FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Plánování cest pro roboty v rekonfigurovatelném skladu
Student:	Vladislav Beneš
Vedoucí:	doc. RNDr. Pavel Surynek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Cílem práce bude navrhnout plánovací systém pro logistiku v rekonfigurovatelném skladu, tj. předpokládáme, že transport zboží v rámci skladu zajišťují roboti, pro které je třeba plánovat bezkolizní cesty. Navíc rozmístění zboží ve skladu nemá pevné schéma, ale plánovací systém sám rozhoduje o místě uskladnění zboží tak, aby bylo vzhledem k plánování cest pro roboty co nejvýhodnější, tj. sklad se neustále rekonfiguruje. Úkoly uchazeče budou následující:

1. Seznámí se s literaturou o plánování cest pro mnoho robotů (MAPF), zejména s řešícími algoritmy.

2. Navrhne a implementuje plánovací systém pro hledání cest a rekonfiguraci skladu, předpokládáme, že systém může být postaven na existujícím řešiči pro MAPF [1].

3. Navržený systém otestuje co do výkonnosti v relevantních scénářích.

[1]. Pavel Surynek. Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. Proceedings of the 28th International Joint Conference on AI

Elektronicky schválil/a Ing. Karel Klouda, Ph.D. dne 30. prosince 2021 v Praze.

Bakalářská práce

PLÁNOVÁNÍ CEST PRO ROBOTY V REKONFIGUROVATELNÉM SKLADU

Vladislav Beneš

Fakulta informačních technologií
Katedra znalostního inženýrství
Vedoucí: doc. Ing. Surynek, Ph.D.
22. června 2022

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2022 Vladislav Beneš. Odkaz na tuto práci.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci: Beneš Vladislav. *Plánování cest pro roboty v rekonfigurovatelném skladu.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Obsah

Poděkování vi

Prohlášení vii

Abstrakt viii

Seznam zkratek ix

1 Teoretická část 5
- 1.1 Multiagentní hledání cest . . . 5
  - 1.1.1 Agenti . . . 5
  - 1.1.2 Evaluace MAPF . . . 6
- 1.2 Používané algoritmy . . . 7
  - 1.2.1 CBS . . . 7
  - 1.2.2 MDD-SAT . . . 9
  - 1.2.3 SMT-CBS . . . 11

2 Analytická část 13
- 2.1 Jak funguje automatizovaný sklad . . . 13
  - 2.1.1 Prostor . . . 13
  - 2.1.2 Položky . . . 14
  - 2.1.3 Roboti . . . 14
- 2.2 Skladová anarchie . . . 15

3 Praktická část 21
- 3.1 Vstup . . . 21
- 3.2 Hlavní cyklus . . . 22
- 3.3 Základní struktura . . . 22
  - 3.3.1 Worker Manager . . . 23
  - 3.3.2 Item Manager . . . 23
  - 3.3.3 Graph . . . 24
  - 3.3.4 Storage Anarchy . . . 25
- 3.4 Implementace algoritmů . . . 25
  - 3.4.1 Storage Anarchy . . . 25
  - 3.4.2 CBS . . . 26
- 3.5 Výstup . . . 33

4 Experimentální část 35
- 4.1 Menší rozměry . . . 36
  - 4.1.1 Běžná konfigurace . . . 36
  - 4.1.2 Storage anarchy . . . 37
  - 4.1.3 Porovnání . . . 37
- 4.2 Větší rozměry . . . 38
  - 4.2.1 Běžná konfigurace . . . 38

4.2.2 Storage anarchy . . . 39
4.2.3 Porovnání . . . 39
4.3 Diskuze . . . 41

5 Závěr **43**

Obsah přiloženého média **47**

Seznam obrázků

1 Příklad automatizovaného skladu s běžnou konfigurací rozmístění položek[1] . . . 1

1.1 Příklad průběhu algoritmu na jednoduchém případě, kdy je algoritmus krajně neefektivní[3] . . . 8
1.2 Příklad reprezentace MAPF pro jednoho agenta do TEG, kdy agent začíná ve vrcholu a a jeho cílem je vrchol c pro časové kroky $t = 5$[5] . . . 10
1.3 Příklad zmenšení TEG1.2 [6] . . . 11

2.1 Robot KIVA přenášející regál [9] . . . 14
2.2 Příklad spojení dvou kritických skupin . . . 15
2.3 Příklad rozdílu mezi sousedy a okolními vrcholy . . . 16
2.4 Příklad spojitého skladu . . . 17
2.5 Příklad nespojitého skladu . . . 17

3.1 Příklad výstupu při použití Storage Anarchy . . . 33

4.1 Výsledky při použití běžné konfigurace . . . 36
4.2 Výsledky při použití *Storage Anarchism* . . . 37
4.3 Výsledky při použití běžné konfigurace . . . 38
4.4 Výsledky při použití *Storage Anarchism* . . . 40

Seznam tabulek

Seznam výpisů kódu

3.1 Zjednodušená verze hlavního cyklu . . . 22
3.2 Algoritmus na vybírání náhodného Node s nejmenším počtem volných sousedů . 26
3.3 Hledání vrcholových konfliktů ve vyšší vrstvě algoritmu CBS . . . 28
3.4 Hlavní cyklus vyšší vrstvy algoritmu CBS . . . 29
3.5 Algoritmus na vyhledání řešení v modifikovaném A* . . . 31
3.6 Nižší vrstva algoritmu CBS . . . 32

Chtěl bych poděkovat své rodině a Adéle Kamešové za podporu a pozitivní myšlenky během těžkých chvil.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 22. června 2022 . . . .

Abstrakt

Tato práce se zabývá mutli-agentním vyhledáváním cest (MAPF) pro roboty ve skladu, ve kterém je umisťování položek, které roboti přemisťují, určováno skladovou anarchií. Ta spočíva v tom, že ve skladu nejsou dané pevná místa na ukládání položek, nýbrž tyto pozice jsou vybírány za běhu a mohou tak být určeny pro uskladnění položky nebo čistě jen pro pohyb robotů. Do větší hloubky tu je rozveden algoritmus SMTCBS, který použit pro řešení problému MAPF, a úvaha pro efektivní skladovou anarchii.

Klíčová slova MAPF, SMTCBS algoritmus, rekonfigurovatelný sklad, skladová anarchie, plánování

Abstract

This work is focused on Multi-agent path finding (MAPF) algorithms for robots in warehouses in which places, where the items will be stored, is decided by warehouse anarchy. Warehouse anarchy doesn't have specified places in the warehouse for storing items, but those places are selected in the progress and thez can be specified for storing items or purely for robot movement. The main focus will be on algorithm SMTCBS and ideas for effective warehouse anarchy.

Keywords MAPF, SMTCBS algorithm, reconfigurable storage, anarchy storage, planning

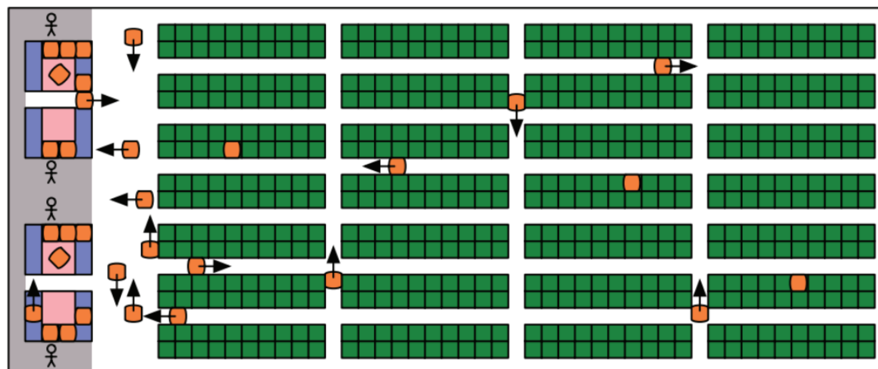
Seznam zkratek

MAPF	Multi agent path finding = Multiagentní vyhledávání cest
SMT	Satisfiability modulo theories
CBS	Conflict based search
SAT	Boolean satisfiability problem
MDD	Multivalue decision diagrams
FIFO	First in first out
TEG	Time expanded graph
MDD	Multi decision diagram

Úvod

Logistika se vždy ukazovala jako velice výnosná a potřebná jakéhokoliv celku, který operuje s větším množstvím položek. Ať se ohlídneme zpět do minulosti, kdy při armádních taženích se musely za armádou tvořit obrovské zástupy vozů, které obsahovaly zásoby pro vojáky, nebo do současnosti, kdy společnosti doručují až 300 tisíc balíků denně svým zákazníkům. Věřím, že spoustu lidí by zajímalo sáhodlouhé povídání o starověkých armádních taktikách, ale tato bakalářská práce je zaměřená na trochu něco jiného a to přesněji automatizované sklady. Roboti, co ve skladu přenáší balíky z místa A na místo B, už nejsou žádnou novinkou. Neunaví se, nebloudí, celkově mají oproti lidem mnohem lepší vlastnosti. Pro navigaci těchto robotů ve skladu se používají algoritmy MAPF (multi-agent path finding). Tyto algoritmy se vyvíjejí a zdokonalují již mnoho let, protože v praxi se mohou využít nejenom pro roboty ve skladu, ale i v jiných oblastech. Rychlá přeprava balíků je jedna věc, ale efektivní skladování a vybírání položek ze skladu je neopomenutelný prvek celého procesu. Kdyby ve skladu bylo zboží jen tak naházeno, samotná přeprava by bez pochyb nemohla být tak rychlá. V této práci se budu zabývat analyzováním více algoritmů na řešení problému MAPF, aplikováním a porovnáním skladové anarchie s běžným přístupem pro skladování položek (každá položka má své dané místo, které se nemění).

Obrázek 1 Příklad automatizovaného skladu s běžnou konfigurací rozmístění položek[1]



Cíl práce

Teoretická část cílem této části je vysvětlit potřebné pojmy pro pochopení problému MAPF a skladové anarchie. Poté rešerše známých algoritmu, které řeší MAPF problém.

Analytická část v této části budu analyzovat problému skladové anarchie a osvětlovat fungování automatizovaného skladu.

Praktická část cílem části praktické je popsání programu, vlasní implementace algoritmu pro skladovou anarchii a řešiče MAPF problému.

Experimentální část zde se zabývám experimenty, analýze získaných dat, porovnáním rekonfigurovatelného skladu s běžnou konfigurací a diskuzí nad výsledky.

Kapitola 1
Teoretická část

1.1 Multiagentní hledání cest

Multiagentní hledaní cest je problém, který se zaobývá vyhledáváním cest pro více robotů ve stejném prostoru, aniž by došlo ke kolizi. Tento problém se reprezentuje pomocí grafu.

▶ **Definice 1.1.** *Klasický MAPF je definován jako trojce*

$$\langle G, s, g \rangle$$

- $G = (V, E)$ *je neorientovaný graf, kde* V *je množina vrcholů, které mohou agenti okupovat, a* E *je množina hran, po kterých se agenti pohybují mezi jednotlivými vrcholy, přičemž každý vrchol grafu* G *náleží množině* V *a každá hrana grafu* G *náleží množině* E.
- $s : [1...k] \rightarrow V$ *mapuje agenta na počáteční vrchol*
- $g : [1...k] \rightarrow V$ *mapuje agenta na konečný vrchol*

1.1.1 Agenti

U MAPF je běžné, že se v grafu vyskytuje více agentů. Kdyby tomu tak nebylo a v grafu by se vyskytoval pouze jeden agent, z MAPF problému by se stal klasický problém vyhledávání cesty v grafu. Proto je potřeba si zadefinovat agenty, jejich akce a pravidla.
Nechť jsou definováni agenti $a_0 \ldots a_k, a \in A$, kdy $k \in N, k \leq |V|$.

- Každý agent okupuje v určitý čas právě jeden vrchol
- Vrchol $v \in V$ nemůže okupovat více než jeden agent $a \in A$ v daný časový krok
- Počet robotů nesmí být vyšší než počet vrcholů kvůli tomu, že roboti nesmějí sdílet dohromady jeden vrchol (viz. pigeonhole principle)

Akce

Předpokládáme, že čas je diskrétní a je rozdělen do **časových kroků**. Každý agent provede právě jednu akci za každý jeden krok. **Akce** agentů se dají rozdělit na dva základní druhy:

- **Pohyb**
  mějme funkci $a(v) = v', v \in V, v' \in V$, kde pokud je ve vrcholu v agent a a vrchol v je spojen s vrcholem v' libovolnou hranou, po uplynutí jednoho časového kroku se agent a přemístí z vrcholu v do vrcholu v'.
- **Čekání**
  mějme funkci $w(v) = v, v \in V$ kde pokud je ve vrcholu v agent a, po uplynutí jednoho časového kroku agent a zůstane v původním vrcholu v.

Plán

Pro sekvenci akcí $\Pi_i = (a_1 \dots a_n)$ agenta i značíme $\Pi_i[x]$ jako lokaci, na které se agent nachází po provedení x akci z Π_i. Formálně $a_x(a_{x-1}(\dots a_1(s(i))))$. Sekvence Π je plánem (resp řešením) agenta i, pokud sekvence začne ve startovním vrcholu $s(i)$ a po jejím dokončením se agent i nachází ve vrcholu $g(i)$. Řešením celého problému je soubor k plánů, kdy plány nejsou v konfliktu.

