F2

Fakulta strojní
Ústav mechaniky, biomechaniky a mechatroniky

Diplomová práce

Optimální řízení toku materiálu v chaotickém skladu s výrobními stroji

Bc. Jan Hrnčíř

14. srpna 2023
Vedoucí práce: Ing. Martin Nečas, MSc., Ph.D.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hrnčíř** Jméno: **Jan** Osobní číslo: **483140**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav mechaniky, biomechaniky a mechatroniky**

Studijní program: **Robotika a výrobní technika**

Specializace: **Robotika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Optimální řízení toku materiálu v chaotickém skladu s výrobními stroji.

Název diplomové práce anglicky:

Optimum control of material flow in a chaotic warehouse with production machines.

Pokyny pro vypracování:

1. Seznamte se s problematikou optimalizačních algoritmů s důrazem na plánování výroby
2. Vytvořte optimalizační matematický model výrobního plánu a řešte jej s využitím vhodného solveru
3. Vytvořte algoritmus pro efektivní vyskladňování a naskladňování skladového materiálu
4. Integrujte optimalizační algoritmus vyskladňování s algoritmem plánování výroby
5. Kriticky zhodnoťe dosasažené výsledy s ohledem na budoucí využití.

Seznam doporučené literatury:

[1] BARTÁK, Roman. Constraint Programming [online]. Praha: Matematicko-fyzikální fakulta, Univerzita Karlova, 1998 [cit. 2023-04-27]. Dostupné z: http://kti.ms.mff.cuni.cz/~bartak/constraints/constrsat.html
[2] BRUCKER, Peter. Scheduling algorithms. 5th. New York: Springer, 2007. ISBN ISBN 978-3-5-0-69515-8. Dostupné také z: https://ftp.idu.ac.id/wpcontent/uploads/ebook/ip/BUKU%20SCHEDULING/Scheduling%20Algorithms.pdf
[3] Google OR-Tools [online]. Mountain View (Kalifornie): Google, 2023 [cit. 2023-04-27]. Dostupné z: https://developers.google.com/optimization/

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Martin Nečas, MSc., Ph.D. odbor mechaniky a mechatroniky FS

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **28.04.2023** Termín odevzdání diplomové práce: **14.08.2023**

Platnost zadání diplomové práce: _____

Ing. Martin Nečas, MSc., Ph.D. podpis vedoucí(ho) práce	prof. Ing. Michael Valášek, DrSc. podpis vedoucí(ho) ústavu/katedry	doc. Ing. Miroslav Španiel, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Poděkování / Prohlášení

Chtěl bych poděkovat celé své rodině za nekonečnou podporu při studiích nejen finanční a za to, že mohu dělat, co mě baví.

V neposlední řadě děkuji Ing. Martinovi Nečasovi, MSc., Ph.D. za vedení mé diplomové práce.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 14. 8. 2023

.....

Abstrakt / Abstract

Diplomová práce se zaobírá řízením pohybu materiálu v automatizovaném chaotickém skladu deskového materiálu. Je rozdělena do tří částí. V první části byla provedena rešerše přístupů k řešení prohledávání stavového prostoru. Na základě těchto principů je vytvořen algoritmus podle požadavků na pohyb materiálu ve skladu. Druhá část obsahuje seznámení s problematikou rozvrhování průmyslové výroby na paralelně pracujících strojích, která je připojena ke skladu. Třetí část pojednává o spojení algoritmu starající se o pohyb materiálu s rozvrhováním výroby paralelních strojů.

Klíčová slova: automatizovaný; chaotický; sklad; deskového; materiálu; ai; umělá inteligence; stavový prostor; prohledávání; LP; ILP; MILP; CP; CSP; programování s omezujícími podmínkami; celočíselné lineární programování; rozvrchování; paralelní; stroje

This master thesis deals with the control of material flow in an automated chaotic panel warehouse. It is divided into three parts. In the first part, research is made about approaches to state-space search. Based on these principles, an algorithm is developed according to the requirements of material flow in the warehouse. The second section describes the problem of industrial production scheduling on parallel machines connected to the warehouse. The third part discusses the integration of the algorithm taking care of material flow with the production scheduling of parallel machines.

Keywords: automated; chaotic, panel; storage; system; warehouse; ai; artificial intelligence; state-space; search; astar; a-star; LP; ILP; MILP; CP; CSP; constraint programmin; integer linear programmin; scheduling; paralel; machines

Title translation: Optimum control of material flow in a chaotic warehouse with production machines

/ Obsah

Tabulky / Obrázky

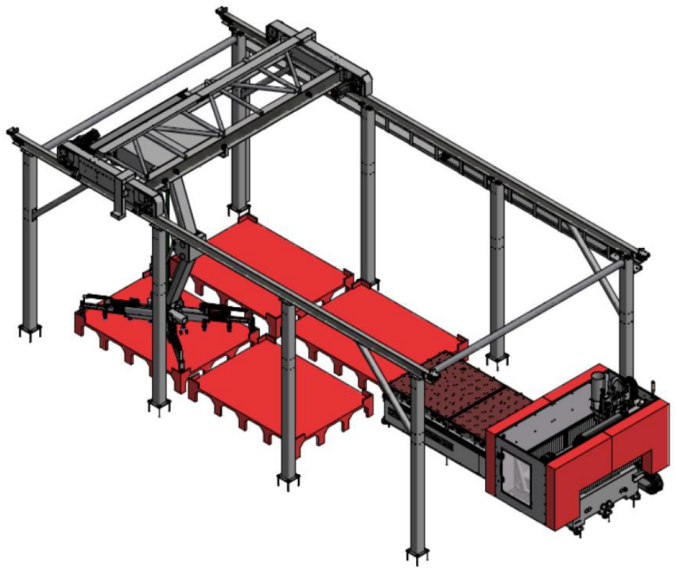
Kapitola 1
Úvod

Tématem práce je řízení pohybu materiálu v automatizovaném chaotickém skladu. Sklad je určen pro dřevozpracující průmysl a jeho součástí jsou CNC obráběcí stroje. S tím souvisí problém rozvrhnutí prací na jednotlivé stroje dle požadovaného denního výrobního plánu. Na jehož základě skladový manipulátor musí být schopen včas zásobit stroje deskami.

Jak už název napovídá, deskový materiál není ve skladu umístěn nijak uspořádaně a jednotlivé desky jsou umístěny na sobě náhodně v několika skladovacích pozicích, například paletách. S tím souvisí problém rychlého a efektivního vyskladnění požadovaného materiálu. Hlavním požadavkem na řízení tedy je, že musí optimalizovat pohyb materiálu ve skladu s ohledem na vyskladnění, výrobu a naskladnění nového materiálu tak, aby požadovaná deska byla dostupná a vyskladněná v co nejkratším čase.

S rozvojem automatizace a robotizace vzniklo několik variant automatických skladů. Příkladem může být ASRS (Automated storage and retrieval system), pro který jsou vymyšleny postupy uskladnění a manipulace se zbožím uvnitř skladu. U tohoto typu skladu roboti nebo manipulátor při pohybu ve skladu může po cestě sebrat nebo uskladnit více zboží, což je hlavní rozdíl od skladu s deskovým materiálem, kde manipulátor může uchopit pouze jednu desku a tu přemístit na jinou pozici. A proto se postupy již vzniklých koncepcí nedají použít. Sklad deskového materiálu je relativně novým typem skladu a přináší tak spoustu výzev a problémů k řešení.

Cílem práce tedy je navržení a implementace algoritmu pro efektivní manipulaci s deskovým materiálem pro konkrétní sklad (obr. 1.1), na jehož vývoji se v rámci projektu podílí Fakulta strojní. Dalším cílem je vytvoření plánování výroby pro několik současně pracujících strojů napojených na automatizovaný sklad.



Obrázek 1.1. Chytrý sklad deskového materiálu od firmy HOUFEK a.s. [1]

Kapitola 2
Algoritmus skladu

Tato kapitola se zaměřuje na vytvoření klíčového algoritmu řídícího systému chaotického skladu pro vyskladňovaní deskového materiálu v daném pořadí metodou prohledávání stavového prostoru. Nejprve je popsána obecná formulace problému a jsou rozebrány postupy pro nalezení optimálního řešení v prohledávání. Dále je formulována reprezentace problému skladu a jeho řešení pomocí A^* algoritmu s vybranou vhodnou heuristikou. Závěr kapitoly pojednává o strategii naskladňování.

2.1 Prohledávání stavového prostoru

Prohledávání je jednou z nejzákladnějších a nejuniverzálnějších metod pro řešení problémů nejen umělé inteligence, ale je i hojně využívané v nejrůznějších oblastech pro řešení reálných problémů.

Robotika. Pro plánování pohybu průmyslových robotů v kloubových souřadnicích s požadavkem na vyhýbání se překážkám musí být provedeno prohledávání v mnoho dimenzionálním prostoru. Dalším příkladem může být navigování mobilních robotů, např. robotické vysavače nebo průzkumní roboti.

VLSI (very large-scale integration) je proces vytváření integrovaných obvodů miliard tranzistorů na jednom čipu (mikroprocesoru nebo paměti). Zde se řeší problém potřeby rozložení miliard komponent a jejich propojení na jednom čipu s cílem milimalizovat velikost, zpoždění v obvodu a maximalizovat výrobu.

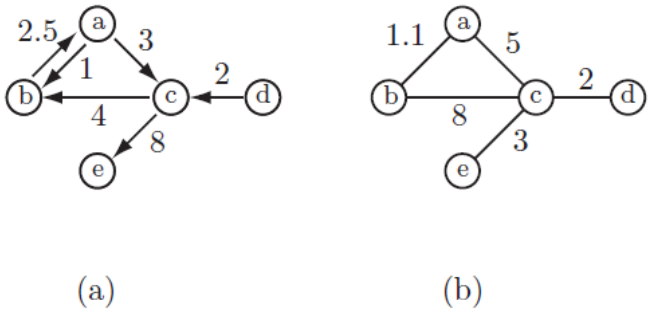
Problém obchodního cestujícího (TSP) je obtížný optimalizační problém navštívení daných měst, každé právě jednou, a vrácením se do výchozího města. Cílem je nalezení nejkratší možné cesty. Přestože patří do skupiny $\mathcal{NP}$*–hard*, bylo vynaloženo veliké úsilí na zlepšení TSP algoritmů.

Další aplikace jsou například hledání cest v GPS navigacích, směrování živého videa v internetu, a mnoho dalších.

Aby prohledávací algoritmy mohli systematicky hledat řešení, je potřeba nejprve vytvořit *formální reprezentaci problému*. Následujících pět komponent formálně definují problém:

- *Stavy* jsou všechny možné situace, které mohou nastat v daném problému. Množinu všech stavů značíme písmenem S. Vezme-li zjednodušený příklad navigace z Prahy do Ostravy, stavy můžou být jednotlivá města. Stavu, ve kterém začíná prohledávání říkáme *iniciální stav* : $V(Praha)$.
- *Akce* popisují množinu všech akcí (značíme A) aplikovatelných v konkrétním stavu. Například ve stavu *V(Praha)* jsou možné akce $A = \{Do(Brno), Do(Plzeň), Do(Pardubice), \dots\}$.
- *Přechody* specifikují co se stane po aplikaci akce $a \in A$ na stav $s \in S$. Pokud je problém deterministický, bude nový stav $s´$ stoprocentně daný akcí a. Ale například v robotice, kde je problém stochastický vlivem nepřesnosti senzorů, působením vnějšího prostředí, atd., může nový stav $s´$ být různý s nějakou pravděpodobností od požadovaného.

- *Test cílového stavu* určuje zda daný stav už je cílový nebo není. Na příkladu navigace se jedná o jednoduchý test, jest-li aktuálně prohledávaný stav je $V(Ostrava)$.
- *Ohodnocení ceny cesty* je celková cena z iniciálního do cílového stavu. Jedná se o součet cen přechodů po cestě. V navigaci mohou být ceny například počet kilometrů z města s do $s´$, doba jízdy nebo kvalita vozovky a podobně.



Obrázek 2.1. Příklad (a) orientovaného a (b) neorientovaného grafu [2]

Stavy, akce a přechody dohromady tvoří *stavový prostor* problému – množinu všech dosažitelných stavů pomocí jakékoli posloupnosti akcí. Stavový prostor je reprezentovaný orientovaným nebo neorientovaným grafem $G = (V, E)$. Kde vrcholy V neboli uzly (nodes) jsou stavy a hrany E (edges) spojující uzly jsou akce.

Řešením problému je sekvence akcí vedoucí z iniciálního stavu do cílového stavu. Kvalitu řešení měříme ohodnocením ceny cesty, přičemž hledáme *optimální řešení*, které má mezi všemi řešeními nejnižší cenu cesty.

Přestože výpočetní výkon dnešních počítačů stále roste, není možné, aby počítač testoval postupně všechny možnosti. Domény problémů bývají často obrovské, v některých případech i nekonečné a cílové stavy mohou být natolik vzdálené, že nemusejí být nalezeny jednoduchými metodami v rozumném čase a s omezenou operační pamětí. Proto vznikla potřeba prohledávat nějak efektivně a byly vytvořeny různé metody na prohledávání stavového prostoru. Tyto metody obvykle dělíme na neinformované (2.1.2) a informované (2.1.3). [2][3]

2.1.1 Měření výkonu metod

Pro potřebu porovnání algoritmů mezi sebou užíváme určitá kritéria. Existují čtyři vlastnosti podle kterých hodnotíme výkon algoritmů.

- *Úplnost* – udává, zda algoritmus garantuje nalezení alespoň jednoho řešení, pokud existuje
- *Optimalita* – nám říká, bude-li nalezené řešení optimální
- *Časová složitost* – čas potřebný k vyřešení problému
- *Prostorová složitost* – dává informaci o tom, kolik bude potřeba operační paměti k řešení problému

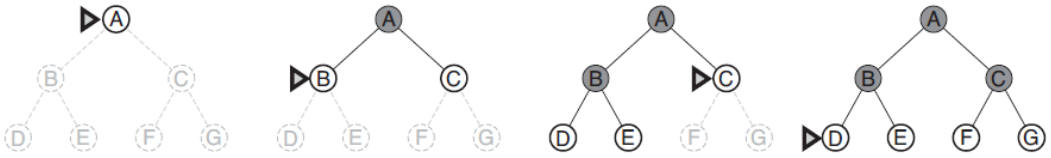
Zatím co úplnost a optimalita udává jakési dobré vlastnosti, které od algoritmu požadujeme, časová a prostorová složitost vypovídá o obtížnosti řešení problému. V teorii computer science typicky obtížnost závisí na velikosti problému, resp. stavového prostoru. Ta je dána součtem $|V| + |E|$, kde $|V|$ je počet stavů a $|E|$ počet akcí mezi

stavy. Problémem je, že při prohledávání máme graf, reprezentující stavový prostor, zadaný implicitně závislý na stavech, akcích a přechodů. Z tohoto důvodu závisí složitost na následujících parametrech: b *branching factor* neboli maximum počtu následujících stavů z jakéhokoli stavu, d *depth* je hloubka nejbližšího cíle, tj. minimální počet akcí provedených z iniciálního stavu do cílového, a m je maximální délka jakékoli cesty v grafu. Protože čas je závislý na konkrétním výpočetním výkonu, časovou složitost měříme počtem prozkoumaných stavů během prohledávání. Prostorová složitost je dána počtem současně uložených stavů v paměti, tím je zaručena nezávislost na platformě a implementaci. [3]

2.1.2 Neinformované prohledávání

Neinformované metody, někdy se také nazývají slepé, nemají žádnou dodatečnou informaci o problému. Mohou pouze procházet systematicky jednotlivé stavy a generovat jejich následovníky. Tyto algoritmy se pouze liší v tom, jak generované následníky procházejí.

- *Breadth-First-Search* (BFS) Prohledávání do šířky je nejjednodušší přistup k řešení. Tento algoritmus prohledává stavový prostor po vrstvách, jak je znázorněno na obr.2.2. Začíná v iniciálním stavu a postupně prochází všechny jeho následníky. Uchovává si frontu nevyřízených stavů a postupně je odebírá. Fronta zajišťuje, že se vždy nejdříve projdou všechny stavy v jedné hloubce než se přejde do nižší hloubky. Tím je také zaručeno nalezení cílového stavu v nejbližší hloubce a kompletnost, resp. optimalita BSF je splněna.



Obrázek 2.2. Příklad běhu BFS algoritmu [3]

Předpokládejme řešení v hloubce d, pak se musí projít b uzlů na první vrstvě, kde každý uzel vygeneruje dalších b uzlů, tj. celkově b^2 uzlů v druhé vrstvě a tak dále. Celkový počet prohledaných uzlů bude tedy roven:

$$b + b^2 + b^3 + \ldots + b^d = O(b^d) \tag{1}$$

Tyto uzly musí být zároveň uloženy v paměti, tedy časová i prostorová složitost roste exponenciálně $O(b^d)$. Ukazuje se, že takto extrémní nárůst paměti je daleko větší problém než časová složitost. Ostatní metody naštěstí nejsou tak náročné na paměť.

