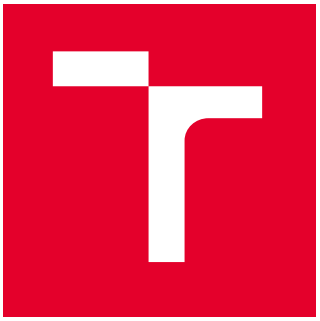
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2023 Bc. Roman Surový



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

ŘÍDICÍ SYSTÉM REGÁLOVÝCH VOZÍKŮ

MULTI-SHUTTLE CONTROL SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Roman Surový**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jiří Přinosil, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Roman Surový ***ID:*** 164946

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Řídicí systém regálových vozíků

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je návrh dílčí koncepce a technické řešení pro řídící systém vozíků, které se pohybují v rámci regálového systému a to jak horizontálně, tak vertikálně. Jedná se zejména o vyřešení problematiky plánování optimální trasy s ohledem na rychlost a bezpečnost. Navržený systém implementujte a ověřte v laboratorních podmínkách.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 19.5.2023

Vedoucí práce: Ing. Jiří Přinosil, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
předseda rady studijního programu

ABSTRAKT

Cielom tejto práce je návrh čiastkovej koncepcie a technické riešenie pre riadiaci systém vozíkov, ktoré sa pohybujú v rámci regálového systému a to ako horizontálne a vertikálne. Ide najmä o vyriešenie problematiky komunikácie medzi jednotlivými vozíkmi a plánovanie optimálnej trasy s ohľadom na bezpečnosť. Navrhnutý systém sa implementuje a overuje v laboratórnych podmienkach. V rámci diplomovej práce je navrhnutý celkový koncept systému, navrhnutý spôsob medzivozíkovej komunikácie a je overený v laboratórnych podmienkach.

KĽÚČOVÉ SLOVÁ

Vertikálna časť, horizontálna časť, regálový systém, trasy, medzivozíková komunikácia, bezpečnosť, Kyvadlová doprava, Dijkstra algoritmus, A* algoritmus

ABSTRACT

The aim of this work is the design of a partial concept and technical solution for the control system of carts that move within the racking system both horizontally and vertically. It is mainly about solving the problem of communication between individual trucks and planning the optimal route with safety in mind. The proposed system is implemented and verified in laboratory conditions. As part of the diploma's work, the overall concept of the system is designed, the proposed method of intercar communication and verified in laboratory conditions.

KEYWORDS

Vertical part, horizontal part, shelving system, routes, inter-carriage communication, safety, Shuttle transport, Dijkstra's algorithm, A* search algorithm

Vysadené pomocou balíčku `thesis` verzie 4.07; `http://latex.feec.vutbr.cz`

SUROVÝ, Roman. *Řídící systém systém regálových vozíků*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 116 s. Diplomová práca. Vedúci práce: Ing. Jiří Přinosil, Ph.D.

Vyhlásenie autora o pôvodnosti diela

Meno a priezvisko autora:	Bc. Roman Surový
VUT ID autora:	164946
Typ práce:	Diplomová práca
Akademický rok:	2022/23
Téma závěrečnej práce:	Řídící systém systém regálových vozíků

Vyhlasujem, že svoju záverečnú prácu som vypracoval samostatne pod vedením vedúcej/cého záverečnej práce, s využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej záverečnej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto záverečnej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podpisuje iba v tlačenej verzii.

POĎAKOVANIE

Rád bych poděkoval vedoucímu diplomové práce panu Ing.Jiřímu Přinosilovi Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Úvod **14**

1 Problematika systému regálových vozíkov **16**
1.1 Skladový kyvadlový systém 16
1.1.1 Definovanie skladového kyvadlového systému 16
1.2 Ako fungujú skladové kyvadlové systémy 16
1.2.1 FIFO first-in-first-out 17
1.2.2 Push-back stojany LIFO 18
1.3 Použitie skladových kyvadlových systémov 19
1.4 Plánovanie viacnásobných kyvadlových vozíkov v automatizovaných skladovacích a zberných systémoch 20
1.5 BFS algoritmus 20
1.5.1 Algoritmy prechodu grafom v dátovej štruktúre 21
1.5.2 Zložitosť algoritmu BFS 21
1.6 Dijkstrov algoritmus 21
1.7 A* star algoritmus 22

2 Simulácia a koncept systému medzivozíkovej komunikácie **25**
2.1 Shuttle systému s horizontálnym aj vertikálnym pojazdom 25
2.1.1 Základné parametre Shuttle systému s horizontálnym aj vertikálnym pojazdom 25

3 Nástroj na vizualizáciu 2D skladu **27**
3.1 Vysvetlenie kódu 27
3.1.1 Main hlavná funkcia 27
3.1.2 Logika Stock simulation 27
3.1.3 Trieda Stock update 29
3.1.4 Trieda Shuttle 30
3.1.5 Trieda StockNode 33
3.1.6 NodeConnection 33
3.1.7 Vylepšenia 33

4 Implementácia algoritmov **36**
4.1 Vylepšenia algoritmov 36
4.1.1 BFS0/BFS 36
4.1.2 Dijkstra 36
4.1.3 A-Star 38

5 Nástroj na vizualizáciu plánovaných úloh **40**
5.0.1 Generovanie máp . . . 40
5.0.2 Časovanie . . . 41
5.1 3D stock . . . 43
5.1.1 3D stock main, a iné časti implementácie . . . 43

6 Simulácia a zhodnotenie efektivity algoritmov **50**
6.0.1 Meranie pre 2D stock . . . 50
6.0.2 Meranie pre 3D stock . . . 53

Záver **63**

Literatúra **66**

Zoznam symbolov a skratiek **68**

Zoznam príloh **69**

A Výpis kódu **71**
A.1 Main . . . 71
A.2 Main pokračovanie . . . 72
A.3 NodeConnection . . . 73
A.4 Shuttle . . . 74
A.5 Shuttle pokračovanie prvé . . . 75
A.6 Shuttle pokračovanie druhé . . . 76
A.7 Shuttle pokračovanie tretie . . . 77
A.8 Shuttle pokračovanie štvrté . . . 78
A.9 Shuttle pokračovanie piate . . . 79
A.10 Shuttle pokračovanie šieste . . . 80
A.11 Shuttle pokračovanie siedme . . . 81
A.12 Shuttle pokračovanie ôsmé . . . 82
A.13 Shuttle pokračovanie deviate . . . 83
A.14 Shuttle pokračovanie desiate . . . 84
A.15 Shuttle pokračovanie jedenáste . . . 85
A.16 Shuttle pokračovanie dvanáste . . . 86
A.17 Shuttle pokračovanie trináste . . . 87
A.18 Shuttle pokračovanie štrnáste . . . 88
A.19 Shuttle3D . . . 89
A.20 Shuttle3D pokračovanie prvé . . . 90
A.21 Stock . . . 91
A.22 Stock pokračovanie prvé . . . 92

A.23 Stock pokračovanie druhé 93
A.24 Stock pokračovanie tretie 94
A.25 StockNode 95
A.26 StockNode3D 96
A.27 StockNode3D pokračovanie prvé 97
A.28 StockSimulation 98
A.29 StockSimulation pokračovanie prvé 99
A.30 StockSimulation pokračovanie druhé 100
A.31 StockSimulation pokračovanie tretie 101
A.32 StockSimulation pokračovanie štvrté 102
A.33 StockSimulation pokračovanie piate 103
A.34 StockSimulation pokračovanie šieste 104
A.35 StockSimulation3D 105
A.36 StockSimulation3D pokračovanie prvé 106
A.37 StockSimulation3D pokračovanie druhé 107
A.38 StockSimulation3D pokračovanie tretie 108
A.39 StockSimulation3D pokračovanie štvrté 109
A.40 StockSimulation3D pokračovanie piate 110
A.41 StockSimulation3D pokračovanie šieste 111
A.42 StockSimulation3D pokračovanie siedme 112
A.43 StockSimulation3D pokračovanie ôsme 113
A.44 StockSimulation3D pokračovanie deviate 114
A.45 StockSimulation3D pokračovanie desiate 115

B Obsah elektronickej prílohy **116**

Zoznam obrázkov

1.1 Shuttle . . . 17
1.2 FIFO . . . 18
1.3 LIFO . . . 19
1.4 Dijkstrov graf . . . 22
1.5 Graf A star . . . 23
2.1 Zobrazenie náčrtku regálovej časti . . . 26
3.1 Cesty jednotlivých nodov a pozície . . . 28
3.2 Zobrazenie prechodu shuttla . . . 30
3.3 Zobrazenie gridov a stavov . . . 34
3.4 Práca desiatich shuttlov . . . 35
5.1 Prepojenia . . . 41
5.2 Cesta . . . 42
5.3 Čas . . . 42
5.4 Prázdne miesta v 3D . . . 45
5.5 Zobrazenie pohybu,čakania a nakladania v 3D . . . 46
5.6 Prechody vozíkov v 3D . . . 47
5.7 3D vizualizácia . . . 48
5.8 Celková 3D vizualizácia . . . 49
6.1 BFS0 2D . . . 50
6.2 BFS 2D . . . 51
6.3 Dijkstra 2D . . . 52
6.4 A-Star 2D . . . 52
6.5 Porovnanie všetkých algoritmov v 2D . . . 53
6.6 Porovnanie BFS, Dijkstra,A-Star 2D . . . 53
6.7 BFS0 3D . . . 54
6.8 BFS 3D . . . 55
6.9 Dijkstra 3D . . . 55
6.10 A-Star 3D . . . 56
6.11 Porovnanie všetkých algoritmov v 3D . . . 57
6.12 Porovnanie BFS, Dijkstra,A-Star 3D . . . 57
6.13 Porovnanie algoritmov pri 50tich meraniach v 3D . . . 58
6.14 Graf práce algoritmov na základe počtu vozíkov . . . 59
6.15 Časoví priemer rozhodnosti algoritmov so zvyšujúcim sa počtom vozíkov 59
6.16 Porovnanie všetkých algoritmov 30 meraní . . . 60
6.17 Porovnanie BFS s A-Star 3D . . . 61
6.18 Porovnanie BFS s Dijkstrou 3D . . . 61
6.19 Porovnanie BFS0 s A-Star . . . 62

6.20 Porovnanie BFS0 s Dijkstrou 62

Zoznam výpisov

A.1 Príklad výpisu kódu Main 71
A.2 Príklad výpisu kódu Main pokračovanie 72
A.3 Príklad výpisu triedy NodeConnnection 73
A.4 Príklad výpisu triedy Shuttle 74
A.5 Príklad výpisu triedy Shuttle pokračovanie prvé 75
A.6 Príklad výpisu triedy Shuttle pokračovanie druhé 76
A.7 Príklad výpisu triedy Shuttle pokračovanie tretie 77
A.8 Príklad výpisu triedy Shuttle pokračovanie štvrté 78
A.9 Príklad výpisu triedy Shuttle pokračovanie piate 79
A.10 Príklad výpisu triedy Shuttle pokračovanie šieste 80
A.11 Príklad výpisu triedy Shuttle pokračovanie siedme 81
A.12 Príklad výpisu triedy Shuttle pokračovanie ôsmé 82
A.13 Príklad výpisu triedy Shuttle pokračovanie deviate 83
A.14 Príklad výpisu triedy Shuttle pokračovanie desiate 84
A.15 Príklad výpisu triedy Shuttle pokračovanie jedenáste 85
A.16 Príklad výpisu triedy Shuttle pokračovanie dvanáste 86
A.17 Príklad výpisu triedy Shuttle pokračovanie trináste 87
A.18 Príklad výpisu triedy Shuttle pokračovanie štrnáste 88
A.19 Príklad výpisu triedy Shuttle3D 89
A.20 Príklad výpisu triedy Shuttle3D pokračovanie prvé 90
A.21 Príklad výpisu triedy Stock 91
A.22 Príklad výpisu triedy Stock pokračovanie prvé 92
A.23 Príklad výpisu triedy Stock pokračovanie druhé 93
A.24 Príklad výpisu triedy Stock pokračovanie tretie 94
A.25 Príklad výpisu triedy StockNode 95
A.26 Príklad výpisu triedy StockNode3D 96
A.27 Príklad výpisu triedy StockNode3D pokračovanie prvé 97
A.28 Príklad výpisu triedy StockSimulation 98
A.29 Príklad výpisu triedy StockSimulation pokračovanie prvé 99
A.30 Príklad výpisu triedy StockSimulation pokračovanie druhé 100
A.31 Príklad výpisu triedy StockSimulation pokračovanie tretie 101
A.32 Príklad výpisu triedy StockSimulation pokračovanie štvrté 102
A.33 Príklad výpisu triedy StockSimulation pokračovanie piaté 103
A.34 Príklad výpisu triedy StockSimulation pokračovanie šieste 104
A.35 Príklad výpisu triedy StockSimulation3D 105
A.36 Príklad výpisu triedy StockSimulation3D pokračovanie prvé 106
A.37 Príklad výpisu triedy StockSimulation3D pokračovanie druhé 107

A.38 Príklad výpisu triedy StockSimulation3D pokračovanie tretie 108
A.39 Príklad výpisu triedy StockSimulation3D pokračovanie štvrté 109
A.40 Príklad výpisu triedy StockSimulation3D pokračovanie piate 110
A.41 Príklad výpisu triedy StockSimulation3D pokračovanie šieste 111
A.42 Príklad výpisu triedy StockSimulation3D pokračovanie siedme 112
A.43 Príklad výpisu triedy StockSimulation3D pokračovanie ôsme 113
A.44 Príklad výpisu triedy StockSimulation3D pokračovanie deviate 114
A.45 Príklad výpisu triedy StockSimulation3D pokračovanie desiate 115

Úvod

Skladové kyvadlové systémy (Shuttle system) sú našim riešením automatizácie skladu, ktorý sa zvyčajne používa v spojení s paletovými regálmi na automatizáciu prepravy paliet z jedného konca regálu na druhý. V tejto práci sme sa zamerali na návrh čiastkovej koncepcie a technické riešenie pre riadiaci systém vozíkov, ktoré sa pohybujú v rámci regálového systému ako horizontálne, tak aj vertikálne. Kde paletové vozíky sa pohybujú automatizovane a len na príkaz operátora. Ide teda hlavne o vyriešenie problematiky komunikácie medzi jednotlivými vozíkmi a plánovanie optimálnej trasy s určitými rýchlosťami a to aj s ohľadom na bezpečnosť.

V prvej časti sme sa zamerali na teoretickú časť problému, kde si zhodnotili, aké sú možnosti a poznania jednotlivého riešenia pre náš shuttle systém. Našou primárnou oblasťou bola na začiatok horizontálna časť problematiky, kde sa odsimulovalo približné uloženie skladu a skladových zásob. To všetko bolo v prostredí simulácie a v laboratorných podmienkach. Rozmiestnenia skladu boli zadané, alebo sa postupne v kóde pridávali. Toto sme spravili či už manuálne, alebo automaticky operátorom, aby bolo zaručené, že sklad sa aj môže zmeniť. To znamená, že sa môže zväčšovať, zmenšovať, alebo inak meniť. To je preto, lebo sa môžu striedať jednotlivé skladové uloženia, pretože je materiál v nich uložený.

Našu simuláciu sme dosiahli pomocou GUI v jave. Sú tam zobrazené jednotlivé miesta v sklade a premiestňovanie vozíkov. Táto scéna mohla byť možno vykreslená a umožňovala nám zobrazovať jednotlivé pohyby vozíkov, shuttlov. Bolo vidno teda nakladanie a vykladanie materiálu. Ako aj jednotlivý postup shuttla. Ďalšia hlavná časť, ktorá nasleduje, je bezpečnosť, pri ktorej by malo byť vidieť jednotlivé stavy shuttla, aby bolo možné eliminovať zrážky alebo iné kolízie. To znamená, že sa tým hlavne vyrieši ich bezpečnosť. Bezpečnosti sa ďalej hlavne rozumie jeho prechod medzi skladovými časťami a zastávkami, pričom aj vyhýbanie sa druhým vozíkom. Keďže je predpoklad, že vozíky môžu mať na prechode medzi jednotlivými zónami aj iné rýchlosti, treba tak dbať obzvlášť na bezpečnosť.

Tiež sme aj hľadali optimálne trasy. Kde vozíky buď hľadali čo najkratšiu trasu k cieľu alebo si našli alternatívnu trasu, čakali, kým sa trasa uvoľní. Samozrejme čakanie je asi tá najkomplikovanejšia časť a snažili sme sa tieto stavy eliminovať. Avšak niekedy je tento stav tiež nevyhnutný, pretože shuttle môže čakať na prácu iného shuttla. Všetky hore spomenuté problémy by mal systém zvládať automaticky a bez zásahu operátora, to znamená, že je úplne automatizovaný. Deje sa tak v horizontálnej aj vertikálnej polohe. Treba bolo najprv urobiť 2D rovinu, teda horizontálnu polohu, a neskôr 3D rovinu, teda aj vertikálnu polohu. V 3D rovine, kde sklad môže mať tiež rôzne veľkosti. Neskôr sme porovnali, implementovali vybrané algoritmy a ďalším poznatkom sa stali merania jednotlivých budúcich vyskúšaných

algoritmov. Ich porovnanie a vykreslenie do jednotlivých grafov. Kde je vidno, ktorý z algoritmov lepšie zvláda náš vykreslený, naprogramovaný sklad. Na základe lepších výpočtov alebo vlastností. Bol porovnaný 2D sklad s 3D skladom na základe závislosti času na počte prác a na veľkosti skladu, použití algoritmov.

1 Problematika systému regálových vozíkov

1.1 Skladový kyvadlový systém

Ako sme v úvode napísali, skladové kyvadlové systémy sú našim riešením automatizácie skladu, ktorý sa zvyčajne používa v spojení s paletovými regálmi na automatizáciu prepravy paliet z jedného konca regálu na druhý.

1.1.1 Definovanie skladového kyvadlového systému

Skladový kyvadlový systém je vlastne mobilný vozík, ktorý nám preváža alebo prepravuje položky v automatizovaných paletových regáloch.

Kyvadlová doprava je systém a typ automatizovaného skladovacieho a vyhľadávacieho systému, ktorý využíva mobilné roboty alebo paletu na horizontálnu jazdu po radoch produktov. Používa sa na skladovanie a vyberanie zásob alebo skladových predmetov, podnosov alebo puzdier v skladovacej pamäti. To znamená, že vozíky presne vedia, kam prísť pre skladovaný tovar. Tieto kyvadlové systémy sú neoceniteľné v skladoch, ktoré používajú metodiku výberu tovaru a majú regály s vysokou hustotou plus vo viacerých prípadoch treba niečo rýchlo priniesť a tu nám pomáhajú hlavne tieto automatické systémy. Prechádzajú hĺbkou nášho regálu, aby sme mohli vyzdvihnúť alebo uložiť palety na oboch koncoch regálovej konštrukcie. V našom prípade sa bude jednať o horizontálne a vertikálne uloženie tovaru. Dokumentáciu môžno nájsť na [1]

Tieto sú ideálne pre sklady s nižším počtom skladového uloženia a s vysokou priepustnosťou, čo vlastne znamená časté nakladanie a vykladanie veľkého počtu paliet na skladovú jednotku. Urýchľujú obeh odchádzajúceho a prichádzajúceho tovaru, pritom vybavujú objednávky a doplňujú tovar, uľahčujú nám správu regálov s hlbokými skladovacími miestami.

To nám umožňuje manažment skladovacích priestorov a tým aj zväčšiť hĺbku regálov, zvýšiť výšku regálových konštrukcií a odstrániť uličky, čím sa maximalizuje každý štvorcový palec skladového priestoru, kedy by sme potrebovali veľké paletové vozíky, ktoré sa musia otáčať, a preto zaberajú miesto.

1.2 Ako fungujú skladové kyvadlové systémy

Skladové kyvadlové systémy využívajú jednu z dvoch možných konfigurácií a tou sú vlastne FIFO alebo LIFO. Tieto dva systémy sú používané aj od toho ako hlavne, či sú čelá stojana prístupné len na jednom konci alebo na oboch. Ovládajú sa buď to

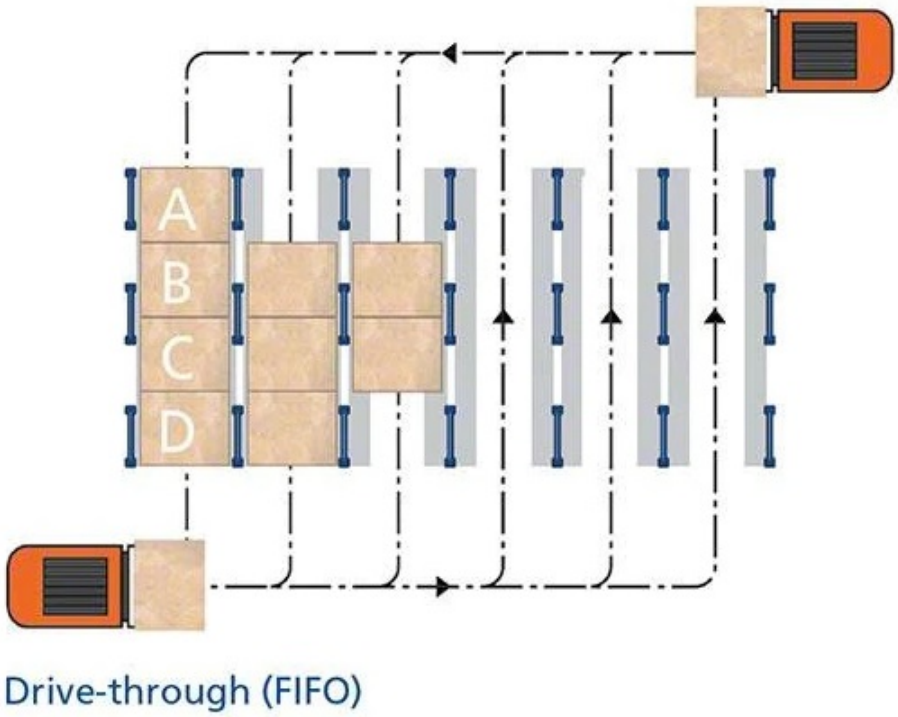
Obr. 1.1: Zobrazenie Shuttlu

diaľkovým ovládaním, rádiovým signálom, alebo pomocou Wi-Fi, alebo po pamäti. Naložené palety sú umiestnené na jednom konci regálu. Ľudský operátor odošle príkaz prostredníctvom správneho signálu k paletovému vozíku. Kyvadlová doprava vyberie naloženú paletu a doručí ju na prvé voľné miesto v jazdnom pruhu. Kyvadlové vozidlá, paletové vozíky jazdia po koľajniciach, ktoré sú integrované do regálovej konštrukcie. Môžu meniť úrovne, a to znamená stúpať a klesať pomocou vertikálnych transportérov alebo skladových výťahov, zvyčajne umiestnených na prednom konci uličky, nie je to však pravidlom, môžu byť aj inde umiestené. To im umožňuje skladovať a vyberať palety na niekoľkých rôznych úrovniach. Dokumentáciu môžno nájsť na [2]

1.2.1 FIFO first-in-first-out

Pre prevádzku FIFO v systéme paliet Shuttle musí byť na konci úrovne umožnený prístup vysokozdvižným vozíkom, čím sa umožní aj prístup k nákladu jednotky, umiestnenému vzadu. To vlastne umožní dokončiť cyklus riadenia zásob FIFO a prvá paleta, ktorá vstúpi do konštrukcie, bude ako prvá radená von. Pred umiestnením posledných paliet do pruhu musí byť AR Shuttle odstránený a umiestnený do nového pruhu. Taktiež FIFO (first-in-first-out) sa tento systém používa tak, aby bol ideálny pre vyrovnávacie jednotky alebo tranzitné sklady, kde sa vstupné a výstupné operácie sa vykonávajú z opačných strán. Keď dráha nie je úplne vyprázdnená a rýchlo

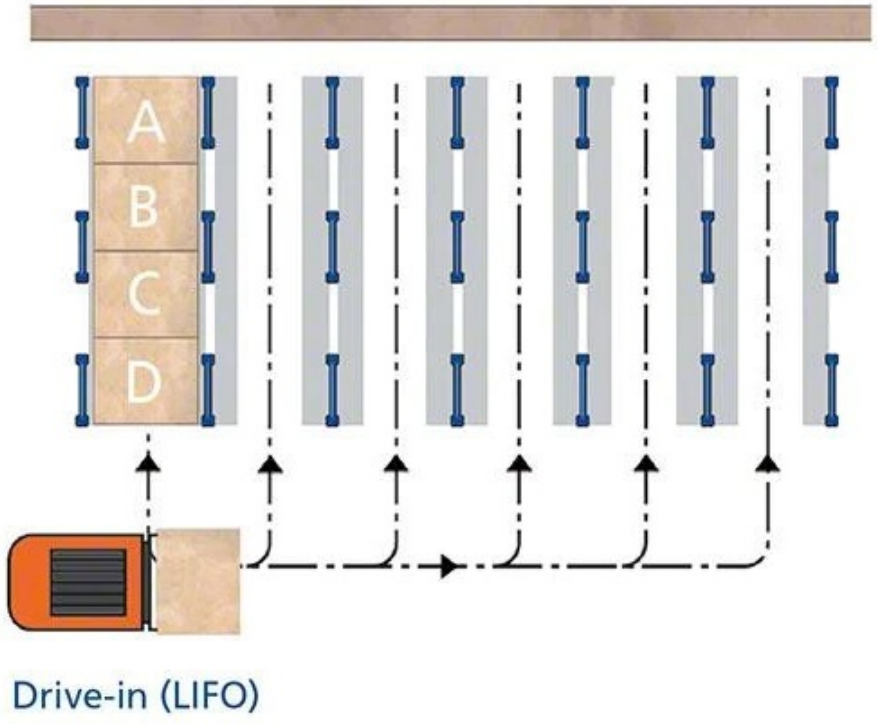
sa tým premiestnia palety tak, že sa priblížia k výstupnej polohe, čo nám umožní umiestniť nové náklady na regál. Dokumentáciu môžno nájsť na [3]



Obr. 1.2: Zobrazenie FIFO skladovania

1.2.2 Push-back stojany LIFO

Naopak pre prevádzku Push-back stojanov v systéme LIFO sa regály využívajú tak, že na prevádzku využijú silu gravitácie, s tým rozdielom, že palety sa ukladajú a vyberajú z rovnakého konca konštrukcie. Každá úroveň regálov má vozík s ložiskami, pripevnený k nosníkom. To nám uľahčuje zatlačenie nákladu dozadu po vložení novej palety a po vybratí sa zvyšok kontrolovane posúva smerom k prednej časti regálu. Táto úprava nám umožňuje pracovať so systémom LIFO a priemernými obratmi zásob. Paletové regály Drive-in/Drive-thru sú tie najpoužívanejšie. V tomto prípade existuje iba jedna prístupová ulička pre vysokozdvižný vozík a fungujú podľa stratégie skladovania LIFO, posledná paleta, ktorá vstupuje, je prvá a ktorá odchádza. Používa sa na tovar, ktorý nepodlieha skaze. Dokumentáciu môžno nájsť na [3]



Obr. 1.3: Zobrazenie LIFO skladovania

1.3 Použitie skladových kyvadlových systémov

Keďže skladové kyvadlové systémy uľahčujú automatické umiestňovanie a vyberanie paliet z hlbokých skladovacích regálov, možno ich použiť na zvýšenie efektivity v hustých skladovacích priestoroch. Malé sklady, ktoré nemajú priestor na nasadenie vysokozdvižných vozíkov, môžu tieto systémy využiť na plnenie objednávok a doplňovanie. Takéto sklady môžu tiež maximalizovať svoj obmedzený skladovací priestor zvýšením výšky a hĺbky regálových konštrukcií.