Konflikt

Mějme dva plány Π_i a Π_j. Mezi plány může dojít k více druhům konfliktů:

- **Vrcholový konflikt**
  nastane právě tehdy, pokud v obou plánech agenti okupují stejné místo ve stejný čas, formálně
  $$\Pi_i[x] = \Pi_j[x]$$
  , kdy x je časový krok
- **Hranový konflikt**
  nastane právě tehdy, pokud si agenti mají ve stejný čas prohodit své pozice, formálně
  $$\Pi_i[x] = \Pi_j[x+1] \wedge \Pi_i[x+1] = \Pi_j[x]$$
  , kdy x je časový krok

1.1.2 Evaluace MAPF

Algoritmy pro řešení MAPF problémů se mohou lišit v efektivnosti. Abychom dokázali efektivně rozlišit, jaký algoritmus odvedl lepší práci, používáme k měření tyto dvě funkce:

- **Makespan**
  počet časových kroků, které jsou potřeba, než všichni agenti dorazí ze svých startovních pozic do cílových pozic, tj.
  $$\mu = \max 1 \leq i \leq |\Pi_i|$$
- **Sum of costs**
  suma všech časových kroků každého agenta nutných k tomu, aby se každý agent dostal ze své startovní pozice do své cílové pozice, tj.
  $$\xi = \sum_{i=1}^{k} |\Pi_i|$$

1.2 Používané algoritmy

Pro řešení MAPF problémů je vymyšleno už nespočet algoritmů. V této práci se budu hlavně zabývat algoritmem SMT-CBS, který vznikl zkombinováním MDD-SAT a CBS algoritmu (jak už vyplývá z názvu). Nejdříve vysvětlím algoritmus CBS, poté MDD-SAT a nakonec objasním jejich zkombinování. Informace k používaným algoritmům jsem čerpal především z článků pana Surynka. [2]

1.2.1 CBS

Definice

Algoritmus CBS (conflict-based search) zaručuje optimální řešení pro instance MAPF problémů. Rozděluje se na dvě vrstvy – ve vyšší úrovní pracuje s binárním stromem, jehož vrcholy znázorňují jednotlivé řešení a v nižší úrovni hledá cesty pro jednotlivé agenty.

- **Vyšší úroveň**
  Jak je už předem řečeno, na vyšší úrovni se pracuje s binárním stromem T, kdy každý vrchol N ve stromu znázorňuje jedno možné řešení. N se skládá z více částí:
  - Množiny omezení $N.omezeni$, která obsahuje soubor trojcí (a_i, v, t), kdy a_i znázorňuje určitého agenta, který se nesmí nacházet ve vrcholu v během určitého časového kroku t
  - Množina k plánů $N.plany$, kdy k je počet agentů figurujících v řešené instanci MAPF problému
  - *sum of costs* plánů $N.\xi$, které jsou obsaženy v $N.plany$
- **Nižší úroveň**
  Odehrává se v jednotlivých vrcholech stromu T, kde ve vrcholu N_i vyhledává daný algoritmus (většinou se používá A*, může se ale použít jakýkoliv algoritmus, který najde řešení pro jednoho agenta) nejkratší cestu pro daného agenta a_i z jeho počátečního vrcholu $s(i)$ do konečného vrcholu $g(i)$ s ohledem na množinu omezení $N_i.omezeni$.

Průběh

Nazačátku máme pouze kořen N binárního stromu T, který má prázdnou množinu omezení $N.omezeni$. Pro každého agenta a_i figurujícího v MAPF problému, který byl zadán, se najde plán Π_i a přidá se do množiny plánů $N.plany$. Následně se zkontroluje bezkoliznost všech plánů v $N.plany$ Pokud jsou plány bezkolizní, algoritmus skončí a vrátí množinu plánů v $N.plany$ jako řešení.
Pokud je však kolize nalezena, budou vytvořeni dva noví následovníci N_i a N_j. Pokud je kolizí v množině plánů vrcholu N nalezeno víc, je brána v potaz jen první nalezená kolize. Této první nalezené kolizi mezi agenty a_i a a_j ve vrcholu v v čase t lze předejít tak, že jeden z agentů a_i nebo a_j nepodstoupí krok do vrcholu v v čase t. Toho docílíme přidáním omezení do nově vytvořených následovníků $N - N_i$ a N_j. Tyto následovníky přesněji vytvoříme následovně:

1. U nového vrcholu N_i zkopírujeme množinu omezení z N a přidáme do ní omezení (a_i, v, t)

$$N_i.omezeni = N.omezeni \cup (a_i, v, t)$$

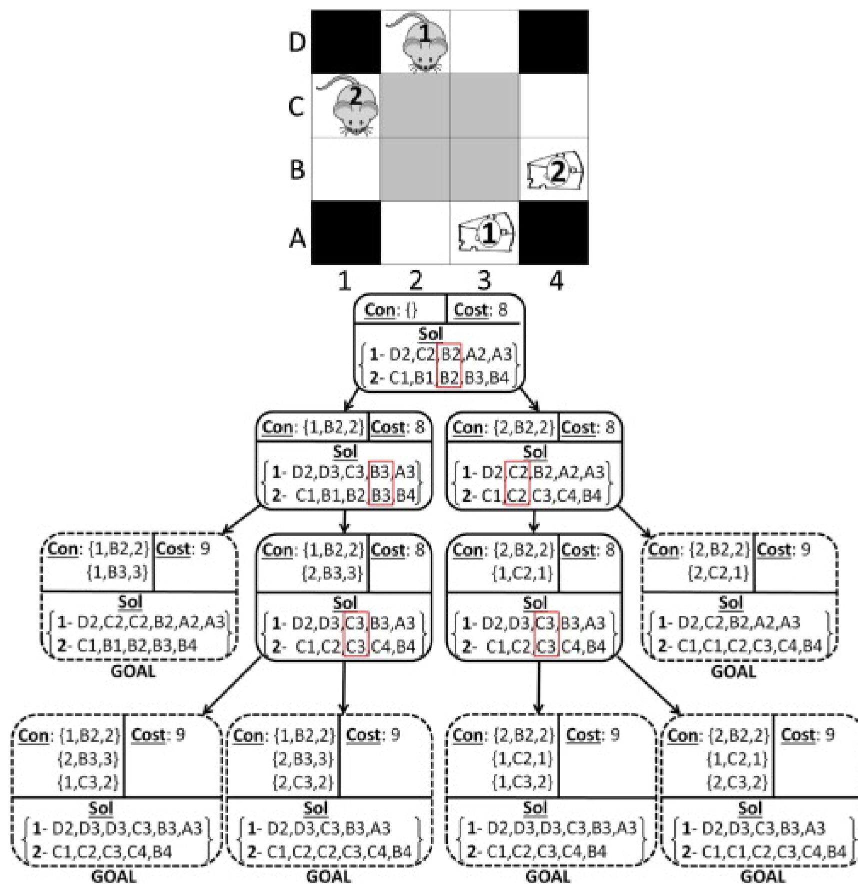
2. U nového vrcholu N_j zkopírujeme množinu omezení z N a přidáme do ní omezení (a_j, v, t)

$$N_j.omezeni = N.omezeni \cup (a_j, v, t)$$

3. Množinu plánů z N zkopírujeme do nově vytvořeného vrcholu N_i resp. N_j, najdeme nový plán pro agenta a_i resp. a_j
4. Vypočítáme nový *sum of costs* pro daný vrchol
5. Zařadíme nově vytvořené N_i a N_j do fronty, kde jsou zařazeny podle velikosti *sum of costs*

Jednotlivé vrcholy N jsou vkládány do fronty $OPEN$, kde jsou řazeny podle velikosti *sum of costs*. Každý krok algoritmu CBS vezme vrchol N_i z fronty $OPEN$ s nejmenším *sum of costs* a zjistí, zda je množina plánů ve vrcholu N_i bezkolizní. Pokud není, vytvoří dva nové následovníky a pokračuje dál.

Obrázek 1.1 Příklad průběhu algoritmu na jednoduchém případě, kdy je algoritmus krajně neefektivní[3]



▶ **Tvrzení 1.2.** *Algoritmus CBS vždy nalezne optimální řešení, pokud takové řešení existuje.*

Algorithm 1 CBS pseudokód

```
1: procedure CBS(G = (V,E), A, s, g)
2:     K.omezeni ← ∅
3:     R.plany ← {nejkratší cesta z s(i) do g(i) pro agenta a_i | i = 1, 2, ..., k}
4:     R.ξ = ∑_{i=1}^{k} ξ(R.plany(a_i))
5:     vlož R do fronty OPEN
6:     while OPEN ≠ ∅ do
7:         N ← min(OPEN)
8:         odstraň N z OPEN
9:         kolize ← najdi kolize v N.plany
10:        if kolize = ∅ then
11:            vrať N.paths jako výsledek
12:        (a_i, a_j, v, t) ∈ kolize
13:        for každé a ∈ a_i, a_j do
14:            N'.plany ← N.plany
15:            N'.omezeni ← N.omezeni ∪ (a, v, t)
16:            N'.plany ← najdi nový plán pro a s ohledem na N'.kolize
17:            N'.ξ = ∑_{i=1}^{k} ξ(N'.plany(a_i))
18:            vlož N' do fronty OPEN
```

1.2.2 MDD-SAT

Algoritmus MDD-SAT zaručuje optimální řešení pro instance MAPF problémů. Jeho hlavní myšlenkou je vytvoření výrokové formule $\mathcal{F}(\xi)$, která není kontradikce právě tehdy, když existuje řešení daného problému MAPF u kterého platí, že *sum of costs* $= \xi$. Můžeme říct, že $\mathcal{F}(\xi)$ je **kompletní výrokový model** MAPF problému, pokud platí:

$$\mathcal{F}(\xi) \text{ není kontradikce} \Leftrightarrow \text{MAPF má řešení se } \textit{sum of costs } = \xi$$

Při vytvoření takové formule $\mathcal{F}(\xi)$, je možné najít optimální řešení, pokud budeme zkoušet formule $\mathcal{F}(\xi_0)$, $\mathcal{F}(\xi_0 + 1)$, $\mathcal{F}(\xi_0 + 2)$, ..., dokud nenajdeme výrokovou formuli $\mathcal{F}(\xi)$, která nebude kontradikce. S hledáním začínáme od ξ_0, kdy $\xi_0 =$ součet délek nejkratších cest všech robotů.

Tvorba formule

K tvorbě formule a jejímu následnému otestování je potřeba znázornit MAPF problém jako výrokovou formuli. Toho docílíme vytvořením *time expanded graph* (TEG)[4], orientovaného acyklického grafu, který reprezentuje vrcholy a hray grafu G v každém možném časovém kroku. Tuto reprezentaci si vytvoříme pro každého agenta zvlášť.1.2 Největší výhodou tohoto algoritmu je právě to, že využívá externí řešiče pro zjištění uspokojitelnisti dané formule.

- **Vrcholy**
  Pro každý vrchol $v \in V$ si vytvoříme vrchol v^t závislý na časovém kroku $t = 0, 1, 2, \ldots, \mu$.
  Pro každý vrchol v^t vytvoříme proměnnou

  $$\mathcal{V}_v^t(a_i)$$

  Tato proměnná je *PRAVDA* právě tehdy, kdy agent a_i okupuje vrchol v v časovém kroku t.

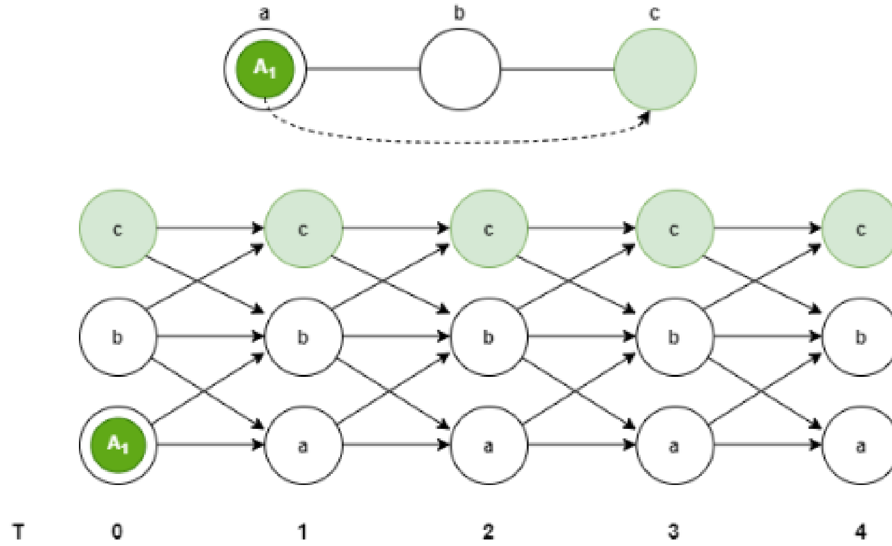
- **Hrany**
  Pro každou hranu $e = \{u, v\}$, kdy $e \in E$ a $u, v \in V$, si vytvoříme orientovanou hranu (u^t, v^{t+1}) závislou na časovém kroku $t = 0, 1, 2, \ldots, \mu - 1$. Každá takto znázorněná hrana reprezentuje

pohyb agenta v čase t, proto je potřeba reprezentovat i čekání (u^t, u^{t+1}).
Pro každou orientovanou hranu (u^t, v^{t+1}) vytvoříme proměnnou

$$\mathcal{E}^t_{u,v}(a_i)$$

Tato proměnná je $PRAVDA$ právě tehdy, kdy agent a_i cestuje hranou $\{u, v\} \in E$ během přechodu z časového kroku t do časového kroku $t + 1$.