Při implementaci podle pseudokódu 2.3 přidáváme uzly nejen do fronty (opened list), ale i do tzv. closed listu, kde si ukládáme již navštívené uzly, aby jsme předešli opětovnému prohledání uzlů v případě cyklu v grafu. [3][4]

```
function BREADTH-FIRST-SEARCH(graph, initial_state)
  opened ← new FIFO queue
  closed ← new empty set
  opened.ADD(initial_state)
  while not opened.IsEmpty() do
    state = opened.POP()
    if state in closed
        continue // state is already visited
    else
        closed.ADD(state)
    end if
    for action in state.actions
        next_state = state.apply(action)
        if next_state is GOAL_STATE return SOLUTION
        else opened.ADD(next_state)
  end while
  return FAILURE // no path found
end function
```

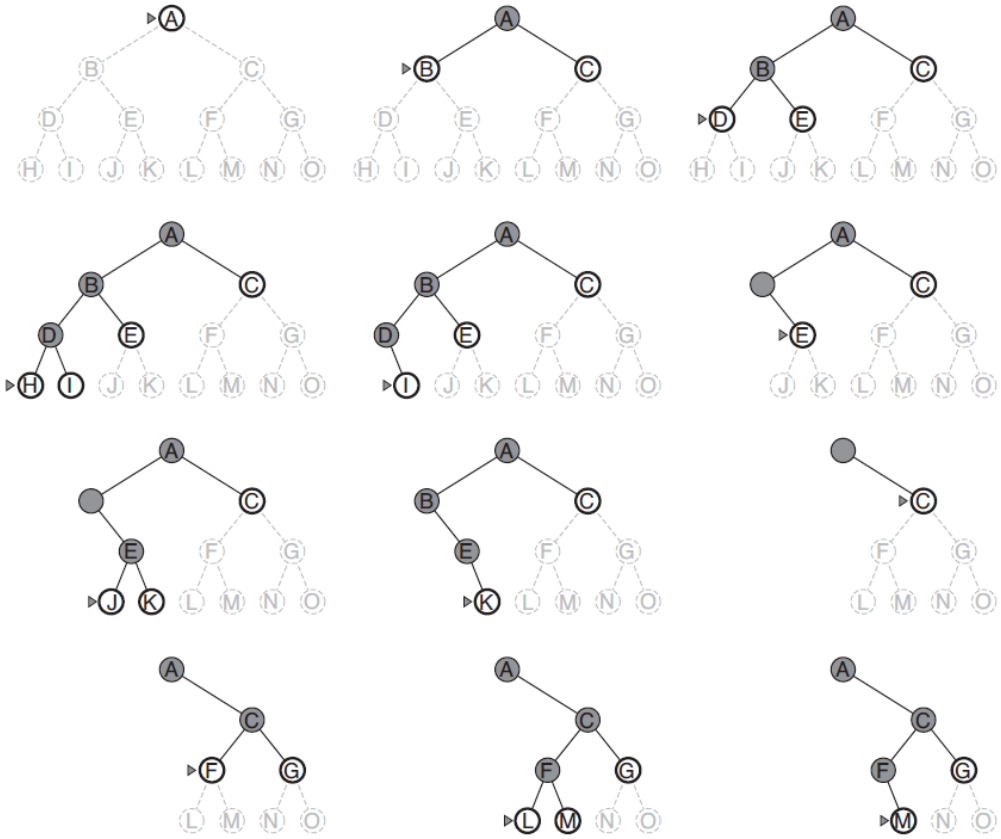
Obrázek 2.3. Pseudokód BFS algoritmu

- *Depth-first search* (DFS) Algoritmus prohledává stavový prostor do hloubky. Začíná v iniciálním stavu a postupně prochází jednotlivé stavy co nejhlouběji. Pokud nenajde cílový stav, vrací se zpět a pokračuje v prohledávání dalších stavů na zásobníku.

  Implementace algoritmu zůstává stejná jako u BFS, jen je potřeba nahradit frontu za zásobník (v pseudokódu 2.3 změníme FIFO queue →LIFO stack). Tím jsme se zbavili největšího problému BFS prostorové složitosti, protože nám stačí pamatovat si pouze stavy na cestě od iniciálního po současný. Již jsme zavedli parametr m, jako maximální délku jakékoliv cesty v již prohledaném prostoru, tedy prostorová složitost DFS je pouze $O(m)$. Touto vlastností je algoritmus vhodný v různých oblastech umělé inteligence, včetně CSP[1].

  Časová složitost je dána velikostí problému $O(b^m)$, přičemž m může být daleko vetší než d a nemusí nalézt optimální řešení, jak je vidět na příkladu 2.4. Kde předpokládáme první nalezené řešení v uzlu H a algoritmus končí, ale druhé předpokládané daleko lepší řešení v uzlu C nebude nalezeno. Nezaručení optimality je hlavní nevýhoda DFS algoritmu. [3][4]

[1] z angl. Constraint Satisfaction Problems



Obrázek 2.4. Příklad běhu DFS algoritmu [3]

- *Iterative deepening depth-first search* (IDDFS) Tento algoritmus kombinuje výhody BFS, jako je úplnost a optimalita, a DFS, zejména pak jeho nízkou prostorovou náročnost $O(d)$. Postupně zvyšuje maximální hloubku prohledávání (limit = 0, 1, 2, ...), dokud nenajde cílový stav. Tímto se podobá DFS a zároveň postupuje po vrstvách, jako je tomu u BFS. Nejprve jsou prohledány všechny stavy v daném limitu než se limit zvýší, tím je zaručena optimalita řešení.

  Časová složitost této strategie se zdá na první pohled neefektivní, protože opakovaně prohledáváme stejné stavy ve vyšších patrech stromu obr. 2.6. Ale při hlubší analýze zjistíme, že asymptotická časová složitost je stejná jako u BFS $O(b^d)$, neboť problém roste exponenciálně neboli nejvíce stavů se nachází v nejnižších patrech stromu a nevadí tedy, že stavy v nejvyšších patrech jsou prohledány více krát.

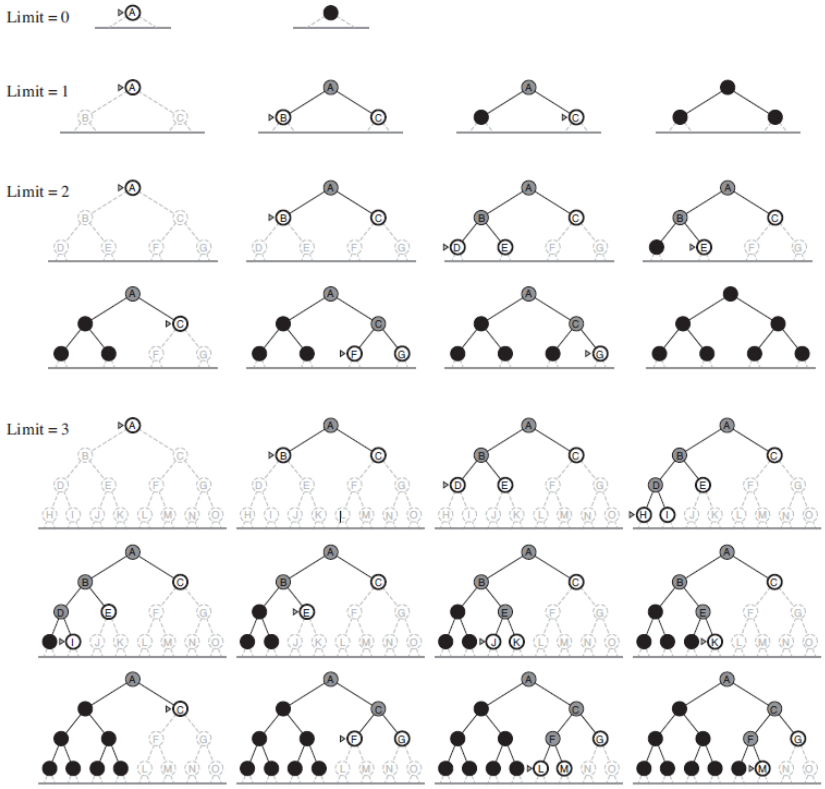
  Obecně je IDDFS algoritmus preferovaný v oblastech, kde je obrovský stavový prostor a hloubka řešení d je neznámá. [3][4]

Kritérium	BFS	DFS	IDDFS
Úplnost	Ano	Ne	Ano
Optimalita	Ano	Ne	Ano
Časová složitost	$O(b^d)$	$O(b^m)$	$O(b^d)$
Prostorová složitost	$O(b^d)$	$O(m)$	$O(d)$

Tabulka 2.1. Porovnání neinformovaných algoritmů [3–4].

```
// Returns true if target is reachable from
// src within max_depth
function IDDFS(src, target, max_depth)
    for limit from 0 to max_depth
      if DLS(src, target, limit) == true
          return true
    return false
end function

function DLS(src, target, limit)
    if (src == target)
        return true;

    // If reached the maximum depth,
    // stop recursing.
    if (limit <= 0)
        return false;

    for adjacent i of src
        if DLS(i, target, limit-1)
            return true

    return false
end function
```

Obrázek 2.5. Pseudokód IDDFS algoritmu [5]



Obrázek 2.6. Příklad běhu IDDFS algoritmu [3]

2.1.3 Informované prohledávání

Informované metody na rozdíl od neinformovaných metod využívají znalosti definovaného problému a tak můžou být mnohem efektivnější. Obecně se tyto metody nazývají *best–first search*, protože prohledávají nejprve stavy s nejnižší hodnotou *ohodnocující funkce* $f(n): S \to \mathbb{R}^+$. Jejich implementace je přímočará jako v případě grafového prohledávaní v BFS 2.3 s tím rozdílem, že postupně prohledávané stavy jsou vkládány do *prioritní fronty* podle ohodnocující funkce.

Výběr ohodnocující funkce je určen strategií konkrétní metody. U většiny metod obsahuje *heuristickou funkci* $h(n)$, která odhaduje nejlevnější cenu cestu od stavu n do cílového stavu. Na příkladu navigace může být heuristická funkce h_L například přímá (letecká) vzdálenost mezi městy.

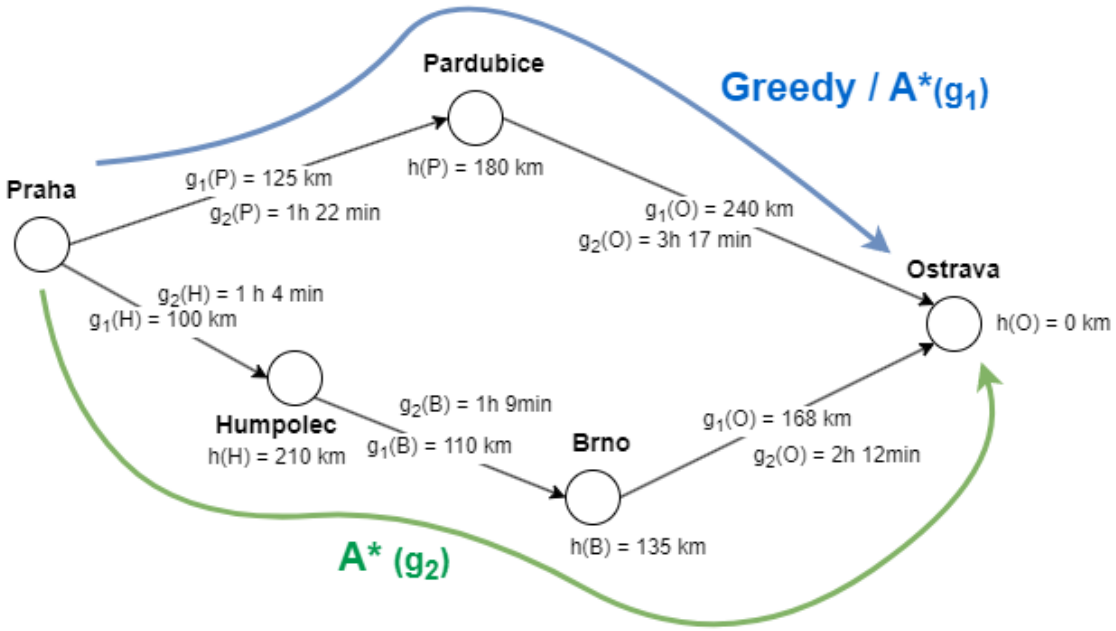
Využití informace z heuristické funkce si ukážeme na dvou algoritmech.

- *Greedy search* nebo také hladové prohledávání. Název vyplývá ze strategie algoritmu, kdy nejdříve prohledává stavy co nejblíže k cíli a předpokládá, že takto se pravděpodobně dostane k cíli co nejrychleji. Stavy jsou v tomto případě ohodnoceny pouze heuristickou funkcí, tedy

$$f(n) = h(n) \tag{2}$$

Tento přístup bohužel nezaručuje optimalitu, pouze hledá nejvíce slibné řešení podle h_L. Což je zřejmé z příkladu na obr.2.7 hledání nejrychlejší cesty z Prahy do Ostravy. V počátečním stavu *V(Praha)* je první prohledaný stav podle h_L *V(Pardubice)*, ve kterém je ihned nalezeno řešení.

V nejhorším případě je časová a prostorová složitost $O(b^m)$, ale na druhou stranu s kvalitní heuristikou se může množství prohledávaných stavů razantně snížit. Tato vlastnost se může hodit v aplikacích s malou pamětí nebo pro obrovské až nekonečné stavové prostory. [3]



Obrázek 2.7. Hledání optimální cesty v navigaci. Praha -> Humpolec -> Brno -> Ostrava celkem 378 km a 3h 25min. Praha -> Pardubice -> Ostrava celkem 365 km a 4h 39min.

- A^* *search* (čteme „A–star“) je nejznámějším algoritmem informovaného prohledávání a přichází s vylepšením ohodnocující funkce, kterou rozšiřuje o informaci o konkretní prohledávané cestě. Při prohledávání jsou vybírány právě následující stavy minimalizující funkci:

$$f(n) = g(n) + h(n) \tag{3}$$

kde $g(n)$ je akumulující cena cesty a $h(n)$ je heuristická funkce. Pro garantování optimálního řešení je kritické zvolení vhodné heuristiky, která musí splňovat dvě vlastnosti.

První z nich je *přípustnost*. Heuristika je přípustná, když nikdy nenadhodnocuje odhad k cílovému stavu. Jinými slovy je vždy optimistická pro všechny stavy ($h(n) \leq h^*(n)\ \forall n \in S$). Pokud je tato vlastnost splněna, A^* je vždy optimální.

Druhá vlastnost je *konzistence*, kterou si můžeme představit jako trojúhelníkovou nerovnost. Pro jakékoli dva stavy n, n' a cenu akce $c(n, n')$ mezi nimi musí platit $h(n) \leq c(n, n') + h(n')$. Platí, že každá konzistentní heuristika je přípustná, a je to silnější vlastnost než přípustnost.

Příkladem takové heuristiky je již zmíněná letecká vzdálenost h_L, která určitě nenadhodnocuje, neboť je řešením zjednodušeného problému navigace. Relaxování neboli zjednodušení problému je jeden z přístupů k nalezení vhodné heuristiky, dále může být přínosné rozdělení problému na podproblémy. Zároveň platí čím více informací heuristika o problému obsahuje, tím méně stavů prohledává. [3][4]

```
function A-STAR-SEARCH(graph, initial_state)
  opened ← new Priority queue
  closed ← new empty set
  opened.ADD(initial_state, COST = 0)
  while not opened.IsEmpty() do
    state = opened.POP()

    if state is GOAL_STATE return SOLUTION PATH
    end if

    if state in closed
      continue // state is already visited
    else
      closed.ADD(state)
    end if

    for action in state.actions
      next_state = state.apply(action)
      next_state.COST += COST_ACTION(action) + HEURISTIC(next_state)
      opened.ADD(next_state, next_state.COST)
    end for

  end while

  return FAILURE // no path found
end function
```

Obrázek 2.8. Pseudokód A–star

Běh A^* si ukážeme na příkladu nalezení nejrychlejší cesty viz obr. 2.7. Stavy jsou řazeny v prioritní frontě podle funkce $f(n) = g_2(n) + h_L(n)$, kde $g_2(n)$ je očekávaná doba jízdy autem mezi úseky v minutách.

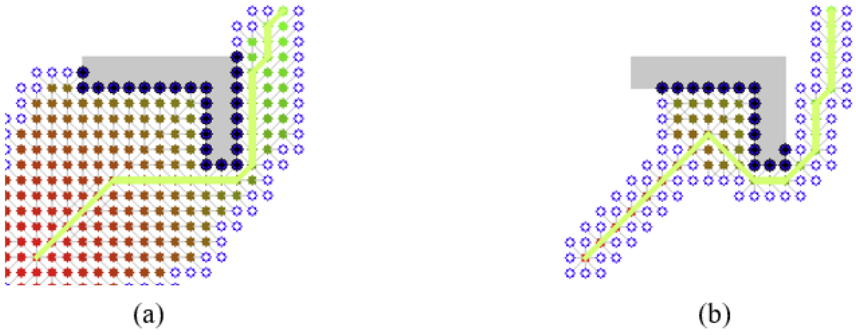
Prohledávaný stav	Prioritní fronta $(g_2(n) + h_L(n))$
Praha	Pardubice(262), Humpolec(274)
Pardubice	Humpolec(274), Ostrava z Pardubic(279)
Humpolec	Brno(268), Ostrava z Pardubic(279)
Brno	Ostrava z Brna(265), Ostrava z Pardubic(279)
Ostrava z Brna	Ostrava z Pardubic(279)
Cíl nalezen	

Tabulka 2.2. Průběh A^* z příkladu navigace 2.7.