Pre maximálnu návratnosť investícií by sa kyvadlové systémy mali používať iba v skladoch, ktoré skladujú obrovské množstvá rovnakého typu SKU, ako sú zariadenia na výrobu, alebo distribúciu potravín a nápojov, spracovanie mäsa, chladiarenské sklady atď. Nie sú však ideálne pre distribučné zariadenia, kde sú viaceré SKU na dráhu a viacero položiek uložených na paletách.

V podstate znižujú zdroje a pracovnú silu, potrebnú na presun veľkých objemov podobných produktov a paliet na jeden pruh. Zvyšujú tiež prevádzkovú bezpečnosť, znižujú cestovný čas a náklady na prácu a zlepšujú produktivitu. Dokumentáciu môžno nájsť na [4]

1.4 Plánovanie viacnásobných kyvadlových vozíkov v automatizovaných skladovacích a zberných systémoch

Problém

$$[F, k|IO^2, open, 2n|C_{max}]$$

pozostáva z naplánovania vozíku s ľubovoľným číslom shuttlov, ktoré vykonávajú príkazové cykly. Príkazový cyklus pozostáva z nasledujúcich krokov. Vozík sa zdvihne na prednom konci I/O k položkám, ktoré sa majú uložiť do skladu, sa presunie na prázdnu skladovaciu pozíciu, aby jednu z nich odovzdal, a potom sa presunie na jedno z niekoľkých daných miest, kde je potrebné položky vyzdvihnúť. Tam sa zo stojana vyberie položka, ktorá sa má vybrať, a teraz prázdna priehradka sa naplní položkou, ktorá sa má vybrať.

Základný problém

$$[F, k|IO^2, open, 2n|C_{max}]$$

je založený na niektorých predpokladoch. Keď sa však na problém pozrieme ako na špeciálny typ CVRP, mnohé z týchto predpokladov môžeme relatívne ľahko uvoľniť. Dokumentáciu môžno nájsť na [12]

1.5 BFS algoritmus

Pre náš systém skladu sme vyskúšali BFS algoritmus. Jeho výhody si popíšeme. Algoritmus BFS v dátovej štruktúre je najprv vyhľadávanie do šírky a je to vlastne algoritmus prechodu grafu, ktorý začína prechádzať grafom od koreňového uzla a skúma všetky susedné uzly. Potom vyberie najbližší uzol a preskúma všetky nepreskúmané uzly. Mnoho problémov je tak v teórii grafov môžno vyriešiť pomocou vyhľadávania do šírky. Pri použití BFS pre prechod možno za koreňový uzol považovať ktorýkoľvek uzol v grafe. Napríklad, nájdenie najkratšej cesty medzi dvoma vrcholmi a (a, b) je určené počtom hrán. V tokovej sieti sa na výpočet maximálneho toku používa Ford-Fulkersonova metóda a keď sa binárny strom serializuje alebo deserializuje namiesto serializácie v zoradenom poradí, strom sa dá rýchlo rekonštruovať. Dokumentáciu môžno nájsť na [5]

Existuje mnoho spôsobov, ako prechádzať grafom, ale medzi nimi je BFS najčastejšie používaným prístupom. Je to rekurzívny algoritmus na vyhľadávanie všetkých vrcholov stromovej alebo grafovej dátovej štruktúry. BFS rozdeľuje každý vrchol grafu do dvoch kategórií, a to navštívené a nenavštívené. Vyberie jeden uzol v grafe, a potom navštívi všetky uzly, susediace s vybraným uzlom.

1.5.1 Algoritmy prechodu grafom v dátovej štruktúre

Prechádzanie grafom je technika vyhľadávania na nájdenie vrcholu v grafe. V procese vyhľadávania sa prechod cez graf používa aj na určenie poradia, v ktorom navštívi vrcholy. Bez vytvárania slučiek, prechod grafom nájde hrany, ktoré sa majú použiť v procese vyhľadávania. To znamená, že pomocou prechodu grafom môžete navštíviť všetky vrcholy grafu bez toho, aby ste museli prechádzať slučkovou cestou. Existujú dva spôsoby prechodu grafov, ktoré sú nasledovné: Breadth-First Search, alebo BFS Algoritmus, hĺbka a prvé vyhľadávanie, algoritmus DFS. Dokumentáciu môžno nájsť na [6]

1.5.2 Zložitosť algoritmu BFS

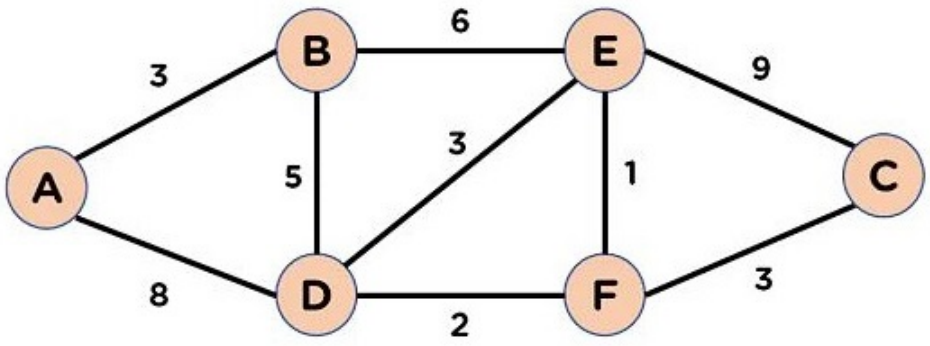
Časová zložitosť algoritmu BFS je vyjadrená v tvare O(V + E), kde V je počet uzlov a E je počet hrán. Priestorová zložitosť algoritmu je O(V). Aplikácie, pre ktoré sa dá využiť BFS algoritmus. Ak by sme chceli vytvoriť index podľa indexu vyhľadávania, pre GPS navigáciu, algoritmy hľadania cesty. V algoritme Ford-Fulkerson na nájdenie maximálneho toku v sieti, detekcia cyklu v neorientovanom grafe a v minimálnom rozpätí. Dokumentáciu môžno nájsť na [8]

1.6 Dijkstrov algoritmus

Dijkstrov algoritmus nám umožňuje nájsť najkratšiu cestu medzi akýmikoľvek dvoma vrcholmi grafu. Odlišuje sa od minimálnej kostry, pretože najkratšia vzdialenosť medzi dvoma vrcholmi nemusí zahŕňať všetky vrcholy grafu. Dijkstrov algoritmus funguje na základe toho, že akákoľvek podcesta B -> D najkratšej cesty A -> D medzi vrcholmi A a D je zároveň najkratšou cestou medzi vrcholmi B a D. Dokumentáciu môžno nájsť na [10]

Dijkstra vlastne používa túto vlastnosť tak, že v opačnom smere nadhodnotí vzdialenosť každého vrcholu od počiatočného vrcholu. Potom môžeme navštíviť každý uzol a jeho susedov, aby sme našli najkratšiu podcestu k týmto susedom. Algoritmus tak používa chamtivý prístup v tom zmysle, že nájdeme ďalšie najlepšie riešenie v nádeji, že konečný výsledok bude našim najlepším riešením celého problému. Časová zložitosť je O(E Log V) kde E je počet hrán a V je počet vrcholov. Priestorová zložitosť:O(V). Tento algoritmus je dobrý na to, aby sme našli najkratšiu cestu, v aplikáciách sociálnych sietí, v telefónnej sieti, a ak chceme nájsť miesta na mape. Dokumentáciu môžno nájsť na [9]

Aplikujme Dijkstrov algoritmus na graf, uvedený na obr. 1.4 a nájdeme najkratšiu cestu z uzla A do uzla C. Na všetky vzdialenosti od uzla a k zvyšku uzlov sú .



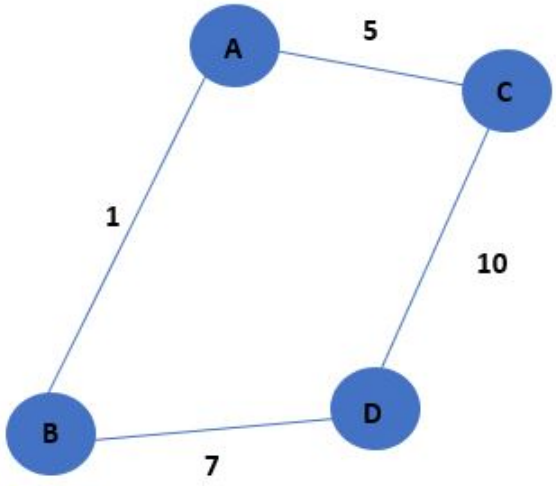
Obr. 1.4: Dijkstrov graf

Výpočet vzdialenosti medzi uzlom A a bezprostrednými uzlami (uzol B a uzol D) pre uzol B, uzol A do uzla B = 3 pre uzol D, uzol A až Uzol D = 8 Vyberieme uzol s najkratšou vzdialenosťou ako aktuálny uzol z nenavštívených uzlov, to znamená uzol B. Výpočet vzdialenosti medzi uzlom B a bezprostrednými uzlami pre uzol E, uzol B do uzla D = 3+5 = 8 pre uzol E, uzol B do uzla E = 3+6 = 9. Ďalej vyberieme uzol s najkratšou vzdialenosťou ako aktuálny uzol z nenavštívených uzlov, to znamená uzol D. Výpočet vzdialenosti medzi uzlom D a bezprostrednými uzlami. Pre uzol E, uzol D k uzlu E = 8+3 = 11 ([9 < 11] > pravda: Takže žiadna zmena) pre uzol F, uzol D k uzlu F = 8+2 = 10. Nasleduje výber uzla s najkratšou vzdialenosťou ako aktuálny uzol z nenavštívených uzlov, znamená uzol E. Výpočet vzdialenosti medzi uzlom E a bezprostrednými uzlami. Pre uzol C, uzol E do uzla C = 9+9 = 18 pre uzol F, uzol E do uzla F = 9+1 = 10. Nakoniec vyberieme uzol s najkratšou vzdialenosťou ako aktuálny uzol z nenavštívených uzlov, to znamená uzol F. Výpočet vzdialenosti medzi uzlom F a bezprostrednými uzlami. Pre uzol C, uzol F na uzol C = 10+3 = 13 ([18 < 13] nepravda: Takže sme zmenili predchádzajúcu hodnotu). Takže po vykonaní všetkých krokov máme najkratšiu cestu z uzla A do uzla C, teda hodnotu 13 jednotiek. Dokumentáciu môžno nájsť na [11]

1.7 A* star algoritmus

Je to vyhľadávací algoritmus, ktorý sa používa na nájdenie najkratšej cesty medzi počiatočným a konečným bodom. Je to veľmi šikovný algoritmus, ktorý sa často používa na prechádzanie mapou na nájdenie najkratšej cesty, ktorou sa má vydať. A* star bol pôvodne navrhnutý ako problém s prechodom grafu, aby pomohol postaviť nejakého robota, ktorý dokáže nájsť svoj vlastný kurz. Stále zostáva široko populárnym algoritmom na prechádzanie grafom. Najprv hľadá kratšie cesty, čím

sa stáva optimálnym a úplným algoritmom. Optimálny algoritmus nájde výsledok s najnižšími nákladmi na problém, zatiaľ čo úplný algoritmus nájde všetky možné výsledky problému. Ďalším aspektom, ktorý robí A* tak silným, je použitie vážených grafov pri jeho implementácii. Vážený graf používa čísla na vyjadrenie nákladov na každú cestu alebo postup. To znamená, že algoritmy môžu ísť cestou s najnižšími nákladmi a nájsť najlepšiu trasu z hľadiska vzdialenosti a času. Neformálne aj povedané, vyhľadávacie algoritmy A* star majú na rozdiel od iných techník prechodu „mozgy“. To znamená, že je to skutočne inteligentný algoritmus, ktorý ho oddeľuje od ostatných konvenčných algoritmov. A tiež stojí za zmienku, že mnoho hier a webových máp používa tento algoritmus na veľmi efektívne nájdenie najkratšej cesty (aproximáciu). Dokumentáciu môžno nájsť na [13]



Obr. 1.5: Graf A star

Ako funguje A* star algoritmus. Zoberme si vážený graf, zobrazený na obr. 1.5, ktorý obsahuje uzly a vzdialenosť medzi nimi. Povedzme, že začíname od bodu A a musíme ísť do bodu D.Teraz, keďže začiatok je v zdroji A, ktorý bude mať nejakú počiatočnú heuristickú hodnotu. Preto sú výsledky f(A) = g(A) + h(A), f(A) = 0 + 6 = 6. Ďalej je potreba prejsť cestou k ďalším susedným vrcholom: f(AB) = 1 + 4, f(AC) = 5 + 2 Teraz treba na cestu k cieľu z týchto uzlov a vypočítajte váhy: f(ABD) = (1+7) + 0, f(ACD) = (5 + 10) + 0. Je jasné, že uzol B nám poskytuje najlepšiu cestu, takže je to uzol, ktorým sa musíme vydať, aby sme sa dostali do cieľa. Dá sa aj tak, že uvažujme, že máme štvorcovú sieť s mnohými prekážkami a dostaneme počiatočnú a cieľovú bunku. Chceme sa dostať do cieľovej bunky (ak je to možné) z počiatočnej bunky čo najrýchlejšie. Tu A* star algoritmus prichádza na záchranu. Algoritmus A* star robí to, že v každom kroku vyberie uzol

podľa hodnoty-’ f ’, čo je parameter rovný súčtu dvoch ďalších parametrov „ g “ a „ h “. V každom kroku vyberie uzol alebo bunku s najnižším „ f “ a spracuje tento uzol/bunku. Keď si zadefinujeme „ g “ a „ h “ čo najjednoduchšie pod g= cena pohybu na presun z počiatočného bodu na dané políčko na mriežke po vygenerovanej ceste tam. A h = odhadované náklady na presun z daného políčka na mriežke do konečného cieľa. Toto sa často označuje ako heuristika, čo nie je nič iné ako druh inteligentného odhadu. Skutočnú vzdialenosť naozaj nepoznáme, kým nenájdeme cestu, pretože v ceste môžu byť rôzne veci (steny, voda atď.). Dokumentáciu môžno nájsť na [14]

2 Simulácia a koncept systému medzivozíkovej komunikácie

2.1 Shuttle systému s horizontálnym aj vertikálnym pojazdom

Keďže výstupom tejto semestrálnej práce je koncept a simulácia medzivozíkovej komunikácie, je treba navrhnúť celkový systém simulácie a aplikáciu, ktorá by nám takú komunikáciu umožňovala.

2.1.1 Základné parametre Shuttle systému s horizontálnym aj vertikálnym pojazdom

Jednotlivé rýchlosti, zrýchlenia a časy:

- Rýchlosť horizontálna medzi regálmi do 5m/s
- Rýchlosť vertikálna a do boku prejazd medzi uličkami do 1m/s
- Zrýchlenie pre všetky smery do 1m/s
- Zmena smeru vozíka vo všetkých smeroch na križovatkách do 2s
- Rýchlosť odovzdania a naberania prepravky na odovzdávacom mieste prebieha súbežne do 3s
- Rýchlosť uskladnenia/vyskladnenia na regálovej pozícii do 5s

Definovanie regálov:

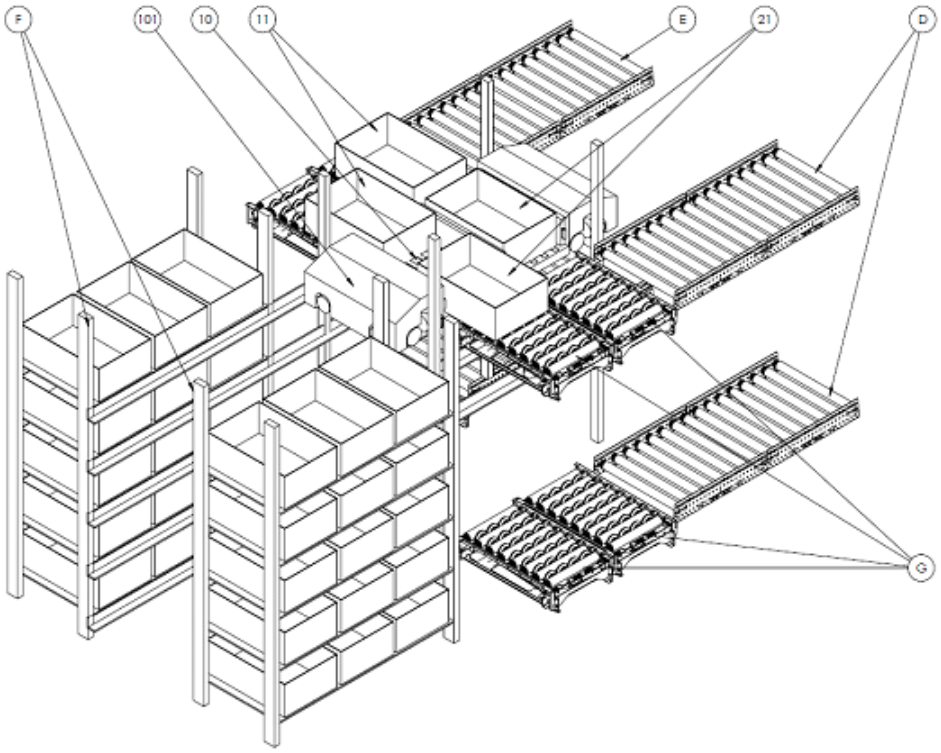
- Štandardné policové regály s výškou 10+m (pri 10m cca. 26 políc)
- Štandardná výška police: 38cm (môže byť aj rôzna v rôznych častiach systému – prepravky môžu byť rôzne vysoké)
- Počet uličiek: max. 30
- Regálová pozícia: definovaná pôdorysom skladovej jednotky, štandardne 0,5m (pri prepravke 600x400mm)
- Orientácia v rámci regálov: každá regálová pozícia má presné umiestnenie v regálovom systéme: ulička, poschodie, pozícia od začiatku uličky. Pre výpočet dôb presunov bude potrebný prepočet na skutočné „ubehnuté“ vzdialenosti.

Prejazdové šachty:

- V systéme budú použité šachty pre vertikálny a bočný prejazd. Každá šachta je buď pre vertikálny alebo pre horizontálny prejazd (nikdy nedochádza ku kríženiu všetkých smerov). Šachty sa môžu nachádzať kdekoľvek v systéme, ale väčšinou budú na začiatku regálov (pred regálmi).
- Horizontálne šachty: v systéme môže byť použité 0-n týchto šácht, reálne 0-2, štandardne 1 horizontálny prejazd nemusí byť v každom policovom poschodí.

Pokiaľ má systém 26 poschodí, môže byť prejazd realizovaný, napr. v každom treťom poschodí.

- Vertikálne šachty: v systéme môže byť použité 0-n týchto šácht, reálne 1-2, štandardne 1-2 Šachty môžu byť použité pre jednosmerný alebo obojsmerný pohyb – malo byť možné definovať
- Odovzdávacie miesta: Systém bude vybavený n-odovzdávacími miestami, ktoré môžu byť rozmiestnené kdekoľvek. Štandardne sa počíta s umiestnením priamo vo vertikálnych šachtách, ale môžu byť aj mimo nich. Štandardné umiestnenie na obr. 2.1.



Obr. 2.1: Zobrazenie náčrtku regálovej časti

3 Nástroj na vizualizáciu 2D skladu

3.1 Vysvetlenie kódu

Celý kód v podstate začína v maine, štandartne je takto volaná hlavná funkcia pri spustení programu. Cez main je to tak vo väčšine prípadov kódov. Predbiehame, ale na úvod povieme, že sme používali prostredie IntelliJ IDEA Community Edition 2023.1, ktoré je zadarmo a dobre sa v ňom programuje, patrí v recenziách medzi tie najlepšie.

3.1.1 Main hlavná funkcia

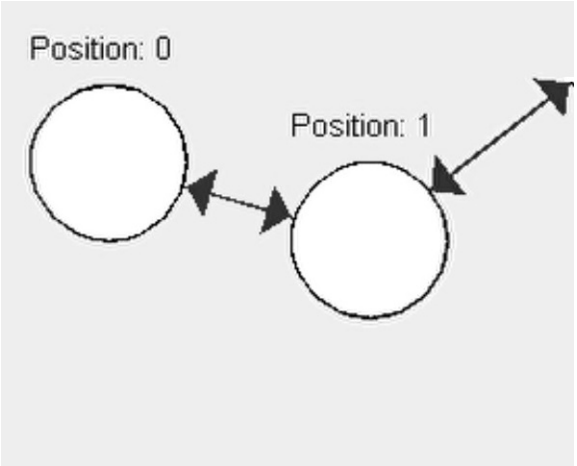
V maine sa vytvára "StockSimulation simulation" trieda alebo objekt. Táto trieda zaobaľuje celý stock a všetky shuttle, zabezpečuje našu simuláciu a celý jej beh. Vlastne sa jedná o GUI, window (okno) a to nám vykresľuje našu simuláciu. Je to taktiež trieda (objekt) a zabaľuje sa nám tam celý stock. Zabezpečuje v podstate celú našu simuláciu. Na to, aby nám to fungovalo, je tam využitý frame work. Volá sa swing, ten sa vlastne používa v JFrame. Na základe neho sa nám tým vytvorí frame. A keďže Frame je vlastne okno, do ktorého hodíme v našom prípade simuláciu alebo v iných prípadoch elementy. Ďalej zavoláme pack, kde sa nám nastavia resolutions. Zavoláme inicializačné funkcie, a tým nakoniec máme zobrazenie. To znamená, že tým vlastne Jframe nám zobrazí simuláciu. Toto je celá naša logika mainu. Main môžno vidieť tu A.1.

3.1.2 Logika Stock simulation

Ďalšou triedou je trieda Stock simulation. Naša logika sa vykonáva v stock simulation, je to naša simulácia skladu. Stock simulation dedí po JPaneli a je to tiež vlastne zo swingu a aj to, že je to knižnica swing na GUI pre Javu. Keď chceme, aby sa niečo zobrazovalo, tak musíme pridať tejto triede, aby dedila (extends) alebo aj dedí po JPaneli. Na pozadí nám to zabezpečuje, aby sa vytvorilo plátno alebo aj niečo, čomu musíme nastaviť veľkosť. Do plátna nám to následne vykresľuje, čo chceme. To je vlastne dedenie a je to kvôli GUI.

Na začiatok vypíšeme, že chceme, aby sa začala stock simulácia. Nastavíme si veľkosť okna v "setPreferredSize", v našom počiatočnom prípade to bolo 600x400, kde sa to dá editovať.

Ďalej prídeme k samotným nódom, ktoré nastavujeme tak, že parameter tohto vytvorenia nódu je jeho pozícia v pixeloch, napríklad, nód v pozícii x = 100 a y = 200.



Obr. 3.1: Cesty jednotlivých nodov a pozície

Keď ich tam pridávame, tak súčasne s tým, aby sme ich spojili, musíme zavolať funkciu connectnodes. Tá nám zabezpečuje, aby sme mali, napríklad, desať nódov. Keďže, ale nechceme vlastne, aby bol každý s každým prepojený. Kvôli tomu je tam táto samostatná funkcia, že je vytvorený nód samostatne a dokážeme ho tak pospájať. Ony sa dajú spojiť tak, že prvý parameter tejto funkcie connectnode je sourceNodeIndex, čiže začiatok, a potom druhý je destinationNodeIndex koniec, to znamená, že napríklad, prvý spojíme s druhým nódom. Podobu, ako vyzerá toto nastavenie môžno vidieť na obr. 3.1 Ďalej môžeme nastaviť maximálnu rýchlosť medzi nódami, zatiaľ v našom počiatočnom prípade na jedna. Pretože naše prechody môžu mať aj iné rýchlosti a tým sme vlastne zabezpečili, aby sme to do budúcnosti vedeli použiť, ak to bude treba. Ďalším parametrom je isBothDirectional, ktorý nám vlastne zabezpečuje, že je connection obojsmerný, alebo aj nemusí byť. To znamená, že môže byť jednosmerný a tým by sme sa vedeli hýbať len jedným smerom. Je to kvôli tomu, aby sa dalo ísť len jedným smerom napríklad horizontálne a potom následne len vertikálne. Je to hlavne do budúcnosti, ak by náš shuttle išiel len jedným smerom. Takto si môžeme nahádzať nódy, koľko ich budeme potrebovať, využívať. Ako vyzerá Stock Simulation môžno vidieť tu A.28

Ďalším bodom je vytvorenie shuttlov. Každý shuttle má svoje joby (práce), ktoré musí vykonávať a samozrejme aké mu ich nastavíme. Môže ich nastavovať, napríklad, operátor. Sú to pozície, na ktoré musí ísť, ktoré musí vykonať, aby dokončil prácu. Get nám vyjadruje spojku, ktorej priradíme napríklad dvojku, čo vlastne v našom prípade znamená, že pôjde na tretí, lebo sa indexuje od nuly. Neskôr toto poupravujeme, aby to pre nás bolo viacej intuitívne. Ako ďalej vidno, na simulácii pozície sú od nuly a shuttle je na tom tiež podobne, čiže od nuly.

Nasledujúci bod v kóde je Timer. Ten sa vytvára preto, lebo je potrebná aktu-

alizácia, alebo updatovanie (refresh). To znamená, že GUI vlastne potrebuje obnovovanie, a my jemu ho musíme nadefinovať, pretože inak by nám ho nezobrazovalo. Vždy mu musíme volať funkciu update, ktorá sa reálne volá repaint. Vykresli nám znova sklad, alebo inak povedané nám vykreslí scénu. Pre timer sme nastavili delay, čas desať, kde to znamená, že sa obnoví každých desať milisekúnd.