Obrázek 1.2 Příklad reprezentace MAPF pro jednoho agenta do TEG, kdy agent začíná ve vrcholu a a jeho cílem je vrchol c pro časové kroky $t = 5$[5]



Pro úspěšnou tvorbu reprezentace MAPF problému do formule, je potřeba přidat omezení. Po přídání omezení můžeme prohlásit, zda $\mathcal{F}(\xi)$ je kompletní výrokový model daného MAPF problému.
Na ukázku zde popíšu příklad jen jednoho omezení. Pokud by čtenář měl zájem se dozvědět více, odkazuji na detailnější článek o tomto tématu. [2]
Jak již víme, určitý vrchol nesmí v daný časový krok okupovat více než jeden agent. 1.1.1 Budeme tedy muset přidat takové omezení, aby v případě výskytu kolize ve formule $\mathcal{F}(\xi)$ neuznala jako kompletní výrokový model. Takové omezení v časovém kroku t lze zapsat

$$\sum_{a_i \in A | v^t \in V} \mathcal{V}^t_v(a_i) \leq 1$$

a následovně zapsat jako klausuli

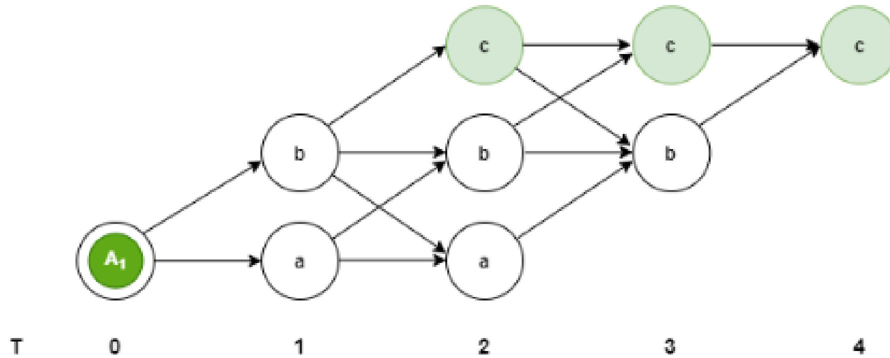
$$\neg\mathcal{V}^t_v(a_i) \vee \neg\mathcal{V}^t_v(a_j)$$

pro všechny možné dvojce agentů $a_i \in A$ a $a_j \in A$.

Pro zmenšení formule lze vzít v potaz dostupnost jednotlivých vrcholů pro agenty v určitý časový krok. Každý agent má dostupné jenom ty vrcholy v, které jsou vzdálené od počátečního vrcholu $s(i)$ resp. konečného vrcholu $g(i)$ agenta a_i maximálně d, kdy d je počet kroků, které jsou potřeba pro dosažení vrcholu v po nejkratší ceste z $s(i)$ resp. $g(i)$ do v. Po takovém zmenšení TEG1.2 se graf nazývá MDD (*Multi-valued Decision Diagram*)1.3.

▶ **Tvrzení 1.3.** *Algoritmus MDD-SAT na bázi SAT řešiče vždy nalezne optimální řešení, pokud takové řešení existuje.*

Obrázek 1.3 Příklad zmenšení TEG1.2 [6]



Algorithm 2 Pseudokód algoritmu pro MAPF na bázi SAT řešiče

1: **procedure** MAPF-SAT(G = (V,E), A, s, g)
2: *plany* ← {nejkratší cesta z s(i) do g(i) pro agenta $a_i | i = 1, 2, \ldots, k$}
3: $\xi = \sum_{i=1}^{k} \xi(plany(a_i))$
4: **while** $OPEN \neq \emptyset$ **do**
5: $\mathcal{F}(\xi)$ ← vytvoř kandidáta na kompletní výrokový model(ξ, G, A, s, g)
6: *řešení* ← SAT řešič$(\mathcal{F}(\xi))$
7: **if** *řešení* ≠ *kontradikce* **then**
8: *plany* ← získej plány(*řešení*)
9: vrať *plany* jako výskedek
10: $\xi = \xi + 1$

1.2.3 SMT-CBS

Kombinace si bere od obou algoritmů jejich efektivní část a nahrazuje tím jejich neefektivitu.
U algoritmu CBS je nejvíce náročné větvení, kdy optimální řešení už mohlo být nalezeno, ale algoritmus i nadále hledá efektivnější řešení.
U MDD-SAT algoritmu je největší přítěží to, že formule vytvořená z omezení TEG1.2 všech agentů i po zmenšení na MDD1.3 je obrovská a zjištění, zda je daná formule kontradikce, trvá externímu řešiči SAT dlouho. Každý cyklus k dané formuli ještě přidává další omezení, takže řešení je více a více složitější.
Algoritmus SMT-CBS má stejně jako CBS dvě úrovně:

- **Vyšší úroveň**
  Ve vyšší úrovni se narozdíl od CBS nepracuje s binárním stromem. Začne se inicializací, ve které se najde soubor nejkratších plánů pro roboty bez ohledu na kolize a poté se vypočítá *sum of costs* těchto plánů. Inicializuje se prázdná množina konfliktů, která se následovně bude plnit pomocí nižší úrovně.
  Následovně se použije cyklus, který připomíná hlavní cyklus v algoritmech založených na SAT řešičích2 — po každé jeho iteraci se zkontroluje, zda nižší úroveň našla platné řešení, pokud ne, inkrementuje se *sum of costs*.

- **Nižší úroveň**
  V nižší úrovni se nejdříve vytvoří formule $\mathcal{F}$, která bude kandidátem pro kompletní výrokový model daného MAPF problému. Rozdíl při tvoření formule $\mathcal{F}$ je takový, že narozdíl od tvoření formule v algoritmu MDD-SAT1.2.2 se do formule $\mathcal{F}$ nezahrnujou všechny možné kolize mezi

agenty, ale jen takové kolize, které byly detekovány již v předešlých iteracích.
Pokud $\mathcal{F}$ není kontradikce, extrahuje se z ní soubor plánů pro agenty, který je následně ověřen, zda v něm nejsou kolize. Pokud nejsou, algoritmus vrátí soubor plánů jako řešení. Pokud však jsou kolize nalezeny, algoritmus je zohlední v dalších iteracích.

Algorithm 3 SMT-CBS pseudokód

1: **procedure** SMT-CBS-Vyssi(G = (V,E), A, s, g)
2: $kolize \leftarrow \emptyset$
3: $plany \leftarrow$ {nejkratší cesta z s(i) do g(i) pro agenta $a_i | i = 1, 2, \ldots, k$}
4: $\xi = \sum_{i=1}^{k} \xi(plany(a_i)$
5: **while** $PRAVDA$ **do**
6: $(plany, kolize) \leftarrow$ SMT-CBS-Nizsi$(kolize, \xi, G, A, s, g)$
7: **if** $plany \neq kontradikce$ **then**
8: vrať $plany$ jako výskedek
9: $\xi = \xi + 1$
10:
11: **procedure** SMT-CBS-Nizsi(kolize, ξ, G = (V,E), A, s, g)
12: $\mathcal{F}(\xi) \leftarrow$ vytvoření kandidáta na kompletní výrokový model s ohledem na kolize$(kolize, \xi, G, A, s, g)$
13: **while** PRAVDA **do**
14: *řešení* $\leftarrow$ SAT řešič$(\mathcal{F}(\xi))$
15: **if** *řešení* $\neq$ *kontradikce* **then**
16: $plany \leftarrow$ získej plány$(\mathcal{F}(\xi))$
17: $kolize \leftarrow$ najdi kolize v $plany$
18: **if** $kolize \neq \emptyset$ **then**
19: vrať $(plany, kolize)$
20: $(a_i, a_j, v, t) \in kolize$
21: **for** každé $a \in a_i, a_j$ **do**
22: $\mathcal{F}(\xi) \leftarrow \mathcal{F}(\xi) \cup \neg\mathcal{V}_v^t(a_i) \vee \neg\mathcal{V}_v^t(a_j)$
23: $konflikty \leftarrow konflikty \cup \{(a_i, v, t), (a_j, v, t)\}$
24: **else**
25: vrať $(kontradikce, kolize)$

Kapitola 2

Analytická část

V této části se budeme věnovat vysvětlení, jak sklad funguje, co je vůbec myšleno spojením skladový anarchismus a jak by se dal implementovat.

2.1 Jak funguje automatizovaný sklad

Pro pochopení části praktické a úvah nad skladovým anarchismem, je potřeba si říct, jak takový automatizovaný sklad funguje. Není dané pravidlo, jak by měl takový sklad fungovat, ale v mé bakalářské práci si budu brát inspiraci z fungování automatizovaných skladů společnosti AMAZON.[7][8]

2.1.1 Prostor

Pro skladování položek je potřeba místo k tomu určené. Takový sklad má (není to pravidlem) čtvercový či obdelníkový půdorys. Celý prostor většinou nemá žádné překážky, které by bránily robotům v pohybu, takže celý prostor je možné využít na skladování či pohyb. Je velice ojedinělé, že by lidé chodili v samotném skladu mezi roboty. Pokud je ale potřeba lidské pomoci, pracovník si vezme speciální vestu, která zaručí, že se roboti budou danému pracovníkovi s vestou vyhýbat. Ve skledu jsou dezignované dvě speciální místa:

- **Nabíjecí stanice**
  Nabíjecí stanice slouží k účelu samo o sobě vypovídajícím – roboti si zde dobíjejí energii pro další fungování. Toto je také jedno z mála míst, kde se častěji vyskytují lidé. V nabíjecích stanicích se roboti berou i na případnou plánovanou údržbu či opravy.

- **Pickery**
  Takzvané *pickery* jsou stanice, které ve skladu obsluhují lidé. Roboti právě sem na vyžádání pracovníka dané stanice, přivezou žádaný regál. Dovezené regály jsou buď samotným pracovníkem doplněny, nebo si pracovník naopak vezme žádané zboží z regálu. Po vykonání potřebné akce pracovníkem stanice dá robotovi příkaz k uklizení regálu zpět na své místo.

Prostor je rozdělen na čtverce o velikosti jednoho regálu. Každý čtverec je označen unikátním QR kódem. Díky těmto QR kódům se mohou roboti ve skladu jednoduše orientovat.

Běžná konfigurace

Konfigurace skladu hraje v efektivitě využití prostoru a pohybu robotů v prostoru zásadní roli. Nejběžnější konfigurací skladu jsou položky umístěny v obdelníkových půldorysech o velikosti $2 \times Y$. Na referenčním obrázku jsou půldorysy položek o velikosti 2×10.1 Mezi těmito půldorysy je volný prostor o velikosti 1 a více robotů, aby se zachovala *spojitost skladu*. 2.1

2.1.2 Položky

Samotné položky ve skladu jsou umisťovány do vysokých regálů, které jsou ze spodu označené QR kódem. Každý regál je při umístění na dané místo ve skladu zapsán do databáze, která mapuje daný QR kód regálu na QR kód daného místa ve skladu. V databázi je také zaznamenáno, jaký regál obsahuje které položky po kolika kusech.
Regál z důvodu efektivity obsahuje více druhů položek. Kdyby regál obsahoval pouze jeden druh položky, při vyžádání dané položky více *pickery* najednou by musel každý *picker* čekat, než *picker*, který si položku vyžádal před ním, dokončí svůj úkon. Proto je po skladu rozmístěno více regálů s malým množstvím dané položky. Toto by se dalo říct stejně po všechy možné položky, které jsou ve skladu skladovány. Proto by se dalo říct, že regály v sobě mají téměř náhodné položky.

2.1.3 Roboti

Samotní roboti ve skladu slouží jen k převezení regálu s položkami z místa A na místo B. Při vyžádání dané položky daným *pickerem* je robot navigován na místo, kde je regál s žádanou položkou umístěn. Po příjezdu robot nadzvihne regál a přiveze ho na žádané místo.
Každý robot ve skladu je omezen pouze jedním pravidlem, a to jest že se musí vyvarovat kolizím. Ať to je kolize s dalším robotem, s lidským pracovníkem nebo s regálem. Kolizi s dalším robotem se dá předejít spolehlivým MAPF řešičem.
Možnost kolize s lidským pracovníkem eliminuje kamera, kterou robot disponuje, na snímání okolí a případným zastavením, pokud je zaznamenána speciální vesta pracovníka.
Kolize s regálem může vzniknout pouze v případě, že robot již převáží jiný regál. Pokud je robot bez regálu, může bez sebemenšího problému jakýkoliv regál podjet.