Na příkladu je vidět, že přestože cílový stav byl objeven již ve druhém cyklu, algoritmus nekončí jako u Greedy prohledávání, neboť ve frontě jsou slibnější stavy. Teprve při vyjmutí cíle algoritmus končí a je zrekonstruována cesta: Praha -> Humpolec -> Brno -> Ostrava celkem 3h 25min (265min).

Dalším poznatkem je, že funkce $g(n)$ určuje cíl minimalizace. Například při zvolení $g_1(n)$, skutečně ujetá vzdálenost mezi městy, A^* nalezne minimální trasu z Prahy do Ostravy.

- *Částečně optimální* A^*. V některých případech i přípustná heuristika prozkoumává stále mnoho stavů a algoritmus je pomalý. Pro zlepšení výkonu můžeme použít nepřípustnou heuristiku, kterou vyrobíme modifikací z přípustné $h_W(n) = \epsilon \cdot h(n)$ pro nějaké číslo vetší než jedna, $\epsilon > 1$. S touto heuristikou bude A^* preferovat prozkoumávání uzlů blíže k cíli. To může způsobit, že řešení bude nalezeno rychleji, ale na rozdíl od optimální heuristiky nebudeme mít záruku, že nalezené řešení je optimální viz obr. 2.9. [2]



Obrázek 2.9. Porovnání (a) přípustné $h(n)$ a (b) vážené (nepřípustné) heuristiky $h_W(n)$ [6]

2.2 Formální reprezentace problému skladu

Při formální reprezentaci problému skladu je nutné zvolit vhodnou abstraktní reprezentaci, která umožní popsat všechny důležité vlastnosti skladového prostoru a umožní provádění operací, jako je přesouvání desek mezi stohy uvnitř skladu či vstupem a výstupem ze skladu. Dále musí být definované testovací cílové podmínky s ohledem na vyskladňovací, resp. naskladňovací požadavky skladu.

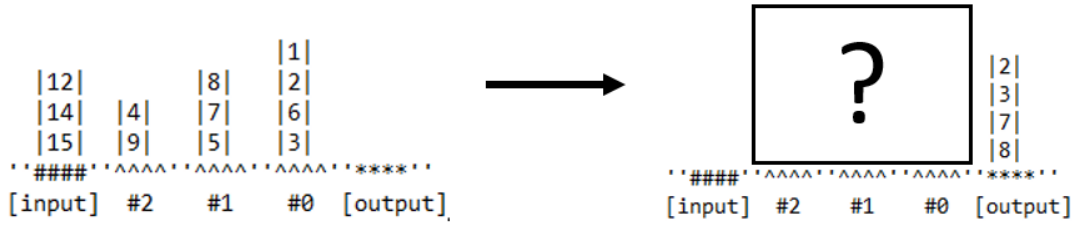
Stav skladu je modelován jako seznam zásobníků. Každý zásobních obsahuje různý počet identifikačních čísel podle aktuálního rozložení desek ve skladu a má také vlastní unikátní identifikační číslo. Výstup, vstup a prázdný zásobník mají záporná identifikační čísla, protože se jedná o speciální pozice ve skladu a jsou tak oddělena od vnitřních pozic pro lepší škálovatelnost skladu.

Akce jsou reprezentovány jako dvojice unikátních čísel zásobníků `(from_id,to_id)` a dávají informaci, ze kterého zásobníku se má deska odebrat a na který zásobník se má deska přesunout.

Součástí modelu musí být také omezující podmínka na maximální výšku stohů, která je dána konstrukčním řešením skladu. V rámci zjednodušení je tato podmínka řešena stanovením maximálního počtu desek, které můžou být vloženy do zásobníku.

Dalším omezením a současně testovací podmínkou je vyskladnění požadované výstupní fronty desek, daná plánováním výroby nebo vytvořena operátorem skladu. Protože sklad je chaotický, není určeno uspořádání desek uvnitř skladového prostoru a předchozí podmínka společně s požadavkem, že na vstupní pozici po naskladnění nesmí být žádná deska, definují cílový stav skladu viz obr. 2.10.

Tímto je reprezentace skladu kompletní a prohledávací algoritmus se snaží najít optimální sekvenci přesunů desek z počátečního stavu do cílového splňující popsané dvě podmínky.



Obrázek 2.10. Definice počátečního a cílového stavu skladu

Aby některý z výše popsaných algoritmů mohl prohledávat stavový prostor skladu muselo být vytvořeno prostředí simulující chování skladu podle omezujících podmínek. Pro tyto účely byla napsána třída `Warehouse`, která obsahuje metody pro manipulaci s deskami a tím může vytvořit jakýkoli stav skladu.

Zde jsou ukázány nejdůležitější metody:

- *Konstruktor třídy* obsahující stav vnitřního uspořádání skladu, vstupní a výstupní zásobník, požadovanou výstupní frontu, příznaky zda se má vyskladňovat, resp. naskladňovat a maximální počet položek, které mohou být umístěny v zásobnících.

```
1  GROUND_ID = -1
2  OUTPUT_ID = -2
3  INPUT_ID = -3
```

```
class Warehouse():
     expanded = 0

     def __init__(self, inicial_state, out_order=None, in_order=None,
                    isInputProccesed = False, isOutputProccesed = True,
                    max_stack_items = 200):
          self.state = inicial_state

          self.input = in_order
          self.output = np.empty(0, dtype=np.int32)

          self.required_order = out_order
          self.isInputProccesed = isInputProccesed
          self.isOutputProccesed = isOutputProccesed
          self.max_stack_items = max_stack_items
```

- *Metoda* `get_moves()` vrací seznam všech možných tahů, které lze provést ve stavu skladu. Tahy jsou generovány na základě pravidel a omezení, které platí pro stavový prostor.

```
def get_moves(self):
    moves = []
    #Pokud je požadované naskladňování a vstupní stoh
    #je neprázdný, přesuň desku na kterýkoli nezaplněný
    #nebo na prázdný stoh.
    if self.isInputProccesed and self.input is not None
    and len(self.input) > 0:
         for to_id, to_stack in enumerate(self.state):
              if to_stack.size == 0
              and ((INPUT_ID, GROUND_ID) not in moves):
                   moves.append((INPUT_ID, GROUND_ID))
              elif to_stack.size > 0
              and to_stack.size < self.max_stack_items:
                   moves.append((INPUT_ID, to_id))
    #Pokud je požadované vyskladňování a přesuň jakoukoli
    #vrchní desku na kterýkoli nezaplněný nebo na prázdný stoh.
    if self.isOutputProccesed:
         for from_id, from_stack in enumerate(self.state):
              if from_stack.size == 0:
                   continue
              object = from_stack[0] # ID desky na vrchu zásobníku
              #Když je vrchní deska následující v požadované výstupní
              #posloupnosti, přesuň ji na výstup.
              if self.output.size < len(self.required_order)
              and object == self.required_order[self.output.size-1]:
                   moves.append((from_id, OUTPUT_ID))
              for to_id, to_stack in enumerate(self.state):
                   if from_id == to_id:
                        continue
                   if len(from_stack) > 1 and to_stack.size == 0
                   and ((from_id, GROUND_ID) not in moves):
```

```
50                          moves.append((from_id, GROUND_ID))
51                          continue
52                    if to_stack.size > 0
53                    and to_stack.size < self.max_stack_items:
54                          moves.append((from_id, to_id))
55          return moves
```

- *Metoda* `move()` provede přesun desky z jednoho zásobníku do jiného zásobníku podle akce `(from_id,to_id)` a tím aktualizuje stav skladu.

```
56  def move(self, move):
57      from_id, to_id = move
58      if from_id == to_id:
59          print("!ERROR:␣invalid␣move␣(from_id␣==␣to_id)!")
60          return
61      if (from_id == GROUND_ID or from_id == OUTPUT_ID
62          or from_id < INPUT_ID or from_id >= len(self.state)) \
63          or (to_id < OUTPUT_ID or to_id >= len(self.state)):
64          print("!ERROR:␣invalid␣move␣outside␣of␣warehouse!")
65          return
66      #vezme ID desku na vrchu zásobníku from_id
67      object_to_move = None
68      if from_id == INPUT_ID:
69          object_to_move = self.input[0]
70          self.input = np.delete(self.input, 0)
71      else:
72          object_to_move = self.state[from_id][0]
73          self.state[from_id] = np.delete(self.state[from_id], 0)
74      #vloží ID desky na vrch zásobníku to_id
75      if to_id == OUTPUT_ID:
76          self.output = np.insert(self.output, 0, object_to_move)
77      elif to_id == GROUND_ID:
78          to_stack, to_stack_id = _find_empty_stack(self.state)
79          self.state[to_stack_id]=np.insert(to_stack, 0, object_to_move)
80      else:
81          to_stack = self.state[to_id]
82          self.state[to_id] = np.insert(to_stack, 0, object_to_move)
```

- *Metoda* `get_neighbours()` vrací všechny dvojice akcí a stavů dosažitelných z aktuálního stavu. Metoda je použita při prohledávání k exploraci stavového prostoru a zvětšuje se zde proměnná `expanded` pro statistiku počtu prozkoumaných stavů.

```
83  def get_neighbors(self):
84      Warehouse.expanded += 1
85      neighbors = []
86      for a in self.get_moves():
87          new_state = self.clone()
88          new_state.move(a)
89          neighbors.append((a,new_state))
90
91      return neighbors
```

- *Metoda* `isDone()` testuje nalezení cílového stavu podle podmínek správné výstupní fronty `isGoalOutput()` a prázdné vstupní pozice `isGoalInput()`. Obě podmínky se dají omezit příznaky podle potřeby skladového systému.

```
92  def isGoalOutput(self):
93       """ výstupní pořadí musí být shodné s požadovaným """
94       return self.get_curr_output() == self.get_goal_output()
95
96  def isGoalInput(self):
97       """ vstupní pozice skladu musí být prázdná """
98       return len(self.input)==0
99
100 def isDone(self):
101      if self.isInputProccesed and self.isOutputProccesed:
102           return self.isGoalInput() and self.isGoalOutput()
103      elif self.isOutputProccesed:
104           return self.isGoalOutput()
105      elif self.isInputProccesed:
106           return self.isGoalInput()
107      else:
108           return False
```

Třída také obsahuje další pomocné funkce pro testování. Funkce `get_random_orders(počet desek, délka posloupnosti)` generuje náhodné pořadí desek pro vstupní pozici nebo požadovanou výstupní posloupnost. Funkce `get_random_state(počet desek, počet stohů)` generuje náhodný počáteční stav vnitřního uspořádání skladu. A nakonec funkce `_find_empty_stack()`, která nalezne první prázdný stoh ve skladu.

Správnou funkčnost třídy lze ověřit v nekonečné smyčce, kde uživatel vybírá z vygenerovaných validních akcí a snaží se splnit cílové podmínky.

```
State (Legend: GROUND_ID=-1, OUTPUT_ID=-2, INTPUT_ID=-3):

                        |7|
  |10|  |4|              |5|
  |11|  |6|        |1|   |3|
  |12|  |8|        |2|   |9|
''####''^^^^''^^^^''^^^^''^^^^''****''
[input]  #3    #2    #1    #0  [output]
Required output order: (4, 5, 1, 8) Work is Done: False
moves = [(-3, 0), (-3, 1), (-3, -1), (-3, 3), (0, 1), (0, -1), (0, 3),
<from> <to>: 1 3
```

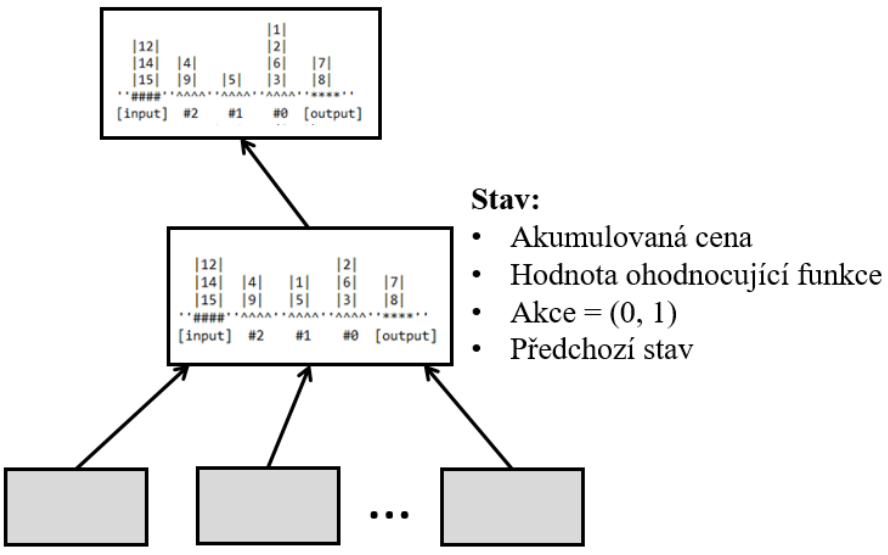
Obrázek 2.11. Simulátor skladu

2.3 Výběr algoritmu a jeho implementace

Problém chaotického skladu má obrovský stavový prostor. Již při sto deskách v jednom stohu je $100! = 9.3 \cdot 10^{157}$ možností, jak desky ve stohu uspořádat. Právě sto desek v každém stohu se dá očekávat v zadané konfiguraci skladu se čtyřmi skladovými pozicemi, pro který je algoritmus vytvářen. Proto byl vybrán pro řešení problému vytažení žádaných desek A–star algoritmus, který vzhledem na velikost problému má výhodné vlastnosti a zvláště v kombinaci se silnou heuristikou může efektivně prohledávat stavový prostor bez nutnosti v tomto případě nemožného prozkoumání všech stavů, což přináší úsporu výpočetního času.

Jeho implementace je přímočará podle pseudokódu 2.8. Jen je potřeba zadefinovat datovou strukturu stavu, která uchovává informaci o aktuálním uspořádání skladu, akumulovanou cenu cesty, hodnotu ohodnocující funkce, podle kterého se řadí stavy v prioritní frontě, a konečně historii (již zmíněnou dvojici akce a stav) pro rekonstrukci nalezené sekvence akcí.

```
1  class State():
2     def __init__(self, warehouse, cost = 0, priority = 0,
3      history = (None, None)):
4          # současné uspořádání skladu
5        self.warehouse = warehouse
6          # akumulovaná cena cesty g(n)
7        self.cost = cost
8          # hodnota ohodnocující funkce f(n) = g(n) + h(n)
9        self.priority = priority
10         # dvojice (akce, predchozí stav)
11       self.history = history
12    # přepsání "<" operátoru, potřebné pro prioritní frontu
13    def __lt__(self, other):
14       return self.priority < other.priority
```



Obrázek 2.12. Datová struktura stavu skladu

Cílem prohledávaní je minimalizace počtu přesunů desek při vyskladňovaní, proto akumulovaná hodnota nových stavů je zvýšena vždy o jedna. Minimalizace přesunů je dostačující při menší počtu skladových pozic, protože rozdíly časů přejezdů manipulátoru mezi jednotlivými pozicemi jsou zanedbatelné. Pro větší skladový prostor by bylo vhodné rozšířit simulátor skladu o informaci umístění skladových pozic a vzdáleností mezi nimi, čímž by se mohl minimalizovat čas přejezdů díky upřednostnění bližších skladových pozic.

Dále do implementace je přidán parametr váhy `weigth` pro možné použití částečně optimálního A^*.

```
15 class AStar():
16    def __init__(self, weigth=1):
17       self.weigth = weigth
18
19    def search(self, start):
20       """ Vrátí nalezenou sekvenci akcí přesunů
21         z počátečního do cílového stavu"""
22       opened = PriorityQueue()
23       closed = {}
24       state = State(start.clone())
25       opened.put(state)
26
27       while not opened.empty():
28          state = opened.get()
29          action, prev_state = state.history
30
31          if state.warehouse.isDone():
32             #vrácení nalezené cesty
33             path = [action]
34             while prev_state is not None:
35                action, prev_state = closed[prev_state]
36                if action is not None:
37                   path.append(action)
38             return list(reversed(path))
39
40          if state.warehouse in closed:
41             continue
42          else:
43             closed[state.warehouse] = (action, prev_state)
44
45          for action, neighbor in state.warehouse.get_neighbors():
46          # vytvoření následujícího stavu a inkrementace jeho ceny
47             next_state = State(neighbor, state.cost + 1, 0,
48                                (action, state.warehouse))
49             next_state.priority = next_state.cost
50                         + self.weigth * next_state.warehouse.heuristic()
51             opened.put(next_state)
52
53       return None
```

2.3.1 Heuristiky

Jak už bylo řečeno, největší síla A^* algoritmu je v jeho heuristice. Nicméně důležité je si uvědomit, že navržená cesta nemusí být skutečným globálním minimem. To znamená, že existuje možnost, že existuje jiná cesta, která by mohla být ještě optimálnější, ale algoritmus ji nepostřehne, protože některé části grafu nebyly prozkoumány. Z toho důvodu je důležité vybrat vhodnou heuristiku pro daný problém podle zmíněných podmínek optimality (přípustnosti a konzistence), aby bylo dosaženo co nejlepších výsledků.