Nasledujúcou funkciou je update, teda keď zavolá repaint sa vlastne vnútorne nakonfiguruje a zavolá si paintComponent. To nám vlastne zabezpečí celú funkciu a vykreslia sa scény.

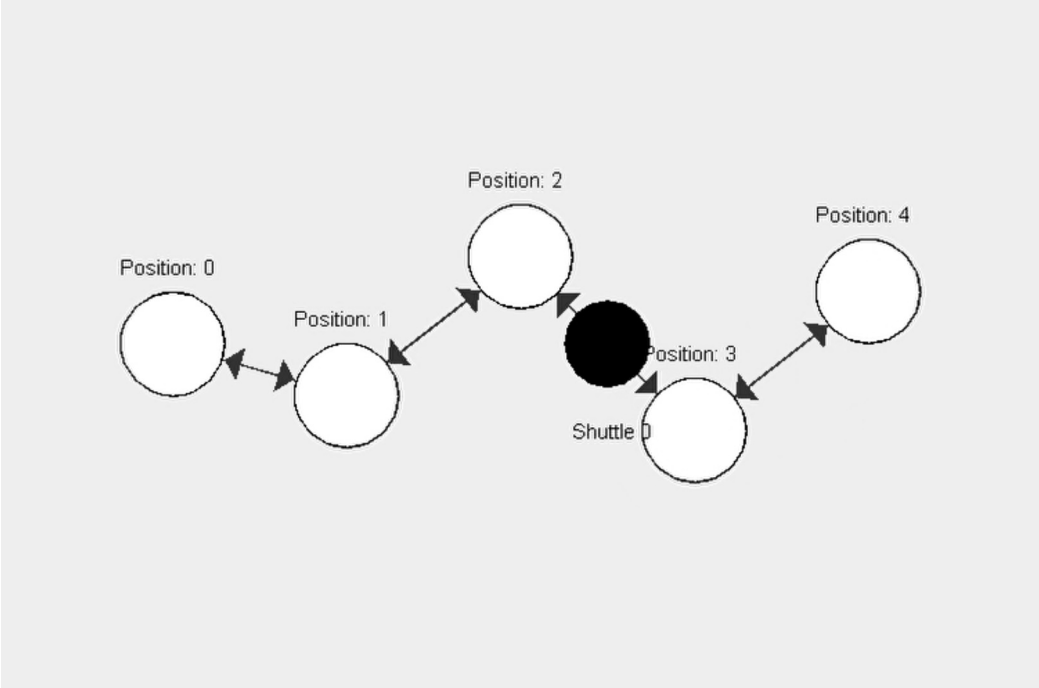
PaintComponet je na začiatku vykreslenia v nód Connections. Ďalej sa tu vykresľujú nódy a nakoniec sa aj vykresľujú shuttle. Každý môže mať aj rôzne farby ako si ich nastavíme. V shuttloch sa vypisuje aj ich identifier, čo znamená, že v sebe nesie názov. Dobré je to kvôli tomu, ak by sme chceli vedieť, do akej pozície shuttle ide, a tak, napríklad, za shuttle.identifier doplníme + "pozícia xy" je to do budúcna, keď to budeme potrebovať. Samozrejme to iste platí pre stock positions, čo sú vlastne pozície v sklade. Tiež sa nám vykresľujú s farbami, aké im nastavíme, v našom prípade sú to momentálne pre shuttle čierne a pre Stock biele, position je tiež čierna. Podobu, ako vyzerá toto nastavenie môžno vidieť na obr. 3.2.

3.1.3 Trieda Stock update

Tu je riešený update celej scény, je to dôležitá časť, kedže update je treba pre refresh. Zo začiatku update skontroluje, či v sklade je požadovaný počet shuttlov. Vieme si aj nastaviť cez maximalShuttlesInStock, koľko shuttlov tam v tom sklade môže byť, to znamená, že ich vieme napríklad obmedziť, či pustiť viac. V našom počiatku sme nastavili jeden a neskôr ich budeme pridávať viac. Toto obmedzenie znamená, že v tejto situácii môže byť v sklade len jeden. Vieme ich však samozrejme nastaviť aj takmer nekonečno. Ak nejaký shuttle dokončí svoju prácu, tak potom sa nastaví do stavu Finish, znamená to, že vypadne zo scény a môže tam v tomto prípade ísť ďalší shuttle. Zabezbečuje nám to funkcia na začiatku updatu. Čiže skontroluje, či je v sklade nejaký shuttle a pridá ho tam. Ak už tam nejaký shuttle je, cez for sa shuttle updatujú, posúvajú sa ďalej. Pozrie sa, kde sa nachádza aktuálne a posunie o nejaký úsek dopredu k cieľu. To nám zabezpečuje náš update. Každý shuttle má svoj update, ktorý sa kontroluje, či prišiel už do cieľa, či už dokončil všetky jobs, práce. Vypočítava sa mu, kde ešte všade má ísť. Všetky tieto rutiny sa zabezpečujú cez tento update. Všetky premenné (ArrayList), ktoré stock obsahuje, sú v public class Stock. Sú tu nódy, čo sú naše pozície v sklade. Nasledujúce sú shuttle, ktoré sú v sklade alebo už boli v sklade, budú v sklade. Ako vyzerá Stock update môžno vidieť tu A.21.

NodeConnection vlastne vie o nódoch v sebe, a s ktorým je spojený, priamo

v nóde. Ak máme nejakú pozíciu v sklade, ona vie (nodes), kto sú jej susedia. Vie si aj vyčítať, ako ďaleko sú od nej, akou rýchlosťou sa môžeme dostať k susedovi. Využíva sa to v tom, keď shuttle sa nachádzajú na nejakej pozícii a chceme, aby nám išli ďalej.



Obr. 3.2: Zobrazenie prechodu shuttla

3.1.4 Trieda Shuttle

Samotný shuttle obsahuje v sebe na začiatku stavy, aké môžu nastať:

- Ready – Znamená, že je pripravený na začatie práce (ešte nezačal, ale je pripravený)
- Moving – Znamená, že sa hýbe niekam medzi dvoma nódami
- Loading – Znamená, že nakladá materiál (nastavenie času nakladania)
- Waiting – Znamená, že čaká na uvoľnenie trasy, napríklad, ak je trasa zablokovaná ešte nie je vo finishe a vlastne stojí. (nehýbe sa)
- Finish – Znamená, že dokončil svoju prácu a dostal sa mimo skladu vyšiel z neho

Nasleduje conuter, ktorý je zo začiatku nastavený na nulu. Je to pre nás len počítanie, koľko je vytvorených shuttlov v programe. Position je pozícia toho shuttlu. Ďalej sú jeho jobs, práce, teda tasky, ktoré musí spraviť, alebo inak povedané práce,

ktoré musí vykonať. Nasledujú pozície, ktoré musí naložiť a jeho aktuálna trasa (currentPath), ktorú vykonáva. Potom aktuálny currentNode, na ktorom sa nachádza. Tieto tri sú v podstate najdôležitejšie, aby sa nám vedel hýbať lokalizovať. Ďalej identifire, to je vlastne len ako sa volá. State je stav, ktorý aktuálne má v triede, čo nám práve robí.

Ďalej máme pomocné metódy ako, napríklad, instock. Ten nám vracia informáciu, či sa shuttle nachádza v sklade. To znamená, že môžu nastať takéto prípady, že shuttle môže byť ešte len v stave ready, ale nie je v sklade, je v stave finish, čo znamena taktiež, že nie je v sklade a dokončil prácu. Ak je v sklade, buď to hýbe, nakladá, alebo čaká (Moving, Loading, Waiting). Funkcia nám vlastne vracia, či je shuttle v sklade fyzicky. Ďalej funkcia nám zisťuje, či jeho môžeme vložiť do scény, skladu. Ak už dokončil prácu, alebo ešte ani nezačal a už je v sklade. Zaisťuje nám, že ho nemôžeme len tak hodiť do skladu, aj druhýkrát. SetStartNode je pomocná funkcia na štart. Dir je na zistenie pozície.

FindParthToCurrentJob je nosná funkcia hľadania cesty. Keď sa shuttle dostane do stavu, kedy si naložil náklad a nemá vytvorenú ďalšiu cestu, tak sa zavolá táto funkcia a tá mu nájde novú cestu. Buď mu nájde novú cestu k nódu, alebo mu nájde novú cestu von zo skladu (vyjde). Vo VisitedNodesMap sa kontrolujú všetky nódy, a v prvotnej verzii bol použitý algoritmus BFS, ktorého znenie sme mohli vidieť v teoretickej kapitole. V našom prípade to funguje tak, že sa ide po každom nóde a hľadá sa ten cieľový a ak ho vieme nájsť, tak sa skonči hľadanie a skončí sa cesta.

Funguje to v podstate tak, že my vieme všetky nódy, a na základe toho si vytvoríme mapu už navštívených nódov (visitNodesMap) do nej si ukladáme všetky nódy, v ktorých sme boli alebo sme sa v nich nachádzali raz. Ideme tak cez všetky nódy (visitNodesMap.put(initPath.get(0)). Na začiatku ideme na počiatočný. čo je náš inicializačný nód. Zistíme. kde ten inicializačný kód má connections to je v (visitedNodesMap.put(connection.destionationNode)). Prejdeme cez všetky jeho connections. Spýtame sa, či už sme sa v jeho connectionNodes raz nachádzali (visitedNodesMap.conectionsKey) a aj na tej pozícii. Ak sme sa nenachádzali, zabezpečuje nám to výkričník. A ak sme sa tam ešte nedostali, tak si ho tam uložíme (visitedNodesMap.put) a vytvoríme si novú cestu (newPath), lebo ešte sme sa v tomto nóde nenachádzali. Takže si vytvoríme novú cestu a ideme stále dookola, až pokým ju nenájdem, alebo neprejdem cez všetky nódy. To nám zabezpečuje while(newNodeFound). Ktorý keď už sme našli všetky nódy, že už sme žiadny nový nenavštívili. Tak z tohto while vyskočíme a ideme do for cyklu, v ktorom skontrolujeme všetky tie nájdené cesty, ktoré boli v predošlom while a ktoré sme navštívili. Prejdeme všetky tie cesty a skontrolujeme, či jedna s ciest nevedie do nášho cieľového nódu (if(part.get(0) == currentNode..) get0 nám vráti prvý nód tej cesty a súčastne sa spýtame, či sa tam v tom nóde aktuálne nachádza shuttle a ešte súčastne posledný nód je nód kam

má ísť shuttle, vlastne kde má jobs, prácu. Ak to tak je a splní sa táto podmienka príkazu, tak nastavíme túto cestu pre shuttle a vyskočíme z tohoto for-u. Znamená to, že potrebujeme nastaviť cestu zo všetkých vygenerovaných, ktorá nám vyhovuje a tú následne nastaviť. Inak povedané, cesta, ktorá nás vedie od toho, kde sme tam, kam chceme isť. To je algoritmus hľadania cesty, neskôr pripíšeme ďalšie update hľadania alebo použijeme iný algoritmus.

Náš public void update, ako sme povedali tak je volaný z triedy stock a znamená, že sa každý jeden shuttle vlastne updatne a zároveň sa aj jeho pozícia updatne. Keď sa nám hýbe, posunie sa o niečo ďalej k nášmu cieľu, ak sa však nehýbe, tak čaká. Ak už dokončil, čo mal zadané, teda prácu, tak sa mu zavolá funkcia, nech si nájde novú prácu. Logika celého update je v tom, že ak ten shuttle už nie je v sklade, tak nepotrebuje žiaden update. Rovno da return inak ak ten shuttle ešte neskončil svoju prácu, tak sa pýtame, či ešte má prácu alebo či má prácu „(job.isEmpty() and curentPath.isEmpty())“ ak má všetko prázdne, tak svoju prácu dokončil a prejde do stavu Finished, čiže všetko dokončil.

Nasleduje funkcia, ktorá zabezpečuje naše nakladanie. Tá je zabezpečená tým, že ak sa shuttle nachádza na požadovanej pozícii, spustí sa nakladanie „start loading procedure“. Čas očakávania na nakladanie sa dá nastaviť v loadingTime. Momentálne je to nastavené tak, že každý má rovnaký čas, do budúcnosti sa to môže zmeniť, aby boli v daných nódoch rôzne časy. Ak dokončí túto prácu, teda nakladanie, tak sa zmení nód na aktuálny a na ten, ktorý dokončil. Odstráni sa mu práca, lebo ju dokončil a následne sa mu nájde nová cesta, aby sa mohol pohnúť. Zmení sa mu stav na moving, čiže sa hýbe a na ďalší smer alebo sa rozbehne. Je tam možnosť, že ešte súčasne, ak má nejakú prácu, ale nemá cestu, tiež sa mu vygeneruje nová cesta. Takáto situácia nastane, iba keď sa vloží. Ak sa vloží, nemá prázdne joby, ale má prázdnu cestu. Takáto situácia môže nastať, ak sa vloží do scény. Ak shuttle má ďalej nejakú cestu a má cieľ, napríklad, má cestu cez štyri nódy, ide postupne prvý, druhý, tretí, štvrtý „currentPath.remuve(index: 0)“. Je tam taktiež kontrola, keď príde do prvého, tak prvý sa mu vymaže z tej cesty, a ďalej môže ísť do druhého. Týmto spôsobom tam nemá uloženú celú cestu, ale iba tú, čo pred ním len je, teda budúcu cestu. Preto sa maže ta pozícia, z ktorej prišiel. Tá sa nám vymaže a prichádza update pozície. Ak ani jedna z predošlých podmienok nebola splnená, znamená, že shuttle sa niekde hýbe a to znamená, že sa môže pohnúť bližšie k svojmu cieľu. Toto je nastavenie „curentPath.get(0).position“. Nastaví sa jeho pozícia buď to na požadovanú, alebo sa posunie k tej požadovanej. Tu je vlastne zatiaľ celá logika posúvania. Ako vyzerá trieda Shuttle môžno vidieť tu A.4.

3.1.5 Trieda StockNode

Táto trieda obsahuje vlastne pozíciu v sklade momentálne Point2D. Nachádza sa tu pozícia naskladania. Obsahuje aj ako dlho sa uskladňuje daný materiál. Ak na tomto mieste prepíšeme Point2D na 3D, mali by nám všetky funkcie ostať a fungovať v podstate 3D sklade. Máme to napísane tak, že to je pripravené na 3D. S tým súvisí aj algoritmus hľadania trasy, ktorý by mal tiež fungovať, tým sa vlastne dostávame k vertikálnej časti práce, ktorá bude v budúcnosti obsiahnutá v tejto práci. Ako vyzerá trieda StockNode môžno vidieť tu A.25.

3.1.6 NodeConnection

Táto trieda obsahuje vzdialenosť medzi jednotlivými nódmi. Nasleduje rýchlosť medzi nódmi. Ďalej pointer destinationNodes. Ako vyzerá trieda NodeConnection môžno vidieť tu A.3.

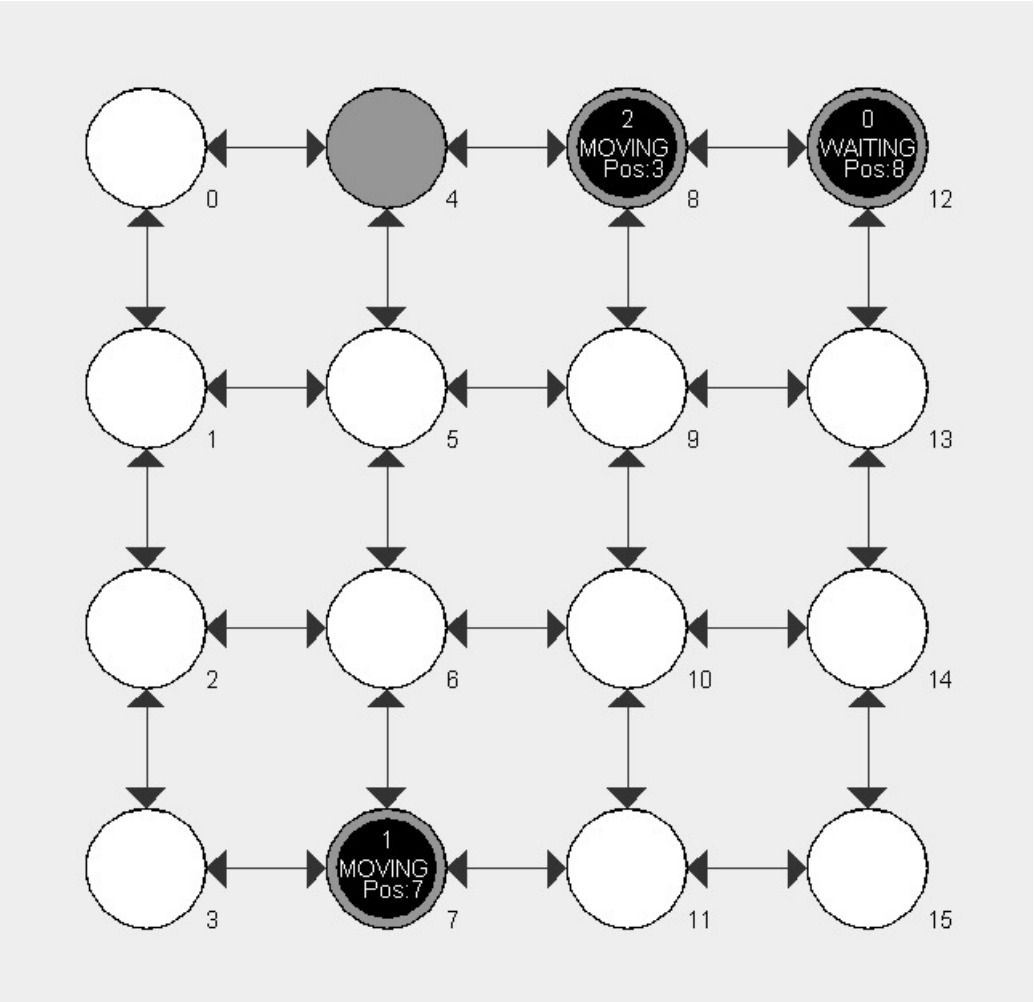
3.1.7 Vylepšenia

V prvom rade sme v kóde pridali generovanie gridu, možno to vidieť na obr. 3.3. Následne sme vyriešili, aby sa momentálne hľadala najrýchlejšia cesta a na shuttly sa vždy vypíše, na akú pozíciu ide a čo práve robí. Ak sa hýbe. píše moving, zastavípíše loading. Je to lepšie, aby sme videli, čo sa práve deje a aké stavy sa vykonávajú, a aj to, na akú pozíciu majú naplánované ísť. Ďalším vylepšením je, že ak sa pozície vykreslia sivou farbou, tak tá pozícia je obsadená, locknutá. Vtedy tam nikto nemôže ísť, lebo má locknutú pozíciu niekym iným. Je to kvôli tomu, aby sa na danú pozíciu nenaplánoval iný shuttle, aby sa v podstate nezrazili, je to pre ich bezpečnosť.

Joby sa aktuálne generujú random. Zaisťuje nám to random generátor. Shuttle momentálne vďaka funkcii vykonáva desať jobov, každému novému vytvorenému shuttlu sme pridali tieto joby. Vykonáva sa to v "stock.shuttles.get.."je to vlastne náhodne generovanie a vždy ho vykonáva nový shuttle, ktorý v podstate vyjde zo skladu. Vďaka tomu, sa mu nahádžu pozície a ony ich postupne vykonávajú.

V stock simulation sme pridali funkciu, aby sa dalo pridávať veľkosť gridu. Ak zväčšíme grid, treba v tomto prípade zväčšiť aj veľkosť plátna, aby sme videli všetky pozície.

V triede Shuttle a currentPath každý nodes dostal parameter occupied, to znamená, že vie, či je obsadený, či do neho môže ísť iný shuttle, a to všetko nám zabezpečuje premenná occupied. Ak je nód obsadený, tak doň nemôže ísť žiadny iný shuttle, iba ten, ktorý si ho predtým obsadil. Je to aj hlavná funkcia, ktorú sme upravili. Ďalej sme upravili, aby do scény mohlo vstupovať viacej shuttlov, konkrétne toto obmedzenie je nastavené v parametri ïnt maximalShuttlesInStock"v triede stock.

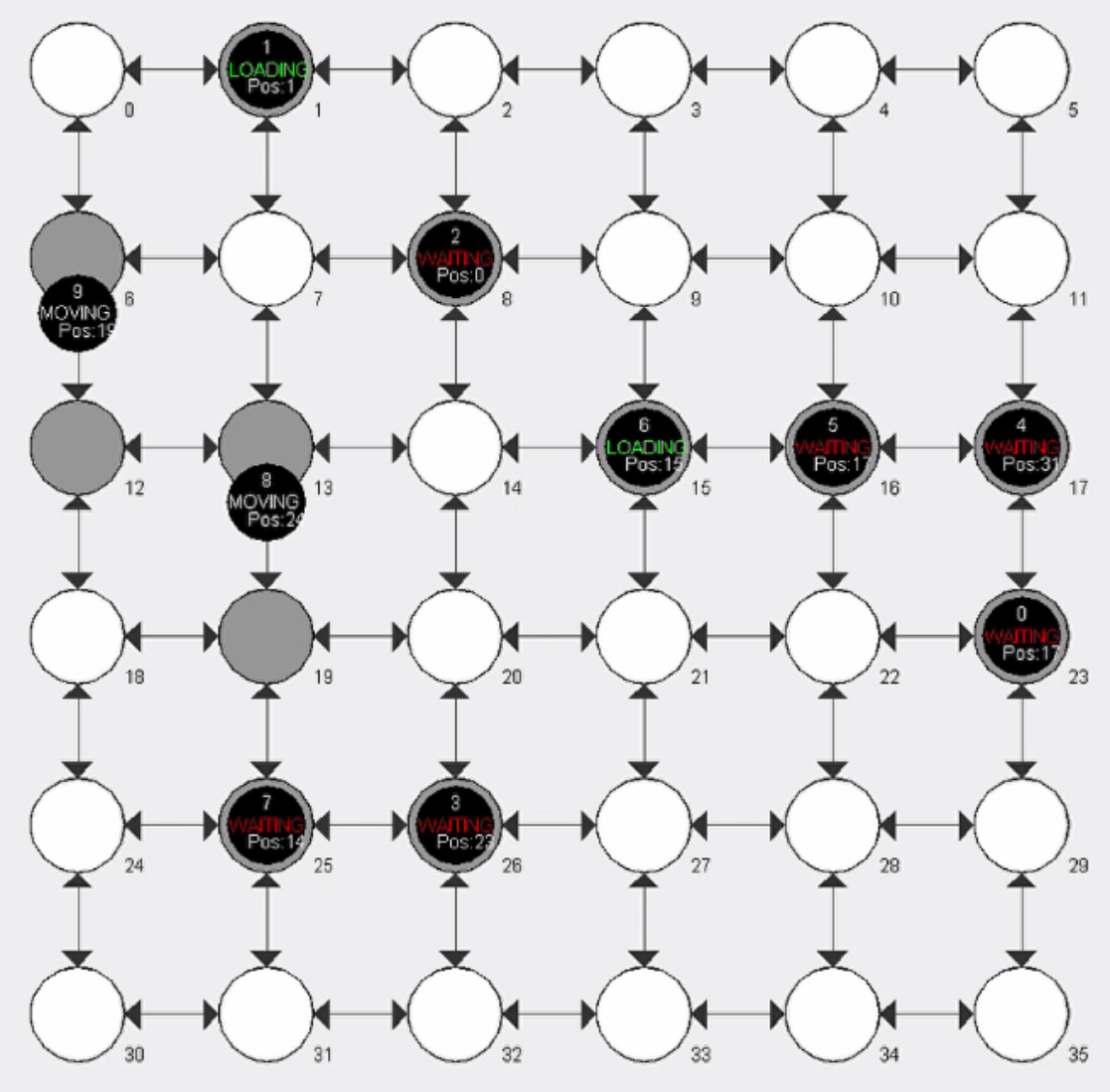


Obr. 3.3: Zobrazenie gridov a stavov

Vždy, keď je volná štartovacia pozícia, tak odtiaľ vyjde ďalší shuttle. Štartovaciu pozíciu sme nastavili v nulovom nóde. Odkiaľ môžu jednotlivé shuttly vychádzať a robiť svoje joby. Jednotlivé shuttle vychádzajú postupne, ako majú nastavené poradové čísla. Joby sa v mometalnej verzii nesortujú od najbližšieho jobu, to bude vyriešené neskôr. Idú vlastne od prvého do posledného. Aj keď napríklad majú dva joby, tak sa vykonajú postupne. Neskôr to bude zaistené. Pridali sme aj farebnú vizualizáciu aktuálneho stavu shuttlu pre podrobnú analýzu a pre lepšiu viditeľnosť, čo sa práve deje so shuttlom.

Vyskúšali sme aktuálne zobrazenie simulácie a chod shuttlov na mriežke 6x6 s dvadsiatimi shuttlami vo fronte, kde naraz ich mohlo byť v sklade desať a vždy si shuttle urobili svoju prácu automatizovane a do konca. Ak sa stane, že dva shuttle sa zablokujú, tak po štyroch sekundách blokovania sa pohne shuttle o náhodnú pozíciu vedľa, aby uvoľnil cestu druhému, nie je to môžno najoptimalnejšie riešenie,

ale to funguje. V našom zadaní bolo povedané, že nejaký buffer tam môže byť. Ešte sme pridali náhodné časy nakladania a náhodné rýchlosti medzi nódmi. Tiež spomenieme, že shuttle si sám hľadá cestu, nie je to tak, žeby nám to riadil nejaký sklad. Aktuálnu podobu nášho simulovaného skladu je vidno na obr. 3.4



Obr. 3.4: Cesty jednotlivých nódov a pozície, práca desiatich shuttlov

4 Implementácia algoritmov

4.1 Vylepšenia algoritmov

4.1.1 BFS0/BFS

Breadth-first search, alebo hľadanie do šírky, jeho skratka je BFS. My sme tento počiatočný algoritmus označili ako nula, pretože to bol prvý algoritmus, ktorý sme implementovali. BFS je teda náš algoritmus a znamená to v našom prípade, že shuttle ide po porade. Znamená tiež, že shuttle má nejaké svoje práce, ktoré sa nezaradujú podľa toho, ktorá je najvýhodnejšia pre neho, ale ide vlastne porade a robí jednu prácu za druhou. Napríklad, aj keby mal najbližšie prácu na desiatom nóde, tak on sa pozrie na svoj zoznam a prvú prácu, ktorú spraví, napríklad, dvadsiatku, ktorá je viacej vzdialená, ale je na zozname uvedená ako prvá a až potom spraví tu prácu desať, ktorá sa vykoná až keď príde na radu. Našou úlohou bolo postupne vylepšovanie kódu a tak sme najprv zlepšili kód BFS0 takže sme ho začali označovať neskôr BFS a BFS0 bol náš pôvodný prvý kód, ktorý sme vytvorili. BFS má vylepšenia, niektoré z tých vylepšení si popíšeme. Breadth-first search sme teda vylepšili a to konkrétne tak, že si po vylepšení zoraďuje najbližšie práce. Ako sa cez čo najmenej nódov dostane k danej práci, ale ignoruje keď ide cez nódy či tá trasa je nejako viacej zdĺhavá, alebo krátka. Jemu je to jedno, ide mu o to, aby nám čo najmenej nódov prešiel.

