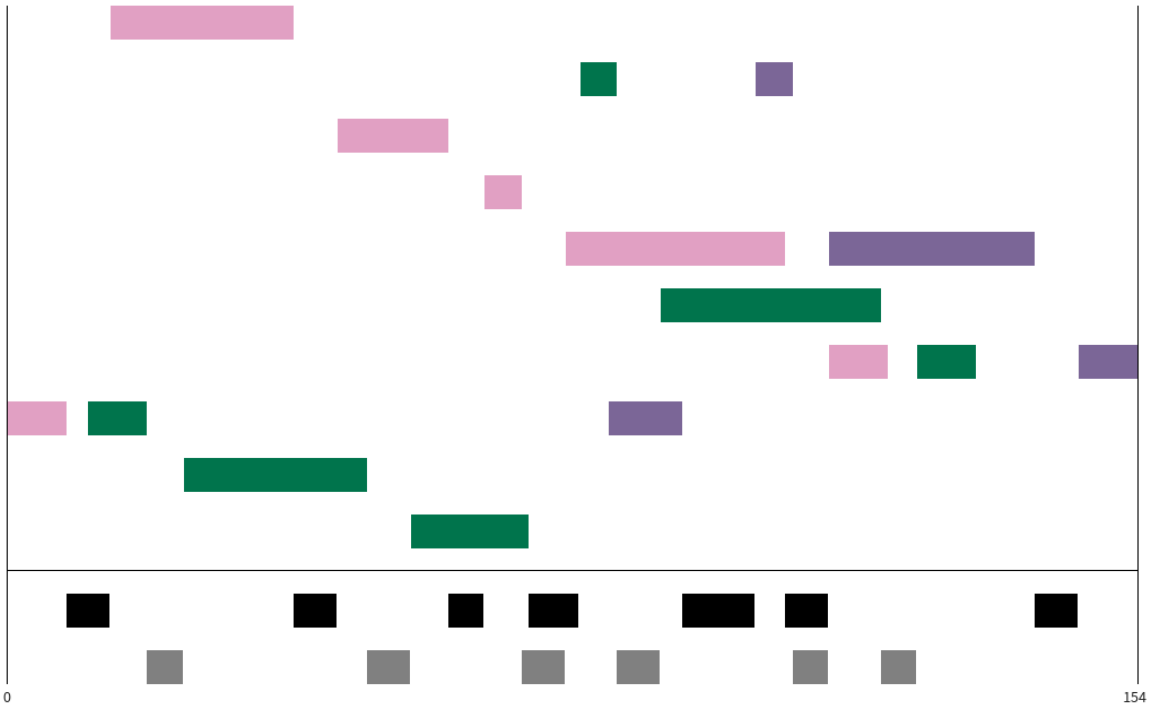
výsledku tedy 20 produktů o 6 operacích a 10 produktů o 4 operacích namáčení) je výsledek s $C_{max} = 1105$ uveden na obrázku 6.2.



Obrázek 6.2: Použití multiplikátoru 10 na úlohu s 2 stejnými produkty s 6 operacemi a 1 produktem se 4 operacemi namáčení. Výrobní čas $C_{max} = 1105$. V horní části se nacházejí operace namáčení, v dolní operace transportu (na stejném řádku se nacházejí operace stejného jeřábu nebo vany). Stejnou barvou jsou označeny namáčecí operace jednoho produktu, v případě transportních operací ve spodní části reprezentuje jedna barva konkrétní jeřáb.

Nakolik je multiplikátor efektivní u jednodprodukových úloh ukazuje obrázek 6.4, obsahující 3 stejné produkty zahrnující 6 operacích namáčení ($C_{max} = 252$). Multiplikované řešení je znázorněno na obrázku 6.5.

Srovnání výsledků s naivním harmonogramem, kdy jsou produkty optimalizační úlohy zpracovávány postupně, je uvedeno v tabulce 6.3. Na těchto datech je viditelný přínos z hlediska zkrácení celkového času výroby: zde beru v potaz srovnání mezi naivním harmonogramem a aplikací nalezeným řešením, úspora času se u úloh menšího rozsahu pohybuje okolo 30 %.