Při hledání heuristiky pro vyskladnění požadovaného pořadí desek uvažujme zjednodušený problém, kde by bylo ve skladu možné přesunout nepotřebné desky na stohy mimo sklad místo na stohy uvnitř skladu, kde by mohly překážet vyskladnění dalších desek v pořadí. S touto myšlenkou byly zformulovány následující tři heuristiky.

1) První heuristika nalezne pro každou desku d z množiny desek D, které zbývá ještě vyskladnit pro splnění požadované výstupní fronty, nejvyšší pozici desky stejného typu a sečte počet desek nad ní neboli zaházení desky d.

$$h_1(n) = \sum_{d \in D} zahazeni(d) + |D| \tag{4}$$

2) Druhá heuristika na rozdíl od první, protože ve výstupní frontě může být více desek stejného typu, pro každý unikátní typ desky z výstupní fronty nalezne n nejlepších pozic a sečte jejich zaházení.

$$h_2(n) = \sum_{d \in U \subseteq D} \sum_{i=1}^{n} zahazeni(d) + |D| \tag{5}$$

   Kde $U \subseteq D$ je podmnožina unikátních desek v požadované výstupní frontě.

3) Třetí heuristika řeší problém, při kterém dochází k překryvu zaházení a některé desky jsou započítány vícekrát, způsobený umístěním požadovaných desek ve stejném stohu. Heuristika je součtem nejhoršího zaházení pro každý stoh a tím tvoří ideální dolní odhad počtu přesunů desek při vyskladňovaní.

$$h_3(n) = \sum_{s \in S} \max_{d \in D} zahazeni(d, s) + |D| \tag{6}$$

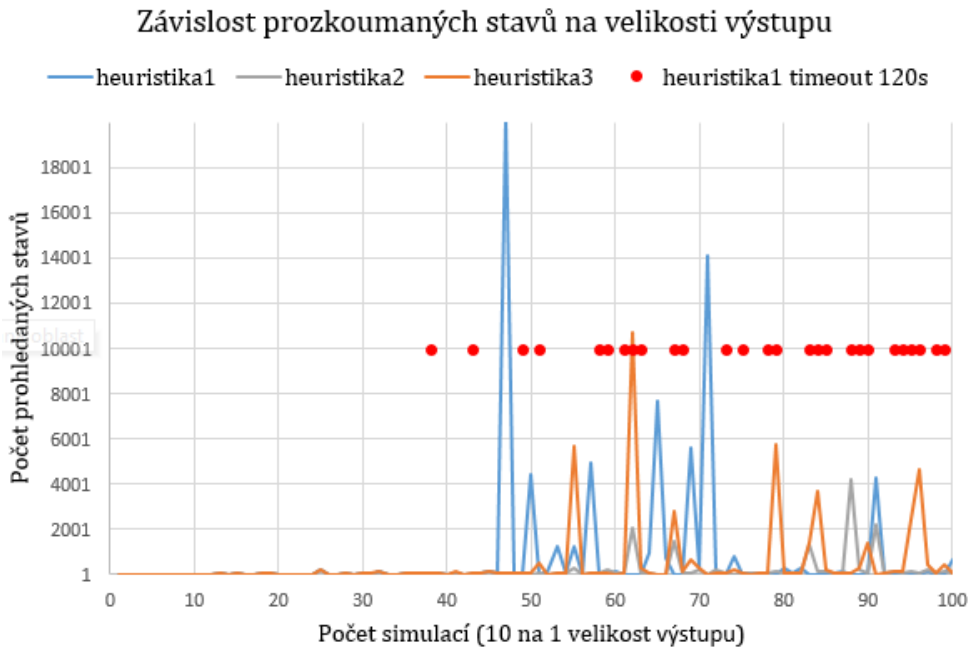
Ke každé heuristice se připočte zbývající počet vyskladňovaných desek $|D|$ a informace o počtu přesunů v aktuálním stavu je kompletní.

Závěr kapitoly se bude zabývat porovnáním vlastností heuristik v simulacích náhodného celkového uspořádání skladu a zjištění účinnosti algoritmu A* při uskladnění různého počtu typů desek a omezením výšky stohů.

2.3.2 Porovnání heuristik

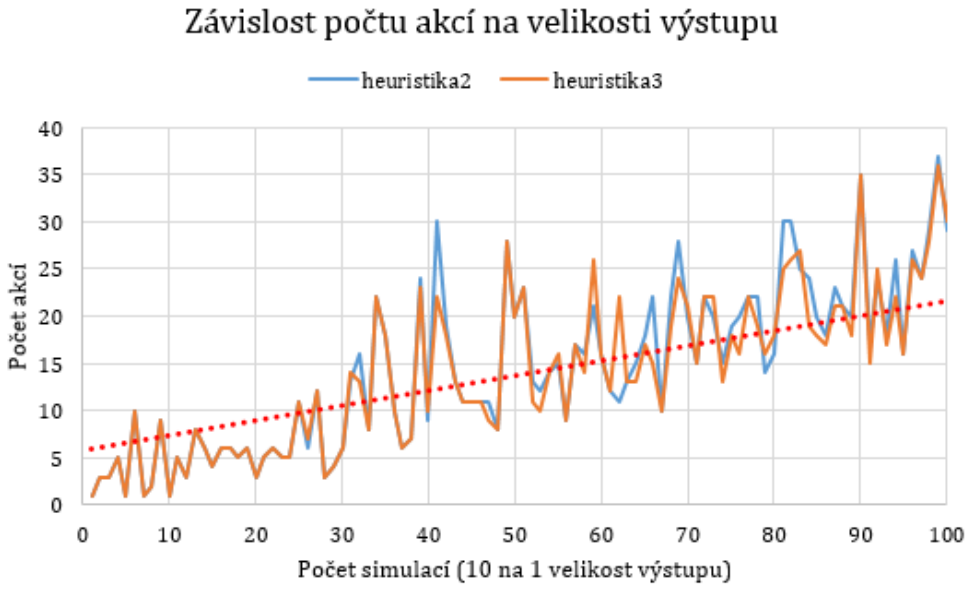
Testování probíhalo za následujících podmínek: Sklad obsahoval celkem 400 desek, rozdělených do 4 stohů. Maximální výška každé pozice byla omezena na 200 desek a první simulace začínaly na 10 typech desek.

Nejprve bylo provedeno testovaní všech tří heuristik při rostoucí velikosti výstupní fronty, z které vyplynulo, že první navržená heuristika není vhodná. Již při velikosti fronty 4 začalo docházet k prohledávaní velkého počtu stavů a čím dál častěji nebylo možné nalézt řešení do dvou minut, kterému docházelo zejména při více desek stejného typu ve frontě. Neúspěšnost první heuristiky je způsobena její špatnou informovaností, kdy se uvažují pouze nejlépe umístěné desky bez ohledu na požadovanou frontu.



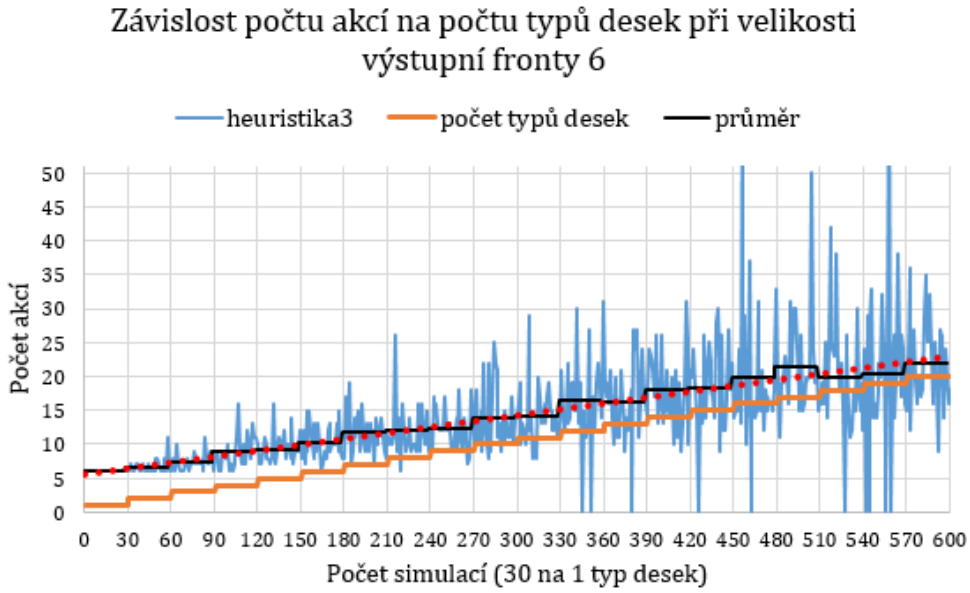
Obrázek 2.13. Závislost prozkoumaných stavů na velikosti výstupní fronty

Při vykreslení závislosti počtu přesunů nalezeného řešení na velikosti výstupní fronty pro zbylé dvě heuristiky pozorujeme lineární závislost, kdy počet přesunů potřebných k vyskladnění roste dvojnásobně na velikosti fronty. Dále od velikosti fronty šest výrazně narůstá prostorová náročnost viz 2.13. Z grafů si můžeme všimnou, že druhá heuristika prohledává méně stavů než třetí, ale zároveň její řešení vyžaduje více přesunů. To je dáno tím, že druhá heuristika stále nadhodnocuje řešení a není přípustná.

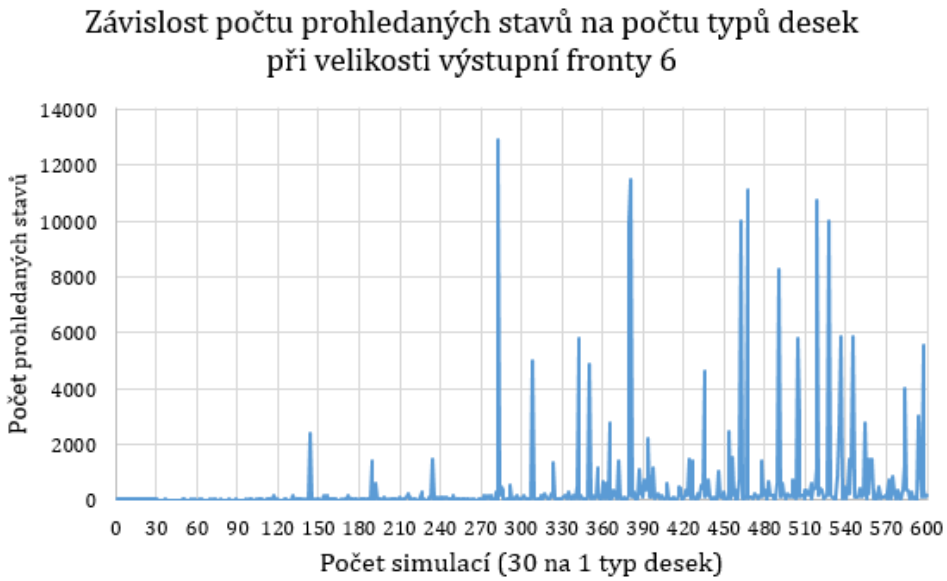


Obrázek 2.14. Závislost počtu akcí na velikosti výstupní fronty

Pro další simulace už byla použita jen třetí jediná přípustná heuristika. Byla zjišťována efektivita a náročnost řešení při rostoucím počtu typů desek. Ze závislosti 2.15 je patrná přímá úměrnost mezi počtem typů desek a přesunů. Z této závislosti můžeme usoudit, že deset typů desek je optimálních pro danou konfiguraci skladu a efektivita přesunů se rovná 50%, tj. na požadovanou délku fronty je potřeba dvojnásobek přesunů. S větším počtem typů desek efektivita přesunů klesá a roste také prostorová složitost a tím i časová náročnost.

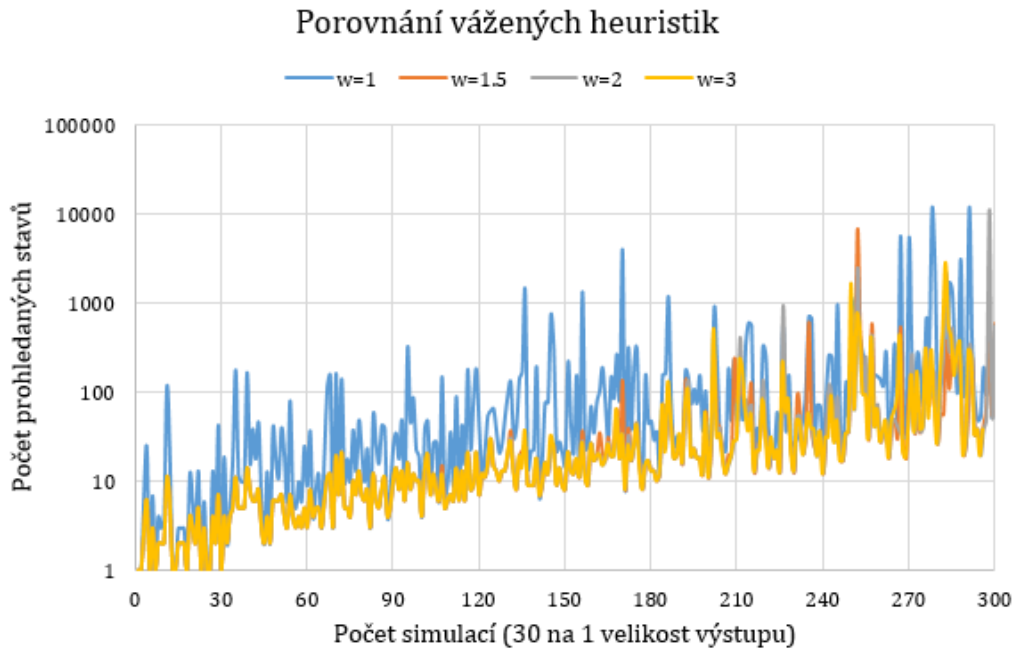


Obrázek 2.15. Závislost počtu akcí při rostoucím počtu typů desek



Obrázek 2.16. Závislost prozkoumaných stavů při rostoucím počtu typů desek

Vzhledem k rychle rostoucí prostorové složitosti a potřebou rychlého plánování v řídícím systému skladu bylo otestováno použití částečně optimálního A^*. Pří vážení heuristické funkce konstantou 1.5, 2 a 3 došlo k řádovému zlepšení prohledávaní, přičemž vážení nemělo skoro žádný vliv na optimalitu nalezeného řešení. V porovnání s neváženou heuristikou byla nalezená řešení větší jen o jednotky přesunů, průměrně o dva přesuny a v 53% byly shodné s neváženou heuristikou.