4.1.2 Dijkstra

Aby sme mohli porovnávať jednotlivé algoritmy tak sme potrebovali implementovať aj iné a tak sme začali Dijkstrou ktorá sa nám zdala najvýhodnejšia pretože má niektoré vylepšenia ktoré si popíšeme. Jej popis nájdete aj v teoretickej časti tejto práce. Ale popíšeme si ju aj tu ako sa nám podarila a aké výhody z nej vyplývajú. Podarilo sa nám teda implementovať do kódu Dijkstru, ktorá by mala zlepšovať naše časy, ktoré sú potrebné pre rýchlejšie dojazdy k materiálu, alebo výklad materiálu. Tento algoritmus vlastne prejde všetky nódy a zistí si, ako výhodné je ísť cez tento nód, alebo cez túto trasu a ide tou, ktorá je najoptimálnejšia. Napríklad shuttle má dvadsať práci a jedna práca je len o jeden nód vedľa, ale trvalo by mu tam sa dostať dlho, povedzme, jednu minútu, a naopak druhá práca je síce len o dva nódy vedľa, ale čas k nej je oveľa kratší a trvalo by mu prísť do neho do tohto nódu kratšie za päť sekúnd tak si zvolí ísť radšej touto cestou teda cestou do tohto druhého nódu. Využíva teda najmä to, že tá trasa môže trvať dlhšie ako je väčší počet nódov. Je to výhodné v tom, že dva nódy medzi sebou nemusia mať tú istú rýchlosť keď cez nich

prechádzame. Môžu mať nejaké spomalenie medzi sebou a teda my potrebujeme, aby sme sa dostali čo najrýchlejšie a časovo lepšie.

V dijkstre sa jedná o to, že na začiatku sa nám vytvorí tabuľka „dijkstraTable“ ktorá pozostáva z toho, že máme tabuľku všetkých nódov a potom je „Pair“ takzvaní „dijkstraPair“. Sme teda v nejakom nóde s vozíkom, currentNode, ten si uložíme rovno medzi zoznam nových nódov a vložíme ho do dijkstraTable a teraz potrebujeme vytvoriť mapu celého prostredia, aby sme vedeli ktorý nód je od nás najbližšie, na to slúži táto tabuľka (vlastne táto tabuľka je mapa, to znamená, že každý nód má nejakú vzdialenosť - dijkstraPair). Keď sa vygeneruje táto tabuľka, budeme vedieť kde je každý jeden nód v tomto danom Stocku (vypýtať si ho) a vráti ti nám to dijkstraPair, ktorý má v sebe parametre cost (skóre pripojenia), ako výhodne sa vieme vlastne dostať do toho nódu a ešte aj samotný nód má v sebe. Postupne tak ideme cez všetky nódy, robíme si všetky prepojenia, a zistíme, že ku ktorému nódu sa ako výhodne vieme dostať. Spravíme si takúto tabuľku tých nódov s tým, že vieme sa z toho aktuálneho nódu dostať do hociktorého a ako ťažké sa k nemu dostať. Máme vlastne takto celý sklad a tým sme si ho aj namapovali. Keď máme takúto tabuľku, tak podľa nej si pozrieme zoznam práci a pozrieme sa ku ktorej práci sa vieme dostať najrýchlejšie. Vozík vlastne prejde teda všetky práce, ktoré musíme vykonať a zistí, ktorá v tej tabuľke je najvýhodnejšia pre seba, aby išiel na ňu ako prvý a keď ho nájde tak ho nastaví a vráti sa na cestu k tomu najlepšiemu nódu (return bestJobPath). Keby sme mali prácu na desiatom nóde a vieme, že nejaký nód je od neho len jeden posuv a je veľmi blízko tak ho vykoná ako prvý v poradí. Aj keby bola o dva nódy bližšia práca a keby bola v zozname týchto práci posledná, tak ona sa priorizuje a bude ako prvá a takisto aj keby sme mali nejakú prácu ktorá je o jeden nód, ale je veľmi vzdialená, alebo je veľmi pomalá teda cesta by bola pomalá tak ide k tomu jednému nódu. Druhý príklad by bol, že sme na nóde tridsať a vytvoríme si takúto mapu a pozrieme sa ako rýchlo sa vieme dostať do nódu tridsaťpäť ako rýchlo sa vieme dostať do nódu sto a zistime si ku ktorému nódu sa vieme dostať čo najrýchlejšie. Vždy sa teda tabuľka počíta voči aktuálnemu nódu kde sa nachádzame. Týmto, že sa vytvorí mapa celého prostredia, alebo ako najbližšie sú ostatné nódy. Kľúč k tej mape je vždy nejaký iný nód a tá mapa nám povie ako sa k tomu nódu vieme dostať čo najoptimálnejšie. Berie teda ohľad aj na rýchlosť aj na vzdialenosti jednotlivých nódov. Vo while sa nám vytvára celá takáto mapa. V stockNode bestJob sa zisťuje ktorá cesta z tých ktorá je na rade nám bude najvýhodnejšia pre náš vozík. Teda je najbližšie k nemu, aby sa k nej dostal čo najrýchlejšie. Výbere sa teda ako najrýchlejšia ďalej sa nastaví ako aktuálna a spustí sa. S tým, že sme pridali medzi nódami, cestami väčšie rýchlosti a spomaľovania, aby boli shuttle oveľa rýchlejšie, pomalšie. To znamená, aby sme videli či tento dijkstra algoritmus funguje dobre a hlavne sme chceli vidieť rozdiel

medzi dijkstrom a bfs algoritmom. Ešte jedna vec ktorú treba spomenúť je, že ak by sme náhodou nenašli žiadnu prácu teda všetky sú obsadené a máme desať práci (celkové) tak vtedy čakáme až sa uvoľní nód, alebo cesta.

4.1.3 A-Star

V podstate štvrtým implementovaným algoritmom je A-Star on vlastne funguje na podobnom princípe ako dijkstra. Predbiehame, ale A-Star je lepší v tom, že keď máme veľmi veľa nódov, obrovské množstvo kombinácii a trvalo by to nám to veľmi dlho ich preskúmať. Čo to znamená si popíšeme. A-Star teda využíva aj smerovanie, že vieme ktorým smerom máme ísť kde cieľ. Shuttle vie kde je jeho cieľ, smer nehľadá smerom mimo neho. Ignoruje tie nódy ktoré sú mimo neho teda cieľa. Oplatí sa to keď je napríklad obrovský sklad. V A-Star ide vlastne o to, že neprechádzame celú mapu nevytvára si teda celú mapu všetkých nódov. Čo môže byť veľmi zdĺhavé. Ide to tak, že si postupne vytvárame mapu od stredu od toho kde sa nachádzame. Ak natrafíme na nód ktorý je naša práca tak hneď ho berieme, tú danú prácu berieme a hneď tú prácu aj vykonáme. Dijkstra ide štýlom, že ona musí prejsť celú mapu. Musí si vytvoriť mapu všetkých nódov. Príkladom je veľká mapa ktorá má tisíc nódov čiže mapa, tabuľka s tisíc hodnotami. A až z tej tabuľky si vytvára, vyberá ktorú prácu zoberie. A-Star robí to, že postupne si vytvára svoje okolie. Pozerá sa čo je najbližšie a vytvára si tak okolité cesty a vždy keď nájde nový nód tak skontroluje či tá práca je pre neho či ten nód má medzi prácami ak ho má tak ho zoberie a rovno ho ide vykonať. Nemusí si tak vytvoriť mapu celého prostredia, ale ide ten ktorý nájde ako prvý. Prvý nód ktorý nájde a má ho ako prácu hneď ho ide vykonať. V kóde sú navštívené nódy (visitedNodesMap) podobné ako v dijkstre. Je to zoznam všetkých nódov ktoré boli navštívené to je kvôli tomu, že keď prechádzame grafom, mapou musíme si pamätať kde sme už boli v akom nóde. Inak by sme sa mohli do nekonečna zacykliť. Je to naša kontrola, aby sme ten nód nemuseli už riešiť, ktorý je tam zapísaný. Už vieme, že je zapísaný niekde. AllPaths je to zoznam všetkých ciest ktoré boli vygenerované sú to cesty tiež k okolitým nódom. S tých ciest sa potom vyberá ktorá má byť vykonaná vždy je to ta posledná v tomto prípade. Potom sa ide do nekonečna a prechádza sa zase všetkými prepojeniami hľadá sa vždy aktuálny noud ktorý je najbližšie. Pretože máme na začiatku totižto len jeden nód. V 3D máme naopak tri možnosti kde ísť x,y,z osi ako prvý sa teda pridáva nód ktorý je najbližšie čiže x,y,z jeden z nich je bližšie než všetky ostatné takže napríklad, že by to bolo naše x. Takže prvý nód bude, že môžeme ísť do x. Potom tým sa ukončí tento cyklus a ideme zase ďalej do while. A pozrieme sa znovu (x už máme zaznamenané), že či z x môžeme ísť do ďalších troch nódov a potom ešte môžeme ísť z toho prvého nódu medzi y a z. Skontrolujeme ktoré s týchto prepojení je najrýchlejšie a zase

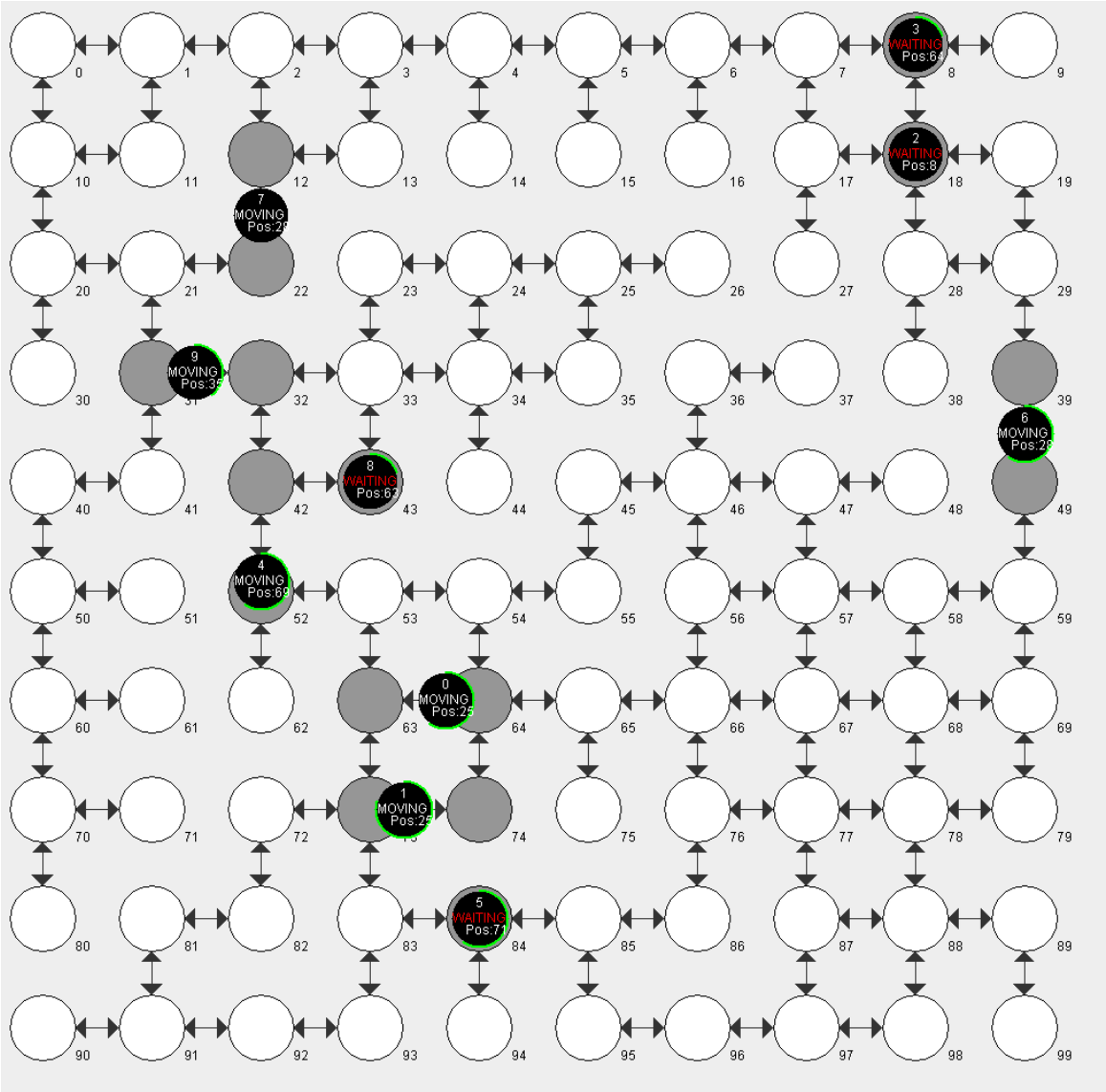
to prepojenie pridáme. Takto vytvárame takú mapu tej najoptimálnejšej cesty. Ideme vlastne tou najoptimálnejšou cestou. Môžeme tak prejsť uhlopriečne celý graf jednou cestou a všetky ostatné by boli horšie a tam by sme našli diaľnicu veľmi rýchlu cestu a ostatné nódy by mohli mať horšie cesty najpomalšie by boli čiže by sme ich vôbec by sme nezarátavali až potom keď by sme vytvorili tu diaľnicu by sa začali zaratávať tie zvyšné. Takto máme spravený A- Star a tak aj funguje. Už to y by bola ako práca ďalej by sa ukončilo skontrolovali by sa či sa niečo našlo ak sa nenašlo tak sa ukončí tento while. Ďalej v cykle kontrolujeme či nejaký s týchto nódov ktoré už sú nájdené nepatria medzi práce či náhodou to x čo sme pridali či náhodou nie je naša práca či tam nemáme ísť ak to tak je tak sa nastaví tento best index ten sa ďalej skontroluje či sa nastavil ak sa nastavil tak mám tu prácu tu x-kovú priorizujeme prihodíme na prvú priečku a vrátime to ako výsledok novej funkcie, že toto je naša nová práca, cesta takto sa ide do nekonečna až kým sa neminú tieto práce ak sa minú tak sa ukonči celý ten shuttle (znamená, že dokončil práce ide von zo skladu) ide preč. To je výpočet findAStar pre jednu prácu tu aktuálnu to sa deje vo findParthcurrentJob (nižšie) a tá sa vždy vola keď už vozík dokončil prácu. Ak už dokončil tak sa skočí do if a skontroluje akou metódou sa má nájsť ďalšia práca. Ak je to napríklad A-Star sa nastaví nová aktuálna práca, cesta na novú. Ak sa náhodou žiadna cesta nenašla tak sa vozík dá do stavu, že čaká do červeného ak sa mu to nepodarí začne sa hýbať nejakými náhodnými cestami. Ak sa nájde tak sa napíše shutlle našiel novu cestu a začne sa hýbať.

5 Nástroj na vizualizáciu plánovaných úloh

5.0.1 Generovanie máp

V stocksimulation sme pridali nahodný generátor. Doteraz to bol len generátor náš nový náhodný generátor je taký, ktorý vždy vygeneruje iné náhodne čísla, mapu, práce pre vozíky. Takže sme vytvorili náhodný generátor, ktorý vždy má označenie podľa set.Seed čísla. Čiže toto hocijaké seed číslo nám vie aj zopakovať tu istú situáciu pre daný algoritmus takto si vlastne môžeme vyskúšať tu istú mapu na algoritmoch, BFS0, BFS, Dijkstra a A-Star. Aby sme videli nejaké rozdiely na algoritmoch a mohli ich jednotlivo porovnať. Tak sme vytvorili v „Generate stock graph“ to, že medzi nódami sme niektoré spojenia vynechali zariaďuje to náš for kde nám vygeneruje mapu, avšak je tam napríklad dvadsať percentná (0,2) šanca, že sa nevytvorí spojenie nódov. Je to vlastne riadok if (random.nextDouble() $<$ 0.9) ktorý nám určuje pravdepodobnosť, že to prepojenie bude vykreslené v tomto prípade na devedesiat percent v scéne. Ono nám to prechádza cez všetky nódy a snaži sa vytvoriť prepojenie. Inak povedané, že je to pravdepodobnosť, že ich spojí, nespojí. Vidno to na obr. 5.1 kde nie sú niektoré prepojenia, cesty k nódom. Ak by toto číslo bolo napríklad jedna celá nula (<=1.0) tak by sa vykreslila celá mapa. Ak by sme aj tento riadok aj zakomentovali tak nám vykresli všetky prepojenia a nódy pretože to nemá vplyv na kód iba v prípade, že chceme niektoré nódy vynechať. Ďalšou výhodou v kóde je to, že ak chceme len určitý algoritmus tak ho jednoducho za komentujeme a spúšťame podľa potreby. Zmeníme teda if tak, že bude napríklad len jeden algoritmus vypočítavať, alebo viac podľa toho ktorý potreujeme.

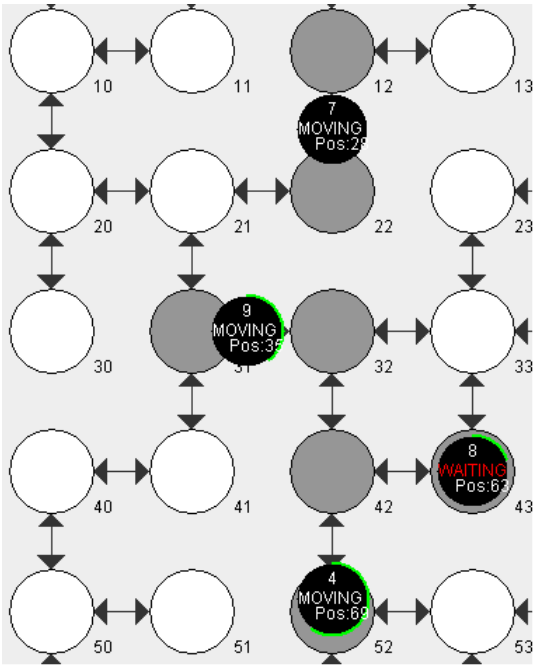
Keďže vidno na mape, že niektoré spojenia, cesty chýbajú. Vyzerá to ako ostrovčeky, alebo jazierka cez ktoré prechádzajú vozíky si musia hľadať cestu. Dá sa tým povedať to, že vozíky musia ísť na niektorých miestach len jednou cestou. Tým by sme sa mali dozvedieť či niektoré algoritmy sú časovo lepšie na tom a o koľko sú lepšie, kde sa vozíky musia viac rozhodovať akou cestou ísť. Vidno to na obr. 5.2, že aby vozíky dostali z nódu dvadsať jediná do napríklad štyridsaťdva tak musia práve tou jednou cestou ísť poprípade si vyberú inú cestu a to hornou, alebo spodnou cestou. Rozdiel teda hlavne môže byť aj v tom, že dijkstra sa rozhodne ísť hornou cestou, lebo vidí podľa tabuľky, že je bližšie, rýchlejšie k danému nódu, ale BFS ide dolom a nepozerá sa že rýchlostne by to bolo lepšie. Takéto rozdiely nám robia vlastne časové oneskorenie, ktoré chceme, aby sa nám prejavili.



Obr. 5.1: Pravdepodobnosť prepojenia jednotlivých nódov

5.0.2 Časovanie

Potrebovali sme kvôli meraniam časovač takže sme ďalej pridali časovač ktorý nám bude časovať algoritmy, tým si ich vieme porovnať časovo podľa toho kedy nám simulácia začne „simulationStart“ a skončí „simulation finished“. Pre nás bol zaujímavý aj čas kedy každému vozíku podľa algoritmu sme počítali čas výpočtu metódy tento čas sme aj spriemerovali. Tento čas vlastne závisí na výpočtu daného algoritmu (spusteného práve algoritmu) a následne sme tie časy priemerovali a zapísali do súboru ako celkový priemer na daný algoritmus. Inak povedané je vlastne čas, ktorý potrebuje vozík na rozhodovanie kam pôjde a priemer všetkých vozíkov na metódu v milisekundách. Pre tieto časy a mapy sme vytvorili textový súbor kde sa nám



Obr. 5.2: Prechody medzi nódami

postupne zapisovali údaje. Vylepšili sme to tak, že len sme pustili kód a načasoval nám tak štyri algoritmy s jednou mapou a jedným gridom. Ktoré vyzerali takto na obr. 5.3

```
Grid: 10x10x5     Seed: 1990589414321677312     Mode: BFS0  Shuttles: 10      Jobs: 40    Milliseconds: 233471    Method milliseconds: 0.21118494464944648
Grid: 10x10x5     Seed: 1990589414321677312     Mode: BFS   Shuttles: 10      Jobs: 40    Milliseconds: 131063    Method milliseconds: 0.12195508365508366
Grid: 10x10x5     Seed: 1990589414321677312     Mode: DIJKSTRA    Shuttles: 10      Jobs: 40    Milliseconds: 120432    Method milliseconds: 0.36081683748169835
Grid: 10x10x5     Seed: 1990589414321677312     Mode: A_STAR      Shuttles: 10      Jobs: 40    Milliseconds: 121283    Method milliseconds: 4.7943792432432435

Grid: 10x10x10    Seed: 3787934947416002560     Mode: DIJKSTRA    Shuttles: 20      Jobs: 50    Milliseconds: 340953    Method milliseconds: 1.4714302367941712

Grid: 10x10x5     Seed: 32604540458710190008    Mode: BFS0  Shuttles: 10      Jobs: 40    Milliseconds: 259508    Method milliseconds: 0.28336134715025907
Grid: 10x10x5     Seed: 32604540458710190008    Mode: BFS   Shuttles: 10      Jobs: 40    Milliseconds: 144626    Method milliseconds: 0.28895689655172413
Grid: 10x10x5     Seed: 32604540458710190008    Mode: DIJKSTRA    Shuttles: 10      Jobs: 40    Milliseconds: 131905    Method milliseconds: 0.5587505966587112
Grid: 10x10x5     Seed: 32604540458710190008    Mode: A_STAR      Shuttles: 10      Jobs: 40    Milliseconds: 130164    Method milliseconds: 0.6607291169451074

Grid: 10x10x5     Seed: 1471332124232706048     Mode: BFS0  Shuttles: 20      Jobs: 20    Milliseconds: 150480    Method milliseconds: 0.26874308724832213
Grid: 10x10x5     Seed: 1471332124232706048     Mode: BFS   Shuttles: 20      Jobs: 20    Milliseconds: 108320    Method milliseconds: 0.3300016129032258
Grid: 10x10x5     Seed: 1471332124232706048     Mode: DIJKSTRA    Shuttles: 20      Jobs: 20    Milliseconds: 104015    Method milliseconds: 0.5912307407407408
Grid: 10x10x5     Seed: 1471332124232706048     Mode: A_STAR      Shuttles: 20      Jobs: 20    Milliseconds: 103740    Method milliseconds: 1.273678

Grid: 10x10x5     Seed: 3496324523366573056     Mode: BFS0  Shuttles: 30      Jobs: 20    Milliseconds: 157003    Method milliseconds: 0.24265942976719854
Grid: 10x10x5     Seed: 3496324523366573056     Mode: BFS   Shuttles: 30      Jobs: 20    Milliseconds: 104079    Method milliseconds: 0.13278454935622316
Grid: 10x10x5     Seed: 3496324523366573056     Mode: DIJKSTRA    Shuttles: 30      Jobs: 20    Milliseconds: 90501     Method milliseconds: 0.30892525979216623
Grid: 10x10x5     Seed: 3496324523366573056     Mode: A_STAR      Shuttles: 30      Jobs: 20    Milliseconds: 90444     Method milliseconds: 2.9710901045296167
```

Obr. 5.3: Úkažka zápisu časov použitej mapy a jednotlivých algoritmov

Tieto údaje sme postupne začali spisovať a prepisovať do Excelu a kde sme si ich mohli porovnávať, vyhodnocovať a spraviť grafy. Čo si popíšeme v ďalšej časti.