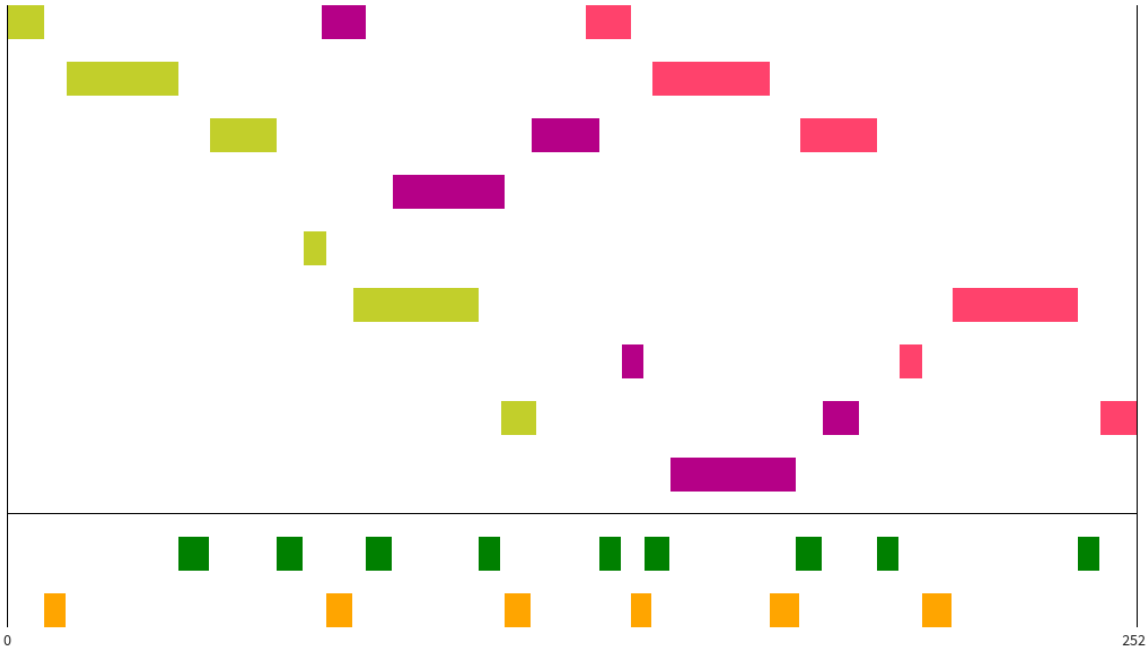
Obrázek 6.3: Ganttův diagram řešení úlohy s 2 stejnými produkty s 6 operacemi a 1 produktem se 4 operacemi namáčení. Výrobní čas $C_{max} = 152$. V horní části se nacházejí operace namáčení, v dolní operace transportu (na stejném řádku se nacházejí operace stejného jeřábu nebo vany). Stejnou barvou jsou označeny namáčecí operace jednoho produktu, v případě transportních operací ve spodní části reprezentuje jedna barva konkrétní jeřáb.

Tabulka 6.3: Srovnání naivního harmonogramu a nalezených výsledků (včetně multiplikovaných) u úloh s třemi stejnými produkty (6 operací) a úloh s dvěma stejnými produkty s 6 operacemi a 1 produktem se 4 operacemi namáčení.

	3 produkty (s)	2 + 1 produkt (s)
Nalezené řešení	252s	154s
Naivní harmonogram	354s	236s
Multiplikované řešení x10	1563s	1105s
Naivní harmonogram s multiplikací x10	3540s	1540s

Následující výsledky byly získány s odlišnými konfiguračními parametry (pravděpodobnost mutace změnou prvku = 0.5, pravděpodobnost mutace změnou pořadí prvku = 0.2, počet měněných jobů při mutaci pořadí prvku = 2). Jedná se o úlohu s 1 produktem o 6 operacích, 2 produkty o 4 operacích a 1 produktem o 5 operacích namáčení (viz obrázek 6.6, zlepšení 45 % oproti naivnímu harmonogramu). Úloha s 2 produkty o 6 operacích a 4 produkty o 4 operacích namáčení (viz obrázek 6.7) dosáhla výrobního času $C_{max} = 313$ a tedy zlepšení zhruba 40 %.