Obrázek 2.1 Robot KIVA přenášející regál [9]

2.2 Skladová anarchie

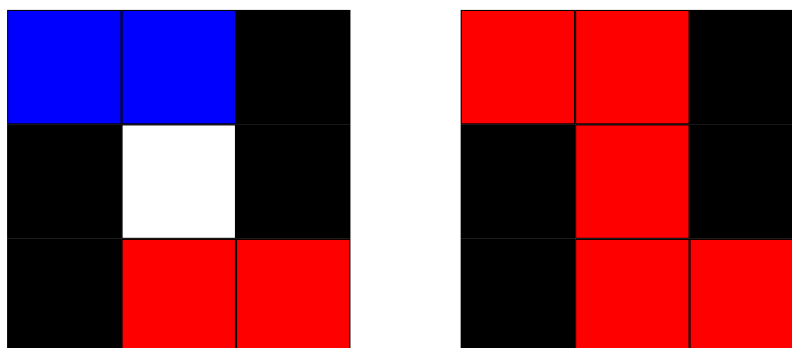
Běžná konfigurace skladu 2.1.1 efektivně řeší pohyb robotů. Většinu času se pohybují jen v rovně a zatáčí minimálně. S efektivitou využití prostoru se to už ale říct nedá. Proto cílem mé práce bylo vyvinout novou konfiguraci skladu, která by mohla potenciál úložného prostoru využít lépe než běžná konfigurace a zachovat si efektivitu pohybu.
Hlavní myšlenkou tohoto návrhu je realizovat sklad s takovou konfigurací, která nemá určené pozice pro uskladňování a pozice pouze pro pohyb, nýbrž udělat takovou konfiguraci, která sama rozhoduje, jakou pozici využije k uskladnění a jakou pouze k pohybu. Tato konfigurace stále musí zaručovat *spojitost skladu.* 2.1

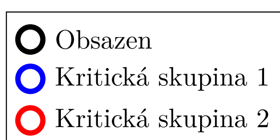
Reprezentace skladu

Pro realizaci nové funkční konfigurace je nejprve pořeba si sklad přetransformovat na graf. Pro naše účely budeme graf reprezentovat jako mřížku, kdy každé její pole (resp. vrchol) bude mít následující parametry:

- **Pozice**
  Reprezentuje pozici vrcholu v mřížce souřadnicemi x (sloupec) a y (řádek)
- **Obsazenost**
  Určuje, zda je daný vrchol využit pro uskladnění položky, či je dané pouze pro pohyb robotů
- **Kritičnost**
  Pokud je vrchol kritický, není možné na daný vrchol umístit položku, jinak by se narušila *spojitost skladu* 2.1
- **Kritická skupina**
  Jestli je vrchol kritický, tento parametr určuje, v jakém kritickém shluku se daný vrchol vyskytuje. Pokud je vrchol označen jako kritický, automaticky vytvoří novou kritickou skupinu. Jestli má daný nový kritický vrchol libovolného souseda také kritického, obě kritické skupiny se sloučí do jedné.
  Sousedi kritické skupiny (tzv. **volní sousedi**) je množina, která obsahuje všechny **nekritické** sousedy vrcholů patřící do dané kritické skupiny. 2.2

Obrázek 2.2 Příklad spojení dvou kritických skupin





- **Sousedi** Každý vrchol má své **sousedy**. Za sousedy vrcholu P považujeme množinu sousedících vrcholů, které se líší souřadnicemi právě o 1.

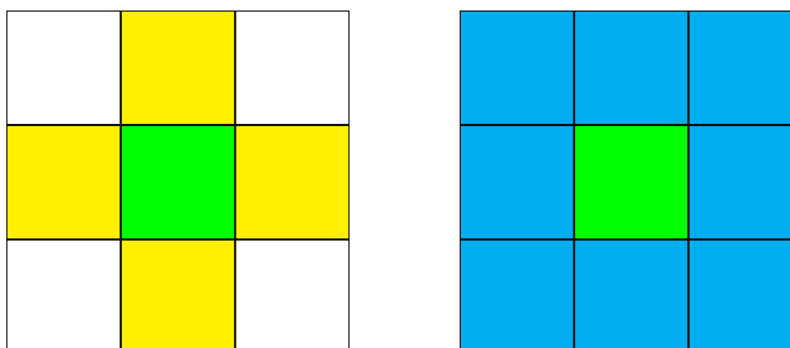
$$P.sousedi = \{(P.x + 1, P.y), (P.x, P.y + 1), (P.x - 1, P.y), (P.x, P.y - 1)\}$$

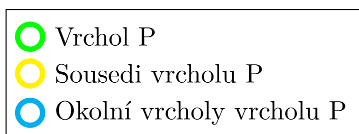
Za sousedy nepovažujeme vrcholy diagonálně umístěné od daného pole 2.3

- **Okolní vrcholy** Každý vrchol v mřížce má také **okolní vrcholy**. Okolní vrcholy jsou množina sousedních vrcholů vrcholu P rozšířená o ty vrcholy, které diagonálně sousedí s vrcholem P 2.3

$$P.okolniVrcholy = \{(P.x + 1, P.y), (P.x, P.y + 1), (P.x - 1, P.y), (P.x, P.y - 1), (P.x + 1, P.y + 1), (P.x + 1, P.y - 1), (P.x - 1, P.y + 1), (P.x - 1, P.y - 1)\}$$

Obrázek 2.3 Příklad rozdílu mezi sousedy a okolními vrcholy





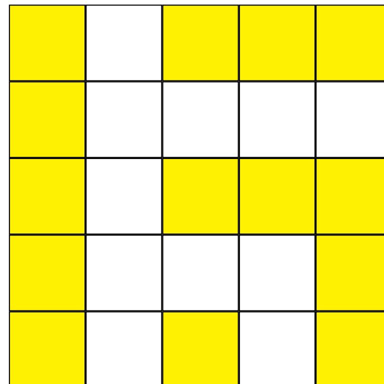
Průchodnost skladu

Položky se většinou do prostoru umisťují podle pravidel dané konfigurace prostoru. Jsou vymezeny přesně místa, kam se mohou položky umístit. 1 Pro efektivitu přepravy je možné každou položku přenést z libovolného místa v prostoru na jiné libovolné místo ve stejném prostoru bez nutnosti posouvání jiné položky. Takovému prostoru budeme říkat **spojitý sklad**.
Jinak řečeno, pokud prostor převedeme na orientovaný graf $G = (V, E)$, kdy každý čtverec označen QR kódem 2.1.1 je vrchol $v \in V$. U každého vrcholu si evidujeme, zda obsahuje či neobsahuje položku. Mezi vrcholy $v_1 \in V$ a $v_2 \in V$ zapíšeme hranu $(v_1, v_2), (v_2, v_1) \in E$, pokud spolu vrcholy sousedí a ani jeden z vrcholů neobsahuje položku. Pokud však jeden z vrcholů v_1, v_2 obsahuje položku, povedeme hranu jen z vrcholu bez položky do vrcholu s položkou. Jestli položku obsahují oba dané vrcholy, hrany neevidujeme. U diagonálně sousedících vrcholů hrany také nezapisujeme.

▶ **Definice 2.1** (Spojitý sklad)**.** *Prostor je* ***spojitý sklad*** *právě tehdy, kdy lze prostor převést na orientovaný, slabě souvislý graf G, a pokud odebereme všechny vrcholy obsahující položku, graf G bude silně souvislý.*

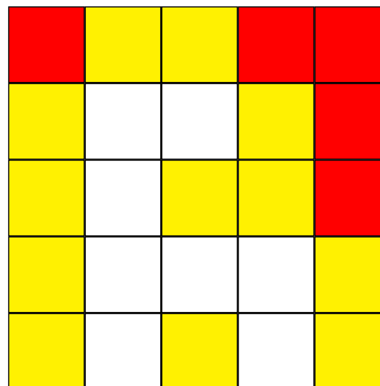
2.4 2.5

Obrázek 2.4 Příklad spojitého skladu



Obsazené dostupné vrcholy

Obrázek 2.5 Příklad nespojitého skladu



Obsazené dostupné vrcholy
Obsazené nedostupné vrcholy

Algoritmus

Při vytváření algoritmu je potřeba dát velký důraz na to, že se za každou cenu musí udržet spojitost skladu. Pokud by sklad spojitý nebyl, efektivnost pohybu robotů ve skladu by se drasticky snížila. Aby byla splněna požadovaná rekonfigurovatelnost, je potřeba mít v algoritmu prvek náhody, díky kterému každá nově vytvořená konfigurace algoritmem může být jiná.

▶ Poznámka 2.2. Podmínkou pro fungování skladové anarchie je sklad, který je na počátku algoritmu spojitý sklad.

Algoritmus má dvě vrstvy, horní a dolní.

- **Horní vrstva**
  Horní vrstva se stará o vybírání správného kandidáta na umístění položky. Na počátku se v horní vrstvě inicializuje fronta F, do které jsou vloženy všechny neobsazené a nekritické vrcholy $v \in V$ orientovaného grafu $G = (V, E)$, který reprezentuje sklad. Vrcholy ve frontě F se seřadí vzestupně podle velikosti počtu neobsazených sousedů. Tato fronta se udržuje aktuální během tvoření nové konfigurace skladu.
  Z fronty F se vybere náhodně jeden vrchol k, který má nejméně neobsazených sousedů. Vrchol k musí splňovat tři podmínky, aby mohl být vybrán jako další vrchol pro uskladnění v nové konfiguraci aniž by porušil skladovou souvislost.

  1. V první podmínce (4 řádek 11-14) se zkontroluje, zda vrchol k nemá mezi jeho sousedy $k.sousedi$ takový vrchol, který je obsazený a má pouze jednoho neobsazeného souseda, kterýmž je vrchol k. Pokud tomu tak je, vrchol k je poslán do dolní vrstvy, kde je označen za kritický a vyhozen z fronty F. Následně se vybere nový vrchol k s nejmenším počtem neobsazených sousedů z fronty F.

  2. V druhé podmínce (4 řádek 15-18) se znovu vyberou sousedi vrcholu k. Pokud je mezi sousedy $k.sousedi$ vrchol s, který je kritický, zkontroluje se kritická skupina, ke které vrchol s patří. Jestli má kritická skupina $s.kritickaSkupina$ vrcholu s pouze jednoho volného souseda 2.2, kterýmž je vrchol k, je vrchol k poslán do dolní vrstvy, kde je označen za kritický, vyhozen z fronty F a poté je vybrán nový vrchol k s nejmenším počtem neobsazených sousedů z fronty F.

  3. Třetí podmínka (4 řádek 19-26) je výpočetně nejnáročnejší, proto se provádí až po prvních dvou podmínkách. Vezme se množina vrcholů okolních vrcholů vrcholu k, ze které se vyloučí každý vrchol, který není obsazen.

     $$O = k.okolniVrcholy \setminus \{\neg(v.obsazen) \in k.okolniVrcholy\}$$

     Pro každý vrchol v množině O se provede následující kontrola: Jestli je absolutní hodnota z rozdílu pozice řádku resp. sloupce vrcholu $v_o \in O$ s každým dalším vrcholem z množiny O vyšší než 1, zavolá se algoritmus DFS, který zkontroluje souvislost grafu, pokud je vrchol k obsazen. Pokud DFS vyvrátí souvislost, je vrchol k poslán do dolní vrstvy, kde je označen za kritický, vyhozen z fronty F a poté je vybrán nový vrchol k s nejmenším počtem neobsazených sousedů z fronty F.

  Pokud je vrchol k po průchodu všemi třemi podmínkami stále nekritický, může na něj být umístěna položka, aniž by se porušila skladová souvislost.

- **Dolní vrstva**
  Dolní vrstva označuje vrcholy za kritické, pokud tomu je tak potřeba, aby se udržela spojitost skladu.2.1 Přijímá vrchol k, který je potřeba označit jako kritický z horní vrstvy. Daný vrchol

označí za kritický, vytvoří mu samostatnou novou kritickou skupinu a vyhodí ho z fronty F, která se vyskytuje v horní vrstvě.
Následně zkontroluje pro každý vrchol $s \in k.sousedi$, zda je s kritický. Pokud tomu tak je, sjednotí tyto dvě kritické skupiny. Sloučení skupin se týka všech kritických vrcholů, které patří do skupiny $s.kritickaSkupina$ a $k.kritickaSkupina$. Poté se stejným způsobem zkontrolují i zbývající nezkontrolované vrcholy $s \in k.sousedi$.
Jako poslední zkontrolujeme počet volných sousedů 2.2 kritické skupiny $k.kritickaSkupina$. Pokud má $k.kritickaSkupina$ právě jednoho volného souseda s_k, pošleme s_k do dolní vrstvy, kde ho označíme za kritický.

► **Tvrzení 2.3.** *Algoritmus Anarchy vždy nalezne konfiguraci pro spojitý sklad. Nalezená konfigurace vždy vytvoří spojitý sklad.*

Při odebírání položky ze skladu je potřeba označit vrchol x, na kterém daná položka byla umístěna, a jeho sousedy, kteří jsou neobsazeni, jako nekritické a zařadí se zpět do fronty F. Pokud jakýkoliv z vrcholů s z $x.sousedi$ rozdělí jeho kritickou skupinu $s.kritickaSkupina$ na dvě skupiny U a V, kdy se tyto skupiny nedotýkají, vytvoří se dvě nové na sobě nezávislé kritické skupiny.