5.1 3D stock

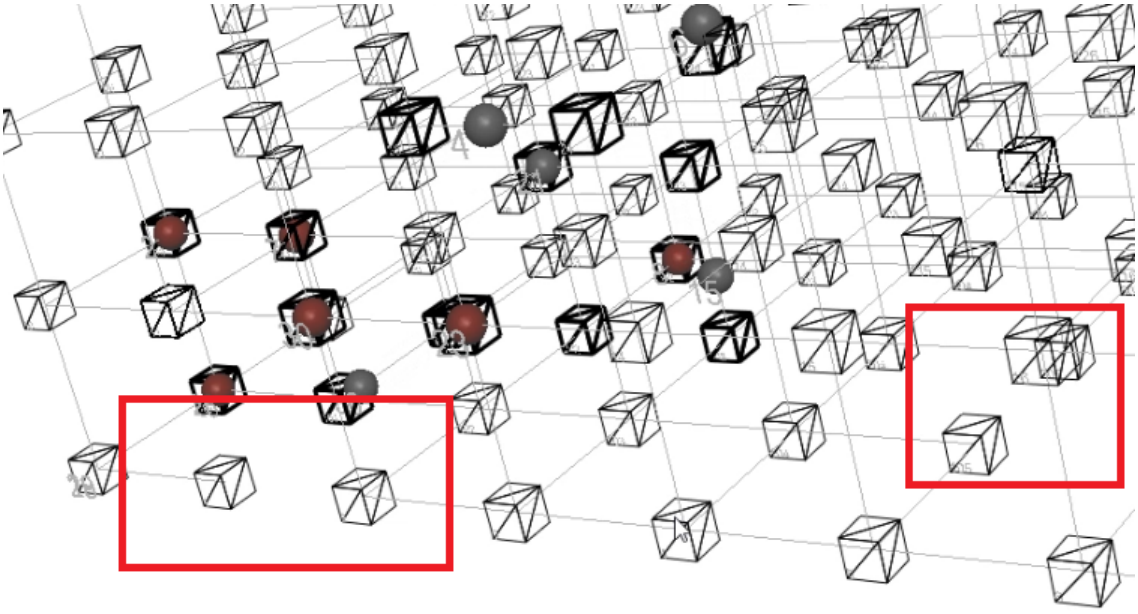
5.1.1 3D stock main, a iné časti implementácie

Na začiatku v main-e sa vygeneruje náhodné číslo teda ten Seed ktorý sme už popisovali. Ďalej nastupuje mód, teda algoritmus s tým náhodným Seedom a vygenerujú všetky štyri typy algoritmov. Ak by sme chceli napríklad jeden algoritmus stačí tieto prvé riadky kde sú typy zmeniť. Keď chceme len napríklad jeden, dva a tak ďalej tak treba ich zakomentovať v kóde, alebo zmeniť if patričným spôsobom. Vytvorí sa nám teda 3D simulácia s pracujúcimi algoritmami. Ak by sa stalo, že by nebola vygenerovaná simulácia to môže nastať len v prípadeˇ, že nie sú všetky nódy spojené. Preto je tam v tomto prípade break teda for. Ak teda neprejde ide sa znovu generovať nová mapa (Seed). Ak to prejde ide sa ďalej a vytvorí sa okno a už ide normálna simulácia, otvorí sa teda okno a pusti sa simulácia a čaká sa, až kým sa to okno nezavrie teda koniec simulácie. Toto čakanie je na to, že sa to okno zavrie a že sa ukončí celá simulácia, zavrie sa a ide sa na ďalšiu. Takže najprv ide na radu BFS0 potom BFS, ďalej DIJKSTRA a nakoniec A-Star. Keď sa všetky tieto štyri algoritmy spravia, tak nám to ukončí tento for a ďalej ide ďalší náhodný seed čiže ďalšia mapa. Ako tomu bolo v 2D stocku tak všetko máme rozdelené do viacerých Java súborov. Oni sú všetky vlastne pôvodné, okrem tých, čo majú na konci 3D. Niektoré úpravy bolo tak treba vykonať, ale väčšina bola pripravená aj na 3D. Prechodom teda z 2D do 3D to bolo tak, že od začiatku sme na to trochu mysleli a kód bol písaní tak, aby nám implementácia vyšla od začiatku, aby s ňou neboli nejaké problémy. Čiže nám úplne väčšina bez problémov preskočila na 3D stock keďže sme s tím aj počítali. Knižnice ktoré sme v práci použili pre 3D sú j3d, jogl java Open Graphics Library aj pre zapisovanie súborov do txt ešte Filewriter s tých dôležitých ešte spomenieme matematické knižnice vecmath.Color3f, vecmath.Vector3d, java.util.Random. Začneme teda v triede shuttle. Tu sú teda nedefinované tieto všetky módy, algoritmy ktorými môžeme spúšťať našu simuláciu. V public enus Method. Takto vypadá Main pre 3D.

Ďalej v triede Shuttle (shuttle update) tá trieda nám vlastne zabezpečuje náš pohyb a tam sme nič skoro nič nemenili, tam sa kontrolovali x, y súradnice (position.getX) tak sme tam ešte pridali z-tovú súradnicu pretože je potrebná pre 3D, ešte plus v if sign(direction.getX a Y) tam bolo tiež potreba navýšenie kde sa nám kontrolovalo znamienko x, y, tak ešte sme pridali z . Ďalej boli úpravy v triede Shuttle, kde bol Point 2D a zmenili sme ho na Point 3D. Potom tá trieda Shuttle3D, tá nám robí to, že ona dedí a má vlastne to isté, čo Shuttle, zdedí naše vlastnosti zo Shuttla. Plus má všelijaké transformačné veci na vizualizáciu. Kde je aj sa náš konštruktér a kde sa nastavuje farba, (setDiffuseColor) momentálne ju máme nastavenú na sivú (0.7f.). Ďalej máme riadok (sphereMaterial.setCapability) všetky tieto farby sa dajú

meniť podľa potreby, (sphereAppearance.setMaterial) nastavujeme materiál, transparenciu, kde ju môžeme buď znížiť teda jej viditeľnosť (tej guli), takže ju môžeme aj urobiť neviditeľnú (hide) využívame to keď nám prídu tie guli (vozíky, shuttle) napríklad do cieľa. V predošlom kóde pre 2D to muselo byť tak, že shuttle keď už nebol v scéne tak sme ho nevykresľovali, ale tu v 3D keď guľa už je v scéne tak sa ona nám vždy vyrendruje, takže keď jej nastavíme transparenciu na 1.0, tak tým ju zviditeľníme a vtedy vlastne nám dokončila prácu a ona zmizne (setVisible). Shuttle transfosformácia (Transform3D) nám určuje kde sa nachádza naša guľa. Za ďalšie (shuttleTG.setCapability) jej môžeme zmeniť túto transformáciu. Kde sa nachádza tým ju dokážeme posúvať. To tam musí byť bez tohoto by sa nám na scéne v podstate nič nedialo. Keď povolíme tak ju môžeme posúvať. V new Sphere vytvoríme tu guľu veľkosťou desať, povolíme zápis na transparenciu. Guľu pridáme do transformácie, aby sa vizualizovala. Pomocou shuttleTG shuttle transformation group zmenou tohto parametra posúvame guľu ona ako sme si povedali je vlastne v scéne my ju len posúvame. Cez setColor jej zmeníme farbu. Set visible jej zmeníme viditeľnosť. Či je viditeľná, alebo nie. To robí náš nový shuttle3D. StockNode3D sa vlastne tiež zmenil a to tak, že dedí tiež po stockNode čo je tiež náš pôvodný. Je obohatený o 3D, vizualizáciu. Je tam vlastne skoro to isté čo máme v predošlej guli. Trošku v inom poradí kvôli prehliadnosti. Najprv sa nastavuje transformácia nódu pozícia x,y,z. Tá sa potom nemení už. Následne sa vytvorí appearance polygónový appearance, aby bol vrátený model. Následne sa nastaví farba. Toto musí mať každý jeden objekt ako sa má vizualizovať. Čiže nastavíme mu, že má byť polygolialny a že má mať farbu čiernu 0.0f. model. Appearance čo sa vytvorí a nastaví sa tým vytvorenie guli. Následne sa nastaví transparencia. Transparencia je tam kvôli tomu keď je obsadený nód. Vlastne lineAttributes nám zabezpečí hrúbku čiary a tým vieme, že nód je obsadený a nemožno do neho ísť teda len jeden shuttle. LineAttributes je defautne 1.0f to je keď nie je obsadený a ak shuttle príde po prepojeniach do nódu tak bude obsadený a nastavíme ho na 3.0f (lineAttributes), tri pixely. Defautne je teda jeden pixel a pri obsadení je to tri pixle (occupied). Ďalej sa vytvorí box (new Box) a pridá sa do scény. Box (nód) má veľkosť desať čiže 10x10x10. Treťou novou triedou je StockSimulation3D. Na začiatku tejto triedy si nastavujeme aký chceme mať grid. Gridsize mometálne je nastavení na desať. GridLayers nám určuje koľko chceme mať vrstiev teraz konkrétne päť, nodeDistance teda vzdialenosť nódu u nás to to nastavenie ostalo na sto. Nechali sme toto nastavenie, aby sme mohli porovnať či sa niečo zmení prechodom do 3D. Snažili sme sa zachovať si nastavenia z pôvodného projektu. Neskôr si to rozoberieme v meraní či sa niektoré hodnoty zmenili. Riadok v kóde gridLayers nám určuje prepojenia, pravdepodobnosť, že to prepojenie bude vykreslené v scéne. Tak ako tomu bolo v predošlom horizontálnom tvare, pri 2D. Ak by tam znova bola jednotka (<=1.0) tak by sa nám znovu vykreslila celá mapa

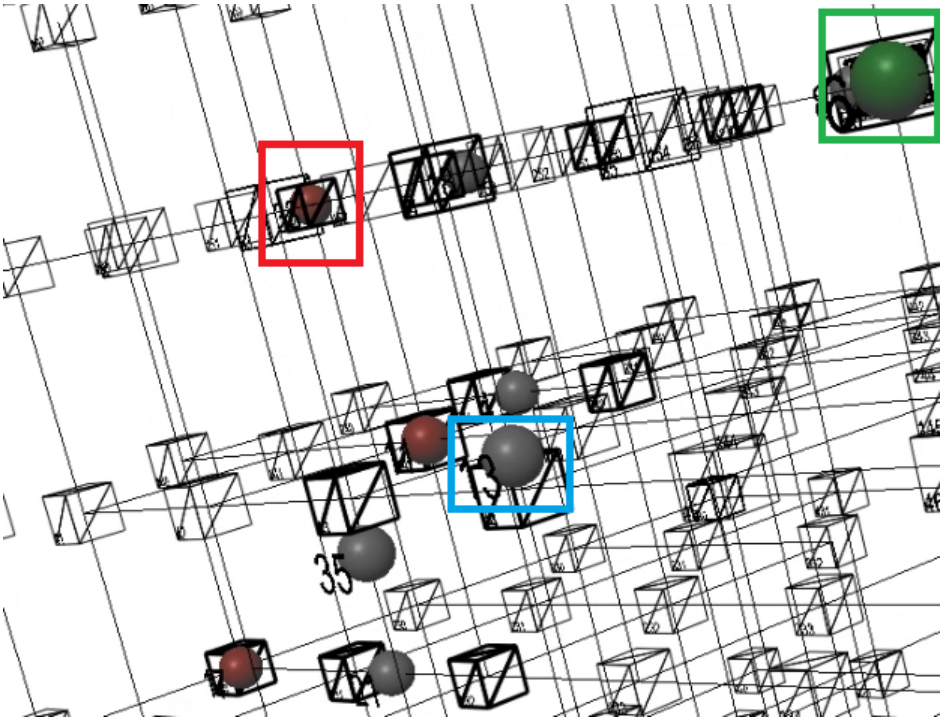
ak tam je zadané napríklad nula celá deväť tak je deväťdesiat percentná úspešnosť vykreslenia nódu, a desať percentná šanca, že sa nód vynechá (nevykreslí) môžeme to vidieť na obr. 5.4 kde v červených rámčekoch chýba nód.



Obr. 5.4: Prázdne miesta na základe percentuálnej úspešnosti v 3D

Je to tam vlastne kvôli tomu, aby sme mohli znovu na simulovať nejaký nedokonali sklad a porovnať algoritmy. V ďalších riadkoch sa nastavuje počet shuttlov a počet práci ako tomu bolo predtým (numberOfShuttles, jobNumber). V new SimpleUniverse, createScene sa vytvára scéna. V Background je to pozadie scény u nás je to biele (1.0f). Nastaví sa ambientné osvetlenie všesmerové (1.0f) na bielu farbu, directional smerové osvetlenie na menšiu farbu (0.3f). Tieto nastavenia sú preto, aby sme mohli vidieť objekty bez nich by sme na scéne nič nevideli. Ďalším krokom sú objekty zo StockNode3D, tým sa pridá do scény každý jeden nód. Nasleduje sa tak každý jeden Shuttle3D, vozík v našom prípade vypadajúci ako guľa. NodeConnection riadok nám znázorňuje, že sa pridajú všetky connections, prepojenia a v LineArray sa nám znázorňujú čiary medzi prepojeniami a pridajú sa tam ako statické objekty a už sa s nimi potom nehýbe. To platí samozrejme aj pre nódy. Pri updateView sa updatne, aktualizuje sklad tu sa všetky vozíky posúvajú. Keď sa poposúvajú tak sa vykreslia, prejde sa tak každý jeden vozík (shuttle). Ak je v sklade nastaví sa mu viditeľnosť a podľa toho či nakladá, čaká, hýbe sa (loading, waiting, running). Pri červenej je to čakanie (1.0f,0.0f,0.0f), nakladanie je zelená(0.0f,1.0f,0.0f) a pohyb znázorňuje čierna(0.7f,0.7f,0.7f). Na obr. 5.5 je vidno jednotlivé znázornenia. Na obrázku je pohyb čiže čierna v modrom rámčeku, aby ju bolo vidieť medzi nódami.

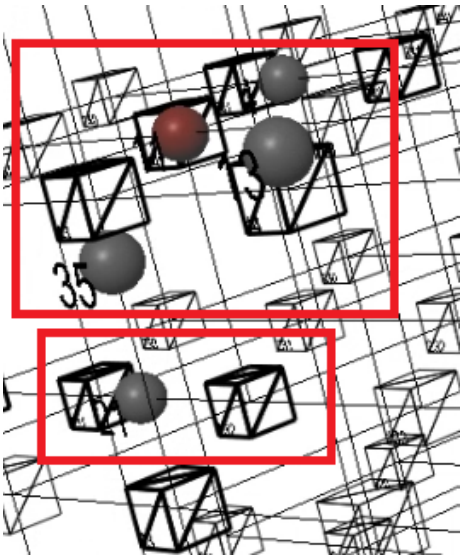
Zelená (nakladanie)v zelenom a čakanie (waiting) v červenom.



Obr. 5.5: Znázornenie pohybu,čakania a nakladania v 3D

Taktiež sa mu nastaví pozícia. Podobne sa to spraví aj s nódami avšak u nich sa updatne len hrúbka čiary čo sme si popísali vyššie keď je nód obsadený, alebo nie. Môžno to vidieť na obr. 5.6 kde v červenom rámčeku môžeme vidieť hrubšie čiary ktoré značia obsadenie nódu.

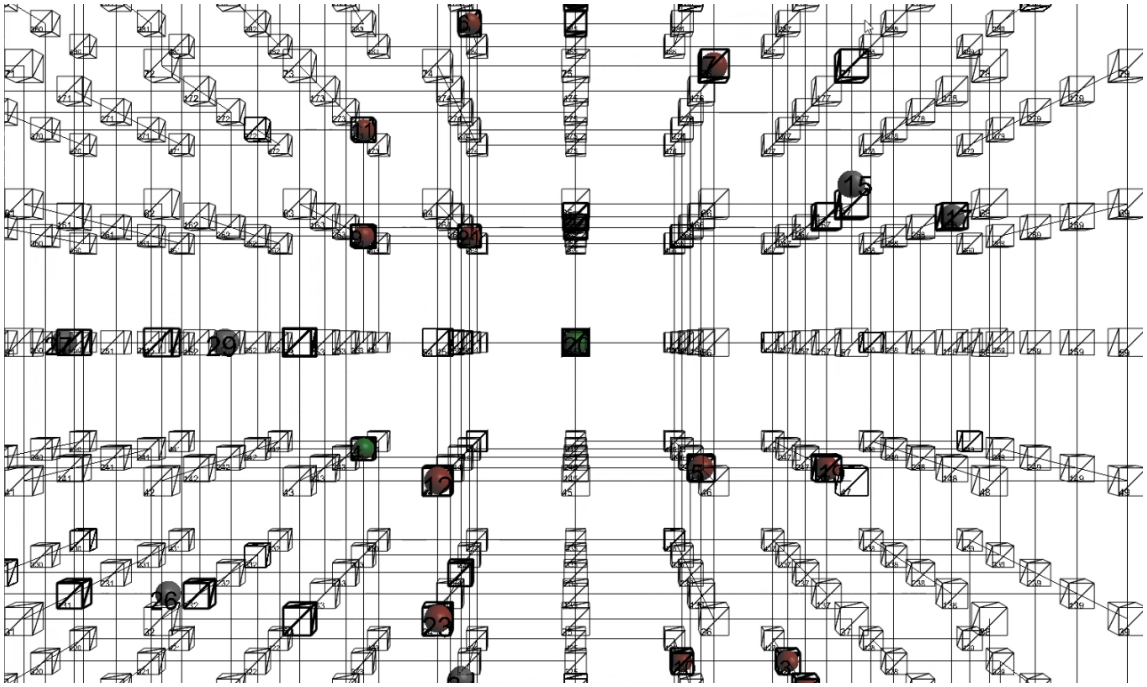
Vo viewTransform sa upravuje kamera podľa toho ako ňou budeme posúvať. Ak sme pohli s kamerou tak, že sme zmenili pohľad viewTransform ktorý nám definuje ako je kamera orientovaná. Ak ňou posúvame meníme aj pohľad ktorý vidíme. Kamera sa definuje pri scéne teda viewTransform kde zo scény sa vytiahne kamera, kamerová transformácia nastaví sa na našu transformáciu, aby sme ju mohli aj meniť. Nastaví sa defaut pozícia kamery, kamera rotácia (Rotation), translation a potom vieme hýbať kamerou do hociktorého smeru. Deje sa to vlastne v keyPressed kde sa nastavuje čo sa má stať ak stlačíme na klávesnici šípky doľava do prava, hore a dole. To má za následok, že sa má posunúť kamera v nejakom smere. Vo mouseDragged je naopak aj myš čo sa stane keď stlačíme ľavé tlačidlo (button) otáčame kamerou, ak stlačíme pravé tlačidlo (button) tam sa môžeme hýbať doľava, doprava a ak chceme ísť hore dole. Ďalej v našom kóde sa spustí timer na updatovanie, spustí sa tak celá simulácia. V časovači (timer) sa vždy nám updatne, ale keď by nastala



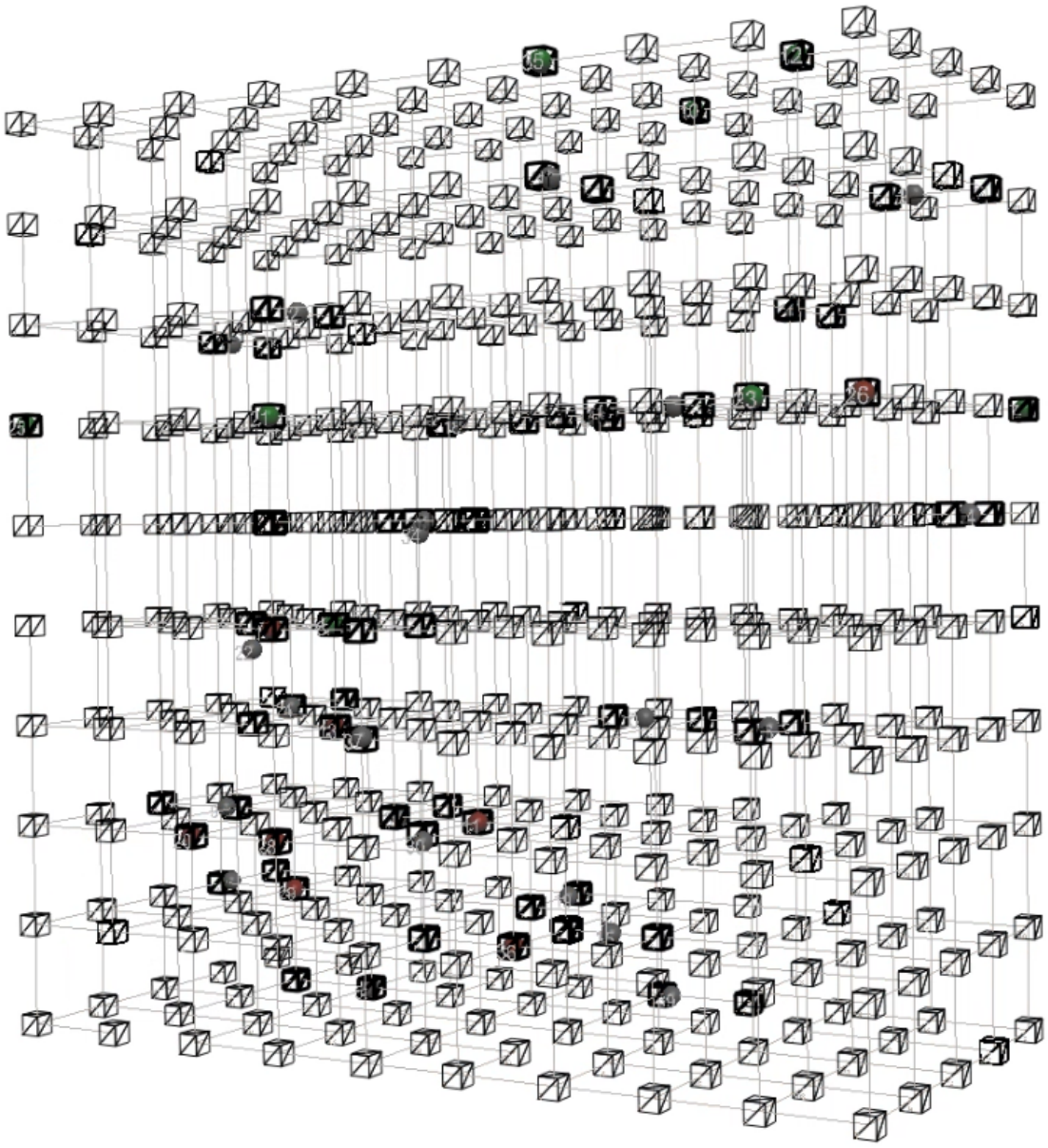
Obr. 5.6: Prechody a obsadenie nódov v 3D

situácia, že sa sklad vyprázdni (stock empty), že tam už nie je žiadny vozík napíše sa teda koniec (finish) ukončenie simulácie (simulation done) a napíše, vypíše sa do súboru txt hodnoty ktoré sme popísali vyššie ako je grid, seed, použitý algoritmus a čas vykonania všetkých práci, priemerní čas na algoritmus teda rozhodovací čas.

Zobrazenie ako sa nám podarila celá vizualizácia teda scéna môžeme vidieť na obr. 5.7 a tým aj celého skladu pri gride 10x10x5 na obr. 5.8.



Obr. 5.7: Zobrazenie 3D vizualizácie



Obr. 5.8: Zobrazenie celkovej 3D vizualizácie na gride 10x10x5

6 Simulácia a zhodnotenie efektivity algoritmov

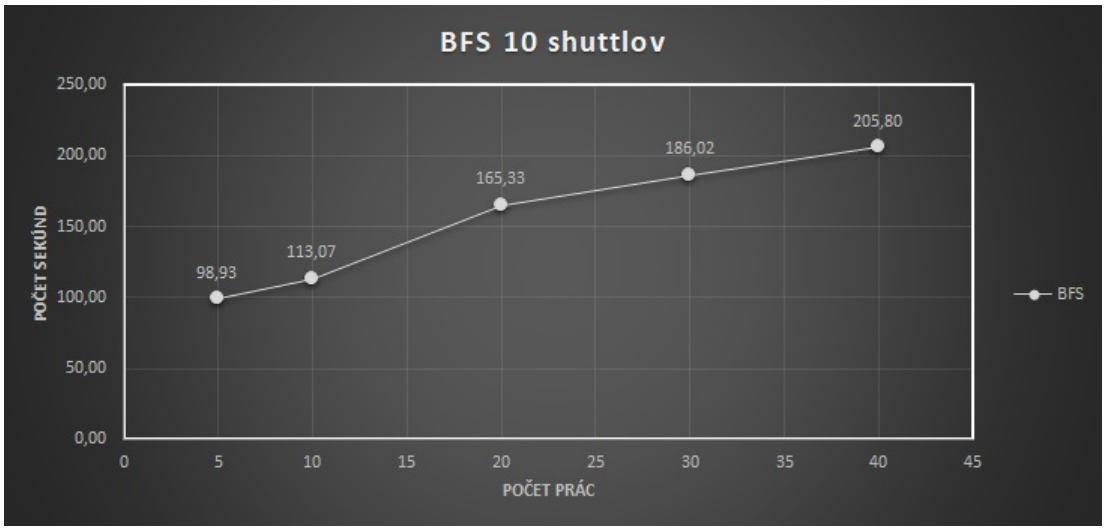
6.0.1 Meranie pre 2D stock

Pristúpime teraz k jednotlivým meraniam v oblasti 2D teda v horizontálnej polohe. Ako bolo napísané tak sme jednotlivé merania získavali pomocou zápisu kedy sme spustili našu simuláciu a znej sme dostavali jednotlivé hodnoty. Meranie prebiehalo najprv na jednom počítači, ale pre zrýchlenie sme sa rozhodli pre viac počítačov pretože jednotlivé merania boli niekedy dosť zdĺhavé, ale primárne sme použili jeden pre lepšie zhodnotenie výsledkov. Meranie prvé ktoré sme uskutočnili bolo BFS0 prvotný náš algoritmus kde sme chceli závislosť času od počtu prác. Grid sme použili 10x10 a do scény sme nasadili desať vozíkov ktoré pracovali a vykonávali jednotlivé práce. Postupne sme si tak odmerali 5,10,20,30,40 prác ktoré vozíky urobili a čas sa nám zapisoval do súboru txt. Merania sme prepisovali do Excelu kde sme následne mohli vytvoriť grafy. Môžeme si priblížiť hneď prvý graf z tohoto merania možno vidieť na obr. 6.1 kde vidno, že s pribúdajúcim sa množstvom prác pribúda aj čas ukončenia všetkých prác.

Obr. 6.1: Graf závislosti času od počtu prác pre algoritmus BFS0 2D

Spomenieme, že sme opakovali merania pre každú škálu teda pre 5 tristo meraní a pre 10 prác tristo meraní a tak ďalej aj pre ostatnú škálu. Tieto merania sme spriemerovali a vyšiel nám počet sekúnd na danú škálu. To isté sme urobili pre BFS čo ako bolo spomenuté je náš vylepšení algoritmus. No tu ako vidno na grafe závislosti času od počtu prác na obr. 6.2 vidíme pokrok v tom zmysle, že nám už klesli

jednotlivé časy. Tým chceme povedať, že tie isté práce náš BFS urobí rýchlejšie. Ako sme vyššie povedali tak algoritmy sú porovnávané na tých istých vygenerovaných mapách, aby sme s určitosťou mohli povedať, že sú buď lepšie, alebo horšie čo sa týka časov. Plus majú tam aj vymedzené cesty ako bolo povedané vďaka generovaniu pravdepodobnosti, že sa nevytvorí prepojenie.