Časy výpočtu jednotlivých jedinců se dají dohledat u konkrétních detailů řešení v aplikaci. Jak lze vypozorovat, pro zpracování všech populací jedinců malých až středních úloh se jedná až o hodiny (ačkoliv některé z řešení je zpravidla nalezeno dříve). U úlohy nejsou



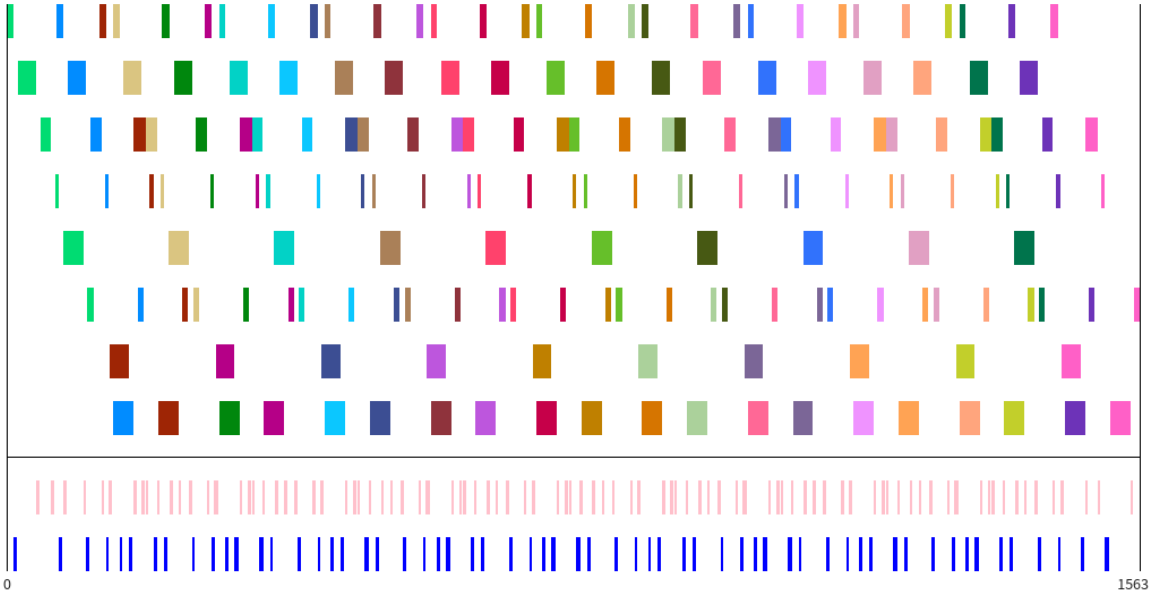
Obrázek 6.4: Ganttův diagram řešení úlohy s 3 stejnými produkty s 6 operacemi namáčení. Výrobní čas $C_{max} = 252$. V horní části se nacházejí operace namáčení, v dolní operace transportu (na stejném řádku se nacházejí operace stejného jeřábu nebo vany). Stejnou barvou jsou označeny namáčecí operace jednoho produktu, v případě transportních operací ve spodní části reprezentuje jedna barva konkrétní jeřáb.

měřeny časy nalezení prvního řešení, ale pouze běh úlohy s konfiguračními parametry, se kterými je nastavena. Vzhledem k tomu, že z časových důvodů je optimalizační část aplikace napsána v jazyce python3, testování probíhalo na úlohách menšího až středního rozsahu (v souhrnném počtu maximálních zhruba 40 uzlů disjunktivního grafu). U úlohy obsahující 8 produktů o 6 operacích namáčení (dohromady 88 uzlů disjunktivního grafu) trvalo vyhodnocení populace přes 15 minut, což ukazuje na nutnost přepsat optimalizační část aplikace do některého nízkoúrovňového jazyka (vhodné by bylo C++). Poté bude možné provádět optimalizace nad úlohami většího rozsahu, a to s předpokládaným mnohonásobným zrychlením. Plnohodnotné srovnání např. s [29] tím pádem není zcela možné - proto zde není uvedeno.

Návrhy na možná vylepšení

Hlavním vylepšením pro rychlejší generování většího množství výsledků je již zmíněné přepsání optimalizačního procesu do C++. Menšími nedostatky jsou generování většího množství rozlišitelných barev pro účely ganttova diagramu (momentálně se barvy berou ze seznamu) a zapamatování si pozic van a konfiguračních parametrů pro každý záznam v historii provedených úloh.