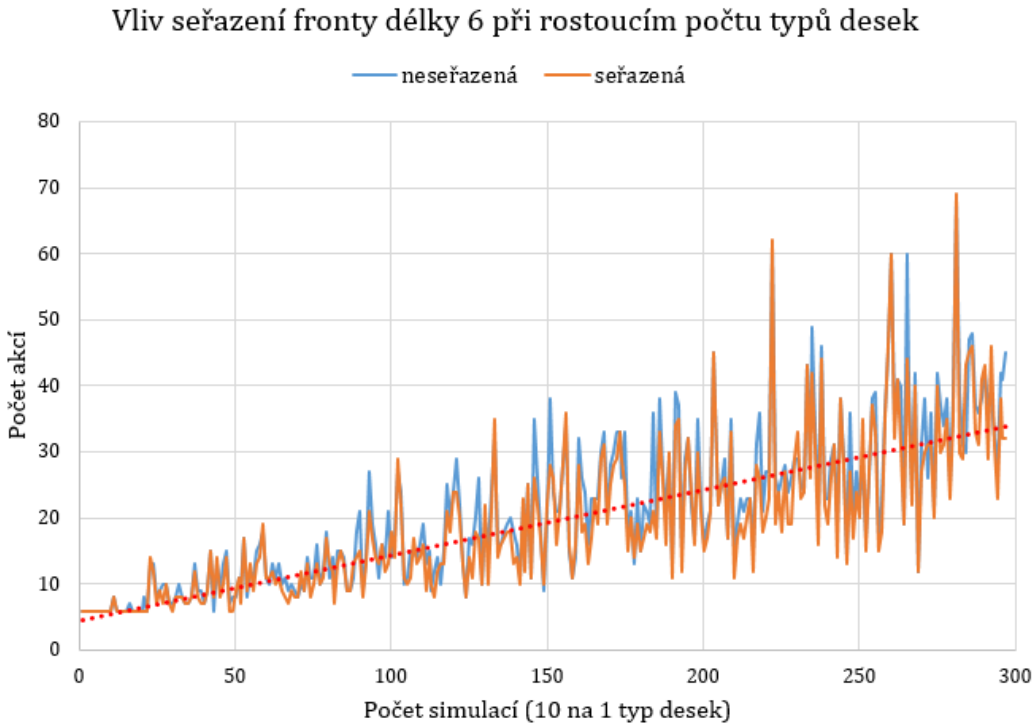


Obrázek 2.17. Porovnání vážených heuristik

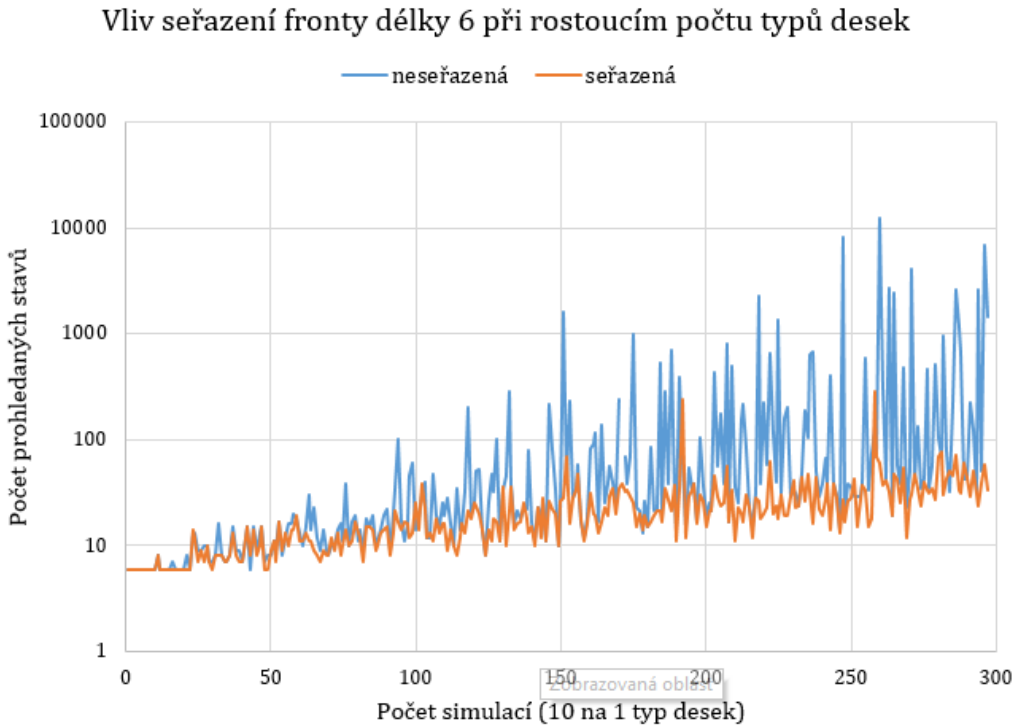
Pokud by výstupní fronta nebyla striktně navázána na plánovanou CNC výrobu, intuice nám napovídá seřadit frontu postupně podle toho jak jsou jednotlivé desky ve skladu zaházené, což by mohlo vést ke zlepšení celého procesu vyskladnění. S touto myšlenkou byly provedeny další dvě simulace.

Nejprve byl zkoumán vliv seřazené fronty konstantní délky šest při rostoucím počtu typů desek ve skladu. Přestože nedošlo k očekávanému zlepšení u počtu akcí (požadovaných přesunů pro vyskladnění), došlo k výraznému zlepšení u počtu prozkoumaných stavů viz obr. 2.18 a obr. 2.19. Počet akcí u neseřazené i seřazené fronty byl více méně stejný a rozdíly v řešení byly zanedbatelné. Co se týče počtu prozkoumaných stavů, zde se jedná o dvanácti násobné zlepšení oproti neseřazené frontě. Pro představu délky trvání výpočtu na notebooku s procesorem Intel i5 7300HQ 2.50GHz je průměrné zlepšení z 1.65 na 0.13 sekund.

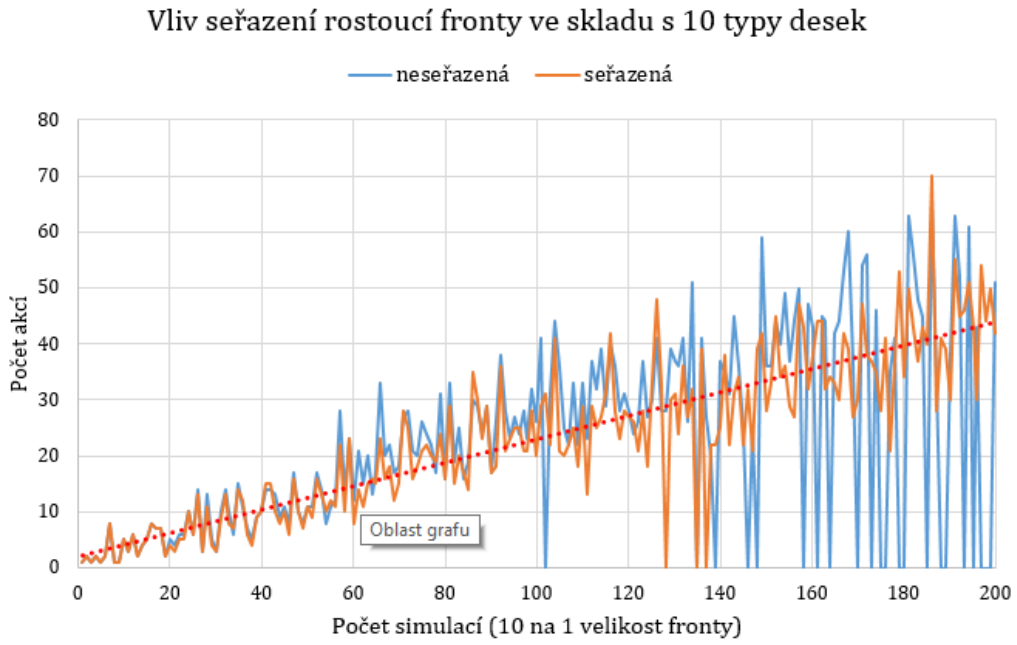
Dále byl zjištěn vliv seřazené rostoucí fronty ve skladu s deseti typy desek viz obr. 2.20 a obr. 2.21. Situace s počtem akcí je obdobná jako v předchozím případě, avšak od velikosti fronty dvanáct dochází čím dál častěji u neseřazené fronty k nevyřešení z důvodu vypršení časového limitu 200 sekund (nulové hodnoty v grafu 2.20). K tomuto nedochází u seřazené fronty díky čtyřnásobnému zlepšení prostorové složitosti a v časech řešení je to zrychlení z průměrných 12.34 na 2.97 sekund. Z tohoto pozorování může být seřazení doporučeno pro vyskladňování většího počtu desek.



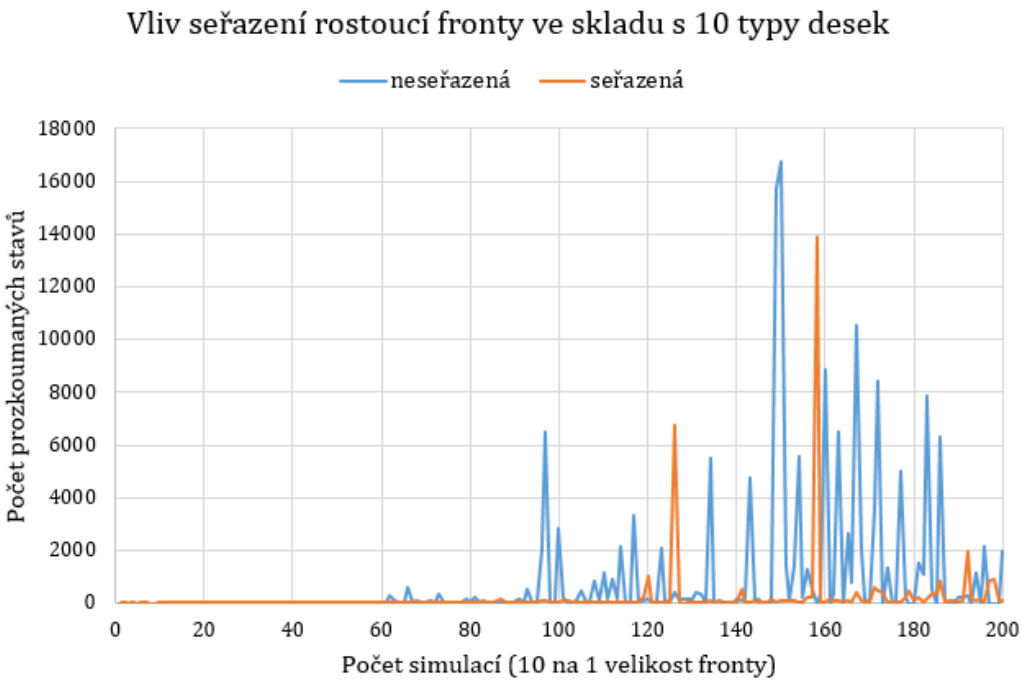
Obrázek 2.18. Vliv seřazené konstantní délky fronty na počet akcí



Obrázek 2.19. Vliv seřazené konstantní délky fronty na počet prozkoumaných stavů



Obrázek 2.20. Vliv seřazené rostoucí délky fronty na počet akcí



Obrázek 2.21. Vliv seřazené rostoucí délky fronty na počet prozkoumaných stavů

2.4 Naskladňovací strategie

Aby mohl být algoritmus skladu kompletní je potřeba ještě rozebrat naskladňovací strategii. Při porovnávání heuristik se ukázalo opravdu výhodné úplné chaotické uspořádání desek ve skladu, proto by mělo být při naskladňování cíleno k udržení této náhodnosti v uspořádaní neboli chaosu. Toho můžeme docílit několika způsoby.

1) Při zavážení vstupní pozice skladu by bylo vhodné (nikoli nutné), aby byly desky předem náhodně promíchány a manipulátor by poté mohl snadno odebírat desky ze vstupní pozice a umísťovat je na náhodně vybranou skladovou pozici.
2) Vytvořením více vstupních pozic, na které by se zavezly desky různých typů, a manipulátor by poté vybíral náhodné dvojice vstupní a skladové pozice.
3) Vnitřním řídicím systémem, který by sledoval vnitřní uspořádání skladu a chytře by udržoval náhodné vnitřní uspořádání vzhledem k požadavku naskladnění určitého typu desky.

Při současném požadavku naskladňování i vyskladňování můžeme použít A^* s rozšířenou heuristikou o počet desek na vstupní pozici. Heuristika zaručí naskladňování desek na pozice ve skladu tak, aby co nejméně překážely vyskladňovaní. Navíc vhodným nastavením ohodnocení přesunů (akcí) ze vstupní pozice do skladu, přesunů mezi vnitřními skladovými pozicemi a ze skladu na výstupní pozici můžeme docílit prioritizace naskladňování nebo vyskladňování.

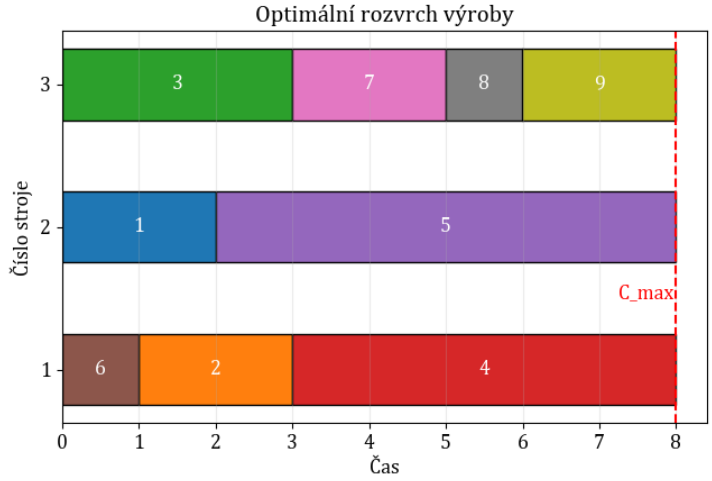
Kapitola 3
Rozvrhování výroby

Rozvrhování výroby může mít výrazný vliv na náklady podniku, proto se tato kapitola zaměřuje na rozvrhování operací s deskovým materiálem na přidružené výrobní stroje chaotického skladu (pila, fréza apod.). Cílem rozvrhování je optimalizovat časové rozvržení pracovních operací, aby byly maximálně využity produkční kapacity a zvýšila se produktivita. Na začátek kapitoly je uvedena terminologie rozvrhování a přístupy k jejímu řešení, poté bude formulován matematický model pro rozvrhování na paralelní stroje a jeho řešení ve vybraném řešiči.

3.1 Terminologie

Rozvrhovací problém se skládá z množiny strojů $\mathcal{M} = \{1, 2, \ldots, m\}$, ke kterým je potřeba přiřadit v čase operace (úkoly) z množiny $\mathcal{T} = \{1, 2, \ldots, n\}$. Operace můžou mít různou délku trvání (*processing time*), požadavek, kdy nejdříve mohou být zpracovány tzv. *release time* například z důvodu čekání na materiál, dále může být požadován požadovaný čas dokončení (*due time*) a nejpozdější čas dokončení (*deadline*) a konkretní typ stroje, na kterém může být operace vykonána. Mezi operacemi můžou existovat závislosti, kdy některá operace vyžaduje dokončení několika předchozích. Všechny tyto omezující podmínky přispívají ke zvýšení náročnosti řešení. Důležité je stanovení cílového kritéria, vůči kterému hledáme optimální řešení. Může to být například minimalizace času konce výroby nebo minimalizace času od due time neboli požadavek na tzv. „just in time“ výrobu.

Pro vizualizaci rozvrhování se používá *Ganttův diagram*. Jedná se graf, kdy na vertikální ose jsou na řádcích vyneseny stroje a na horizontální ose je časové měřítko. Do řádků jsou přiřazovány obdélníky reprezentující úkoly, které označují začátek, konec a dobu trvání operace.



Obrázek 3.1. Ganttův diagram

Podle požadavku na zmíněné omezující podmínky se problémy rozdělují do různých skupin, pro které byly vyvinuty odlišné algoritmy podle složitosti daného problému. Rozvrhovací skupiny klasifikujeme *Grahamovou notací* $\alpha|\beta|\gamma$, kde α popisuje charakteristiku strojů, β popisuje charakteristiku úkolů s omezujícími podmínkami a γ popisuje cílové kritérium. [7][8]

1) *Charakteristika strojů* α obsahuje maximálně dva prvky. α_1 určuje vlastnost strojů viz níže a α_2 označuje počet strojů 1,2, ... nebo obecně m.
   - P značí paralelní identické stroje, na kterých může probíhat zpracování úkolů
   - Q je vlastně stejné jako P s tím rozdílem, že každý stroj může mít jinou rychlost zpracování
   - R je nejobecnější, neboť rychlost zpracovaní závisí jak na stroji, tak na typu úkolu. Rychlost je určena maticí (stroje x úkoly)
   - J job–shop, kdy každý úkol je rozdělen na několik podúkolů, které musí být zpracovány každý na jiném stroji v určeném pořadí
   - F flow–shop je speciální varianta J, kde počet podúkolů je stejný jako počet strojů a podúkoly jsou zpracovány postupně na všech strojích
   - O open–shop je definován jako F bez určeného pořadí podúkolů
2) *Charakteristika úkolů* β je složena nejvýše ze šesti prvků $\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6$
   - $\beta_1 = prmt$ *(preemption)* určuje, že úkol může být kdykoli přerušen a pokračování lze navázat i na jiném stroji.
   - $\beta_2 = prec$ *(precedence)* popisuje závislosti mezi úkoly, kdy úkol nemůže začít bez dokončení předchozích. Tyto závislosti se dají znázornit orientovaným acyklickým grafem.
   - $\beta_3 = r_i$ požadavek, kdy nejdříve se může s úkolem začít, může být pro každý úkol rozdílný
   - β_4 specifikuje dobu trvání úkolů, která může být pro všechny stejná $p_j = t$. Pokud je β_4 vynechána je doba úkolů volitelná.
   - $\beta_5 = d_i$ určuje každému úkolu deadline, do kterého musí být dokončen.
   - β_6 se přidává v případě, kdy úkoly musejí být zpracovány po várkách
3) *Charakteristika cílového kritéria* γ

   Cílové kritérium je popsáno funkcí $f = \gamma$, kterou se zpravidla snažíme minimalizovat. Zde bude uvedeno jen několik nejběžnějších příkladů:
   - C_{max} je nejspíše nejčastější kritérium minimalizace času dokončení[1] posledního úkolu, $C_{max} = \max_{i \in \mathcal{T}}\{C_i\}$. Toto kritérium navíc při rozvrhování na paralelní stroje se snaží rovnoměrně přidělovat jednotlivé úkoly.
   - $\sum_{i \in \mathcal{T}} C_i$ kritérium součtu časů dokončení úkolů si můžeme představit jako minimalizaci doby čekání jednotlivých úkolů.
   - $\sum_{i \in \mathcal{T}} w_i \cdot C_i$ je stejné jako předchozí kritérium, ale časy dokončení jsou váženy parametrem w_i vyjadřující prioritu úkolů.
   - $\max_{i \in \mathcal{T}}\{C_i - d_i\}$ Dalším příkladem může být minimalizace maximálního zpoždění.