Algorithm 4 Skladová anarchie pseudokód

```
1: procedure ANARCHY(G = (V,E))
2:     F ← každý vrchol řazený podle počtu hran vedoucích na volné vrcholy
3:     kandidat = NULL
4:     while PRAVDA do
5:         if kandidat ≠ NULL then
6:             F = F ∖ {kandidat}
7:             vrať kandidat jako výsledek
8:         kandidat ← libovolný vrchol s nejmenším počtem hran ∈ F
9:         if ¬kandidat.kriticky then
10:            for každý soused ∈ kandidat.sousedi do
11:                if soused.volniSousedi = 1 & soused.obsazen then
12:                    Oznac-kriticky(kandidat)
13:                    kandidat = NULL
14:                    přeruš for cyklus
15:                if soused.kriticky & soused.kritickaSkupina.volniSousedi = 1 then
16:                    Oznac-kriticky(kandidat)
17:                    kandidat = NULL
18:                    přeruš for cyklus
19:            if kandidat ≠ NULL then
20:                okolniVrcholy ← každý obsazený vrchol sousedící i diagonálně s kandidat
21:                for každý vrchol ∈ okolniVrcholy do
22:                    if vrchol má nejbližší vrchol v okolniVrcholy vzdálen > 1 then
23:                        použij DFS na zkontrolování souvislosti grafu při použití kandidat
24:                        if graf není souvislý then
25:                            kandidat = NULL
26:                            přeruš for cyklus
27:        else
28:            F ← F ∖ {kandidat}
29:
30: procedure OZNAC-KRITICKY(vrchol)
31:     if ¬vrchol.kriticky then
32:         F ← F ∖ {vrchol}
33:         vrchol.kriticky = PRAVDA
34:         vrchol.kritickaSkupina ← vytvoř kritickou skupinu s vrchol
35:         for každý soused ∈ vrchol.sousedi do
36:             if soused.kriticky then
37:                 vrchol.kritickaSkupina merge soused.kritickaSkupina
38:         if vrchol.kritickaSkupina.volniSousedi = 1 then
39:             for každý soused ∈ vrchol.kritickaSkupina do
40:                 if ¬soused.obsazen then
41:                     Oznac-kriticky(soused)
42:                     přeruš for cyklus
```

Kapitola 3
Praktická část

V této kapitole bude popsána implementace vytvořeného programu, algoritmy, které byly použity, jak byly tyto algoritmy pozměneny ve srovnání s běžnou implementací a celkové fungování programu. Zásadní části programu budou popsány detailněji.
Program nyní funguje čistě na porovnání konfigurace, kterou vytvoří algoritmus *Storage Anarchy*, s běžnou konfigurací. Program tedy naplní sklad maximálně počtem položek, které jsou dány jako argument, a posléze celý sklad vyprázdní. Jako výstup vrátí celkový počet uskladněných položek, součet všech kroků, které roboti učinili 1.1.2, a počet časových kroků potřebných k dokončení procesu. 1.1.2

3.1 Vstup

Vstupem jsou parametry, které se zadají jako argument při spouštění programu z příkazového řádku. Argumenty jsou následující:

- **row**
  Určuje počet řádků, ze kterého se bude skládat graf, který reprezentuje sklad
  Omezení: Minimální velikost 7

- **col**
  Určuje počet sloupců, ze kterého se bude skládat graf, který reprezentuje sklad
  Omezení: Minimální velikost 7

- **robots**
  Určuje počet robotů, kteří budou ve skladu pracovat
  Omezení: Minimální velikost 1
  Omezení: Maximální velikost $(row \times col) - 1$

- **items**
  Určuje počet položek, které se pokusí program do skladiště uložit
  Omezení: Minimální velikost 1

- **anarchy**
  Tento parametr zapíná/vypíná tvoření nových konfigurací. Pokud parametr nastavíme na `TRUE`, program vytvoří podle algoritmu *Storage Anarchy* novou konfiguraci. Pokud parametr nastavíme na `FALSE`, program použije přednastavenou konfiguraci.
  Přednastavená konfigurace je sklad, který obsahuje co nejvíce svislých sloupců $2 \times (row - 2)$. Mezi každým sloupcem je mezera $2 \times (row - 2)$ pro snadnější pohyb robotů. mezi okraji skladu a sloupcem je vždy minimálně jeden volný *Node* pro udržení skladové spojitosti. 2.1

- **vebrose**
  Pokud je režim *vebrose* zapnut, je každou iteraci v hlavním cyklu textově znázorněn aktuální stav skladu

3.2 Hlavní cyklus

Celý program je rozdělen do tří částí, kdy každá část má jiný účel. Jednotlivé části mohou fungovat nezávisle na sobě, pokud mají patřičný vstup. Je to z důvodu, že kdyby bylo potřeba nějákou z nich poupravit či úplně změnit, bude potřeba minimálního zásahu do ostatních. Tyto části se vyskytují v hlavním cyklu, který je jednotlivě a postupně spouští, dokud se nedojde k výsledku. Hlavní cyklus má následovnou strukturu:

Výpis kódu 3.1 Zjednodušená verze hlavního cyklu

```
1  void main(arguments) {
2      Arguments args = parseArguments(arguments);
3      server.init(arguments);
4      while(true) {
5          server.assignWorkToFreeWorkers();
6          server.findSolution();
7          server.update();
8          if(server.jobDone()){
9              break;
10         }
11     }
12     return result;
13 }
```

Při spuštění se zpracujou argumenty programu a inicializuje se `Server`3.3. Zde se vytvoří mapa, inicializujou roboti a položky. Následně se zahájí hlavní cyklus, kde se opakují již zmíněné tři části:

- **Přidělení práce**
  Funkce `assignWorkToFreeWorkers()`
  Zde se dezignuje, jaký robot bude mít jakou prácí. Zpracovávají se požadavky, které jsou uměle zadány serverem, zadávají se cílové pozice pro jednotlivé roboty a hledají se místa ve skladu pro jednotlivé položky. Při zadávání cílových pozic je vždy brán takový robot, který je cílové pozici nejblíže.
- **Hledání řešení**
  Funkce `findSolution()`
  Po dezignování startovních (resp. aktuálních pozic robotů) a cílových pozic ve funkci `assignWorkToFreeWorkers()` se zde spustí algoritmus na řešení MAPF problému.
- **Aktualizace**
  Funkce `update()`
  Zde se zkontroluje, zda nedojde v nějakém kroku ke kolizi, pokud daný algoritmus ve funkci `findSolution()` neošetřuje určité scénáře. Poté se provedou kroky a akce jednotlivých robotůa aktualizuje se status robota, pokud svojí práci již dokončil.

3.3 Základní struktura

Hlavní třídou programu je třída `Server`, která se stará o komunikaci mezi roboty, skladem, uživatelem a položkami. `Server` zpracovává parametry zadané od uživatele, vytvoří mapu, inicializuje všechny roboty, a následně rozdává příkazy jednotlivým robotům. Součástí této třídy je

i primitivní vizualizér na případné debugování nebo vizualizaci vytvořené konfigurace. Obsahuje všechny další třídy, které jsou potřebné ke správnému fungování celého programu. Tyto třídy jsou následovné:

3.3.1 Worker Manager

Tato třída uchovává instanci každého fungujícího robota ve skladu, stará se o jejich pohyb a o jejich akce. Také se stará o všechna data o robotech (např. pozice, obsazenost, pracovní status...), které v případě potřeby poskytuje třídě `Server` či jiným třídám.
Kromě robotů se třída stará i o *pickery*. 2.1.1 Hlídá jejich obsazenost a zda jsou právě používány. Distribuje tyto informace dalším třídám, které si o ně řeknou.

Robot

Tato třída reprezentuje samotného robota pracujícího ve skladu. `Robot` má následující parametry:

- **current_node**
  Reprezentovaná ID daného vrcholu, ve kterém se nachazí
- **id**
  Identifikátor daného robota.
- **work_status**
  Určuje, za má robot zadanou práci, či vyčkává na zadání dalšího úkolu
- **item**
  Odkaz na položku, kterou robot právě přenáší. Pokud žádnou nepřenáší, je tento parametr prázdný
- **destination_node**
  Reprezentovaná ID vrcholu, který má robot za úkol dosáhnout
- **action**
  Definuje akci, kterou robot provede po příjezdu do finální pozice

3.3.2 Item Manager

Tato třída se stará o všechny položky, které jsou ve skladu nebo se do skladu teprve plánují umístit. Dané položky do skladu umisťují, odebírají a hledají jejich pozice.

Item

Tato třída reprezentuje samotnou položku umístěnou skladu. `Item` má následující parametry:

- **current_node**
  Reprezentovaná ID daného vrcholu, ve kterém se nachazí
- **id**
  Identifikátor dané položky
- **current_robot**
  Odkaz na robota, který položku právě přenáší. Pokud žádný robot položku nepřenáší, je tento parametr prázdný
- **storage_node**
  Reprezentovaný ID vrcholu, na kterém se daná položka hodlá uložit

- **deposit_node**
  Reprezentovaný ID vrcholu *pickeru*, na kterém se daná položka hodlá ze skladu odnést

3.3.3 Graph

Třída, která reprezentuje sklad v grafové podobě. Graf má podobu dvourozměrného pole, kdy každý prvek je třídá `Node`. Pamatuje si rozměry celého skladu, ID takových pozic, kde se vyskytují *pickery*, a uschovává si jednotlivé pozice, které jsou určeny pro uskladnění, ve speciální pomocné třídě `StorageAnarchismNodeContainer`. Nenechme se zmást názvem, tato pomocná třída je sice speciálně přizpůsobená pro kompatibilitu s *Storage Anarchy* 4 algoritmem, ale dokáže i jednodušše poskytovat volné vrcholy pro uložení položek během používání běžné konfigurace. 2.1.1
V grafu figuruje jedna důležitá funkce `isContinuous(vrcholy)`. Tato funkce pomocí známého algoritmu DFS zkontroluje, zda se i po obsazení vrcholů, které jsou dané jako parametry funcke, jedná o spojitý sklad. 2.1

Node

Tato třída reprezentuje jednotlivé vrcholy v grafu G, který je reprezentací daného skladu, se kterým program pracuje. `Node` má následující paramerty:

- **position**
  Pozice je reprezentovaná souřadnicemi x a y, které specifikují umístění ve skladu
- **id**
  Identifikátor daného vrcholu.
- **robot**
  Odkaz na robota, který se ve vrcholu právě nachází. Pokud se žádný robot ve vrcholu nenachází, je tento parametr prázdný
- **item**
  Odkaz na položku, která se ve vrcholu právě nachází. Pokud se žádná položka ve vrcholu nenachází, je tento parametr prázdný
- **node_type**
  Tento parametr určuje, zda je vrchol určen jako *picker*, úložný prostor, průchozí prostor nebo překážka
- **robot_id_dest**
  Reprezentovaný ID robota. Určuje, jaký robot má na daný vrchol úkol dojít. Pokud žádný robot nemá tento vrchol daný jako finální destinaci, parametr je prázdný
- **robot_action**
  Určuje, jakou akci plánuje robot, který má tento vrchol daný jako finální destinaci, provést

  Následující parametry jsou potřebné pro fungování *Storage Anarchy* algoritmu:

- **cluster**
  Pokud je vrchol kritický, tento parametr má uložený odkaz na kritickou skupinu, ve které se vrchol nachází.
- **num_of_free_neighbours**
  Udržuje počet volných sousedů.

- **num_of_free_non_critical_neighbours**
  Udržuje počet volných nekritických sousedů.

Jak už bylo dříve řečeno, graf reprezentuje sklad. Každý `Node` reprezentuje jeden čtverec o velikosti jednoho regálu. 2.1.1 Jako sousedy daného `Node` x pokládáme takové okolní `Node`, které jsou přilehlé k x vertikálně či horizontálně. (stejně jako jsme definovali sousedy při analýze algoritmu *Storage Anarchy* 2.2) Robot, který se nachází v x tedy může provést maximálně pět různých pohybů — pohyb z x do libovolného souseda x nebo může počkat ve vrcholu x.

3.3.4 Storage Anarchy

Tato třída neuchovává žádné zásadní parametry. Obsahuje však algoritmus *Storage Anarchy* 4, který je nezbytný pro tvoření rekonfigurovatelného skladu. Implementaci algoritmu vysvětlím později v této kapitole.

3.4 Implementace algoritmů

Zde rozeberu detailněji vlastní implementaci algoritmů, které jsem použil. Přesněji se budu soustředit na algoritmy *Storage Anarchy* a *CBS*. K těmto algoritmům zmíním i některé jejich pomocné třídy.

3.4.1 Storage Anarchy

Jak již bylo řečeno v analytické části, tento algoritmus realizuje novou konfiguraci pro sklad, která zaručuje *spojitost skladu.* 2.1 Je volán ve funkci `assignWorkToFreeWorkers()` 3.2, přesněji v tu chvíli, když je robotu zadána práce pro uskladnění položky ve skladu. 3.4.2.1
Jak je viděno v pseudokódu *Storage Anarchy* 4, nejdříve je potřeba vybrat volný vrchol s nejmenším počtem volných sousedů z fronty F. Fronta F se udržuje s aktuálním počtem volných sousedů každého robota ve třídě `Graph`. Při vybírání takového vrcholu je potřeba ho najít rychle a efektivně. Při zvětšování velikosti skladu se také zvyšuje počet míst, kde je možné položku uložit, takže iterativní prohledávání je neefektivní. Proto je implementována třída `StorageAnarchismNodeContainer`, reprezentující frontu F, která se efektivně stará o vybírání, vkládání a reorganizaci jednotlivých `Node`.
Algoritmus *Storage Anarchy* 4 je kromě zmiňované pomocné třídy `StorageAnarchismNodeContainer` detailně popsán v analytické části, proto zde nebudu samotný kód popisovat.