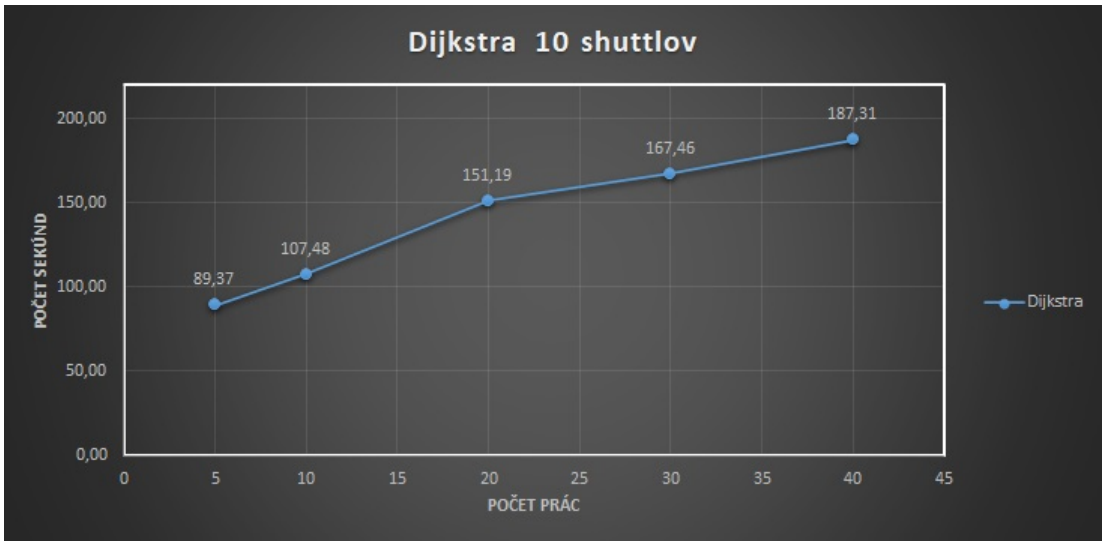
Obr. 6.2: Graf závislosti času od počtu prác pre algoritmus BFS 2D

Ďalším našim grafom bola dijkstra, ktorá ako bolo spomenuté má lepšie vlastnosti keďže si porovnáva vo svojej tabuľke jednotlivé cesty, rýchlosti medzi nódami. Graf pre dijkstru môžno vidieť na obr. 6.3. Nechceme predbiehať, ale vidno na základe časov, že sú určite lepšie a skoro nikdy s určitosťou môžeme povedať, že nie horšie.

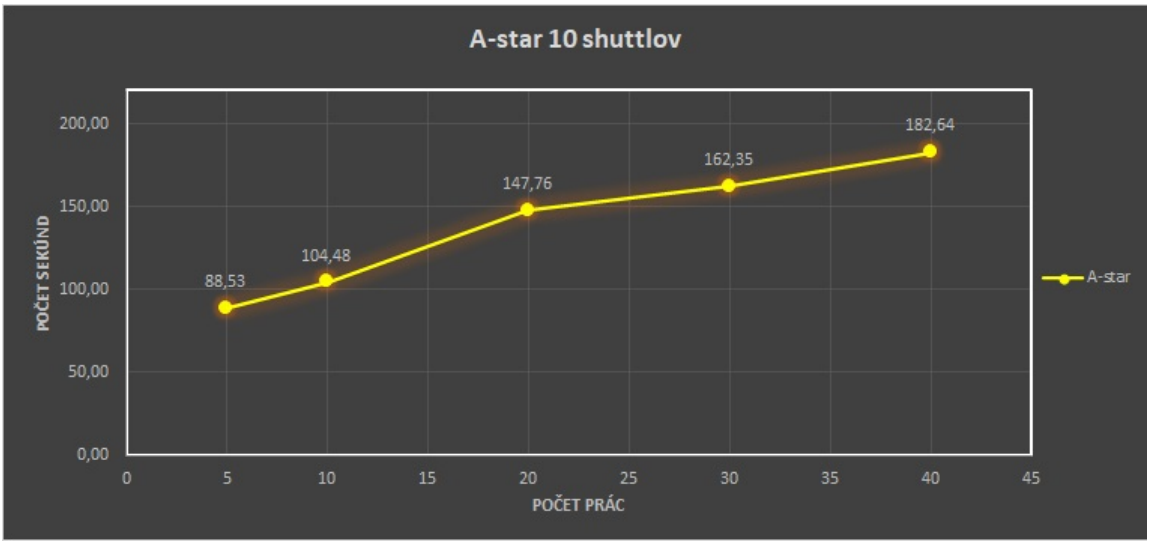
Pre posledný algoritmus máme graf na obr. 6.4. Ten vychádza tiež trošku lepšie oproti BFS a výrazne oproti BFS0, ale už nie tak oproti dijkstre pretože sú porovnateľne na tom skoro istom princípe avšak pre A-Star by bola väčšia výhoda ako bolo spomenuté prechod cez veľmi veľa nódov keďže si tabuľku robí od centra. Ako sme povedali používame tú istú mapu takže tu ten rozdiel nie oproti dijkstre tak znateľní.

Všetky štyri hodnoty grafu sme pre lepšiu porovnateľnosť dali do jedného môžeme to vidieť na obr. 6.5 kde vidno, že dijkstra a A-Star sú lepšie algoritmy. Toto možno vidieť aj v tabuľke nameraných a porovnávaných jednotlivých hodnôt, tabuľka počtu prác.

Posledným grafom pre 2D stock teda pre simuláciu sme sa rozhodli spraviť pre porovnanie len troch algoritmov sú nimi BFS, Dijkstra a A-Star. Kde sú zobrazené

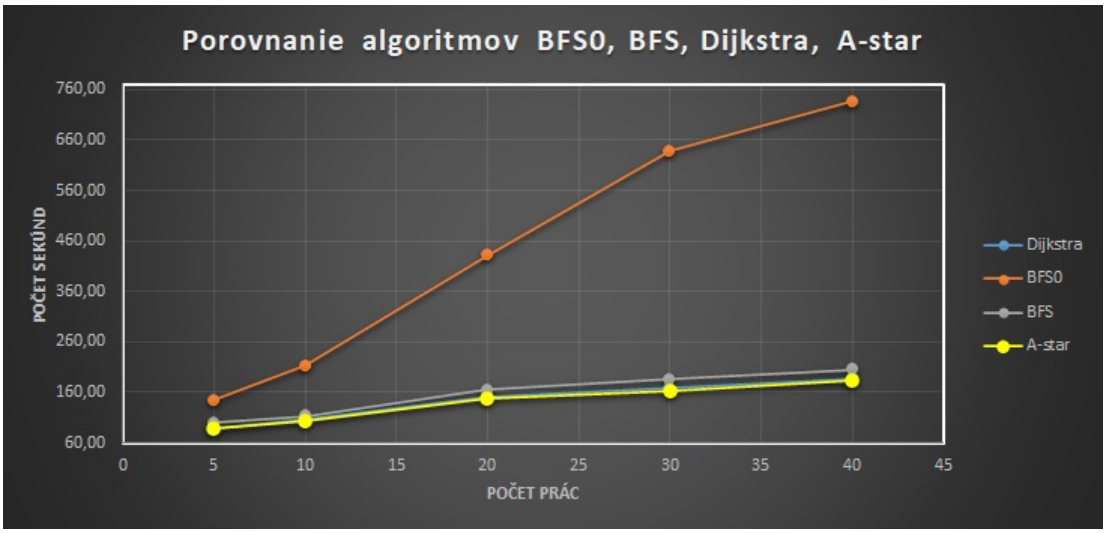


Obr. 6.3: Graf závislosti času od počtu prác pre algoritmus Dijkstra 2D

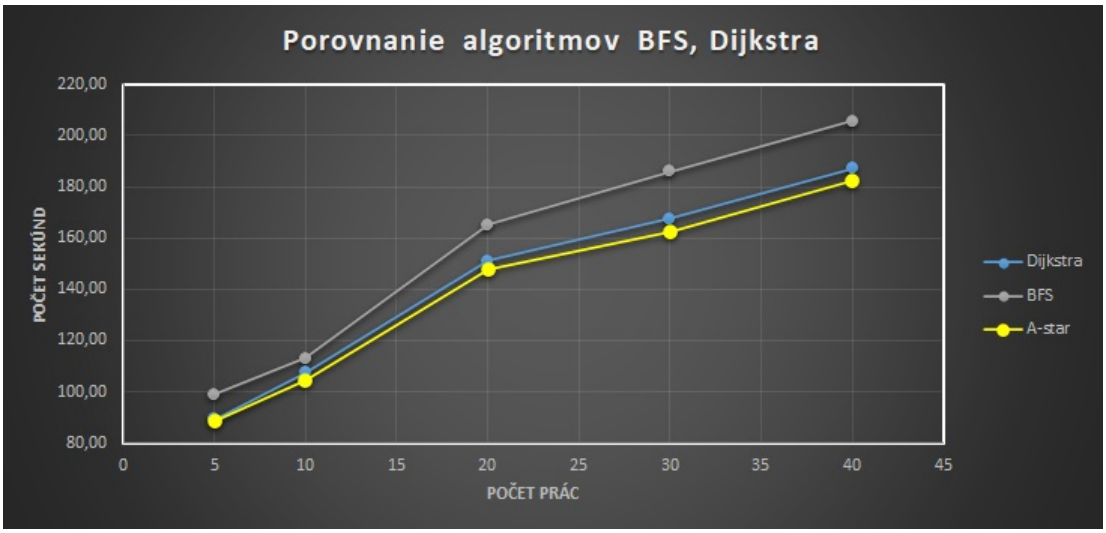


Obr. 6.4: Graf závislosti času od počtu prác pre algoritmus A-Star 2D

o niečo lepšie rozdieli týchto troch pretože v predošlom grafe keď tam bolo BFS0 tak zatienilo ostatné časy. Zobrazenie je na obr. 6.6 Toľko teda ku horizontálnej časti presunieme sa k 3D kde bude zaujímavé vidieť či nám časy stúpnu alebo klesnú keďže budeme v 3D oblasti.



Obr. 6.5: Graf závislosti času od počtu prác pre všetky algoritmy v 2D



Obr. 6.6: Graf závislosti času od počtu prác pre BFS, Dijkstra,A-Star algoritmy v 2D

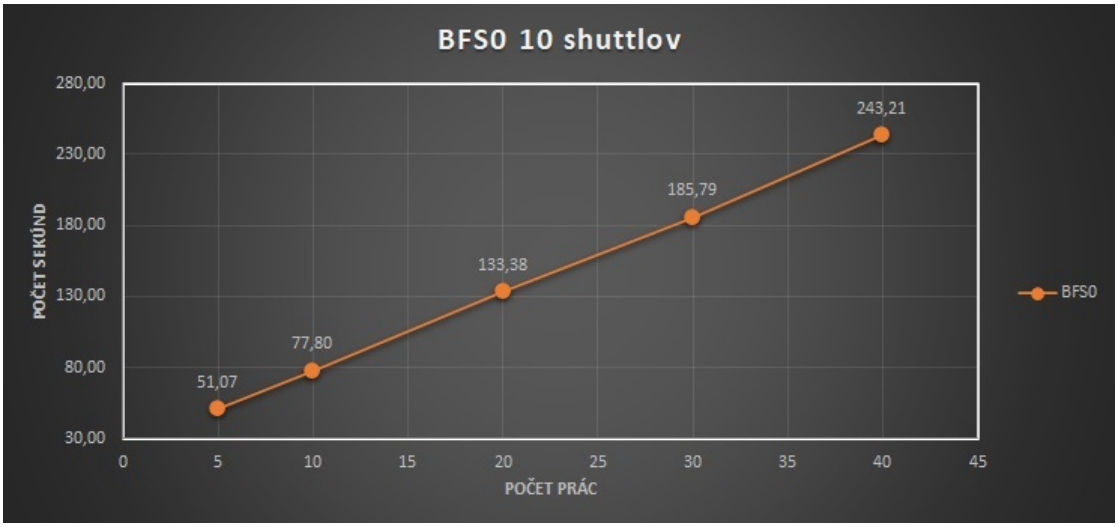
6.0.2 Meranie pre 3D stock

Presunieme sa k ďalšej časti a to je meranie 3D stocku a zistíme či budeme mať lepšie, alebo horšie časy oproti 2D. Hneď s prvého grafu pre BFS0 na obr. 6.7 sme si mohli všimnúť keď ho porovnávame s 2D skladom, že tie časy sú o niečo menšie. A keďže sme sa snažili napodobniť tie isté podmienky ako pri 2D tak to bolo malou záhadou, ale pri lepšom zamyslení je to vlastne jasné. Vysvetlime si teda prečo tomu tak je. Grid sme zvolili tiež 10x10 avšak keďže tu nám už pribudla tretia rovina tak tú sme

Tab. 6.1: Tabuľka počtu spriemerovaných časov od počtu prác pre jednotlivé algoritmy

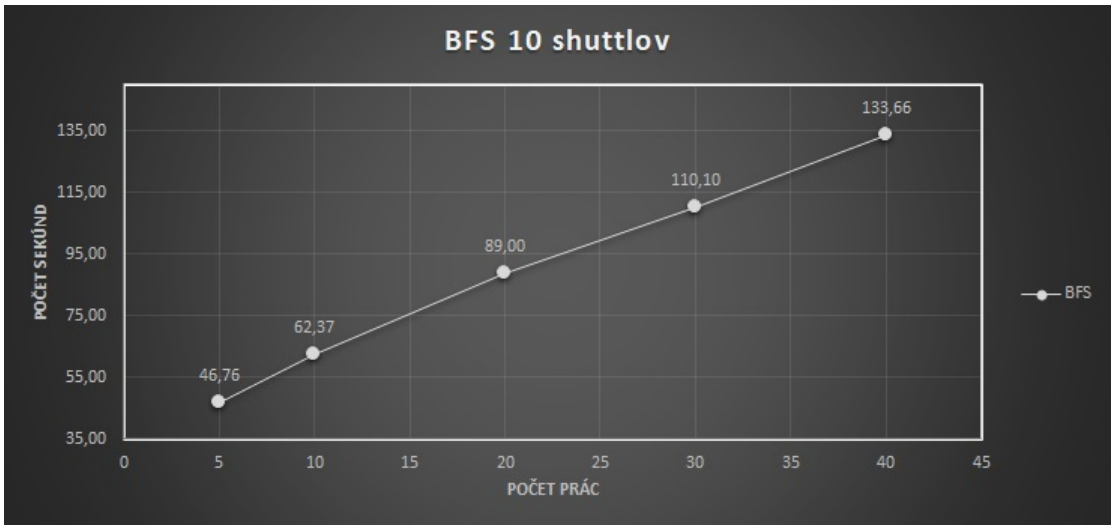
Počet prác	BFS0 poč. sekúnd	BFS poč. sekúnd	Dijkstra poč. sekúnd	A* poč. sekúnd
5	144,52 s	98,93 s	89,37 s	88,53 s
10	213,23 s	113,07 s	107,48 s	104,48 s
20	430,92 s	165,33 s	151,19 s	147,76 s
30	637,57 s	186,02 s	167,46 s	162,35 s
40	736,24 s	205,80 s	187,31 s	182,64 s

na začiatok zvolili na 5 tak nám vznikla scéna o veľkosti 10x10x5. Vozíkov, shuttlov sme do scény pustili tiež desať ako tomu bolo v predošlom meraní a skúmali sme 5, 10, 20, 30 a napokon 40 prác. Tie časy u BFS0 sú skoro dá sa povedať polovičné a je to teda zapríčinené práve tou 3D rovinou pretože vozíky tak sa môžu viacej obiehať. Nie je to vylepšením BFS0 ten ostal pôvodný. Urobili sme teda ako v predošlom 2D stocku tristo meraní na jednu škálu teda na 5, 10 a tak ďalej. Len pre istotu, aby sme si boli istý a niečo nám neuniklo. Tie časy sme znovu spriemerovali a z nich je teda náš prvý graf pre 3D rovinu.



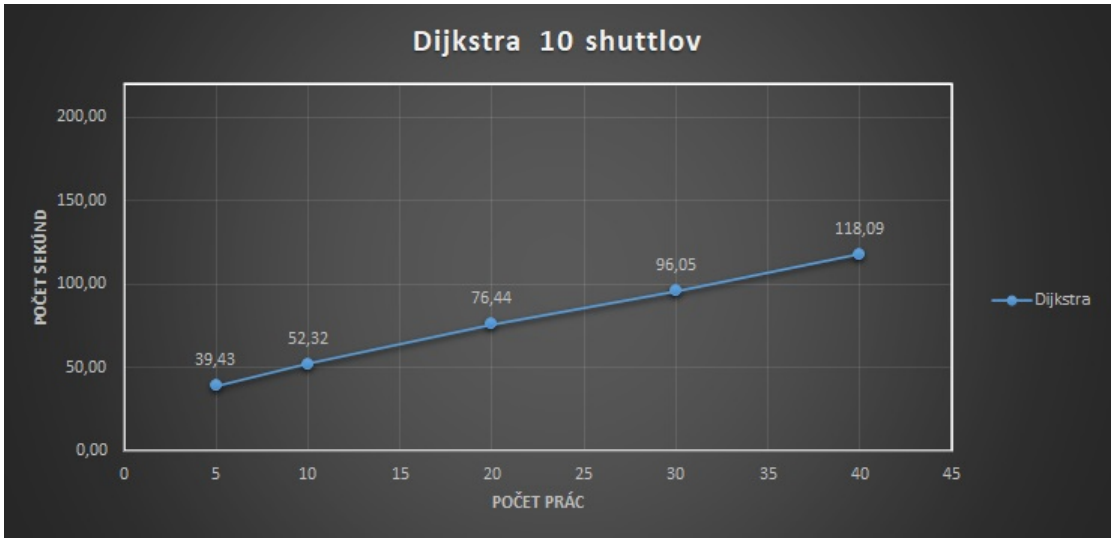
Obr. 6.7: Graf závislosti času od počtu prác pre algoritmus BFS0 3D

U druhého grafu pre BFS teda pre druhý náš algoritmus sme dali rovnaké podmienky a rovnaký počet spriemerovaných meraní čiže tristo na jednu škálu. na obr. 6.8 môžeme vidieť, že hodnoty sú nižšie ako tomu bolo pri 2D stocku a tu ešte nižšie aj vďaka 3D rovine.



Obr. 6.8: Graf závislosti času od počtu prác pre algoritmus BFS 3D

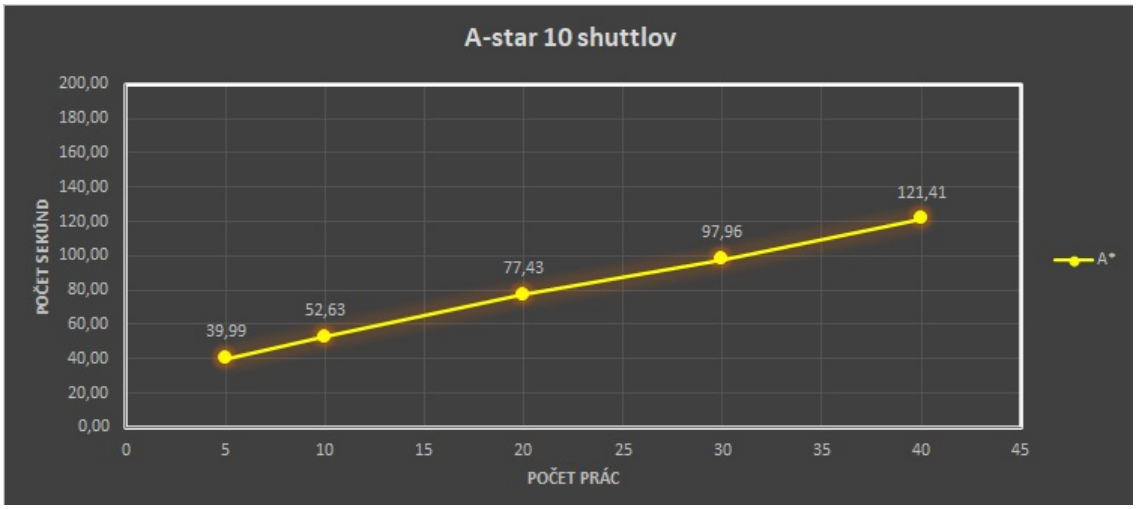
U nasledujúceho grafu sme nastavili tie isté podmienky ako pri ostatných. V 3D rovine sme chceli teda vidieť tiež závislosť času na počte prác s použitým dijkstry. Na obr. 6.9 môžno vidieť, že dijkstra tiež vyhráva časovo oproti predošlým dvom algoritmom a je tomu tak preto, lebo ako bolo už viac krát povedané tak má na tom zásluhu spomínaná tabuľka, vyhľadávacia tabuľka. Kde si dijkstra algoritmus vie ktorou cestou sa má vydať a ktorá je rýchlejšia, ktorá pomalšia a jednotlivé vzdialenosti k nódom. Všetko to má vo svojej tabuľke.



Obr. 6.9: Graf závislosti času od počtu prác pre algoritmus Dijkstra 3D

U A-Star na obr. 6.10 je situácia tiež podobná čiže menší čas oproti predošlým

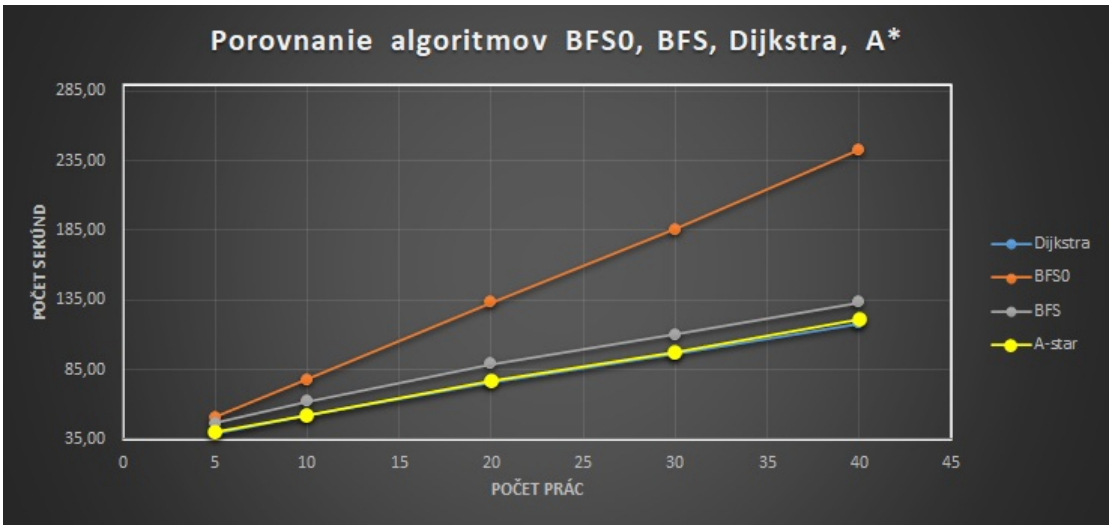
algoritmom a menší aj čo sa týka roviny. Tu platí, že A-Star bude lepší len v prípade veľkého počtu nódov oproti dijkstre a oproti BFS, BFS0 bude vždy. Vidno to aj na grafoch na obr. 6.11 a na obr. 6.12 kde dijkstra s A-Star majú podobné časi je to spôsobené aj tým, že v 3D rovine ako sme povedali tak sa vozíky môžu lepšie obchádzať a tým nevznikajú tak povediac zápchy na cestách k jednotlivým nódom. Preto tu je ten graf skôr viacej lineárny a s pribúdajúcim sa počtom jednotlivých prác stúpa čas skoro rovnomerne u všetkých algoritmov. Výpisom k druhej tabuľke pre 3D algoritmy je vidieť, že tomu tak je. U všetkých spomenutých grafoch boli urobené merania ktoré boli spriemerované, aby sme dosiahli čo najlepší výsledok a urobili sme merania rovnakým spôsobom ako 2D sklade čiže na jednu škálu vyše tri sto meraní. To nám síce zaberalo veľa času čo si povieme na ďalšom grafe, ale výsledky boli o to lepšie.



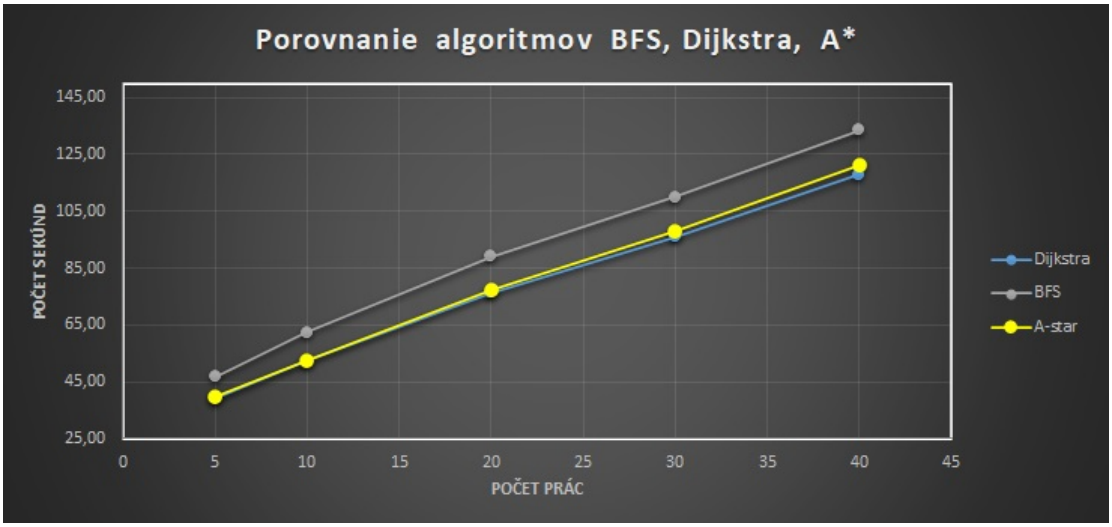
Obr. 6.10: Graf závislosti času od počtu prác pre algoritmus A-Star 3D

Ďalším poznatkom ktorý sme urobili bolo porovnanie na obr. 6.13, aby bolo vidno rozdiel koľko času by bolo treba pre splnenia práci keďže predpoklad je,že aplikácia by mala bežať nepretržite. Je to dobré spomenúť pretože od toho zavisí použitie metódy. Tu bolo teda urobených 50 meraní na každý jeden algoritmus a vypočítaný čas za koľko by danú prácu spravili je vidno, že pri porovnaní algoritmu BFS0 s dijkstrou je to polovičný čas čiže skoro 100 minút to predstavuje obrovský čas. BFS oproti dijkstre robí viac ako 10 minút čo je tiež veľmi veľa keď si predstavíme, že nám v podstate zaleží na každej nadbytočnej sekunde.

Hneď budúci graf na obr. 6.14 je príkladom toho, že so zvyšujúcim sa počtom pracujúcich vozíkov, shuttlov stúpa čas lineárne až po kým nebude viacej stretov.