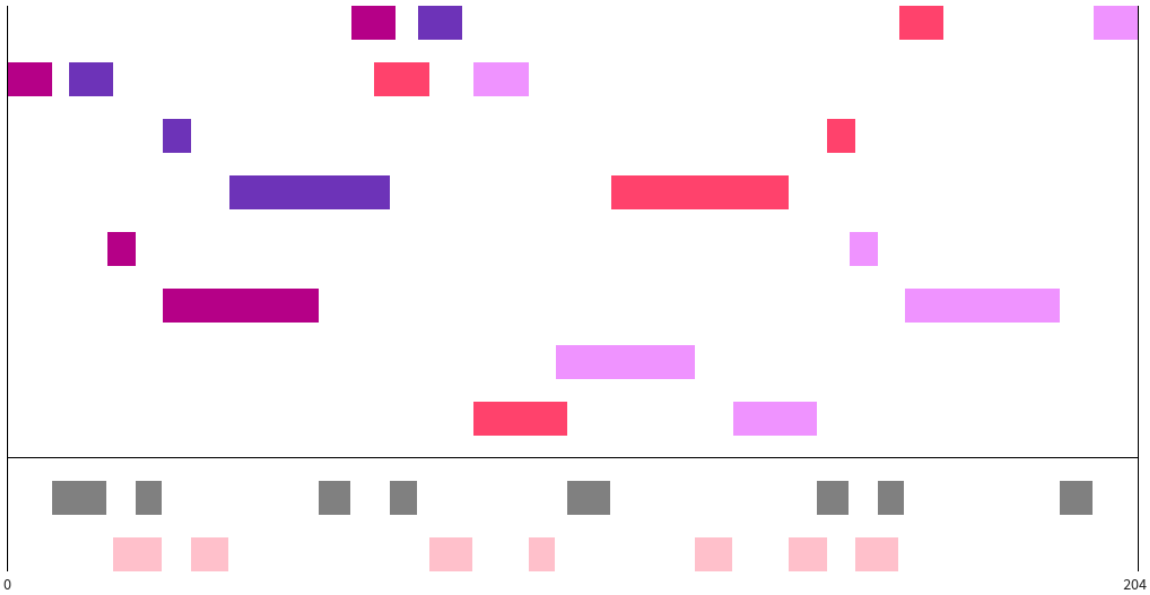
Rozšířením, které může vést ke zvýšení rychlosti nalezení validního řešení, je optimalizace parametrů genetického algoritmu (tento proces je časově a výpočetně náročný - nejprve je třeba provést přepis do C++). Optimalizaci je možné provést tak, že budou automaticky postupně spouštěny optimalizační úlohy s různým zadáním a různou kombinací konfigurač-



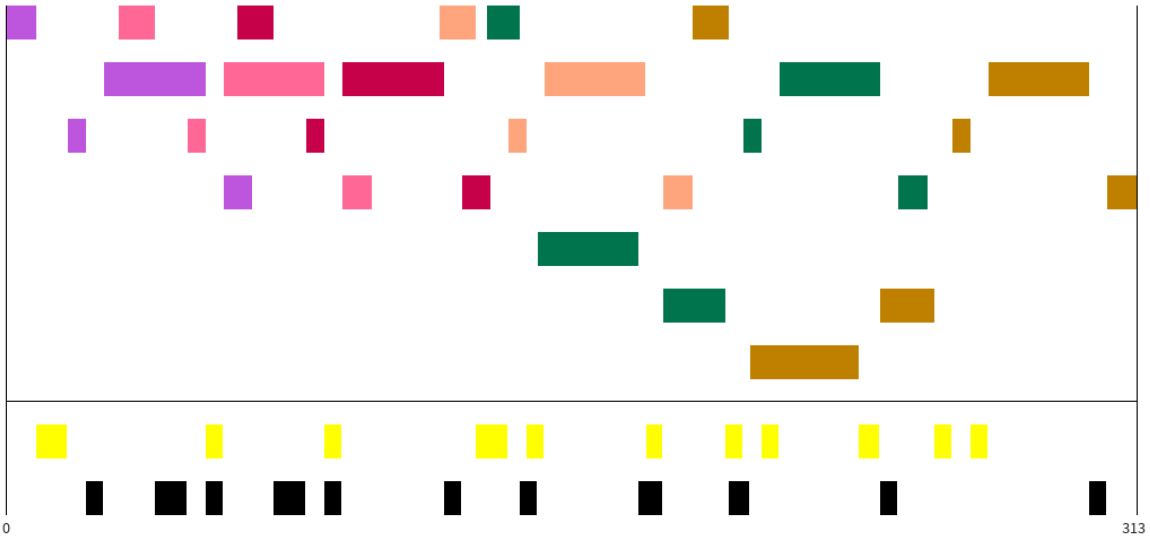
Obrázek 6.5: Použití multiplikátoru 10 na úlohu s 3 stejnými produkty s 6 operacemi namáčení. Výrobní čas $C_{max} = 1105$. V horní části se nacházejí operace namáčení, v dolní operace transportu (na stejném řádku se nacházejí operace stejného jeřábu nebo vany). Stejnou barvou jsou označeny namáčecí operace jednoho produktu, v případě transportních operací ve spodní části reprezentuje jedna barva konkrétní jeřáb.