3.4.1.1 StorageAnarchismNodeContainer

Pro uskladňování jednotlivých `Node` rozlišujeme pouze čtyři možnosti. Pokud se jedná o *spojitý sklad* 2.1, není možnost, aby jakýkoliv volný `Node`, který je označen k ukládání položek, ze skladu měl méně než jednoho volného souseda. Důkaz ponechávám na čtenářovi. V programu je sklad znázorněn jako mřížka, tímpádem žádný vrchol nemůže mít více než čtyři sousedy. Takže všechny volné `Node`, které jsou označeny k ukládání položek, je možné rozdělit na čtyři skupiny podle počtu volných sousedů. Tyto skupiny reprezentuji statickým polem o velikosti čtyři.
Pro efektivní mazání vrcholů jsou tyto skupiny reprezentovány jako mapy, které jsou seřazeny podle ID jednotlivých `Node`, aby šlo vyhledávat v logaritmickém čase.
Finální struktura vypadá tedy následovně:

```
1 vector<map<uint64_t, shared_ptr<Node>>> free_node_container;
```

Při vybírání `Node` s nejmenším počtem volných sousedů se může ideálních `Node` naskytnout více než jeden. V tuto chvíli zde figuruje náhoda, která jednoho z ideálních `Node` vybere. Díky této náhodě při vybírání může algoritmus *Storage Anarchy* vytvořit více konfigurací pro velikostně stejný sklad.3.2

Výpis kódu 3.2 Algoritmus na vybírání náhodného Node s nejmenším počtem volných sousedů

```
std::shared_ptr<Node>
    &StorageAnarchismNodeContainer::getAnyNodeWithLowestNumOfNeighbours() {
    for(int i = 0; i < free_node_container.size(); ++i) {
        if(!free_node_container[i].empty()) {
            int best_score = -1;
            std::vector<int> best_result_ids;
            for(const auto &it : free_node_container[i]) {
                int tmp = it.second->getNumOfFreeNeighbours()
                    - it.second->getNumOfFreeNonCritNeighbours();
                if(best_score < tmp) {
                    best_result_ids.clear();
                    best_result_ids.push_back(it.second->getId());
                    best_score = it.second->getNumOfFreeNeighbours()
                        - it.second->getNumOfFreeNonCritNeighbours();
                }
                else if(best_score == tmp) {
                    best_result_ids.push_back(it.second->getId());
                }
            }
            auto it = best_result_ids.begin();
            std::advance(it, rand() % best_result_ids.size());
            return free_node_container[i].find(*it)->second;
        }
    }
    throw NoAvailableStorage();
}
```

V algoritmu 3.2 si můžeme povšimnout, že zde figuruje ještě počet kritických sousedů daného vrcholu. Vybíráme tedy vrchol s nejmenším počtem sousedícíh volných vrcholů a nejvyšším počtem sousedících kritických vrcholů. Díky tomuto výběru sice omezíme náhodnost dané konfigurace, ale sklad, který nedosáhne své maximální kapacity, bude tak mít všechny položky uskladněné blíže u sebe. Tento krok je vhodný pro zajištění efektivity pohybu robotů ve skladu, pokud sklad není maximálně naplněn.

3.4.2 CBS

Pro řešení MAPF problému jsem využil algoritmus CBS. Tento algoritmus je volán ve funkci `update()` 3.2, pokud je potřeba najít nové plány pro agenty. K tomu dojde v takových případech, kdy robot dokončí svojí práci a potřebuje nový plán, aby se úspěšně dostal do nového cílového vrcholu, nebo pokud dojde ke kolizi mezi robotem s prací a robotem, co čeká na zadání další práce.
Než začnu s komentováním implementace CBS algoritmu, je potřeba si osvětlit, jak funguje přiřazování práce.

3.4.2.1 Dělení práce

Požadavek na práci uchovává třída `Request`, která jí rozděluje na dvě skupiny — zvednutí položky nebo položení položky. Každá práce kromě rozdělení do dané skupiny je ještě specifikována, zda se jedná o uložení položky do skladu nebo o její vyvezení ze skladu. Třída `Request` je uložená ve

třídě `Server`. Po inicializování třídy `Server` se zavolá funkce `requestStoringAllItems()`, která naplní třídu `Request` požadavkama pro zvednutí a uložení všech položek zadaných jako vstup. Dělení práce se odehrává ve funkci `assignWorkToFreeWorkers()`. Zde se vybere ze třídy `Request` požadavek, který se skládá z ID dané položky, se kterou chceme manipulovat, a proměnné, která určuje, zda se jedná o ukládání nebo vyvážení.

- **Zvednutí**
  Při zvedání položky je prvním úkolem lokalizovat danou položku. Pokud je již ve skladu, jedná se o vyvážení položky ze skladu. V tomto případě se najde pomocí *manhattonské vzdálenosti* nejbližší robot, který je bez práce, a dá se mu za úkol položku vzít a odvézt k volnému *pickerovi*.
  Pokud se položka ve skladu nevyskytuje, jedná se o uskladnění položky do skladu. V tomto případě se vybere volný *picker*, který položku robotovi poskytne. Následně se najde nejbližší robot k danému pickerovi pomocí *manhattonské vzdálenosti* a zadá se mu za úkol položku vyzvednou u *pickera* a odvézt na místo dané klasickou konfigurací či algoritmem *Storage Anarchy*.4
  Při vyzvedávání používáme na měření vzdálenosti robota od jeho potencionálního cíle *manhattonskou vzdálenost*, protože roboti bez nákladu mohou podjíždět jednotlivé položky bez jakýchkoliv potíží, tímpádem je měření velice přesné.
- **Položení**
  Požadavky pro pokládání se tvoří ve fuknci `update()`. Pokud robot dokončí zvedání položky, vytvoří se nový úkol na položení položky na místo, které bylo určeno při vytváření zvedacího úkolu. Tímto místem je *picker*, pokud se jedná o vyvážení položky, nebo místo vybrané algoritmem *Storage Anarchy*, jestli se položka ukládá do skladu.

Pokud robot dokončí svůj daný úkol, přejde do stavu hlídkování. Vybere se náhodná pozice ve skladu a ta se určí robotovi k hlídkování. Pokud robot na pozici dorazí, dostane se do stavu nečinnosti a vyčkává na další zadaný úkol.

3.4.2.2 Algoritmus

Při inicializování algoritmu CBS se nejdříve vytvoří dvě mapy `map_without_items` a `map_with_items`, které reprezentujou sklad. Tyto mapy se liší v tom, že jedna z nich reprezentuje položky jako překážky a druhá položky ignoruje.
Algoritmus CBS, který jsem implementoval, se liší od vysvětleného CBS v části teoretické. 1 Provedené rozdíly nejdříve popíšu ve vyšší úrovni a následně ve nižší úrovni.

- **Vyšší úroveň**
  Vyšší úroveň vrací mapu, která obsahuje ID robota jako klíč a plán, reprezentovaný pomocnou třídou `Solution`, jako hodnotu. Místo stromu je použita množinu vrcholů za účelem ušetření místa. Implementovaná vyšší úroveň3.4 je téměř stejná jako vyšší úroveň popsaná v teoretické částí 1, liší se znatelně pouze ve dvou částech:
  - Hledání konfliktů
    Na řádku 11 až 12 se prověřuje, zda řešení vybraného vrcholu `node` z množiny vrcholů `tree` má konflikt. Vždy se zohlední pouze první nalezený konflikt.
    Konflikty jsou buď vrcholové nebo hranové. 1.1.1 Hranové konflikty řeším běžným způsobem, ale vrcholové konflikty berou v potaz `work_status` robotů, kterých se konflikt týká. Pokud má robot r_1 zadanou práci a nastane konflikt s robotem r_2, který práci nemá, ve vrcholu v a čase t, zohlední se ve vyšší vrstvě jen omezení (r_2, v, t). Tímto se omezí počet větvení ve vyšší vrstvě (viz. 3.3 řádek 19 až 55).

Výpis kódu 3.3 Hledání vrcholových konfliktů ve vyšší vrstvě algoritmu CBS

```
vector<Constrain> TreeNode::findVertexConflict
    (const unique_ptr<WorkerManager> &worker_manager) const {
    int timestep = 0;
    vector<Solution> available_solutions;
    for (const auto &solution : paths) {
        available_solutions.push_back(solution.second);
    }
    map<Position, uint64_t, positionComparator> possible_conflicts;
    while(available_solutions.size() > 1) {
        for(auto solution = available_solutions.begin();
            solution != available_solutions.end();) {
            Position position = solution->getTimestepPosition(timestep);
            if(position == Position(-1, -1)) {
                solution = available_solutions.erase(solution);
            }
            else if(!possible_conflicts.insert(
                {position, solution->getRobotId()}).second) {
                auto second_conflict = possible_conflicts.find(position);
                vector<Constrain> result;
                RobotAction first_robot_action =
                    worker_manager->getRobot(solution->getRobotId())->getAction();
                RobotAction second_robot_action =
                    worker_manager->getRobot(second_conflict->second)->getAction();
                if(first_robot_action != IDLE && first_robot_action != PATROL) {
                    result.emplace_back(
                        Constrain(second_conflict->second,
                        second_conflict->first, timestep));
                }
                if(second_robot_action != IDLE && second_robot_action != PATROL) {
                    result.emplace_back(
                        Constrain(solution->getRobotId(),
                        position, timestep));
                }
                if(result.empty()) {
                    if ((first_robot_action == IDLE &&
                        second_robot_action == IDLE) ||
                        (first_robot_action == PATROL &&
                        second_robot_action == PATROL)) {
                        return {Constrain(solution->getRobotId(),
                            position, timestep),
                            Constrain(second_conflict->second,
                                second_conflict->first, timestep)};
                    } else {
                        if (first_robot_action == IDLE)
                            result.emplace_back(
                            Constrain(solution->getRobotId(),
                                position, timestep));
                        else
                            result.emplace_back(
                            Constrain(second_conflict->second,
                                second_conflict->first, timestep));
                    }
                }
                return result;
            }
            else {
                ++solution;
            }
        }
        possible_conflicts.clear();
        ++timestep;
    }
    return {};
}
```

- Duplikace vrcholů
  Díky zásadním zásahům do nižší vrstvy se může ve vyšší vrstvě vyskytnout takový vrchol, který již v množině vrcholů máme. Takového vrcholu se chceme zbavit, aby nedošlo ke zbytečnému větvení. K tomu dochází na řádcích 21 až 26. 3.4

Výpis kódu 3.4 Hlavní cyklus vyšší vrstvy algoritmu CBS

```
1  map<std::uint64_t, Solution>
2  HighLevelSolver::solve(const unique_ptr<WorkerManager> &worker_manager) {
3
4      TreeNode root = TreeNode();
5
6      if(low_level_solver.findPaths(root, worker_manager))
7          tree.insert(root);
8
9      while(!tree.empty()) {
10         auto node = tree.begin();
11         pair<bool, vector<Constrain>> conflicts =
12             node->getFirstConflict(worker_manager);
13         if(!conflicts.first) {
14             TreeNode final_node = *tree.begin();
15             return final_node.getPaths();
16         }
17         //Conflict found
18         for(const auto &constrain : conflicts.second) {
19             TreeNode successor = *node;
20             successor.addConstrain(constrain);
21             if(low_level_solver.findPath
22                 (successor, worker_manager, constrain.robot_id)) {
23                 auto result = tree.insert(successor);
24                 if(!result.second)
25                     cout << "Duplicate tree node found" << endl;
26             }
27
28         }
29         tree.erase(node);
30     }
31     throw runtime_error("Storage is not continuous");
32 }
```

- **Nižší úroveň**
  Nižší úroveň3.6 je značně víc modifikovaná než vyšší úroveň. Její hlavní účel je najít cestu pro jednoho robota s ohledem na omezení získané z vyšší vrstvy. Na hledání cesty pro jednoho robota v prostoru je ideální algoritmus A*. Tento algoritmus ale nemůžeme v našem případě použít, protože roboti mohou provést akce čekání a pohyb na na již nasvštívený vrchol. Tyto akce originální A* zakazuje, proto bylo potřeba A* modifikovat, aby dokázal vždy najít správné řešení.
  Při zanoření do nižší vrstvy se nejdříve vybere mapa, ve které se robot pohybuje. Tato mapa je označena jako proměnná `used_map`. Pokud robot nese položku, cesta se bude hledat v `map_without_items`, jestli robot položku nenese, využije se `map_with_items`. Tato mapa slouží k zapisování informací o tom, kdy a jaký vrchol byl navštíven z jakého vrcholu. Poté se inicializuje množina již navštívených pozic `visited_pos` a množina otevřených vrcholů `open_cells`. Otevřené vrcholy jsou reprezentovány třídou `Cell`.