Obr. 6.11: Graf závislosti času od počtu prác pre všetky algoritmy v 3D



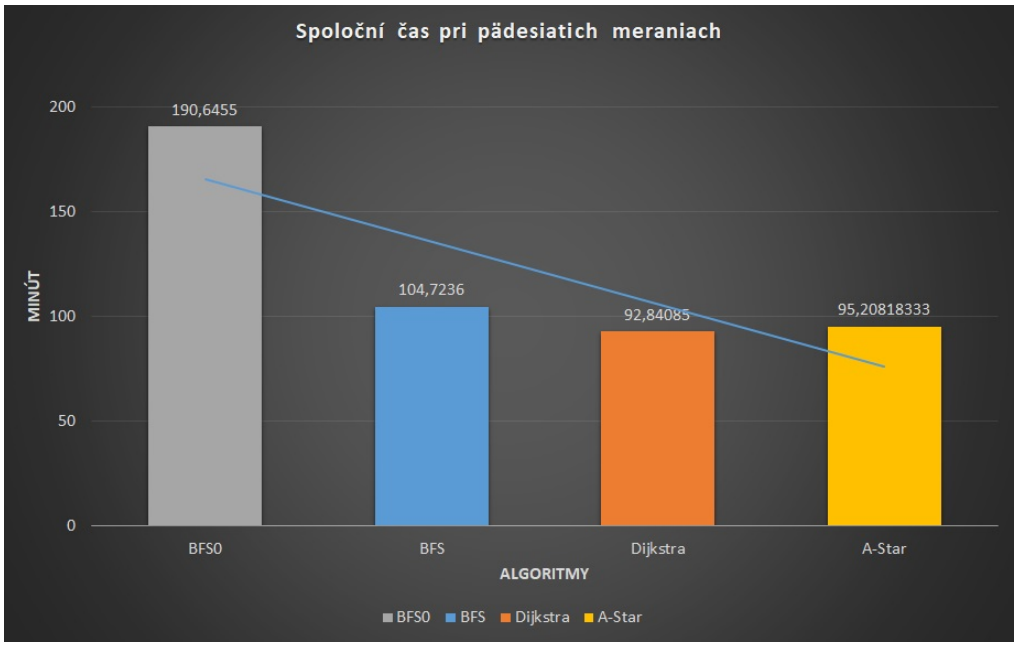
Obr. 6.12: Graf závislosti času od počtu prác pre BFS, Dijkstra,A-Star algoritmy v 3D

Tým by sa čas ešte zvýšil. Ak by sme brali do úvahy buď mali menši sklad je výhodnejšia dijkstra. Ak naopak veľký tak A-Star. Veľmi veľa vozíkov spôsobuje stret a aj zvýšenie počtu prác je tak potreba viacej času vidno to na grafe, že keď je ich viac čas je dlhší a to vďaka viacej stretov vozíkov.

Do nasledujúceho grafu sme vložili nové údaje, pretože sme každému vozíku začali počítať priemerný čas výpočtu na daný algoritmus. Znamená, že je to vlastne čas ktorý potrebuje vozík na rozhodovanie kam pôjde. Tento čas sme následne spriemerovali zo všetkých pracujúcich vozíkov na metódu, algoritmu. Graf môžno vidieť

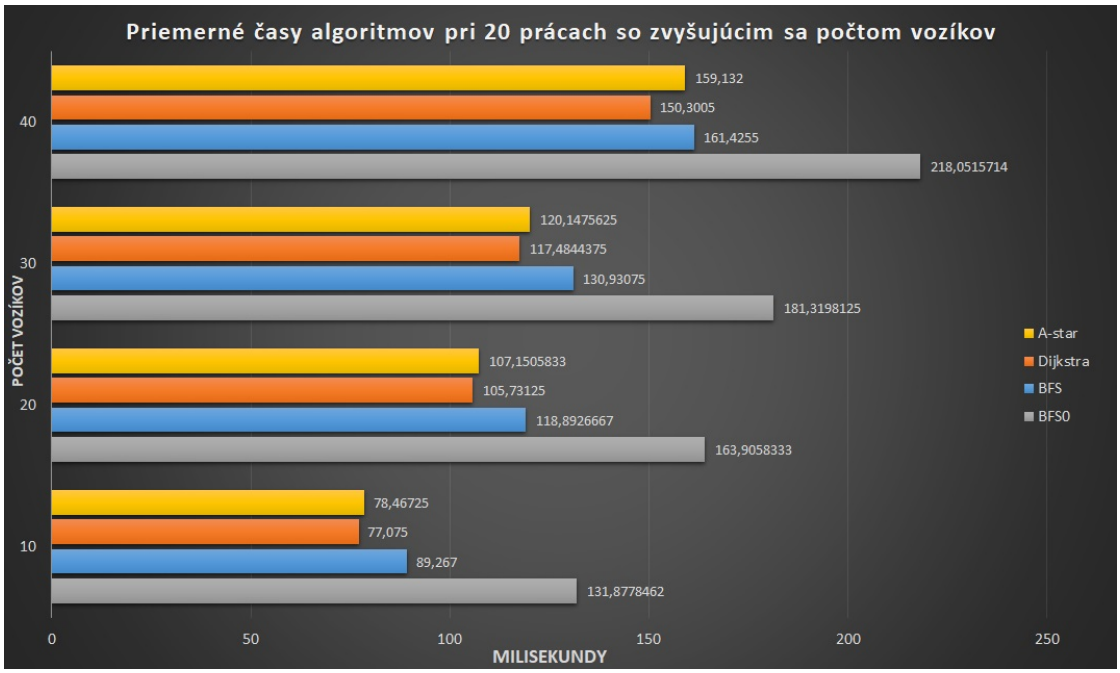
Tab. 6.2: Tabuľka počtu spriemerovaných časov od počtu prác pre jednotlivé algoritmy, 3D

Počet prác	BFS0 poč. sekúnd	BFS poč. sekúnd	Dijkstra poč. sekúnd	A* poč. sekúnd
5	51,07 s	46,76 s	39,43 s	39,99 s
10	77,80 s	62,37 s	52,32 s	52,63 s
20	133,38 s	89,00 s	76,44 s	77,43 s
30	185,79 s	110,10 s	96,05 s	97,96 s
40	243,21 s	133,66 s	118,09 s	121,41 s



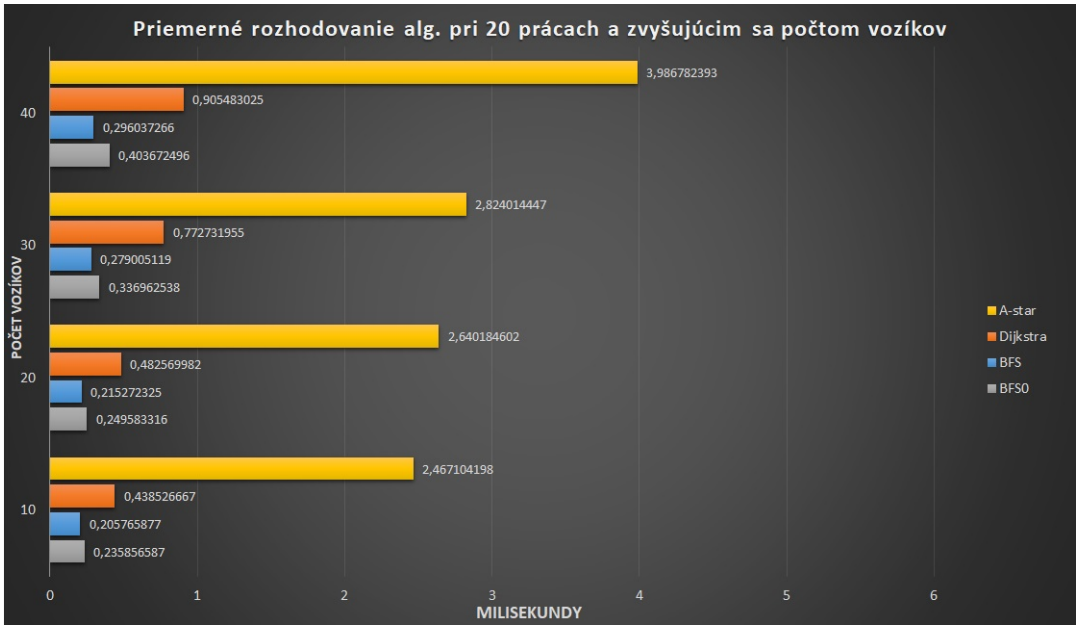
Obr. 6.13: Graf porovnania pre algoritmy v 3D pre 50 prác spriemerovaných časov

na obr. 6.15. Je na ňom veľmi dobre poznať, že každý algoritmus potrebuje iný čas rozhodovania. Z grafu vyplýva, že každý jeden algoritmus s pribúdajúcim sa množstvom vozíkov potrebuje viac času to značí oneskorenie kedy sa musí rozhodnúť kam pôjde. Najhoršie z našich meraní dopadol práve A-Star je to trochu pochopiteľné a logické z toho hľadiska, že je to z pohľadu programovania najzložitejší algoritmus a teda musí prebiehať najviac výpočtov. Avšak treba podotknúť, že rozprávame v milisekundách a tým si to bežní užívateľ nemusí hneď všimnúť. Problém by mohol nastať môžno len vtedy keby vozíkov bolo enormné množstvo, ale tu to A-Star zachraňuje práve veľkosťou skladu kde je lepšie použiť práve jeho ak by v danom sklade bolo naopak veľa vozíkov tak našim odporúčaním je určite dijkstra ktorá



Obr. 6.14: Graf práce algoritmov na základe počtu vozíkov

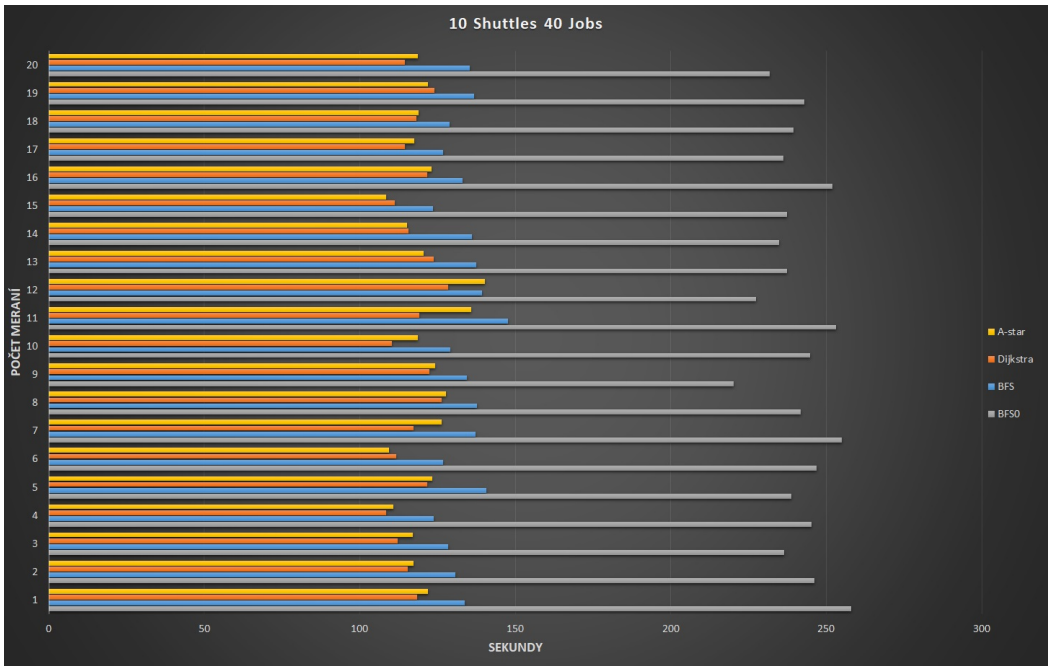
je na tom lepšie čo sa týka oneskorenia.



Obr. 6.15: Graf časového priemeru rozhodnosti algoritmov so zvyšujúcim sa počtom vozíkov

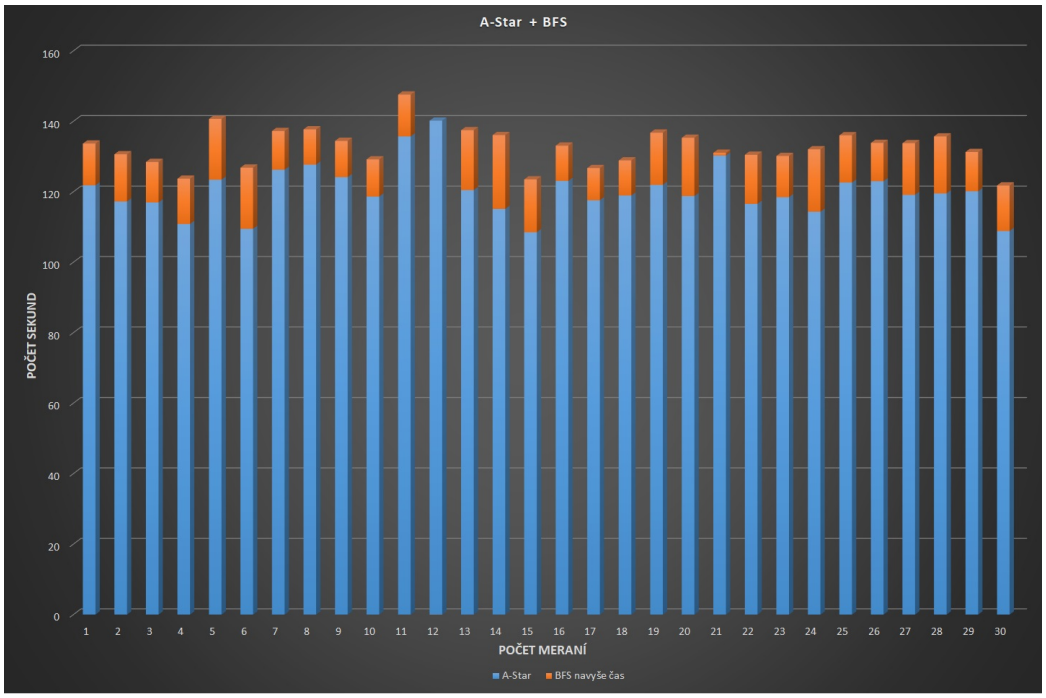
Na obr. 6.16 ďalej vidno porovnanie jednotlivých časov na algoritmy pri 30 me-

raniach. Tento graf sme zvolili práve preto, že tiež na ňom vidno, ktorý algoritmus lepšie pracuje. Ako tomu bolo u predošlých grafov, tak aj tu najlepšie vychádzajú A-Star a dijkstra. Odporúčanie, ktorý algoritmus vybrať by sme volili tak, že podľa počtu nódov a veľkosti skladu. Ako bolo povedané, každý z nich má svoje pre a proti. A-Star na veľký sklad a dijkstra naopak na menší.

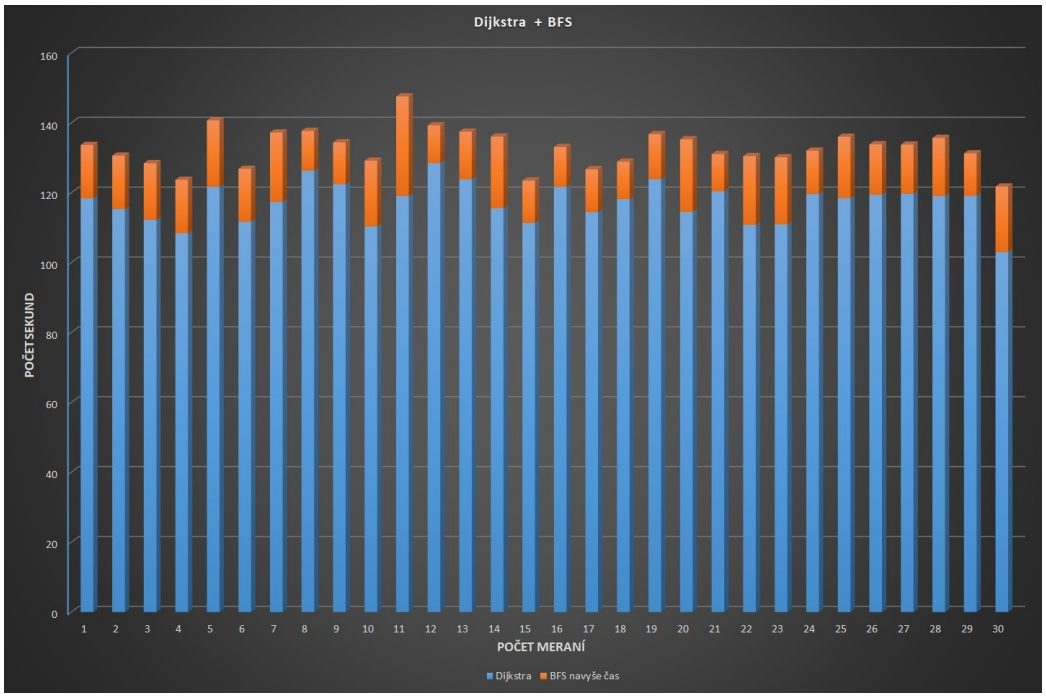


Obr. 6.16: Graf Porovnanie všetkých algoritmov 30 meraní

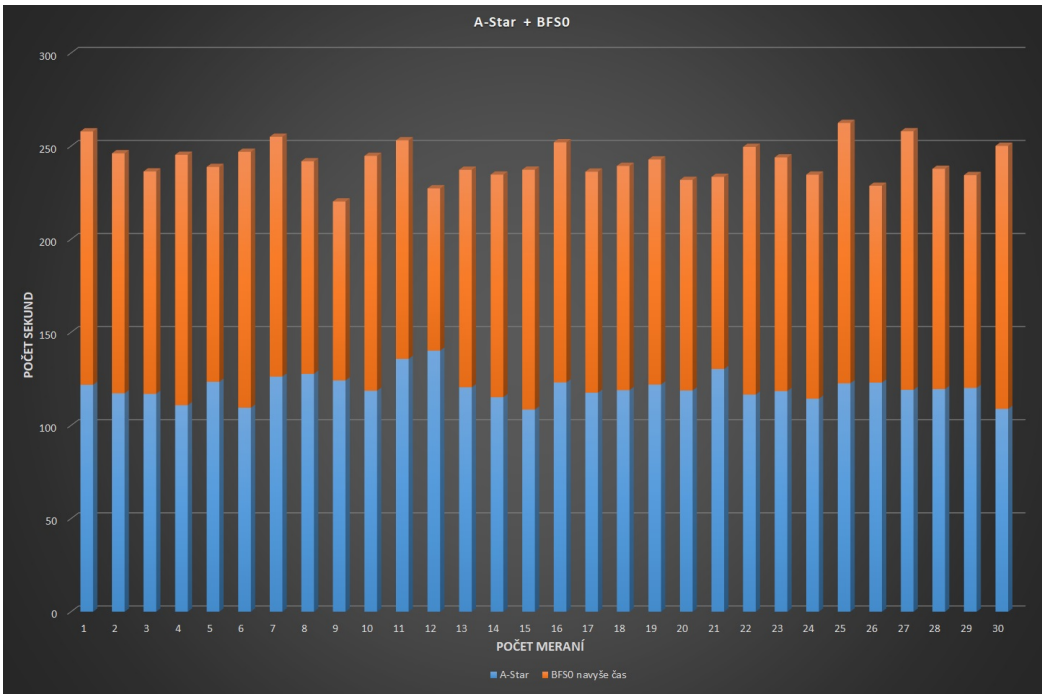
Ďalšími štyrmi grafmi na obr. 6.18, na obr. 6.17, na obr. 6.19, na obr. 6.20 sme chceli poukázať na to koľko by bolo treba času pri použití vybraného algoritmu. Znovu musíme podotknúť, že najlepšie dosahované časy sú práve pre dijkstru a A-Star. Konečným rozhodnutím teda môžeme konštatovať, že algoritmy a celý sklad fungujú veľmi dobre a podarilo sa nám aj splniť naše ciele zo zadania.



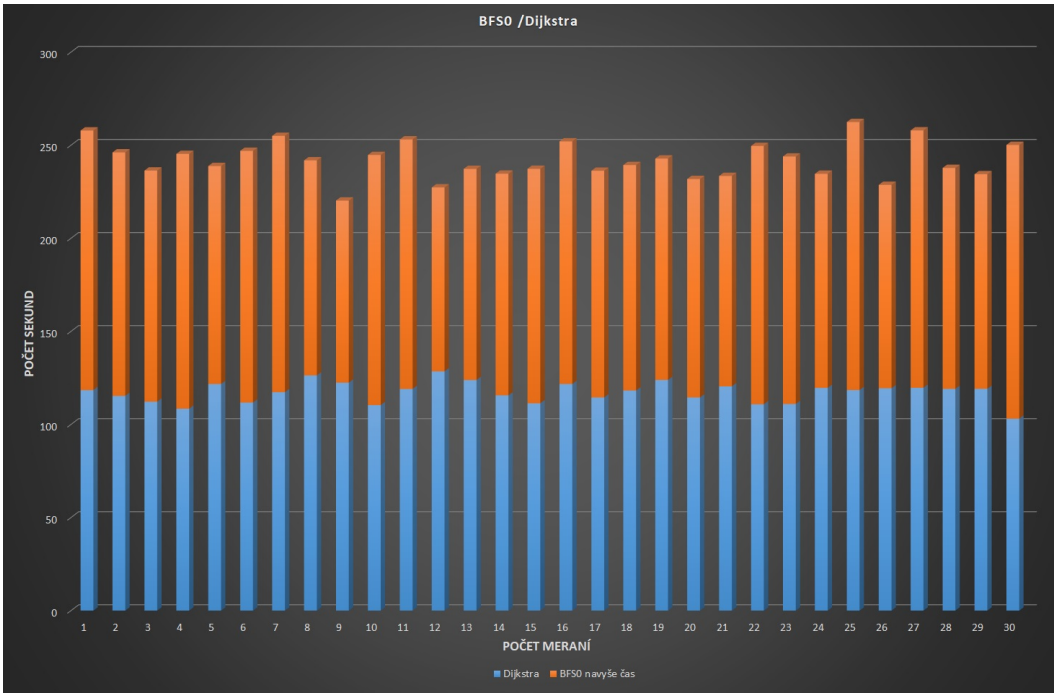
Obr. 6.17: Porovnanie BFS s A-Star 3D



Obr. 6.18: Porovnanie BFS s Dijkstrou 3D



Obr. 6.19: Porovnanie BFS0 s A-Star



Obr. 6.20: Porovnanie BFS0 s Dijkstrou

Záver

V našej práci sme sa zameriavali na skladové kyvadlové systémy (Shuttle system), ktoré sú našim riešením automatizácie skladu. V prvej časti sme sa zamerali na teoretickú časť problému, kde sme si vysvetlili, ako by sa uvedený problém dalo vyriešiť. Ďalej v tejto diplomovej práci sme sa zamerali na návrh čiastkovej koncepcie a technické riešenie pre riadiaci systém vozíkov, ktoré sa pohybujú v rámci regálového systému horizontálne, vertikálne. Nám sa to podarilo na základe GUI, window (okno) a to nám vykresľuje našu simuláciu v jave. Toto okno, alebo plátno, môže byť nastavené a môžno doň nasádzať jednotlivé nódy a cesty, skladové miesta. Tie môžu byť rôzne, keďže boli nastavené pomocou pixelov a neskôr do gridu. Kde je vidno celý simulovaný sklad a jeho pozície vidno na plátne, jednotlivé nódy a aj cesty tiež môžno vykresliť rôznymi farbami. To isté platí pre shuttle (vozíky), pri ktorých sme volili čiernu farbu, a vykreslenie biele sme volili naopak ako skladové miesta. Bolo to pre lepšiu viditeľnosť, a pre cesty medzi prechodom nódami sme tiež zvolili čiernu farbu. Shuttle sa taktiež aktuálne zobrazuje čiernou farbou. Nie je to však nutnosť, a je možná zmena na základe nastavenia.

Nasledujúcim bodom, ktorý sme pridali do scény, je aj prechod medzi jednotlivými nódmi. Bolo to na základe toho, aby sme mohli vidieť simuláciu prechodov. A tým, ako sa nám pohybujú a kam smerujú, sme pre shuttle zvolili sivú farbu, je to aj preto, aby sme vedeli zaistiť bezpečnosť a aby sme sledovali tento stav. Tento stav nám umožňuje napovedať budúci pohyb a predošlý pohyb, aby sa shuttle (vozíky) nezrazili. Rieši to našu problematiku komunikácie medzi jednotlivými shuttlami (vozíkmi). Ešte sme pridali aj samotné statusy, čo shuttle práve robí v našom prípade. Sú to: Ready znamená, že je pripravený na začatie práce, ak ešte nezačal, ale je pripravený; Moving znamená, že sa hýbe niekam medzi dvoma nódami; Loading znamená, že nakladá materiál, tu je potreba aj nastavenia času nakladania, my sme si nastavili čas dve sekundy, avšak môže sa meniť v závislosti od nákladu materiálu. Waiting znamená, že čaká na uvoľnenie trasy, napríklad, ak je trasa zablokovaná, ešte nie je vo finishe a vlastne stojí, nehýbe sa. Finish zobrazuje, že dokončil svoju prácu a dostal sa mimo skladu, vyšiel z neho. Keď vojde do skladu, čiže do scény, hýbe sa k najbližším jobom, prácam. Ak dokončí svoje joby, práce, vracia sa na to isté miesto, kde začal. To isté platí pre ostatné shuttle, vozíky ktoré vstupujú do scény. Do scény vstupujú postupne, ale vystupujú po dokončení práce.

Vyskúšali sme aktuálne zobrazenie simulácie a chod shuttlov na mriežke 6x6 s dvadsiatimi shuttlami vo fronte, kde ich naraz bolo v sklade desať a vždy si shuttle urobili svoju prácu automatizovane a do konca. Ešte sme pridali náhodné časy nakladania a náhodné rýchlosti medzi nódmi. Tiež spomenieme, že shuttle si sám hľadá cestu, nie je to tak, že by nám to riadil nejaký sklad.