ních parametrů aplikace. Tyto výsledky budou zaznamenávány a postupně je možné dojít k dobře fungující kombinaci parametrů pro konkrétní typ/velikost zadané úlohy.

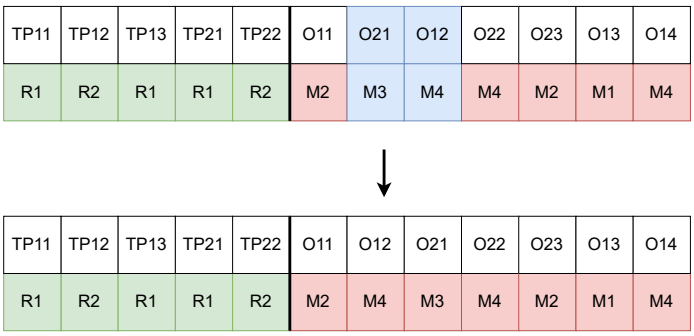
V rámci genetického algoritmu je možným rozšířením otestování nového typu mutačního operátoru, kdy n-krát dojde k náhodnému přehození dvou náhodně zvolených namáčecích nebo transportních operací. Tento postup je znázorněn na obrázku 6.8. Také mám v plánu otestovat novou verzi mutace pořadí operací, která po zvolení ID mutovaného produktu přehodí pořadí pouze některých jeho operací (namáčení nebo transportu).



Obrázek 6.6: Ganttův diagram řešení úlohy s 1 produktem s 6 operacemi, s 2 stejnými produkty s 4 operacemi a jedním produktem s 5 operacemi namáčení. Výrobní čas $C_{max} = 204$. V horní části se nacházejí operace namáčení, v dolní operace transportu (na stejném řádku se nacházejí operace stejného jeřábu nebo vany). Stejnou barvou jsou označeny namáčecí operace jednoho produktu, v případě transportních operací ve spodní části reprezentuje jedna barva konkrétní jeřáb.



Obrázek 6.7: Ganttův diagram řešení úlohy s 2 produkty s 6 operacemi a s 4 stejnými produkty s 4 operacemi namáčení. Výrobní čas $C_{max} = 313$. V horní části se nacházejí operace namáčení, v dolní operace transportu (na stejném řádku se nacházejí operace stejného jeřábu nebo vany). Stejnou barvou jsou označeny namáčecí operace jednoho produktu, v případě transportních operací ve spodní části reprezentuje jedna barva konkrétní jeřáb.



Obrázek 6.8: Navrhovaný alternativní způsob mutace chromozomu: n přehození pořadí dvou operací namáčení nebo transportu.

Kapitola 7

Závěr

Tato práce řeší problém optimalizace výroby u mořicích linek, které pro transport produktů mezi jednotlivými namáčeními ve vanách využívají mostových jeřábů. Těmto transportním jednotkám jsou přiděleny konkrétní operace přesunu produktu, a to tak, aby nedocházelo ke střetu dvou produktů v jedné vaně. Požadavkem bylo seznámit se principem fungování mořicích linek a následně navrhnout a implementovat aplikaci, která seřadí manipulace transportních jednotek za účelem zvýšení výrobní efektivity linky. Dalším požadavkem bylo tuto aplikaci na několika případech otestovat a prokázat tak zvýšení efektivity výroby.