  - **Cell**
    Tato třída reprezentuje vrchol prohledávaný modifikovaným A*. Má následující parametry:

- **pos**
  Pozice vrcholu ve skladu
- **obstacle**
  Prozrazuje, jestli je vrchol prostupný
- **parents**
  Multimapa, kdy klíč je časový krok, ve kterým do daného vrcholu robot vstoupil, a hodnota je pozice, ze které přišel
- **heuristic_distance**
  Manhattonská vzdálenost daného vrcholu od cílového vrcholu
- **traveled_distance**
  Počet kroků, které robot musel provést, než se do daného vrcholu dostal
- **sum_distance**
  Součet `heuristic_distance` a `traveled_distance`
- **score**
  Skóre, podle kterého se vrcholy řadí do množiny otevřených vrcholů

Množina otevřených vrcholů si nepamatuje parametr `parents`, slouží čistě k pamatování ostatních parametrů. Parametr `parents` se vždy bere z proměnné `used_map`. Do `open_cells` a `visited_pos` vložíme startovní vrchol.
Po inicializování (řádek 5 až 20 3.6) začne hlavní cyklus (řádek 21 až 62 3.6) modifikovaného A*. Z `open_cells` se vybere vrchol `open_cell` s nejmenším parametrem `score`, uloží se jeho parametry `heuristic_distance`, `traveled_distance` a `sum_distance` do odpovídajícího vrcholu v `used_map` a poté se `open_cell` vymaže z `open_cells`.
Následně pro každého souseda vrcholu `cell` se zjistí, zda je daný soused v daný časový krok obsažen v omezeních. Mezi sousedy počítáme i samotný vrchol `cell`, aby algoritmus bral v potaz akci čekání. Jestli je vybraný sousední vrchol obsažen v omezeních, soused se ignoruje. Pokud tomu tak není, je vrchol označen jako `successor_cell`, vložen do množiny `visited_pos`, do jeho parametru `parents` je vložen vrchol `cell` s odpovídajícím časovým krokem a nakonec jsou přepočítány hodnoty `heuristic_distance`, `traveled_distance` a `sum_distance` pro `successor_cell`.
Pokud se vrchol `successor_cell` nerovná cílovému vrcholu, je vrcholu vypočítáno skóre a následně vložen do `open_cells`. Skóre je závislé na tom, zda se vrchol již nacházel v množině `visited_pos` nebo jestli algoritmus provedl akci čekání. V těchto případech se skóre vynásobí určitou konstantou, aby byl expandován později.
Bylo implementováno i zvyšování skóre, pokud je `successor_cell` nalezen v daný čas v souboru plánů ostatních agentů, kteří již mají své cesty dané ve vyšší vrstvě. Avšak z důvodu triviální implementace ho ve výpisu kódu nezmiňuji. 3.6 Díky upravování skóre pro rychlejší běh algoritmu může najít i **neoptimální** řešení.
Jestli vrchol `successor_cell` odpovídá cílovému vrcholu, je pomocí funkce `tracePath()` nalezeno řešení a to je posléze posláno do vyšší vrstvy.
Fuknce `tracePath()` 3.5 na vrácení řešení začíná zkontrolováním, zda modifikovaný algoritmus neudělal žádný krok, protože se startovní vrchol rovnal cílovému vrcholu. Pokud tomu tak je, vrátí jako řešení startovní vrchol jako důkaz akce čekání. V ostatních případech začíná hlavní cyklus (řádek 14 až 28) a cílový vrchol se nastaví jako prohledávaný vrchol. Každou iteraci v hlavním cyklu najde v parametru prohledávaného vrcholu `parents` takovou pozici, která odpovídá časovému kroku, ve kterém byl prohledávaný vrchol navšíven. Pokud nalezne jakýkoliv odpovídající vrchol, nastaví ho jako nový prohledávaný vrchol, sníží časový krok o jedna a zařadí nový prohledávaný vrchol do řešení. Pokud nový prohledávaný vrchol odpovídá startovnímu vrcholu, vrátí se celé řešení.

Výpis kódu 3.5 Algoritmus na vyhledání řešení v modifikovaném A*

```
vector<Position> LowLevelSolver::tracePath(Map *used_map,
    const Position &start_pos, const Position &end_pos) {
    vector<Position> result;
    if(start_pos == end_pos &&
        used_map->getCell(end_pos).traveled_distance == 1) {
        result.push_back(start_pos);
        return result;
    }

    Position current_pos = end_pos;
    int current_timestep = used_map->getCell(end_pos).traveled_distance;
    result.push_back(end_pos);

    while(true) {
        pair<int, Position> move = used_map->getCell(current_pos)
            .findElementInParents(current_timestep);
        Position tmp_position = move.second;
        int tmp_traveled_distance = move.first;

        if(tmp_traveled_distance == current_timestep) {
            if(tmp_position == start_pos && current_timestep == 1) {
                break;
            }
            result.push_back(tmp_position);
            current_pos = tmp_position;
            --current_timestep;
        }
    }
    return result;
}
```

Výpis kódu 3.6 Nižší vrstva algoritmu CBS

```
Solution LowLevelSolver::solve(const shared_ptr<Robot> &robot,
                              const Position &start_pos,
                              const Position &end_pos,
                              const TreeNode &tree_node) {
    Map* used_map;
    set<Position> visited_pos;
    if(robot->getItem() == nullptr)
        used_map = &map_without_items;
    else
        used_map = &map_with_items;

    set<Cell, cellComparator> open_cells;
    used_map->getCell(start_pos).heuristic_distance = 0;
    used_map->getCell(start_pos).traveled_distance = 0;
    used_map->getCell(start_pos).sum_distance = 0;
    used_map->getCell(start_pos).score = 0;

    open_cells.insert(used_map->getCell(start_pos).copyCellNoParents());
    visited_pos.insert(start_pos);

    while(!open_cells.empty()) {
        auto open_cell = open_cells.begin();
        Cell *cell = &used_map->getCell(open_cell->getPosition());
        cell->traveled_distance = open_cell->traveled_distance;
        cell->sum_distance = open_cell->sum_distance;
        cell->heuristic_distance = open_cell->heuristic_distance;
        open_cells.erase(open_cells.begin());

        for(const Position &possible_move : possibleMovesPositions) {
            Position new_pos = possible_move + cell->getPosition();
            if(used_map->isValid(new_pos) &&
                !tree_node.isInConstrains(
                Constrain(robot->getId(), new_pos, cell->traveled_distance))) {
                Cell* successor_cell = &used_map->getCell(new_pos);
                bool visited = !visited_pos.insert(new_pos).second;

                successor_cell->addParent(
                    pair<int, Position>(cell->traveled_distance + 1,
                    cell->getPosition()));
                successor_cell->heuristic_distance = used_map->manhattanDistance(
                    new_pos, end_pos);
                successor_cell->traveled_distance = cell->traveled_distance + 1;
                successor_cell->sum_distance =
                    successor_cell->heuristic_distance +
                    successor_cell->traveled_distance;

                if(new_pos == end_pos) {
                    return {robot->getId(), tracePath(used_map,start_pos,end_pos)};
                }
                if(successor_cell->getPosition() == cell->getPosition()) {
                    successor_cell->score = floor(successor_cell->sum_distance*1.5);
                    open_cells.insert(successor_cell->copyCellNoParents());
                }
                else {
                    successor_cell->score = new_sum_distance;
                    if(visited)
                        successor_cell->score *= 2;
                    open_cells.insert(successor_cell->copyCellNoParents());
                }
            }
        }
    }
    throw NoPathFound();
}
```

3.5 Výstup

Jako výstup programu je počet položek z daného argumentu *items* na vstupu, které se povedlo úspěšně do konfigurace skladu uložit, součet počtu provedených akcí každého pracujícího robota (*sum of costs* 1.1.2) a počet časových kroků nutných pro dokončení celého procesu (*makespan* 1.1.2). Jako mezivýstup je textové znázornění skladu, kde jsou roboti označeni jako čísla podle jejich ID a položky jako znak ´*´. Místa, které mají funkci *pickera* jsou označeny písmenem ´P´.

Obrázek 3.1 Příklad výstupu při použití Storage Anarchy

```
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |*  |*  |*  |*  |   |P  |   |*  |*  |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |*  |   |   |   |*  |*  |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |*  |*  |   |*  |*  |*  |   |   |*1 |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |*  |   |*  |*  |*  |*  |   |*2 |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |*  |   |*  |   |   |   |   |*3 |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |*  |   |   |   |*  |   |   |   |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |   |   |*  |   |*  |   |*  |*  |*  |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |*  |*  |*  |   |*  |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |*  |   |   |   |*  |   |*  |*  |*  |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |*  |   |*  |*  |*  |   |   |   |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |*  |   |*  |   |   |   |*  |*  |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
|*  |   |*  |*  |*  |   |P  |   |*  |*  |   |*  |
+---+---+---+---+---+---+---+---+---+---+---+---+
TOTAL MOVES: 3363
TOTAL ITEM STORED: 75
TIME SPENT: 1121
```

Kapitola 4
Experimentální část

V této části budu popisovat postup při experimentech, analyzovat získaná data a následně budu pokračovat diskuzí o získaných výsledcích a případných využití algoritmu *StorageAnarchy* v praxi.
Experiment spočívá v porovnávání výstupů programu při používání algoritmu *Storage Anarchy* na vytvoření nové konfigurace skladu s používání běžné konfigurace skladu. S vedoucím práce jsme usoudili, že nejpřínosnější data se získají při úplném naplnění skladu a následném úplném vyprázdnění. Sice se nebude experimentovat s rekonfigurovatelností skladu, ale většina případů, ke kterým může během rekonfigurování skladu přijít, budou s dostatečným počtem experimentů také do získaných dat zahrnuty.
Nejdříve začnu experimentovat s menšími rozměry skladu a s menším počtem robotů, analyzuji jejich konfiguraci a počet učiněných kroků. Poté budu postupně zvyšovat rozměry skladu i počet dostupných robotů. Sklad vždy zcela naplním a poté vyprázdním.
Pro experimenty jsem si zvolil nutný volný prostor kolem *pickerů*. Každý *picker* je umístěn buď v horní nebo dolní části skladu přilehlý ke stěne. Každý z nich kolem sebe musí mít všechny okolní vrcholy volné.
Jako **běžnou konfiguraci** jsem při testování zvolil prostor, který má po stranách vertikálně umístěné sloupce o velikosti $1 \times Y - 2$, kdy Y je počet řádků ve skladu. Dále má ve zbývajícím prostoru vertikálně umístěné sloupce o velikosti $2 \times Y - 2$. Každý sloupec mezi sebou a dalším sloupcem má prázdný sloupec pozic $2 \times Y - 2$ pro možnost pohybu.
Počet *pickerů* je závislý na velikosti skladu.

$$počet\ pickerů = 2 \times \lfloor \frac{počet\ sloupců}{7} \rfloor$$

4.1 Menší rozměry

Tyto experimenty probíhali na skladech o rozměru 7×7, 8×8, 9×9 a 10×10 s počtem robotů v rozmezí 1 až 6. Každý experiment opakuji 100×, poté vypočítám průměr z výsledků.

4.1.1 Běžná konfigurace

Zde je očividné, že počet položek je přímo závislý na rozměru skladu. Jiný parametr tento výsledek měnit nemůže. Počet robotů nemá absolutně žádný vliv na statický sklad. 4.1a
U *sum of costs* 4.1b si můžeme povšimnout, že u všech menších otestovaných velikostech se velikost rapidně začne zvyšovat při více jak dvou robotech. Důvod tohoto zvyšování je nutnost provedení pohybů potřebných pro úhyb z kolize. Při porovnání 4.1c můžeme usoudit, že při více jak 4 robotech se *sum of costs* rapidně zvyšuje bez značného zlepšení *makespan*.

Obrázek 4.1 Výsledky při použití běžné konfigurace

Počet položek				
Bežná	Velikost skladu			
Počet robotů	7×7	8×8	9×9	10×10
x	16	26	33	46

Sum of costs				
Běžná	Velikost skladu			
Počet robotů	7×7	8×8	9×9	10×10
1	336, 02	634, 95	897, 57	1.393, 10
2	344, 78	638, 70	920, 90	1.416, 70
3	417, 69	790, 68	1066, 44	1.644, 03
4	511, 92	880, 04	1.262, 24	1.931, 28
5	594, 16	1.060, 78	1.452, 83	2.172, 00
6	694, 44	1.172, 16	1.647, 36	2.509, 93

Makespan				
Běžná	Velikost skladu			
Počet robotů	7×7	8×8	9×9	10×10
1	336, 02	634, 95	897, 57	1.393, 10
2	177, 53	315, 02	465, 98	713, 84
3	141, 07	250, 17	359, 19	556, 31
4	125, 38	225, 03	309, 64	486, 65
5	110, 23	212, 06	290, 56	434, 39
6	115, 73	195, 36	274, 56	415, 48

4.1.2 Storage anarchy

U algoritmu *Storage Anarchy* počet uložených položek závisí také na velikosti prostoru a navíc i na prvku náhody. Počet robotů nemá na počet prvků absolutně žádný vliv. 4.2a
Při pozorování nárůstu *sum of costs* 4.2b od druhého robota můžeme usoudit, že podle získaných dat roste lineárně s každým dalším robotem. U *makespan* je tomu podobně jako u běžné konfigurace, a to že s čtvrtým robotem se hodnota již závratně nezlepšuje.

Obrázek 4.2 Výsledky při použití *Storage Anarchism*

Počet položek				
Anarchismus	Velikost skladu			
Počet robotů	7×7	8×8	9×9	10×10
1	18	28,54	38,31	50,14
2	18	28,65	39,40	49,35
3	18	28,91	38,61	50,69
4	18	28,17	39,20	50,15
5	18	28,48	39,31	50,25
6	18	28,24	39,62	50,94

Sum of costs				
Anarchismus	Velikost skladu			
Počet robotů	7×7	8×8	9×9	10×10
1	392,34	714,90	1.114,11	1.575,52
2	434,26	699,76	1.088,34	1.571,52
3	473,97	887,91	1.334,97	1.903,95
4	567,36	1.049,40	1.599,92	2.232,72
5	636,33	1.149,33	1.855,83	2.598,04
6	730,02	1.348,80	2.119,68	2.972,46

Makespan				
Anarchismus	Velikost skladu			
Počet robotů	7×7	8×8	9×9	10×10
1	392,34	714,90	1.114,11	1.575,52
2	198,77	350,22	544,17	793,13
3	157,18	278,78	450,20	621,02
4	140,76	249,66	405,92	563,78
5	127,26	229,45	371,16	519,61
6	121,67	224,80	353,28	495,41

4.1.3 Porovnání

Algoritmus *Storage Anarchy* průměrně dokáže tvořit konfigurace s vyšší kapacitou o 3.5 položky než použitá běžná konfigurace. Čím větší prostor, tím je rozdíl kapacity markantnější ve prospěch algoritmu *Storage Anarchy*.
Při porovnávání efektivity pohybu je na tom lépe běžná konfigurace. Při nejmenší instanci je rozdíl v *sum of costs* přibližně 10%, ve větších instancí dosahuje rozdíl přibližně 16%. Podobně tomu je tak i u *makespan* — u menších instancí je *Storage Anarchy* o přibližně 11% horší, u větších instancí tento rozdíl činí přibližně 15%.