[1] z angl. Completion time

Rozvrhovací problémy často patří do skupiny $\mathcal{NP}$–hard, neboli neexistuje algoritmus, který by jej zvládl vyřešit v polynomiálním čase. V takovém případě je výhodné ho převést na problém kombinatorické optimalizace, pro který existují dva osvědčené přístupy k řešení: *Integer linear programming (ILP), Constraint programming (CP).*

3.2 Lineární a celočíselné programování

ILP přistupuje k rozvrhovacímu problému čistě matematicky a hledá nejlepší řešení minimalizující dané kritérium na množině přípustných řešení za použití dobře známých algoritmů pro řešení úloh LP a ILP.

3.2.1 Lineární programování

Lineární program je optimalizační problém minimalizace nebo maximalizace lineární cílové funkce (1) za splnění omezujících podmínek určených soustavou lineárních rovnic a nerovnic (2)(3). Formálně se zapíše v maticovém tvaru

$$\begin{aligned} min \quad & \mathbf{c}^\mathsf{T}\mathbf{x} && (1)\\ za\ podminek: \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} && (2)\\ & \mathbf{x} \geq \mathbf{0} && (3) \end{aligned}$$

kde $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{x} \in \mathbb{R}^n$ a $\mathbf{c} \in \mathbb{R}^n$. Množina přípustných řešení se nazývá polyhedr (mnohostěn) a pokud existuje řešení, nachází se právě ve vrcholu polyhedru nebo je řešením celá jeho hrana. V úlohách LP se můžeme často setkat s potřebou maximalizace, ale většina řešičů je napsána pouze pro případ minimalizace. Naštěstí je převod z maximalizace na minimalizaci jednoduchý, $\max(\mathbf{c}^\mathsf{T}\mathbf{x}) = -\min(-\mathbf{c}^\mathsf{T}\mathbf{x})$.

Nejznámějším algoritmem na řešení LP je *simplexová metoda*, která systematicky hledá optimum ve vrcholech tak, že prochází vrcholy podél hran polyhedru tak, aby se hodnota cílové funkce zmenšovala nebo alespoň nezhoršovala.[9]

3.2.2 Celočíselné lineární programování

Celočíselné lineární programování se od LP liší pouze tím, že požadujeme, aby doména řešení $\mathbf{x}$ byla pouze celočíselná, nejčastěji binární hodnota 0 nebo 1. Binární hodnota například říká, zda je daný úkol přiřazen ke konkrétnímu stroji nebo není.

$$\min \{\mathbf{c}^\mathsf{T}\mathbf{x} \mid \mathbf{A}\mathbf{x} \geq \mathbf{b},\ \mathbf{x} \in \mathbb{Z}^n\} \tag{4}$$

Na první pohled se může zdát, že zmenšením domény došlo ke zjednodušení problému, ale opak je pravdou, protože množinu přípustných řešení už netvoří konvexní polyhedr, ale množina izolovaných bodů, a nelze použít standardní algoritmy. Na rozdíl od LP, který lze vyřešit v polynomiálním čase, je vyřešení ILP obecně $\mathcal{NP}$–hard.

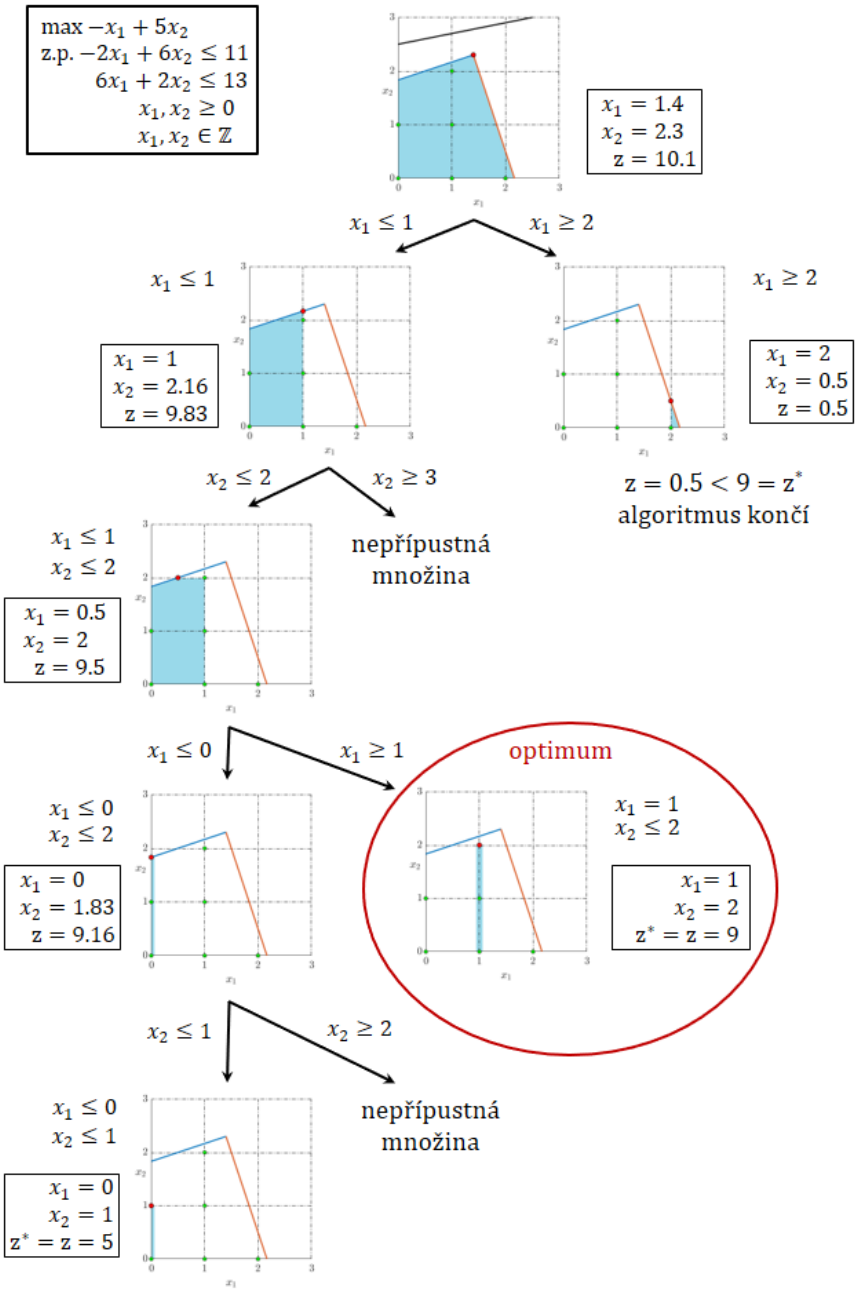
Můžeme se pokusit vyřešit ILP relaxováním na LP převedením binárních podmínek $x_i \in \{0, 1\}$ na nerovnice $0 \leq x_i \leq 1$ a použitím simplexové metody získat přibližné řešení. Bohužel zaokrouhlením můžeme dostat celočíselné řešení mimo přípustnou množinu nebo i kdyby bylo zaokrouhlení přípustné, může být daleko od optima. Přesto je přibližné řešení užitečné v sofistikovanějších metodách, například metoda *větví a mezí (Branch and Bound).*

Dalším přístupem je *enumerační metoda*, která prochází a ohodnocuje všechna přípustná řešení. Vzhledem k exponenciálnímu růstu proměnných je ale vhodná pouze pro malé úlohy. [7][9]

3.2.3 Branch and Bound

Branch and Bound je nejefektivnější metoda na řešení úlohy ILP na jejíž principu funguje řada dostupných i komerčních řešičů.

Princip metody je následující: relaxováním úlohy na LP získáme odhad řešení $\mathbf{x}^*$ a prostor řešení je rozdělen na dvě poloviny zavedením podmínek $\mathbf{x} \leq \lfloor \mathbf{x}^* \rfloor$, resp. $\mathbf{x} \geq \lfloor \mathbf{x}^* \rfloor + \mathbf{1}$. Dělení na menší a menší podprostory dochází dokud není nalezeno přípustné celočíselné řešení. Poté se algoritmus vrací k ještě neprozkoumaným a nerozděleným větvím. Je udržována dolní mez (při maximalizaci), resp. horní mez (při minimalizaci) nalezeného řešení k eliminaci větví, které by výsledek ještě zhoršovaly. Tímto redukuje počet větví a zvyšuje efektivitu prohledávání. [7]



Obrázek 3.2. Branch and Bound algoritmus

3.3 Constraint programming

Zatím co ILP přistupuje k problému čistě matematicky a opírá se o lineární algebru, CP vychází z poznatků výzkumu umělé inteligence a soustředí se na složitost kombinatorického problému a bylo pro něj navrženo spoustu výkonných algoritmů. Hlavní myšlenkou je procházení stavového prostoru řešení za pomocí grafových algoritmů a eliminace velké části prostoru, který nesplňuje omezující podmínky.

Problémy řešené CP se nazývají Constraint Satisfaction Problems (CSP) a jsou definovány jako trojice (X, D, C):

- X je množina proměnných, $\{X_1, X_2, \ldots, X_n\}$
- D je množina domén pro každou proměnou, $\{D_1, D_2, \ldots, D_n\}$
- C je množina omezujících podmínek mezi proměnnými

Příkladem CSP je problém obarvení mapy složené z regionů X_i pomocí tří barev $D_i = \{\textit{červená, modrá, zelená}\}$ přičemž sousední regiony nesmějí mít stejnou barvu. To zajistíme přidáním podmínek $X_i \neq X_j$, pokud region X_i sousedí s X_j. Na tomto příkladu je vidět, že hodnoty nemusejí být pouze aritmetické jako u ILP. Stejně tak podmínky nejsou omezeny jen na nerovnosti, ale můžou být zavedeny logické i globální podmínky (například *alldifferent*). Tímto může být sestavení modelu přímočařejší a snazší.

Řešení CSP provádí tzv. *backtracking*, jedná se vlastně o prohledávání do hloubky, kdy jsou proměnným přiřazovány hodnoty z domén, a přiřazování pokračuje dokud nejsou porušeny omezující podmínky. Poté se vrací zpět a zkouší ještě nepřiřazené hodnoty z domén proměnných. Takto by mohlo dojít k neefektivnímu prohledání celého obrovského prostoru, proto byly navrženy různé heuristiky a přístupu zajištující konzistenci podmínek a domén. [3][7][10]

3.3.1 AC–3

AC–3 je nejznámější algoritmus hranové konzistence[2] využívaný k redukci prostoru před samotným prohledáváním. CSP je hranově konzistentní pokud všechny hodnoty proměnným splňují omezující podmínky. Jinými slovy pro každou hodnotu z D_i musí existovat hodnota z D_j, aby podmínka mezi (X_i, X_j) byla konzistentní. *AC–3* si udržuje frontu všech proměnných, mezi kterými je omezující podmínka, a postupně kontroluje dvojice z fronty. Pokud je nějaká doména D_i zmenšena, některé hodnoty v ní nesplňují konzistenci a byly vymazány, musí být znovu zkontrolovány podmínky související s X_i a tyto podmínky (X_k, X_i) jsou přidány do fronty. [3][7][10]

Příklad CSP s *AC–3*:

$$X = \{\{1,2,3\}, \{1,2,3\} \mid X_1 < X_2\}$$

Fronta $Q = \{X_1 < X_2, X_2 > X_1\}$. Po kontrole 1. podm.: $X = \{\{1,2\}, \{1,2,3\}\}$
Fronta $Q = \{X_2 > X_1\}$. Po kontrole 2. podm.: $X = \{\{1,2\}, \{2,3\}\}$

Ještě musí být zkontrolována podmínka $X_1 < X_2$, protože jsme v X_2 smazali hodnotu 1. Tato podmínka už nijak neovlivní domény a algoritmus AC–3 končí. Při zavedení cílové funkce $\max f(X) = X_1 + X_2$ nalezneme řešení $f(X) = 5$ pro $X = \{\{2\}, \{3\}\}$.

[2] z angl. Arc Consistency

3.4 Rozvrhování na n identických strojů

Po seznámení s teorií můžeme přistoupit k vytvoření matematického modelu rozvrhování výroby. Předpokládáme vytvořený denní výrobní plán, pro nějž je potřeba naplánovat rozvržení zpracování desek na několik současně pracujících strojů. Přičemž operace zpracování desek nesmějí být přerušeny a z povahy problému může být současně na jednom stroji zpracovávána pouze jedna deska. Cílem je minimalizovat výrobní čas daného plánu. Tento rozvrhovací problém můžeme zapsat Grahamovou notací (5) a patří do skupiny $\mathcal{NP}$–hard, proto je výhodné ho formulovat jako celočíselný lineární program a řešit jej pomocí výkonných programů (např. CPLEX, GORUBI, OR-TOOLS).

$$P \mid\mid C_{max} \tag{5}$$

Pro model zavedeme parametry a proměnné popisující rozvrhovací úlohu:

- $\mathcal{T} = \{1, 2, \ldots, n\}$ množina operací s deskami ve výrobním plánu
- $\mathcal{M} = \{1, 2, \ldots, m\}$ množina dostupných strojů
- $\mathbf{P} = \{p_{mt}\}$ matice časů zpracování t-té operace na m-tém stroji
- $\mathbf{X} = \{x_{tm}\}$ matice proměnných přiřazující zpracování t-té operace na m-tý stroj

Dále musíme definovat omezující podmínky. Přiřazení každé operace k právě jednomu stroji nám zajistí podmínky nad maticí $\mathbf{X}$, kde součet každého řádku se musí rovnat jedné a díky binárním proměnným matice je jednoznačně určeno, na kterém stroji má být operace provedena. Minimalizační kritérium je dáno vztahem $C_{max} = \max_{i \in \mathcal{T}}\{C_i\}$, neboli hledáme nejdéle pracující stroj. Čas dokončení každého stroje získáme vymaskováním matice $\mathbf{P}$ maticí $\mathbf{X}$ a součtem příslušných řádků, čímž dostáváme nelineární cílovou funkci.

$$C_{max} = \max\{\sum_{t \in \mathcal{T}} p_{mt}\, x_{tm} \mid \forall m \in \mathcal{M}\} \tag{6}$$

Jak víme, cílová funkce pro řešič ILP musí být lineární, a maximum musíme převést na nerovnice. To není problém, protože platí $\max\{x_1, \ldots, x_n\} = z \;\Leftrightarrow\; z \geq x_1, \ldots, z \geq x_n$ a výsledný ILP model můžeme popsat takto:

$$min \quad C_{max} \tag{7}$$

$$z.\,p.: \quad \sum_{t \in \mathcal{T}} p_{mt}\, x_{tm} \leq C_{max} \quad \forall m \in \mathcal{M} \tag{8}$$

$$\sum_{m \in \mathcal{M}} x_{tm} = 1 \quad \forall t \in \mathcal{T} \tag{9}$$

$$x_{tm} \in \{0, 1\} \quad \forall t \in \mathcal{T}\ \forall m \in \mathcal{M} \tag{10}$$

$$C_{max} \in \mathbb{R} \tag{11}$$

Výhodou modelu je, že má relativně málo proměnných $|\mathcal{M}||\mathcal{T}|$, protože řešiče ILP nemají většinou problém se stovkami tisíc proměnných a podmínek. Výhodná je také reprezentace výrobních časů v matici $\mathbf{P}$, která umožňuje definovat rozdílné časy operace podle typu stroje. Ne vždy totiž pracujeme se stroji stejného výrobce nebo se stroje mohou lišit stářím, výkonem motorů, řeznými rychlostmi a podobně. Proto všechny tyto vlastnosti mohou být do modelu zahrnuty při plánování výroby v podobě simulovaných časů operací v matici $\mathbf{P}$.

3.5 Rozvrhování na n dedikovaných strojů

Jak již bylo zmíněno při zpracování desek často pracujeme s více typy stroji, proto je vhodné předchozí model rozšířit o možnost plánování operací na dedikované stroje a zajistit tak přibližně stejné dokončení prací na všech strojích ve výrobě například do konce směny.

Rozvrhovací model (7) je snadno rozšiřitelný na n typů strojů. Pro názornost a přehlednost bude zformulován rozvrhovací model pro dva typy strojů.

Nejprve zavedeme množinu pro každý typ s různým počtem strojů m_1, resp. m_2. Operace rozdělíme podle toho, na který stroj jsou určeny, a definujeme pro ně matice časů a přiřazovacích proměnných.