Práce obsahuje teoretický návrh a implementaci heuristiky, která řeší zadaný problém: seřazení manipulací transportní jednotky tak, aby byla dodržena produktová receptura a časy expozic ve vanách s kyselinou, a zároveň byla dostatečně vysoká produkční kapacita. Aplikace je postavena na architektuře klient-server: ke správě parametrů zařízení výrobní linky, nastavení parametrů optimalizační úlohy a správě optimalizačních úloh slouží webové rozhraní. Parametry transportní jednotky byly implementovány jako nastavitelné časové údaje, specifické pro každé jednotlivé transportní zařízení. Stejným způsobem byly vyřešeny konfigurační parametry u jednotlivých van. Aplikace vizualizuje navržené řešení pomocí ganttova grafu a 2D animace. Taktéž vytváří abstraktní textové instrukce určené pro řídící jednotku transportního zařízení (obsahující časový údaj a stanovený pohyb zařízení): tyto instrukce jsou použitelné pro automatizované řízení linky. Optimalizační aplikace byla implementována v jazyce python3 a následně otestována na několika zvolených úlohách menšího rozsahu. Na zvolených úlohách bylo v porovnání s naivními časovými harmonogramy dosaženo zkrácení výrobního času o 30-45 %. Pomocí testování bylo nalezeno úzké hrdlo optimalizace a navrženo řešení (přepsání optimalizačního procesu do jazyka C++).

Kromě zefektivnění optimalizačního procesu se nabízí možnost rozšíření v podobě zavedení vyhýbacích podmínek pro transportní jednotky. Zde velmi záleží na funkcionalitě mořicí linky: zda se všechny jeřáby pohybují nad vanami na jedné kolejnici, zda mají po celé délce kolejnice vyhýbací kolejnice do kterých mohou zajet (aby si nepřekážely v pohybu), zda všechny jeřáby mohou obsluhovat všechny vany apod. Prostým řešením je zavedení přídavných omezení prostoru řešení funkcemi, které kladou další nároky na jedincem reprezentované řešení optimalizační úlohy.

Projekt byl pro mě osobně přínosem, protože mi umožnil nahlédnout do praktických způsobů řešení plánovacích úloh. Samotnou Shifting bottleneck heuristiku (s přepisem do jazyka C++ a v prohledávacím prostoru bez tolika striktních omezení) nyní plánuji využít jako součást mikroslužby aplikace, která bude odpovídat za efektivní plánování událostí.

Literatura

[1] Adams, J., Balas, E. a Zawack, D. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science*. INFORMS. 1988, sv. 34, č. 3, s. 391–401. ISSN 00251909, 15265501.

[2] Amjad, M., Butt, S., Kousar, R., Ahmad, R., Agha, M. et al. Recent Research Trends in Genetic Algorithm Based Flexible Job Shop Scheduling Problems. *Mathematical Problems in Engineering*. Únor 2018, sv. 2018, s. 1–32. DOI: 10.1155/2018/9270802.

[3] Amraoui, A. a Elhafsi, M. An Efficient New Heuristic for the Hoist Scheduling Problem. *Computers and Operation Research*. Říjen 2015, sv. 67. DOI: 10.1016/j.cor.2015.10.006.

[4] Balas, E., Lenstra, J. a Vazacopoulos, A. One Machine Scheduling With Delayed Precedence Constraints. *Report Department of Operations Research Statistics and System Theory*. 1993, sv. 9304, s. 1–34.

[5] Blazewicz, J., Pesch, E. a Sterna, M. Disjunctive graph machine representation of the job shop scheduling problem. *European Journal of Operational Research*. Únor 2000, sv. 127, s. 317–331. DOI: 10.1016/S0377-2217(99)00486-5.

[6] Chaudhry, I. A. a Khan, A. A. A research survey: review of flexible job shop scheduling techniques. *International transactions in operational research*. Květen 2016, sv. 23, s. 551–591. DOI: 10.1111/itor.12199.

[7] Darnell, B. *Tornado - Authentication and Security* [online]. 2019 [cit. 2021-04-24]. Dostupné z: http://web-static.stern.nyu.edu/om/faculty/pinedo/scheduling/hurink/sl3.pdf.

[8] Deb, K., Pratap, A., Agarwal, S. a Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*. 2002, sv. 6, č. 2, s. 182–197. DOI: 10.1109/4235.996017.

[9] Erickson, J. *Algorithms* [online]. [cit. 2019-04-24]. Dostupné z: http://algorithms.wtf.

[10] Garey, M. R., Johnson, D. S. a Sethi, R. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research*. Květen 1976, sv. 1, č. 2, s. 117–129. DOI: 10.1287/moor.1.2.117.

[11] Gawiejnowicz, S. *Models and algorithms of time-dependent scheduling*. Springer, 2020. ISBN 978-3-662-59362-2.