4.2 Větší rozměry

Tyto experimenty probíhali na skladech o rozměru 15×15, 20×20 a 25×25 s počtem robotů v rozmezí 1 až 8. Každý experiment opakuji přibližně 10×, poté vypočítám průměr z výsledků. Z důvodu použití CBS na řešení MAPF problému je při větších experimentech možnost delšího vypočítávání výsledku. Je to z důvodu, že modifikovaný A* 3.6, který bere v potaz i akci čekání, může zavinit neadekvátní expandování ve vyšší vrstvě. 3.4 Hledání řešení může trvat mimořádně dlouho, pokud je zadán velký prostor s více roboty a použije se *Storage Anarchy*.

4.2.1 Běžná konfigurace

Zde je stejně jako u menších rozměrů jasné, že počet položek je přímo závislý na rozměru skladu. 4.1a
U hodnoty *sum of costs* 4.3b stejně jako u menších rozměrů je zde hranice, kdy se začne hodnota rapidně zvyšovat. Tato hranice činí pět, u rozměru 25×25 sedm, pracujících robotů. U *makespan* 4.3c je tomu stejně — při překročení hranice pěti robotů již není zlepšení moc znatelné.

Obrázek 4.3 Výsledky při použití běžné konfigurace

Počet položek			
Bežná	Velikost skladu		
Počet robotů	15×15	20×20	25×25
x	98	192	289

Sum of costs			
Běžná	Velikost skladu		
Počet robotů	15×15	20×20	25×25
1	4.608, 46	11.976, 20	22.963, 20
2	4.650, 40	12.048, 00	23.124, 40
3	4.738, 95	12.352, 50	23.427, 80
4	4.844, 35	12.207, 20	23.676, 90
5	4.874, 67	12.554, 00	23.732, 20
6	5.298, 05	13.353, 00	23.821, 20
7	5.689, 13	14.450, 80	23.884, 90
8	6.044, 80	15.379, 20	24.004, 00

Makespan			
Běžná	Velikost skladu		
Počet robotů	15×15	20×20	25×25
1	4.608, 46	11.976, 20	22.963, 20
2	2.318, 73	6.024, 00	11.562, 70
3	1.575, 10	4.117, 50	7.809, 30
4	1.173, 09	3.051, 80	5.919, 20
5	967, 47	2.510, 80	4.784, 40
6	873, 60	2.225, 50	3.953, 50
7	812, 75	2.064, 40	3.369, 10
8	755, 60	1.922, 40	3.000, 50

4.2.2 Storage anarchy

Při používání *Storage Anarchy* se při experimentech na větších rozměrech projevila modifikace A* v algoritmu CBS na řešení MAPF problému. Díky možnosti čekání robotů na místě a složitosti mapy vytvořené *Storage Anarchy* (př. 3.1) došlo k extrémnímu množství expandovaných vrcholů ve vyšší vrstvě algoritmu CBS. Tímto se mnohonásobně zvýšil potřebný čas na vyřešení daného problému. Proto je během experimentů nastaven *timeout* po deseti minutách hledání řešení. Pokud program nedokončil zadaný úkol, je vytvořena predikce z mezivýsledků. Tyto predikce jsou označeny symbolem ´*´.
Počet položek 4.4a, které je možné uskladnit, mají v porovnání s menšími hodnotami mezi sebou větší rozdíly. Náhodný výběr z prvků v algorimtu *Storage Anarchy* je ve větších prostorech větší, tímpádem je větší možnost výběru takového prvku, který v minulém běhu vybrán nebyl.
Sum of costs 4.4b zde měla zvláštní výkyv. Od experimentu s jedním robotem do experimentu se čtyřmi roboty se *sum of costs* proti očekávání zmenšoval. To nastalo díky vytvořené konfiguraci algoritmem *Storage Anarchy*. Konfigurace spíše připomíná bludiště, a při používání menšího počtu robotů musí roboti pokrýt celý prostor sami. Při výskytu většího počtu robotů je při zadávání práce ve funkci `assignWorkToFreeWorkers()` 3.2 vybrán nejbližší volný robot, tímpádem je zmenšena možnost cestování jednoho robota přes celou konfiguraci.
Makespan 4.4c se podle očekávání s počtem pracujících robotů zmenšoval. Zásadní pokles můžeme zpozorovat od experimentu s jedním robotem do experimentu se čtyřmi roboty.

4.2.3 Porovnání

V průměru *Storage Anarchy* při experimentech ve větších rozměrech vytvoří konfigurace, které pojmou o přibližně 21.6 položek víc než běžná konfigurace.
Při porovnávání efektivity pohybu je na tom opět lépe běžná konfigurace. Při nejmenší instanci je rozdíl v *sum of costs* přibližně 20%, ve větších instancí dosahuje rozdíl přibližně 15%. Podobně tomu je tak i u *makespan* — u menších instancí je *Storage Anarchy* o přibližně 25% horší, u větších instancí tento rozdíl činí přibližně 19%.
Měření je nepřesné z důvodu zmiňovaném výše 4.2.2. Nepřesnost u větších instancí s více roboty může být vyšší.

Obrázek 4.4 Výsledky při použití *Storage Anarchism*

Počet položek			
Bežná	Velikost skladu		
Počet robotů	15 × 15	20 × 20	25 × 25
1	118,50	210,20	317,70
2	117,90	209,70	318,90
3	117,80	208,70	316,90
4	118,00	209,300	∗319,30
5	116,80	∗209,90	∗318,20
6	∗117,30	∗208,30	∗318,10
7	∗117,00	∗209,20	∗319,90
8	∗116,60	∗208,80	∗317,40

Sum of costs			
Běžná	Velikost skladu		
Počet robotů	15 × 15	20 × 20	25 × 25
1	5.803,40	13.625,50	26.372,40
2	5.700,10	13.530,00	26.112,20
3	5.698,10	13.572,20	25.975,40
4	5.540,30	13.284,40	∗25.879,70
5	5.850,10	∗13.983,10	∗26.699,70
6	∗6.431,40	∗14.514,90	∗27.240,10
7	∗7.173,50	∗16.091,10	∗28.270,20
8	∗8.166,70	∗16.974,70	∗30.594,10

Makespan			
Běžná	Velikost skladu		
Počet robotů	15 × 15	20 × 20	25 × 25
1	5.803,40	13.625,50	26.372,40
2	2.850,70	6.765,10	13.056,20
3	1.909,70	4.524,20	8.807,30
4	1.385,00	3.867,80	∗6.973,10
5	1.229,40	∗2.947,10	∗5.780,60
6	∗1.117,30	∗2.712,20	∗5.203,80
7	∗1.086,40	∗2.591,10	∗4.906,40
8	∗1.007,60	∗2.397,50	∗4.718,50

4.3 Diskuze

Při porovnávání menších a větších rozměrů lze usoudit, že sice nové konfigurace vytvořené pomocí *Storage Anarchy* algoritmem dokážou sice uskaldnit více pložek, ale s nárůstem rozměrů daného skladu se efektivita pohybu drasticky zhoršuje. Pokud ale algoritmus *Storage Anarchy* pracuje v menších prostorech, efektivita není o tolik horší a stále dokáže daná konfigurace uchovat více položek než běžná konfigurace. Proto při praktickém využití by sice nešlo uvažovat o aplikování *Storage Anarchy* algoritmu na velký sklad, protože možnost umístění více položek by nedokázal kompenzovat nárůst času, který je potřeba pro vyzvednutí položky, ale v menších prostorech, u kterých je potřeba maximálně využít potencionál úložného prostoru, by bylo možné tento oalgoritmus efektivně aplikovat.
Další aplikování algoritmu *Storage Anarchy* by se mohlo využít ve velkých skladech. Algoritmus by se neaplikoval na celý sklad, ale pouze na menší části skladu, kdy by mezi těmito částmi vedly cesty, které by je spojovali s celým skladem a se všemi *pickery*. Každou část by mělo na starost takový počet robotů, jaký si zákazník určí. Pokud mu záleží na rychlosti dopravy položek, počet robotů by byl vyšší, pokud by zákazníkovi však záleželo na úsporném provozu robotů, počet robotů by byl menší. V tomto případě by se mohlo využít větší kapacity konfigurace vytvořené algoritmem *Storage Anarchy*, kdy efektivita pohybu by nebyla o tolik horší, než při běžné konfiguraci.

Kapitola 5
Závěr

Cílem této práce bylo seznámit se s řešícími algoritmy pro MAPF problém, navrhnout a implementovat plánovací systém pro hledání cest a rekonfiguraci skladu a poté otestovat výkonnost rekonfigurovatelných skladů a srovnat s běžným skladováním co do výkonnosti a efektivnosti.

Výsledkem je program napsaný v C++ využívající algoritmus CBS pro řešení hledání cest pro roboty v rekonfigurovatelném skladě. Implementace skladového anarchismu přijímá více parametrů pro testování nejvýhodnější konfigurace pro daný počet robotů, velikost a tvar skladu. Je možnost využít buď skladový anarchismus, nebo bežné skladování s pevnými místy pro položky. Program po každém spuštění provede dva testy: uložení všech položek do skladu do určité konfigurace a poté vyložení všech položek ze skladu s konfigurací, která byla dána prvním testem. Po každém běhu program jako výstup ukáže *makespan sum of costs* a počet uložených polžek. Právě díky tomu lze otestovat efektivita skladového anarchismu s bežným skladováním.

Výsledek experimentů určil algoritmus *Storage Anarchy* jako efektivní při využití potencionálu úložného místa ve skladu a jako méně efektivní při pohybu a přepravě jednotlivých položek. Zvětšení úložného prostoru je průměrně o 8% vyšší při porovnání s běžnou konfigurací. Časový rozdíl je průměrně o 15% vyšší při porovnání s běžnou konfigurací. S velikostí skladu se tyto hodnoty zvyšují.

Pro větší efektivitu a spolehlivost výsledků by bylo vhodné nahradit algoritmus CBS efektivnějším algoritmem, který by dokázal vždy vyřešit MAPF problém a najít optimální řešení pro danou instanci v adekvátním čase. Pro lepší efektivitu by také mohlo být vhodné využívat více vláken pro rychlejší fungování programu. Dále by mohlo být implementováno UI pro lepší přehlednost a vypracování generátoru map. Po implementaci všech zmíněných věcí by mohl program fungovat jako ideální testovací prostředí pro nové MAPF algoritmy a nové nápady pro konfiguraci skladu.

Bibliografie

1. WURMAN, Peter R.; D'ANDREA, Raffaello; MOUNTZ, Mick. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*. 2008, roč. 29, č. 1, s. 9. Dostupné z DOI: 10.1609/aimag.v29i1.2082.
2. SURYNEK, Pavel. Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. 2019, s. 1177–1183. Dostupné z DOI: 10.24963/ijcai.2019/164.
3. SHARON, Guni; STERN, Roni; FELNER, Ariel; STURTEVANT, Nathan R. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*. 2015, roč. 219, s. 40–66. ISSN 0004-3702. Dostupné z DOI: https://doi.org/10.1016/j.artint.2014.11.006.
4. SURYNEK, Pavel. Time-expanded graph-based propositional encodings for makespan-optimal solving of cooperative path finding problems. *Ann. Math. Artif. Intell.* 2017, roč. 81, č. 3-4, s. 329–375. Dostupné z DOI: 10.1007/s10472-017-9560-z.
5. ČAPEK, Martin. *DPLL(MAPF): Integration of a SAT Solver and Multi-Agent Path Finding*. 2021. Dostupné také z: https://dspace.cvut.cz/handle/10467/99534.
6. ČAPEK, Martin. *DPLL(MAPF): Integration of a SAT Solver and Multi-Agent Path Finding*. 2021. Dostupné také z: https://dspace.cvut.cz/handle/10467/99534.
7. SONIC, Tech. *How Amazon Warehouse robots work* [online]. 2021 [cit. 2022-04-18]. Dostupné z: https://www.youtube.com/watch?v=dLJ6gM4AQI4.
8. SONIC, Tech. *Inside Amazon's Smart Warehouse* [online]. 2020 [cit. 2022-04-18]. Dostupné z: https://www.youtube.com/watch?v=IMPbKVb8y8s&t.
9. LEGACY, Devin Connors. *Amazon's Latest Warehouse is a Robot Winter Wonderland* [online]. 2014 [cit. 2022-04-06]. Dostupné z: https://www.escapistmagazine.com/amazons-latest-warehouse-is-a-robot-winter-wonderland/.

Obsah přiloženého média

```
 readme.txt ....................... stručný popis obsahu média
 src
   impl ........................... zdrojové kódy implementace
     build ........................ pomocná složka na kompilaci a spuštění programu
     CMakeList.txt ................ pomocný soubor pro kompilaci
     README.md .................... stručný popis programu a návod na spuštění
     src .......................... složka obsahující zdrojový kód
   thesis ......................... zdrojová forma práce ve formátu LaTeX
 text ............................. text práce
   thesis.pdf ..................... text práce ve formátu PDF
```