- $\mathcal{M}_1 = \{1, 2, \ldots, m_1\}$ množina dostupných strojů prvního typu
- $\mathcal{M}_2 = \{1, 2, \ldots, m_2\}$ množina dostupných strojů druhého typu
- $\mathcal{T}_1 = \{1, 2, \ldots, n_1\}$ množina operací s deskami na prvním stroji
- $\mathcal{T}_2 = \{1, 2, \ldots, n_2\}$ množina operací s deskami na druhém stroji
- $\mathbf{P} = \{p_{mt}\}$ matice časů zpracování t-té operace na m-tém stroji prvního typu
- $\mathbf{P}' = \{p'_{mt}\}$ matice časů zpracování t-té operace na m-tém stroji druhého typu
- $\mathbf{X} = \{x_{tm}\}$ matice proměnných t-té operace na m-tém stroji prvního typu
- $\mathbf{Y} = \{y_{tm}\}$ matice proměnných t-té operace na m-tém stroji druhého typu

Nyní může sestavit model pro dedikované stroje obdobně jako pro identické stoje.

$$min \quad C_{max} \tag{12}$$

$$z.\,p.: \quad \sum_{t \in \mathcal{T}_1} p_{mt}\, x_{tm} \leq C_{max} \quad \forall m \in \mathcal{M}_1 \tag{13}$$

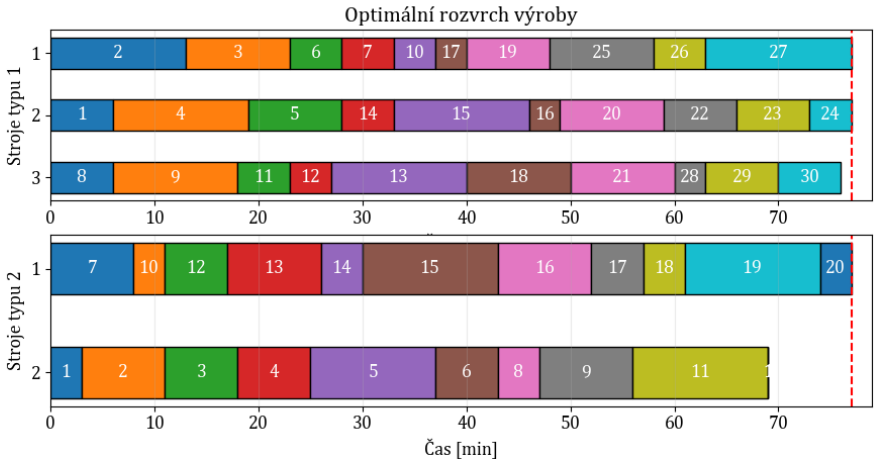
$$\sum_{t \in \mathcal{T}_2} p'_{mt}\, y_{tm} \leq C_{max} \quad \forall m \in \mathcal{M}_2 \tag{14}$$

$$\sum_{m \in \mathcal{M}_1} x_{tm} = 1 \quad \forall t \in \mathcal{T}_1 \tag{15}$$

$$\sum_{m \in \mathcal{M}_2} y_{tm} = 1 \quad \forall t \in \mathcal{T}_2 \tag{16}$$

$$x_{tm} \in \{0, 1\} \quad \forall t \in \mathcal{T}_1\ \forall m \in \mathcal{M}_1 \tag{17}$$

$$C_{max} \in \mathbb{R}, \quad y_{tm} \in \{0, 1\} \quad \forall t \in \mathcal{T}_2\ \forall m \in \mathcal{M}_2 \tag{18}$$



Obrázek 3.3. Rozvrh na dedikované stroje

3.6 Prioritní rozvrhování na identické stroje

Často je potřeba plánovat s ohledem na následující proces ve výrobě, například montáž, neboli potřebujeme, aby všechen materiál potřebný pro montáž byl vyroben do určitého času. Proto byla snaha vytvořit model, který by upřednostňoval skupiny operací.

Prioritizace operací můžeme docílit určením společného deadlinu d_t každé skupině a dostáváme rozvrhovací problém:

$$P \mid d_t \mid C_{max} \tag{19}$$

Protože musíme pracovat s koncem operací, zavedeme proměnou začátku operace s_t a konec získáme přičtením trvání operace p_t. První dvě omezující podmínky přiřazení operace a minimalizace C_{max} jsou shodné s předchozími modely. Dále musíme zavést omezení konce intervalu operací z hora kritériem C_{max} (23) a deadlinem d_t (24). Nakonec musíme zamezit překryvu intervalů operací, pokud jsou rozvrhnuty na stejném stroji. Tedy chceme, aby omezení platilo jen za určitých podmínek. Při takovém požadavku se ve formulaci ILP používá *Big–M metoda*, kdy nějaké veliké číslo (volí s ohledem na ostatní proměnné) při nesplnění podmínek převáží nerovnici a omezující podmínka se stane „neaktivní“. *Big–M metoda* byla použita právě u omezující podmínky překryvu (25). Podmínka má smysl pouze v případě, kdy je operace t rozvržena před operací $t^{'}$. Tento stav vyjadřují dodatečně zavedené binární proměnné $\delta_{tt'}$.

$$min \quad C_{max} \tag{20}$$

$$z.\,p.: \sum_{m \in \mathcal{M}} x_{tm} = 1 \quad \forall t \in \mathcal{T} \tag{21}$$

$$\sum_{t \in \mathcal{T}} p_t \, x_{tm} \leq C_{max} \quad \forall m \in \mathcal{M} \tag{22}$$

$$s_t + p_t \leq C_{max} \quad \forall t \in \mathcal{T} \tag{23}$$

$$s_t + p_t \leq d_t \quad \forall t \in \mathcal{T} \tag{24}$$

$$s_t + p_t \leq s_{t'} + M\,(3 - x_{tm} - x_{t'm} - \delta_{tt'}) \quad \forall t, t^{'} \in \mathcal{T}\ \forall m \in \mathcal{M} \tag{25}$$

$$\delta_{tt'} + \delta_{t't} = 1 \quad \forall t, t^{'} \in \mathcal{T} \quad t \neq t^{'} \tag{26}$$

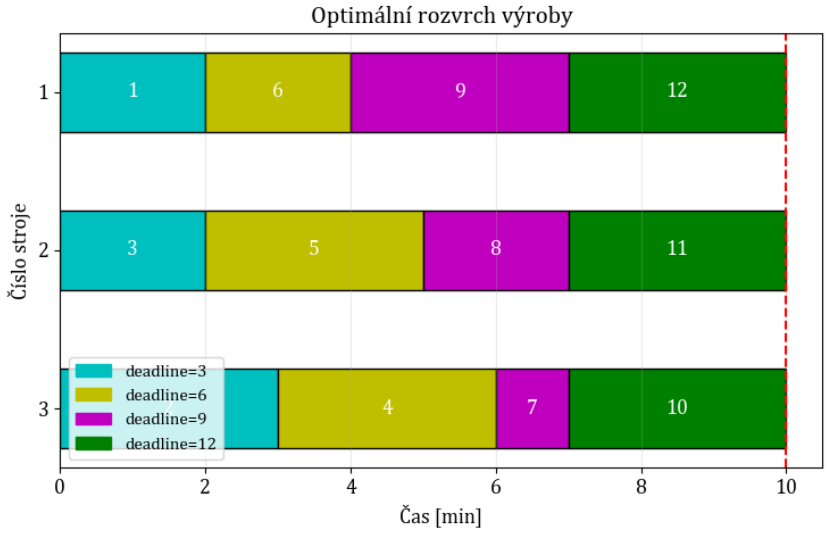
$$x_{tm} \in \{0, 1\} \quad \forall t \in \mathcal{T}\ \forall m \in \mathcal{M} \tag{27}$$

$$\delta_{tt'} \in \{0, 1\} \quad \forall t, t^{'} \in \mathcal{T} \tag{28}$$

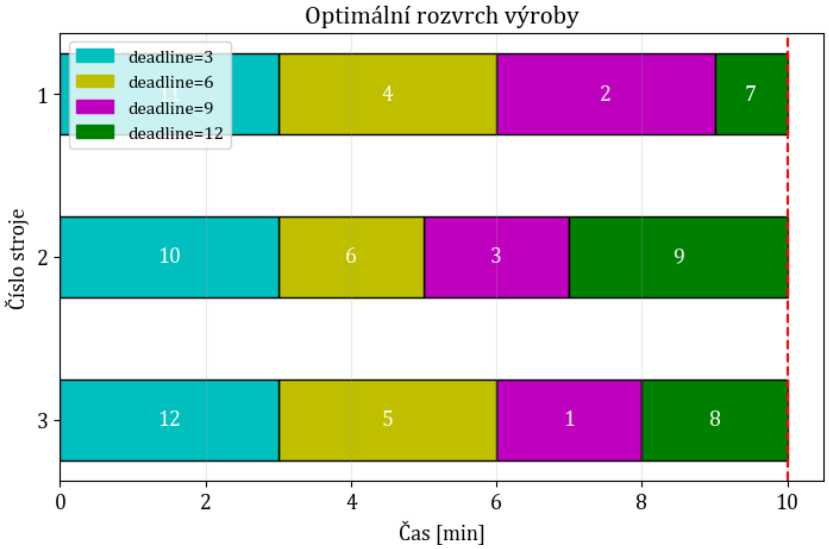
$$s_t \in \mathbb{R} \quad \forall t \in \mathcal{T} \tag{29}$$

$$C_{max} \in \mathbb{R} \tag{30}$$

Na obrázcích 3.4 a 3.5 můžeme vidět příklad rozvrhování s rozdílným přiřazením deadlinů k jednotlivým operacím.



Obrázek 3.4. Prioritní rozvrh 1



Obrázek 3.5. Prioritní rozvrh 2

3.7 Implementace knihovnou OR–tools

Pro implementaci byl zvolen open–source software OR–Tools od Google [11], který poskytuje vlastní řešiče kombinatorické optimalizace LP, ILP, CP, i specializované algoritmy pro TSP, a má také dobrou dokumentaci. OR–Tools je vytvořeno v C++, ale zároveň podporuje širokou škálu jazyků (Python, Java, C#). Jeho název je odvozen od *operačního výzkumu* (jiný termín pro kombinatorickou optimalizaci), byl prvně použit za druhé světové války, kdy se řešili logistické problémy s dodávkami materiálu na frontu.

Pro vyřešení rozvrhu operací byla vytvořena funkce `solve_schedule()`, která na svém vstupu očekává počet strojů a operací pro jednotlivé typy strojů. Dále pak matice časů zpracování, které jsou pro potřeby testování generovány náhodně v rozmezí tří až patnácti minut (průměrné časy operací ve dřevozpracujícím průmyslu).

```
if __name__ == "__main__":
    num_machines = [3, 2, 2]
    num_tasks = [120, 80, 60]
    all_tasks_process_time = []
    for tasks, machines in zip(num_tasks, num_machines):
        all_tasks_process_time.append(
          (np.random.randint(3, 15,(tasks, 1))*np.ones((1,machines))).T)
    print("Scheduling..")
    solve_schedule(num_machines, num_tasks, all_tasks_process_time,True)
```

Funkce nejprve vytvoří objekt řešiče, resp. modelu ILP z knihovny OR–Tools a vytvoří cílovou reálnou proměnnou `makespan`[3] (C_{max}) metodou `Var()` , kde jí nastaví povolený rozsah od nuly do nekonečna. Následuje smyčka, která rozbalí vstupní data, vytvoří proměnné a omezující podmínky pro každý typ podle modelu pro identické stroje (7).

```
def solve_schedule(machinesNums, tasksNums, process_times, showResult):
    #vytvoření ILP řešiče/modelu
    solver = pywraplp.Solver("Makespan Minimalizaion",
        pywraplp.Solver.SCIP_MIXED_INTEGER_PROGRAMMING)
    #přidání cílové reálné proměnné
    makespan = solver.Var(0, solver.infinity(), False, "makespan")
    #list pro matice přiřazovacích proměnných pro každý typ stroje
    tasksOnMachine = []
    for num_machines, num_tasks, all_tasks_process_time
                        in zip(machinesNums, tasksNums, process_times):
        #vytvoření množin indexů pro stroje a operace
        all_machines = range(num_machines)
        all_tasks = range(num_tasks)
        #vytvoření jedné přiřazovací matice reprezentované jako slovník
        #s binárními proměnnými
        tasks = {}
        for t in all_tasks:
            for m in all_machines:
                tasks[(t,m)] = solver.IntVar(0,1,f'task_{t}-{m}')
        tasksOnMachine.append(tasks)
```

[3] angl. termín pro čas dokončení poslední operace

```
21          #přidání podmínek zaručující přiřazení
22          #operace pouze k jednomu stroji
23          for t in all_tasks:
24              solver.Add(sum(tasks[(t,m)] for m in all_machines) == 1)
25          #přidání podmínek shora omezující konec poslední operace
26          for m in all_machines:
27              solver.Add(sum((all_tasks_process_time[m,t]*tasks[(t,m)])
28                                          for t in all_tasks) <= makespan)
```

Po zavedení všech potřebných proměnných a omezujících podmínek zbývá definovat cílovou funkci a požádat řešič o vyřešení modelu. Pokud je nalezeno optimální řešení, vytisknou se hledané hodnoty `makespan` a přiřazovacích proměnných x_{tm}. Nakonec je ze získaných dat sestaven Ganttův diagram.

```
29      #definice cílové funkce
30      solver.Minimize(makespan)
31      #spustění řešení modelu
32      status = solver.Solve()
33
34      #výpis nalezeného řešení
35      if status == pywraplp.Solver.OPTIMAL:
36          print('================ Solution ================')
37          print(f'Solved in {solver.wall_time():.2f} milliseconds')
38          print(f'in {solver.iterations()} iterations')
39          print(f'Optimal makespan = {solver.Objective().Value()} time')
40
41          #zpracovaní každého stroje
42          typeMachineID = 0
43          for num_machines, num_tasks, all_tasks_process_time
44                      in zip(machinesNums, tasksNums, process_times):
45              tasks = tasksOnMachine[typeMachineID]
46              typeMachineID += 1
47
48              all_machines = range(num_machines)
49              all_tasks = range(num_tasks)
50              processVal = np.zeros_like(all_tasks_process_time)
51              for m in all_machines:
52                  for t in all_tasks:
53                      val = tasks[(t,m)].solution_value()
54                      processVal[m,t] = val
55              if showResult:
56                  print(f"Assign task to machines {typeMachineID}:")
57                  print(processVal)
58                  #
59                  #
60                  # Zpracovaní proměnných a tvorba Ganttova diagramu
61                  #
62                  #
63      else:
64          print('The solver could not find an optimal solution.')
65          return None
```

3.8 Testování rozvrhování

Testování probíhalo na notebooku s procesorem Intel i5 7300HQ 2.50GHz a jeho cílem bylo ověření schopnosti nasazení v průmyslové praxi, kde je potřeba naplánovat jak směnnou, tak celodenní výrobu.

3.8.1 Rozvrhování dedikovaných strojů

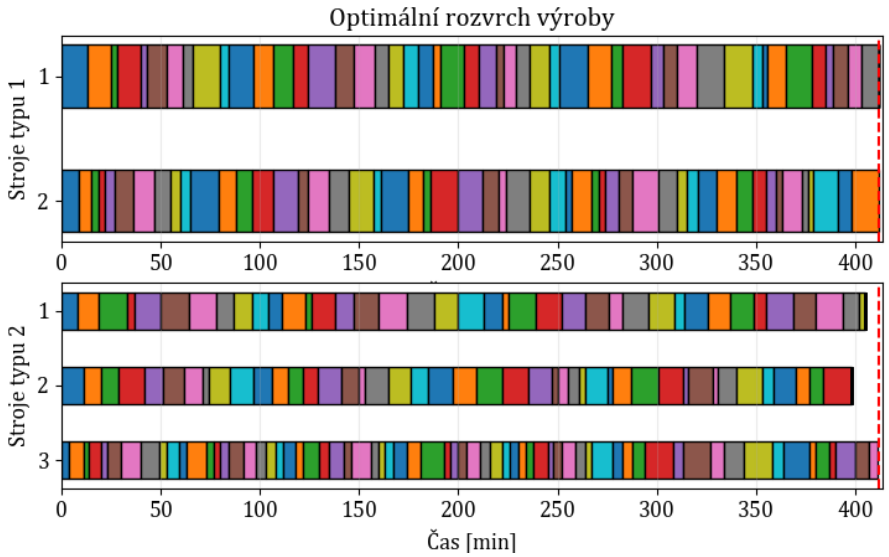
S požadavkem směnné výroby byly provedeny dva testy v konfiguraci s dvěma typy strojů ve výrobě (s dvěma jednoho a se třemi druhého typu). Je důležité podotknout, že pro skončení prací všech strojů v přibližně stejném čase je potřeba plánovat počet operací na jednotlivých typech strojů rovnoměrně.

Pro osmi hodinovou směnu byly stanoveny počty operací na 100 a 150. Dále pro celodenní dvanácti hodinovou výrobu byly stanoveny na 160 a 240 operací. Průměrné časy operací byly náhodné z intervalu od tří do patnácti minut.