[12] Genova, K., Kirilov, L. a Guliashki, V. A Survey of Solving Approaches for Multiple Objective Flexible Job Shop Scheduling Problems. *Cybernetics and Information Technologies*. Červenec 2015, sv. 15, s. 3–22. DOI: 10.1515/cait-2015-0025.

[13] Hurink, J. *Scheduling Lecture 3* [online]. 2005 [cit. 2021-04-24]. Dostupné z: http://web-static.stern.nyu.edu/om/faculty/pinedo/scheduling/hurink/sl3.pdf.

[14] Hurink, J. a Knust, S. Tabu search algorithms for job-shop problems with a single transport robot. *European Journal of Operational Research*. 2005, sv. 162, č. 1, s. 99–111. DOI: https://doi.org/10.1016/j.ejor.2003.10.034. ISSN 0377-2217. Logistics: From Theory to Application.

[15] Jakubčíková, L. *Povrchové úpravy materiálů na bázi železa*. Brno, CZ, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství. Dostupné z: http://hdl.handle.net/11012/67665.

[16] Kramer, O. *Genetic algorithm essentials*. Cham, Switzerland: Springer, 2017. ISBN 978-3-319-52156-5.

[17] Kuendee, P. a Janjarassuk, U. A comparative study of mixed-integer linear programming and genetic algorithms for solving binary problems. In: *2018 5th International Conference on Industrial Engineering and Applications (ICIEA)*. 2018, s. 284–288. DOI: 10.1109/IEA.2018.8387111.

[18] Lageweg, B. J., Lenstra, J. K. a Kan, A. H. G. R. Job-Shop Scheduling by Implicit Enumeration. *Management Science*. Prosinec 1977, sv. 24, č. 4, s. 441–450. DOI: 10.1287/mnsc.24.4.441.

[19] Nieberg, T. *Scheduling - single machine scheduling* [online]. 2010 [cit. 2021-04-24]. Dostupné z: https://www.or.uni-bonn.de/lectures/ss10/scheduling_data/sched10_2.pdf.

[20] Nouri, H. E., Belkahla Driss, O. a Ghédira, K. *A Classification Schema for the Job Shop Scheduling Problem with Transportation Resources: State-of-the-Art Review*. Leden 2016. 1-11 s. ISBN 978-3-319-33623-7.

[21] Pinedo, M. *Scheduling : theory, algorithms, and systems*. New York: Springer, 2012. ISBN 978-1-4614-1986-0.

[22] rivin), I. R. (https://math.stackexchange.com/users/109865/igor. *Count of permutations with N fixed order elements (chromozomes)* [Mathematics Stack Exchange]. URL:https://math.stackexchange.com/q/4082352 (version: 2021-03-30). Dostupné z: https://math.stackexchange.com/q/4082352.

[23] ryash, H. abu a Tamimi, A. Comparison Studies for Different Shortest path Algorithms. *International Journal of Computers and Applications*. Květen 2015, sv. 14. DOI: 10.24297/ijct.v14i8.1857.

[24] Sotskov, Y. a Shakhlevich, N. NP-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics*. 1995, sv. 59, č. 3, s. 237–266. DOI: https://doi.org/10.1016/0166-218X(95)80004-N. ISSN 0166-218X. Dostupné z: https://www.sciencedirect.com/science/article/pii/0166218X9580004N.

[25] Stein, C. *Branch and bound for Lmax* [online]. 2004 [cit. 2021-04-24]. Dostupné z: http://www.columbia.edu/~cs2035/courses/ieor4405.S04/bb.pdf.

[26] Stojanovic, I., Brajevic, I., Stanimirovic, P., Kazakovtsev, L. a Zdravev, Z. Application of Heuristic and Metaheuristic Algorithms in Solving Constrained Weber Problem with Feasible Region Bounded by Arcs. *Mathematical Problems in Engineering*. Červen 2017, sv. 2017, s. 13. DOI: 10.1155/2017/8306732.

[27] Vrajitoru, D. Large Population or Many Generations for Genetic Algorithms? Implications in Information Retrieval. In: *Soft Computing in Information Retrieval: Techniques and Applications*. Heidelberg: Physica-Verlag HD, 2000, s. 199–222. DOI: 10.1007/978-3-7908-1849-9_9. ISBN 978-3-7908-1849-9.

[28] Váňa, P. *Redukční tavenina - účinná předúprava při moření vysocelegovaných ocelí* [online]. 2017 [cit. 2021-04-24]. Dostupné z: http://www.ekomor.cz/wp-content/uploads/2017/02/%C4%8Cl%C3%A1nek-v-%C4%8Dasopise-Focus-Nerez-2016-Reduk%C4%8Dn%C3%AD-tavenina-min.pdf.

[29] Zhang, Q., Manier, H. a Manier, M. A. A disjunctive graph and shifting bottleneck heuristics for multi hoists scheduling problem. *IFAC Proceedings Volumes*. 2011, sv. 44, č. 1, s. 6963–6968. DOI: https://doi.org/10.3182/20110828-6-IT-1002.03153. ISSN 1474-6670. 18th IFAC World Congress.

[30] Zhang, Q., Manier, H. a Manier, M.-A. A genetic algorithm with tabu search procedure for flexible job shop scheduling with transportation constraints and bounded processing times. *Computers Operations Research*. 2012, sv. 39, č. 7, s. 1713–1723. DOI: https://doi.org/10.1016/j.cor.2011.10.007. ISSN 0305-0548.

[31] Zhang, Q., Manier, H. a Manier, M.-A. A modified shifting bottleneck heuristic and disjunctive graph for job shop scheduling problems with transportation constraints. *International Journal of Production Research*. Taylor Francis. 2014, sv. 52, č. 4, s. 985–1002. DOI: 10.1080/00207543.2013.828164.

Příloha A

Struktura databáze

Tabulka A.1: vany - struktura kolekce.

_id	
sign	označení vany
nazev	název vany
position	pozice vany
exp_min	minimální expozice ve vaně
exp_max	maximální expozice ve vaně
drain	čas okapu nad vanou
dive	čas ponoření produktu do vany

Tabulka A.2: jeraby - struktura kolekce.

_id	
sign	
nazev	
empty_move	čas prázdného přejezdu o jednu pozici
move	čas naloženého přejezdu o jednu pozici
raise_low	čas zdvihnutí produktu do nízké pozice
raise_high	čas zvednutí produktu do vysoké pozice

Tabulka A.3: konfig - struktura kolekce.

_id	
_default	označuje dokument s výchozími konfiguračními parametry
mutace_single_pocet_namaceni	počet operací namáčení, které mají být mutovány při mutaci změnou prvku
mutace_single_pocet_transporty	počet operací transportů, které mají být mutovány při mutaci změnou prvku
mutace_order_pocet	počet jobů, které mají být mutovány při mutaci změnou pořadí
mutace_order	pravděpodobnost mutace změnou pořadí
mutace_single	pravděpodobnost mutace změnou prvku
generace	
populace	
nekonecno	nekonečné generování
cores	počet procesů pro paralelizaci výpočtu fitness hodnoty v populaci

Tabulka A.4: produkty - struktura kolekce.

_id	
sign	
nazev	
kroky	seznam seznamů, které obsahují konkrétní ID alternativních van, ve kterých může být proveden konkrétní krok při výrobě produktu

Tabulka A.5: historie - struktura kolekce.

_id	
nazev	
start	datum zahájení úlohy
end	pokud je úloha řádně dokončena, jedná se o datum dokončení
jeraby	seznam ID jeřábů použitelných pro tuto úlohu
produkty	seznam slovníků obsahujících ID jeřábu, název produktu a počet kusů tohoto produktu
multiply	hodnota, kterou se má ve výsledku násobit počet zpracovávaných produktů této úlohy

Tabulka A.6: users - struktura kolekce.

_id	
email	e-mailová adresa uživatele
salt	salt pro ověření hesla
password	zakódované heslo uživatele

Příloha B

Výchozí hodnoty konfiguračních parametrů aplikace

Tabulka B.1: Výchozí hodnoty konfiguračních parametrů aplikace.

Parametr	Výchozí hodnota
Pravděpodobnost mutace změnou prvku	0,35
Počet měněných operací namáčení při mutaci změnou prvku	3
Počet měněných operací transportu při mutaci změnou prvku	3
Pravděpodobnost mutace změnou pořadí prvku	0,35
Počet měněných jobů při mutaci pořadí prvku	1
Počet generací	100
Populace	50
Počet procesů (CPU cores) pro paralelní ohodnocení jedinců	6
Nekonečné generování	False