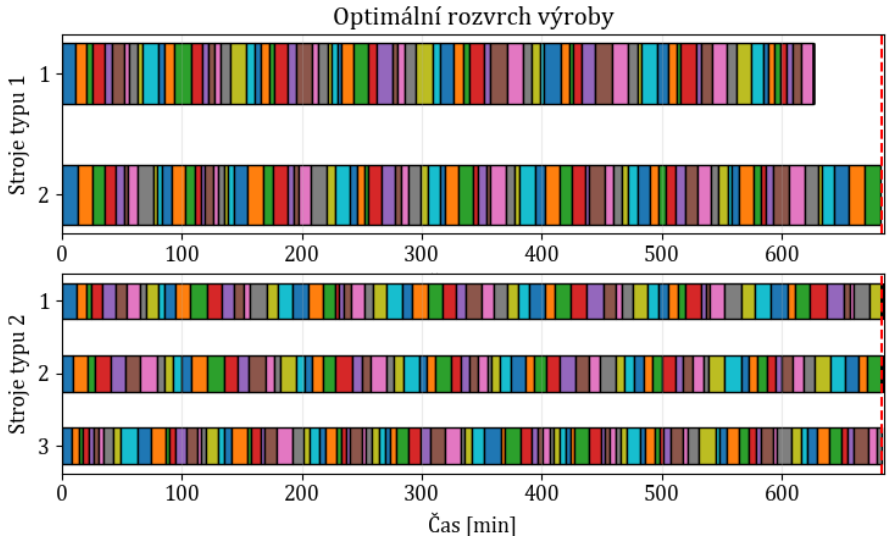
Směna	Čas řešení [s]	Počet iterací	C_{max} [min]
Osmihodinová	0.121	190	440
Dvanáctihodinová	0.186	256	694

Tabulka 3.1. Průměrné hodnoty (z 5 hodnot) při rozvrhování směn (řešič ILP)

V obou případech byly voleny počty operací tak, aby výsledný rozvrh měl rezervu pro případná nečekaná zdržení. Díky nízkému počtu proměnných (651 pro 8h, 1 041 pro 12h) a omezujících podmínek (255 pro 8h, 405 pro 12h) je rozvrhování velice rychlé, čímž umožňuje snadné odebrání, resp. přidání operací a přepočítání celého rozvrhu.



Obrázek 3.6. Osmihodinová směna



Obrázek 3.7. Dvanáctihodinová směna

3.8.2 Prioritní rozvrhování

Testování prioritního rozvrhování probíhalo na třech identických strojích. Počet a časové intervaly operací byly shodné jako v předchozích testech pro tři stroje. Při implementaci modelu s deadliny (20) v řešiči ILP trvalo řešení rozvrhování extrémně dlouho kvůli velkému počtu proměnných (22 951 pro 8h, 58 321 pro 12h) a omezujících podmínek (89 853 pro 8h, 230 163 pro 12h). Už při 45 operacích zabralo řešení 30 minut a bylo provedeno 1 827 446 iterací. Model byl proto přeformulován do CP řešiče, který si vedl díky pokročilým metodám backtrackingu o poznání lépe viz tab. 3.2. Hlavně podmínka *Big–M* (25) v modelu jde formulovat v CP pomocí logické konjunkce a mnoho nepotřebných podmínek při hledání řešení může být odstraněno. Přesto řešení dvanáctihodinové směny trvalo 70 minut a bylo provedeno 4 698 670 iterací (s $C_{max} = 708$). Pro efektivnější model by musely být použity pokročilejší techniky např. *Branch and Price*.

Směna	Čas řešení [s]	Počet iterací	C_{max} [min]
Osmihodinová	47.99	399572	437

Tabulka 3.2. Průměrné hodnoty (z 5 hodnot) při prioritním rozvrhování směn (řešič CP)

Výsledky testů ukázaly, že oba modely jsou vhodné pro použití v praxi a budou tak vhodným pomocníkem při plánování výroby.

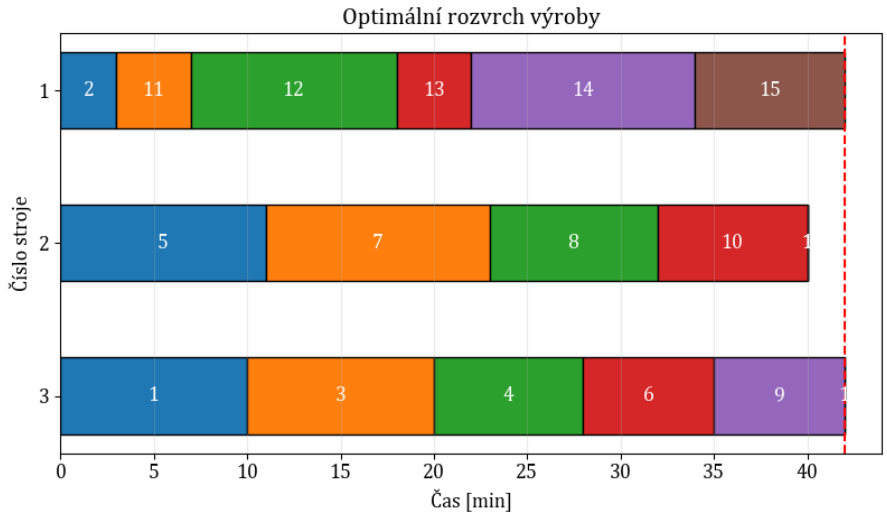
Kapitola 4
Spojení algoritmů vyskladňovaní a rozvrhování výroby

Po vytvoření algoritmů vyskladnění a rozvrhování je potřeba tyto dva moduly propojit. Ze získaného rozvrhu pro každou desku víme, v jakém čase a na kterém stroji má být zpracována. Z těchto informací musíme vytvořit frontu desek v přesně určeném pořadí, ve kterém jsou potřeba odbavovat. Tato fronta bude následně zpracována vyskladňovacím algoritmem a desky mohou být rovnou přesouvány na konkrétní stroje jim určené nebo mohou být připravené v požadovaném pořadí na vyskladňovací pozici skladu v případě, kdy výrobní časy desek budou krátké a skladový manipulátor by nezvládal výrobu zásobit. Druhý přístup je obzvlášť výhodný ve spojení dvou manipulátorů, kdy jeden by se staral o uspořádání desek a druhý by už pouze přesouval desky z již už seřazené výstupní pozice skladu.

Získaný rozvrh výroby můžeme reprezentovat dvěma maticemi `optimSchedule` a `startTime`. V první matici jsou uložena identifikační čísla desek a ve druhé časy ve kterých se má začít konkrétní deska zpracovávat. Řádky matic odpovídají jednotlivým strojům a desky jsou v nich seřazeny podle rozvrhu.

$$optimSchedule = \begin{bmatrix} 2 & 11 & 12 & 13 & 14 & 15 \\ 5 & 7 & 8 & 10 & 0 & 0 \\ 1 & 3 & 4 & 6 & 9 & 0 \end{bmatrix} \tag{1}$$

$$startTime = \begin{bmatrix} 0 & 3 & 7 & 18 & 22 & 34 \\ 0 & 11 & 23 & 32 & 40 & 40 \\ 0 & 10 & 20 & 28 & 35 & 42 \end{bmatrix} \tag{2}$$



Obrázek 4.1. Rozvrh výroby

Matice shodných rozměrů můžeme výhodně procházet po sloupcích a řadit tak desky do výsledné fronty desek vzestupně podle časů ve sloupcích matice `startTime`. Pouze u prvního sloupce je strategie obrácená, aby měl manipulátor čas na zásobení ostatních strojů. Fronta desek pro konkretní případ na obr.4.1 bude:

$$fronta = [5, 1, 2, 11, 12, 3, 7, 13, 4, 14, 8, 6, 10, 15, 9] \tag{3}$$

Z poznatků o vyskladňovacím algoritmu víme, že je výhodné zpracovávat fronty do velikosti šest s ohledem na efektivnost vyskladnění, a tak rozdělíme frontu a připravíme požadavky pro vyskladňovací algoritmus.

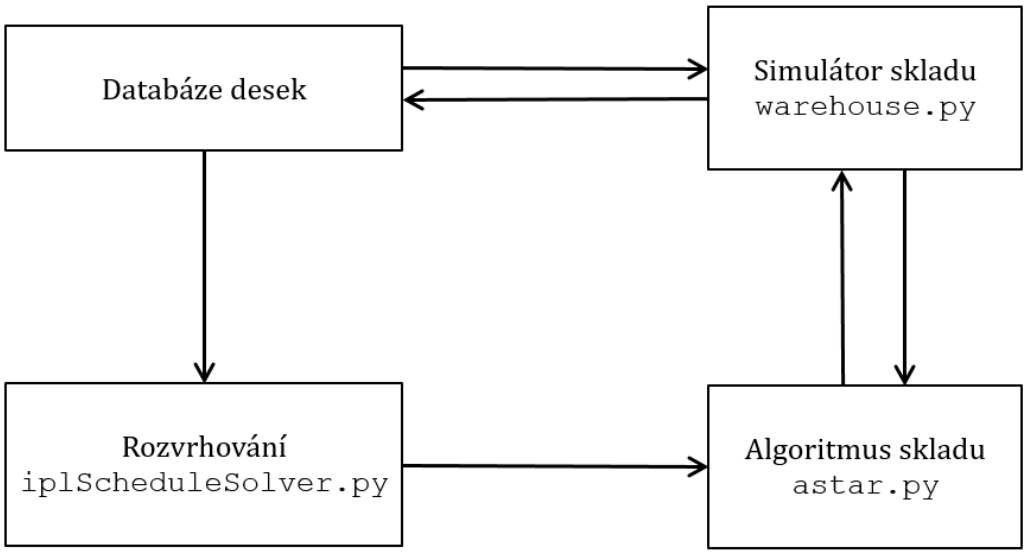
$$fronta = [\ [5, 1, 2, 11, 12, 3], [7, 13, 4, 14, 8, 6], [10, 15, 9]\ ] \tag{4}$$

O toto uspořádání desek a tvorbu fronty se stará funkce `create-batch()` v modulu rozvrhování výroby.

```
def create-batch(optimSchedule, startTime):
    row, col = optimSchedule.shape
    batch = [] # list pro výslednou frontu
    #řazení prvních desek sestupně
    subSchedule = optimSchedule[:,0] #první sloupec
    subTime = startTime[:,1] #druhý sloupec
    while subSchedule.size > 0: #not is empty
        machineID = np.argmax(subTime)
        batch.append(subSchedule[machineID]+1)
        #np.delete(matice, i) vymaže prvek v matici na i-té pozici
        subTime = np.delete(subTime, machineID)
        subSchedule = np.delete(subSchedule, machineID)
    #řazení ostatních desek vzestupně
    q = PriorityQueue()
    for r in range(row):
        for c in range(1,col):
            if optimSchedule[r,c] == 0:
                continue
            q.put((startTime[r,c], optimSchedule[r,c]))

    while not q.empty():
        batch.append(q.get()[1]+1)

    return batch
```

4.1 Architektura aplikace

Závěrem poslední kapitoly popíšeme část architektury plánované aplikace řídícího systému skladu. Komunikace mezi vytvořenými moduly bude následující. Nejdříve bude vytvořena požadovaná fronta z dostupných desek z databáze skladu rozvrhovacím modulem nebo případně ručně operátorem skladu. Fronta bude dále zpracována A–Star algoritmem, který potřebuje simulátor skladu. Proto bude do simulátoru nahrán aktuální stav skladu z databáze. Po získání posloupností akcí od A–Star budou akce provedeny reálným skladem a současně budou aplikovány na simulátor. Nakonec bude aktualizována databáze podle simulátoru a celý proces se může opakovat.



Obrázek 4.2. Architektura plánované aplikace skladu

Kapitola 5
Závěr

V úvodní části práce jsme byli seznámeni s řešenou problematikou automatizovaného chaotického skladu deskového materiálu. Byly představeny cíle a problémy, které jsou spojeny s tímto relativně novým typem automatizovaného skladu.

První kapitola se nejprve zaměřuje na rešerši algoritmů vhodných k prohledávání stavového prostoru, jakožto zvoleného přístupu k řešení optimálního pohyby materiálu ve skladu s ohledem na vyskladňování, resp. naskladňování a na přidružené zpracovaní materiálu na paralelně běžících CNC výrobních strojích. Byly definovány metody pro měření výkonu těchto algoritmů. Algoritmy jsou porovnávány podle způsobu práce s pamětí a počtu prohledaných stavů raději něž pomocí času, který je relativní a se stále rostoucím výkonem procesorů by tento přístup nebyl objektivní. V další části této kapitoly byl vytvořen simulátor formální reprezentace stavového prostoru skladu definující stavy, akce, přechody mezi stavy (přesun desky z jednoho stohu na druhý) a konečně definice cílového stavu daného požadovanou frontou desek na výstupní pozici skladu a prázdnou vstupní naskladňovací pozicí. Vzhledem k obrovskému množství kombinací ve skladovém prostoru byl vybrán A–star algoritmus, pro který byla vytvořena heuristika vyhledávající vždy nejméně zarovnané desky a odhadující počet přesunů do cílového stavu. Tím je dosaženo efektivního a rychlého vyskladňování. Nakonec kapitoly byla provedena analýza náročnosti algoritmu na počtu typů desek ve skladu, ze které vyšlo nejlépe s ohledem na prostorovou i časovou složitost deset typů desek při velikosti výstupní fronty šest. V této konfiguraci je vyskladňování nejefektivnější a je potřeba dvojnásobného počtu přesunů na požadovanou frontu. S rostoucím počtem typů desek účinnost počtu vyskladněných desek ku počtu potřebných přesunů klesá.

Druhá kapitola se zabývá rozvrhováním výroby na paralelně běžících strojích, které jsou součástí skladového systému. Protože rozvrhování je rozsáhlá problematika, začátek kapitoly nejdříve obsahuje seznámení s rozvrhováním a přehledem terminologie. Paralelní rozvrhování se řadí mezi $\mathcal{NP}$–hard problémy kombinatorické optimalizace, na jehož řešení byly popsány dva základní přístupy k řešení, Integer linear programming a Constraint programming. V závěru kapitoly byly formulovány matematické modely pro rozvrhování. První model zahrnuje možnost, kdy je ve výrobě více typů strojů. Druhý model je vhodný pro prioritizaci operací, která je docílena stanovením deadlinů skupině operací. Díky tomu, že modely umožňují vyřešení směnného výrobního plánu do méně než jedné sekundy, resp. minuty v případě rozvrhování s deadliny, můžou být v praxi užitečným pomocníkem při plánování výroby a zvyšovat tak ziskovost podniku.

Vytvořený algoritmus vyskladňování spolu se simulátorem skladu je dobře škálovatelný a bylo by vhodné jeho rozšíření v případě použití pro sklady s více pozicemi. Do rozšíření by mohly být vneseny například odhady časů přejezdů manipulátoru a informace o umístění skladových pozic umožnující minimalizaci najeté trajektorie manipulátoru.

Závěrem můžeme říci, že všechny stanovené cíle diplomové práce byly splněny.

Literatura

[1] Jan Ledr a Martin Nečas. *Odborná zpráva o postupu prací a dosažených výsledcích za rok 2022, Chytrý automatický sklad*.

[2] Kevin M. Lynch a Frank C. Park. *MODERN ROBOTICS*. preprint version vyd.. Cambridge: Cambridge University Press, 2017. ISBN 9781107156302.

[3] Stuart J. Russell a Peter Norvig. *Artificial Intelligence: A Modern Approach*. Third Edition vyd.. Upper Saddle River, New Jersey 07458: Pearson Education, Inc., 2010. ISBN 0-13-604259-7.

[4] Richard E. Korf. *Iterative-Deepening-A*: An Optimal Admissible Tree Search.* In: Aravind K. Joshi, eds. *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*. Morgan Kaufmann, 1985. 1034–1036 [cit. 2023-07-04] . https://cse.sc.edu/~mgv/csce580f09/gradPres/korf_IDAStar_1985.pdf.

[5] Geeksforgeeks.org. *Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)*. [cit. 2023-07-08] . https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/.

[6] Wikipedia. *A* search algorithm*. [cit. 2023-07-10] . https://en.wikipedia.org/wiki/A*_search_algorithm.

[7] Zdeněk Hanzálek. *Combinatorial Optimization*. [cit. 2023-07-27] . https://rtime.ciirc.cvut.cz/~hanzalek/KO/index.htm.

[8] Peter BRUCKER. *Scheduling algorithms*. 5th vyd.. New York: Springer, 2007. ISBN ISBN 978-3-540-69515-8. https://ftp.idu.ac.id/wp-content/uploads/ebook/ip/BUKU%20SCHEDULING/Scheduling%20Algorithms.pdf.

[9] Tomáš Werner. *Optimalizace*. [cit. 2023-07-28] . https://cw.fel.cvut.cz/b231/_media/courses/b0b33opt.

[10] Roman BARTÁK. *Constraint Programming*. 1998 [cit. 2023-07-30] . http://ktiml.mff.cuni.cz/~bartak/constraints/.

[11] *Google OR-Tools*. Mountain View (Kalifornie), 2023 [cit. 2023-08-06] . https://developers.google.com/optimization/.

Příloha A
Seznam přiložených souborů

Zdrojové soubory budou přiloženy v archivu `source.zip`, který bude obsahovat soubory uvedené níže a skripty použité pro analýzu výkonnosti `astar.py` v podadresáři `/stats`.

`warehouse.py`	Simulátor formální reprezentace skladu
`astar.py`	Algoritmus pro optimální pohyb materiálu ve skladu
`ilpScheduleSolver.py` `ilpScheduleTypeOfMachinesSolver.py`	Program pro rozvrhování výroby na identických a na dedikovaných strojích
`schedule-plus-astar.py`	Ukázka spojení `astar.py` s rozvrhováním výroby
`ilp-model.py`	ILP model pro rozvrhování s deadliny
`cp-model.py`	CP model pro rozvrhování s deadliny