Tu sme nadviazali na pokračovanie a pridávali sme rôzne vylepšenia. Jedným z nich bolo, že sme pridali, respektíve upravili algoritmus BFS, pôvodný sme označili ako BFS0. Ďalším krokom bolo pre nás implementovať aj iné vyhľadávacie algoritmy a porovnať ich prípadné časy. Tak sme skúsili aj iné algoritmy, u nás bol použitý algoritmus dijkstra, ktorý nám vylepšuje jednotlivé časy vykonávaných prác a to vďaka tomu, že si ukladá v sebe tabuľku všetkých nódov, jednotlivé rýchlosti a prepojenia medzi nimi. Takže vie, ktorou cestou to bude rýchlejšie a ktorou naopak nie. Čo je pre nás veľmi výhodné, keďže BFS algoritmus také nerobí. Nasledujúcim krokom sme zvolili ešte jeden algoritmus a je ním A-Star. Je to porovnateľne rovnaký algoritmus ako dijkstra, ale má pár výhod, ktoré sa nám hodili, a jedna z nich je, že ak máme veľký sklad, teda veľký počet nódov, tak A-Star je určený práve na tento prípad, keďže si neukladá v sebe celú mapu, teda tabuľku, ale iba časť z nej a berie nódy centralizovane. Dijkstra tak potrebuje vždy celú mapu a celú oblasť nódov, tým sa vlastne od A-Star líši a má to vplyv aj na rozhodovanie aj na celkový počet času. Teda oneskorenie, kedy vozík sa musí rozhodovať akou cestou sa vydá. Tieto algoritmy sme vyskúšali v našich laboratórnych podmienkach a to konkrétne na sklade o veľkosti gridu 10x10. Následne, aby sme algoritmy mohli porovnávať, tak sme si jednotlivé časy, údaje nechali vypisovať do súboru txt a tie sme prepisovali do excelu, kde bolo možné s nimi pracovať. Vytvorili sme grafy hodnôt, kde bolo vidno závislosti času od počtu jednotlivých prác. Využívali sme pri tom všetky algoritmy. Nám vychádzali najlepšie časy práve pre algoritmy dijkstra a A-Star. Spomenieme, že každý má výhodu v niečom inom. Dijkstra pri rozhodovaní, ako vyplýva z grafov, a naopak A-Star pri čo najväčšom počte nódov, kedy je veľmi veľký sklad. Takže by sme odporučili pri výbere algoritmu sa zamerať na veľkosť skladu. Ďalším poznatkom, ktorý spomenieme, že bol rozdiel medzi 2D skladom a 3D skladom. Keďže sme sa zameriavali aj na vertikálnu časť práce, kde nám sklad v podstate narástol. Z 2D sme ho previedli do 3D roviny, a všetky spomenuté problémy, ktoré boli vyriešené v tejto práci pre horizontálnu časť sme vyriešili aj pre vertikálnu časť. Všetky spomenuté časti ostali podobné. Vozíky sa nám zmenili na 3D teda na guľaté objekty a nódy tiež vyzerajú ako jednotlivé boxy. Symbolika tiež ostala rovnaká, pohyb, čakanie a nakládka. Avšak tu nódy len zhrubnú teda majú hrubšiu čiaru keď sú obsadené, alebo budú v budúcnosti obsadené. Ešte jedno vylepšenie sme použili, a to, že sme vložili nové údaje, pretože sme každému vozíku začali počítať priemerný čas výpočtu na daný algoritmus. Znamená, že je to vlastne čas, ktorý potrebuje vozík na rozhodovanie, kam pôjde. Tento čas nám veľmi dobre určuje, že každý algoritmus potrebuje iný čas rozhodovania. V podstate s pribúdajúcim sa množstvom vozíkov potrebuje viac času, to značí oneskorenie, kedy sa musí rozhodnúť, kam pôjde. Najhoršie z našich meraní dopadol práve A-Star, je to trochu pochopiteľné a logické z toho hľadiska, že je to z pohľadu programovania najzložitejší

algoritmus a teda musí prebiehať u neho najviac výpočtov. Avšak treba podotknúť, že sa rozprávame v milisekundách a tým si to bežný užívateľ nemusí hneď všimnúť. Problém by mohol nastať môžno len vtedy, keby vozíkov bolo enormné množstvo, ale tu to A-Star zachraňuje práve veľkosťou skladu, kde je lepšie použiť práve jeho. Ak by v danom sklade bolo naopak veľa vozíkov, tak našim odporúčaním je určite dijkstra, ktorá je na tom lepšie čo sa týka oneskorenia. Takže môžeme s určitosťou povedať, že najlepšie časy budú pre dijkstru a A-Star, čo sa týka splnených prác. Našim výstupom práce je teda aj celkový počet meraní, ktoré sme uskutočnili a bolo ich približne 3000, kde to vyžadovalo veľa času. Konštatujeme, že algoritmy a celý sklad fungujú veľmi dobre a podarilo sa nám aj splniť naše ciele zo zadania.

Literatúra

[1] River systems: Warehouse Automation *River systems* [online]. 2022 [cit. 29. 6. 2022]. Dostupné z URL: <https://6river.com/what-is-a-warehouse-shuttle-system/>.

[2] Pallet Shuttle System: Pallet Shuttle (AR Shuttle) *Pallet Shuttle System* [online]. 2022. Dostupné z URL: <https://www.ar-racking.com/en/storage-systems/automated-warehouses/pallet/pallet-shuttle>.

[3] Interlake mecalux: Industrial rack types for warehouses *Interlake mecalux* [online]. 2022 [cit. 3. 5. 2022]. Dostupné z URL: <https://www.interlakemecalux.com/blog/types-industrial-pallet-racks-warehouse>.

[4] Shuttle Systems: Understanding Warehouse Shuttle Systems and Their Applications *Shuttle Systems* [online]. 2022 [cit. 2. 12. 2021]. Dostupné z URL: <https://www.swisslog.com/en-gb/case-studies-and-resources/blog/understanding-warehouse-shuttle-systems>.

[5] Programiz: BFS algorithm *Programiz* [online]. 2022 [cit. 3. 5. 2019]. Dostupné z URL: <https://www.programiz.com/dsa/graph-bfs>.

[6] Breadth First Search alebo BFS pre graf: Breadth-First Traversal *Breadth First Search alebo BFS pre graf* [online]. 2022 [cit. 8. 11. 2022]. Dostupné z URL: <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>.

[7] Stack overflow: Drawing a line with arrow in Java *Stack overflow* [online]. 2022 [cit. 4. 7. 2015]. Dostupné z URL: <https://stackoverflow.com/questions/4112701/drawing-a-line-with-arrow-in-java/4112875#4112875>.

[8] Stack overflow Bfs: Bfs Implementation using ArrayList in Java *Stack overflow Bfs* [online]. 2022 [cit. 7. 6. 2018]. Dostupné z URL: <https://stackoverflow.com/questions/36084437/bfs-implementation-using-arraylist-in-java>.

[9] Geeksforgeeks Dijkstra's: Dijkstra's Shortest Path Algorithm *Geeksforgeeks Dijkstra's* [online]. 2022 [cit. 31. 8. 2022]. Dostupné z URL: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>.

[10] Programiz dijkstra: Dijkstra's Algorithm *Programiz dijkstra* [online]. 2022 [cit. 3. 5. 2019]. Dostupné z URL: <https://www.programiz.com/dsa/dijkstra-algorithm>.

[11] Simpli learn dijkstra: What Is Dijkstra's Algorithm and Implementing the Algorithm through a Complex Example *Simpli learn dijkstra* [online]. 2022 [cit. 3. 5. 2019]. Dostupné z URL: <https://www.simplilearn.com/tutorials/cyber-security-tutorial/what-is-dijkstras-algorithm?tag=Dijkstra>.

[12] Science direct: Multi-shuttle crane scheduling in automated storage and retrieval systems *Science direct* [online]. 2022 [cit. 2. 2. 2022]. Dostupné z URL: <https://www.sciencedirect.com/science/article/abs/pii/S0377221722000819>.

[13] Geeksforgeeks A* Search: A* Search Algorithm *Geeksforgeeks A* Search* [online]. 2022 [cit. 30. 5. 2022]. Dostupné z URL: <https://www.geeksforgeeks.org/a-search-algorithm/>.

[14] Simpli learn: A* Algorithm Concepts and Implementation *Simpli learn* [online]. 2022 [cit. 15. 11. 2022]. Dostupné z URL: <https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm>.

Zoznam symbolov a skratiek

BSF Prehľadávanie do šírky – Breadth-first search
FIFO Prvý dnu, prvý von– First In First Out
LIFO Posledný dnu, prvý von– Last In First Out

Zoznam príloh

A Výpis kódu **71**

A.1 Main . . . 71
A.2 Main pokračovanie . . . 72
A.3 NodeConnection . . . 73
A.4 Shuttle . . . 74
A.5 Shuttle pokračovanie prvé . . . 75
A.6 Shuttle pokračovanie druhé . . . 76
A.7 Shuttle pokračovanie tretie . . . 77
A.8 Shuttle pokračovanie štvrté . . . 78
A.9 Shuttle pokračovanie piate . . . 79
A.10 Shuttle pokračovanie šieste . . . 80
A.11 Shuttle pokračovanie siedme . . . 81
A.12 Shuttle pokračovanie ôsmé . . . 82
A.13 Shuttle pokračovanie deviate . . . 83
A.14 Shuttle pokračovanie desiate . . . 84
A.15 Shuttle pokračovanie jedenáste . . . 85
A.16 Shuttle pokračovanie dvanáste . . . 86
A.17 Shuttle pokračovanie trináste . . . 87
A.18 Shuttle pokračovanie štrnáste . . . 88
A.19 Shuttle3D . . . 89
A.20 Shuttle3D pokračovanie prvé . . . 90
A.21 Stock . . . 91
A.22 Stock pokračovanie prvé . . . 92
A.23 Stock pokračovanie druhé . . . 93
A.24 Stock pokračovanie tretie . . . 94
A.25 StockNode . . . 95
A.26 StockNode3D . . . 96
A.27 StockNode3D pokračovanie prvé . . . 97
A.28 StockSimulation . . . 98
A.29 StockSimulation pokračovanie prvé . . . 99
A.30 StockSimulation pokračovanie druhé . . . 100
A.31 StockSimulation pokračovanie tretie . . . 101
A.32 StockSimulation pokračovanie štvrté . . . 102
A.33 StockSimulation pokračovanie piate . . . 103
A.34 StockSimulation pokračovanie šieste . . . 104
A.35 StockSimulation3D . . . 105

A.36 StockSimulation3D pokračovanie prvé 106
A.37 StockSimulation3D pokračovanie druhé 107
A.38 StockSimulation3D pokračovanie tretie 108
A.39 StockSimulation3D pokračovanie štvrté 109
A.40 StockSimulation3D pokračovanie piate 110
A.41 StockSimulation3D pokračovanie šieste 111
A.42 StockSimulation3D pokračovanie siedme 112
A.43 StockSimulation3D pokračovanie ôsme 113
A.44 StockSimulation3D pokračovanie deviate 114
A.45 StockSimulation3D pokračovanie desiate 115

B Obsah elektronickej prílohy **116**

A Výpis kódu

A.1 Main

Na výpisu A.1 vidíme Main v jazyku java.

Výpis A.1: Príklad výpisu kódu Main

```
import javax.swing.*;

public class Main {
    public static void main(String[] args) {

        //new StockSimulation3D();
        while (true) {

StockSimulation3D.randomSeed = Math.round(Math.random()
* Long.MAX_VALUE);

 for (int i = 0; i <= 3; i++) {
      if (i == 0) Shuttle.method = Shuttle.Method.BFS0;
 else if (i == 1) Shuttle.method = Shuttle.Method.BFS;
 else if (i == 2) Shuttle.method = Shuttle.Method.DIJKSTRA;
 else if (i == 3) Shuttle.method = Shuttle.Method.A_STAR;

             // Reset static counters
             StockNode.counter = 0;
             Shuttle.counter = 0;
             Shuttle.methodTimesNs.clear();

             // Create stock simulation object
  StockSimulation3D simulation = new StockSimulation3D();

          if (!simulation.stock.allConnected()) {
              simulation.updateTimer.stop();
              System.out.println("Generated␣stock␣with␣seed␣"
  + StockSimulation3D.randomSeed + "␣is␣not␣fully␣connected");
                    break;
                }
```

A.2 Main pokračovanie

Na výpisu A.2 vidíme Main pokračovanie v jazyku java.

Výpis A.2: Príklad výpisu kódu Main pokračovanie

```
                                                                    1
// Create JFrame object for visualisation                           2
          JFrame frame = new JFrame();                              3
          frame.add(simulation);                                    4
                                                                    5
                                                                    6
// Add simulation panel into frame                                  7
          frame.pack();                                             8
                                                                    9
                                                                    10
 // Block frame resizing                                           11
          frame.setResizable(false);                                12
                                                                    13
                                                                    14
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);             15
                                                                    16
                                                                    17
 // Set window visible                                             18
          frame.setVisible(true);                                   19
                                                                    20
                                                                    21
 // Wait for simulation to finish                                  22
           while (frame.isVisible()) {                              23
                    try {                                           24
                        Thread.sleep(1000);                         25
                                                                    26
                   } catch (InterruptedException e) {               27
                        e.printStackTrace();                        28
                    }                                               29
                }                                                   30
            }                                                       31
        }                                                           32
    }                                                               33
}                                                                   34
```

A.3 NodeConnection

Na výpisu A.3 vidíme triedu NodeConnection v jazyku java.

Výpis A.3: Príklad výpisu triedy NodeConnnection

```
public class NodeConnection {                                          1
  double distance;     // Metric distance between nodes               2
  double maxSpeed;     // Maximal speed between nodes                 3
  StockNode destinationNode;  // Destination node                     4
                                                                       5
  // Calculates the cost of connection,                                6
  the smaller the cost, the faster the connection is                   7
    public double cost() {                                             8
        return distance / maxSpeed;                                    9
    }                                                                  10
    public double score() {                                            11
        return maxSpeed / distance;                                    12
    }                                                                  13
}                                                                      14
```

A.4 Shuttle

Na výpisu A.4 vidíme triedu Shuttle v jazyku java.

Výpis A.4: Príklad výpisu triedy Shuttle

```
import javax.vecmath.Point3d;
import java.util.*;
import static java.lang.Math.abs;

public class Shuttle {// Definition of shuttle states
 public enum State {
READY,// Shuttle is ready to be deployed into stock
MOVING,// Shuttle is in movement
LOADING,// Shuttle is loading items in stock node position
WAITING,
// Shuttle is blocked and is waiting for clearing the path
FINISHED // Shuttle has finished job in stock
    }
    public enum Method {
        BFS0,
        BFS,
        DIJKSTRA,
        A_STAR
    }
static int counter = 0; // Counter of all created shuttles
int index;  // Index of current shuttle
Point3d position = new Point3d();
// Current position of shuttle
ArrayList<StockNode> jobs = new ArrayList<StockNode>();
// List of all shuttle jobs to be done
ArrayList<StockNode> curentPath = new ArrayList<StockNode>();
// Currently planned shuttle path
StockNode currentNode;
// Current stock node where shuttle is positioned
State state = State.READY;
// Current state of shuttle
long arriveTime = 0;
// Start time of loading items from stock node
long waitTime = 0; // Start time of shuttle wait state
int jobCount = 0; // Count of shuttle jobs
```

A.5 Shuttle pokračovanie prvé

Na výpisu A.5 vidíme triedu Shuttle pokračovanie prvé v jazyku java.

Výpis A.5: Príklad výpisu triedy Shuttle pokračovanie prvé

```
public static ArrayList<Long> methodTimesNs =
new ArrayList<>();
public static Method method = Method.DIJKSTRA;
    public Shuttle()
    {// Set shuttle index
        index = counter++;
    }// Returns true if shuttle is in currently stock
    public boolean inStock() {
        return state == State.MOVING || state
== State.LOADING || state == State.WAITING;
    }
    public void setStartNode(StockNode node) {
      jobCount = jobs.size();
      currentNode = node;
      position = new Point3d(currentNode.position.getX(),
currentNode.position.getY(), currentNode.position.getZ());
        jobs.add(node); // Set start node also as end node
    }
    // Returns direction of movement of shuttle
    private Point3d dir(Point3d p1, Point3d p2) {
        Point3d direction = new Point3d(p2.getX()
- p1.getX(), p2.getY() - p1.getY(),p2.getZ()
- p1.getZ());
        double sum = abs(direction.getX())
+ abs(direction.getY())+ abs(direction.getZ());
        direction.set(direction.getX() / sum,
direction.getY() / sum, direction.getZ() / sum);
        return direction;
    }
  // Returns speed of currently used node connection
  double currentConnectionSpeed() {
if (curentPath.size() > 1) {
for (int i = 0; i < curentPath.get(0).connections.size(); i++)
{
if (curentPath.get(0).connections.get(i).destinationNode
== curentPath.get(1)) {
return curentPath.get(0).connections.get(i).maxSpeed;
```

A.6 Shuttle pokračovanie druhé

Na výpisu A.6 vidíme triedu Shuttle pokračovanie druhé v jazyku java.

Výpis A.6: Príklad výpisu triedy Shuttle pokračovanie druhé

```
   }                                                                    1
            }                                                           2
        }                                                               3
        return 0;                                                       4
    }                                                                   5
    int sign(double value) {                                            6
        if (value == 0) {                                               7
            return 0;                                                   8
        }                                                               9
        else {                                                          10
            return (int)Math.round(value / abs(value));                 11
        }                                                               12
    }                                                                   13
 // Returns the best path to current job                                14
 private ArrayList<StockNode> findBFS0() {                              15
 Set<StockNode> visitedNodesMap = new HashSet<>();                      16
// Map of already visited nodes for algorithm                          17
 ArrayList<ArrayList<StockNode>> allPaths =                             18
new ArrayList<ArrayList<StockNode>>();                                  19
// List of all possible paths
 ArrayList<StockNode> initPath = new ArrayList<StockNode>();            20
 initPath.add(currentNode);                                             21
// Add current node to start of path                                   22
 allPaths.add(initPath); // Create initialisation path                   23
 visitedNodesMap.add(initPath.get(0));                                  24
 boolean newNodeFound = true;                                           25
// Flag if new path was found in last search loop                      26
 while (newNodeFound) {                                                 27
 newNodeFound = false;                                                  28
 for (int i = 0; i < allPaths.size(); i++) {                            29
 ArrayList<StockNode> path = allPaths.get(i);                           30
for (NodeConnection connection:path.get(path.size()                     31
- 1).connections) {                                                     32
if (!visitedNodesMap.contains(connection.destinationNode)               33
&& !connection.destinationNode.occupied) {                              34
visitedNodesMap.add(connection.destinationNode);                        35
newNodeFound = true;    // Set flag to true                             36
```

A.7 Shuttle pokračovanie tretie

Na výpisu A.7 vidíme triedu Shuttle pokračovanie tretie v jazyku java.

Výpis A.7: Príklad výpisu triedy Shuttle pokračovanie tretie

```
  ArrayList<StockNode> newPath = new ArrayList<StockNode>();
  newPath = (ArrayList<StockNode>)path.clone();
  newPath.add(connection.destinationNode);
  allPaths.add(newPath);
                    }
                }
            }
        }
        // Find desired path from all the found paths
        for (ArrayList<StockNode>path:allPaths) {
if (path.get(0) == currentNode
&& path.get(path.size() - 1) == jobs.get(0)) {
                return path;
            }
        }
        return new ArrayList<>();
    }
    // Find the shortest path to any job in list
    private ArrayList<StockNode> findBFS() {
        Set<StockNode> visitedNodesMap = new HashSet<>();
        // Map of already visited nodes for algorithm
        ArrayList<ArrayList<StockNode>> allPaths =
new ArrayList<ArrayList<StockNode>>();
// List of all possible paths
        ArrayList<StockNode> initPath =
new ArrayList<StockNode>();
initPath.add(currentNode);// Add current node to start of path
        allPaths.add(initPath);// Create initialisation path
        visitedNodesMap.add(initPath.get(0));
        boolean newNodeFound = true;
// Flag if new path was found in last search loop
        while (newNodeFound) {
            newNodeFound = false;
            for (int i = 0; i < allPaths.size(); i++) {
                ArrayList<StockNode> path = allPaths.get(i);
for (NodeConnection connection:path.get(path.size()
- 1).connections) {
```

A.8 Shuttle pokračovanie štvrté

Na výpisu A.8 vidíme triedu Shuttle pokračovanie štvrté v jazyku java.

Výpis A.8: Príklad výpisu triedy Shuttle pokračovanie štvrté

```
if (!visitedNodesMap.contains(connection.destinationNode)
&& !connection.destinationNode.occupied) {
     visitedNodesMap.add(connection.destinationNode);
     newNodeFound = true;    // Set flag to true
ArrayList<StockNode> newPath = new ArrayList<StockNode>();
     newPath = (ArrayList<StockNode>)path.clone();
     newPath.add(connection.destinationNode);
     allPaths.add(newPath);
                    }
                }
            }
        }

        // Find desired path from all the found paths
        for (ArrayList<StockNode>path : allPaths) {
            for (int i = 0; i < jobs.size(); i++) {
                if (path.get(0) == currentNode
&& path.get(path.size() - 1) == jobs.get(i)) {
                    if (i == jobs.size() - 1) {
  // If found job is last index
                        if (jobs.size() == 1) {
  // If only last job is left (out of stock)
                            return path;
                        }
                    }
                    else {
                        Collections.swap(jobs, 0, i);
 // Set nearest job node as current job
                        return path;
                    }
                }
            }
        }
        return new ArrayList<>();
    }
```

A.9 Shuttle pokračovanie piate

Na výpisu A.9 vidíme triedu Shuttle pokračovanie piate v jazyku java.

Výpis A.9: Príklad výpisu triedy Shuttle pokračovanie piate

```
 class DijkstraPair                                                    1
    {                                                                  2
        double cost;   // Score of connection                         3
        StockNode node;                                                4
public DijkstraPair(double conCost, StockNode stockNode) {             5
            cost = conCost;                                            6
            node = stockNode;                                          7
        }                                                              8
    }                                                                  9
    private ArrayList<StockNode> findDijkstra() {                      10
                                                                       11
// Dijkstra table, contains <vertex, distance, prev vertex>           12
Map<StockNode, DijkstraPair> dijkstraTable = new HashMap<>();          13
                                                                       14
        Set<StockNode> visited = new HashSet<>();                      15
        Set<StockNode> newNodes = new HashSet<>();                     16
        newNodes.add(currentNode);                                     17
dijkstraTable.put(currentNode, new DijkstraPair(0.0, null));          18
                                                                       19
      while (visited.size() != newNodes.size()) {                      20
         Set<StockNode> newNodesOld = new HashSet<>(newNodes);         21
                                                                       22
          // Go through all new nodes                                 23
          for (StockNode newNode: newNodesOld) {                       24
                                                                       25
                // If new node was visited before, skip it           26
                if (visited.contains(newNode))                         27
                    continue;                                          28
                                                                       29
                // Loop through all neighbours of the new node       30
          for (NodeConnection con : newNode.connections) {             31
                                                                       32
                    // Skip node that is occupied                      33
                    if (con.destinationNode.occupied)                  34
                        continue;                                      35
```

A.10 Shuttle pokračovanie šieste

Na výpisu A.10 vidíme triedu Shuttle pokračovanie šieste v jazyku java.

Výpis A.10: Príklad výpisu triedy Shuttle pokračovanie šieste

```
if (dijkstraTable.containsKey(con.destinationNode)) {
// If current path is already better, do not add new
   if (dijkstraTable.get(con.destinationNode).cost
< dijkstraTable.get(newNode).cost + con.cost())
                            continue; }
                    // Add new best path
                    dijkstraTable.put(con.destinationNode,
new DijkstraPair(dijkstraTable.get(newNode).cost
+ con.cost(), newNode));
                    newNodes.add(con.destinationNode);
                }
                // Add new node into visited nodes
                visited.add(newNode);
            }
        }
        StockNode bestJob = jobs.get(0);
        double bestScore = Double.MAX_VALUE;
        int bestIndex = -1;
        for (int i = 0; i < jobs.size() - 1; i++) {
            if (dijkstraTable.containsKey(jobs.get(i))) {
                if (dijkstraTable.get(jobs.get(i)).cost
< bestScore) {
bestScore = dijkstraTable.get(jobs.get(i)).cost;
                    bestJob = jobs.get(i);
                    bestIndex = i;
                }
            }
        }
        // If best job was not found
        if (bestIndex == -1) {
            // If only last job left (going home)
            if (jobs.size() == 1)
                bestJob = jobs.get(0);  // Go home
            else
                return new ArrayList<>();
        }
```

A.11 Shuttle pokračovanie siedme

Na výpisu A.11 vidíme triedu Shuttle pokračovanie siedme v jazyku java.

Výpis A.11: Príklad výpisu triedy Shuttle pokračovanie siedme

```
 else if (bestIndex != 0) {
            // Set nearest job node as current job
            Collections.swap(jobs, 0, bestIndex);
        }
        // Prepare best job path
        ArrayList<StockNode> bestJobPath = new ArrayList<>();
        while (true) {
            bestJobPath.add(0, bestJob);
            if (dijkstraTable.containsKey(bestJob)
&& dijkstraTable.get(bestJob).node != null)
                bestJob = dijkstraTable.get(bestJob).node;
            else
                break;
        }
        // Check if path to best job was found successfully
        if (bestJobPath.get(0) == currentNode
&& bestJobPath.get(bestJobPath.size() - 1) == jobs.get(0))
            return bestJobPath;
        else
            return new ArrayList<>();
    }
    private double getPathCost(ArrayList<StockNode> path) {
        double cost = 0.0;
        for (int i = 0; i < path.size() - 1; i++) {
            for (NodeConnection connection :
path.get(i).connections) {
                if (connection.destinationNode ==
                path.get(i+1)) {
                    cost += connection.cost();
                    break;
                }
            }
        }
        return cost;
    }
```

A.12 Shuttle pokračovanie ôsmé

Na výpisu A.12 vidíme triedu Shuttle pokračovanie ôsmé v jazyku java.

Výpis A.12: Príklad výpisu triedy Shuttle pokračovanie ôsmé

```
                                                                    1
    private ArrayList<StockNode> findAStar() {                      2
        Set<StockNode> visitedNodesMap = new HashSet<>();           3
// Map of already visited nodes for algorithm                       4
        ArrayList<ArrayList<StockNode>> allPaths =                  5
new ArrayList<ArrayList<StockNode>>();                              6
// List of all possible paths                                       7
        ArrayList<StockNode> initPath =                             8
new ArrayList<StockNode>();                                         9
        initPath.add(currentNode);                                  10
// Add current node to start of path                                11
        allPaths.add(initPath); // Create initialisation path       12
        visitedNodesMap.add(currentNode);                           13
while (true) {// Find next node with lowest cost                    14
            double lowestCost = Double.MAX_VALUE;                   15
ArrayList<StockNode> newBestPath = new ArrayList<>();               16
for (ArrayList<StockNode> path : allPaths) {                        17
double currentPathCost = getPathCost(path);                         18
// Calculate current path cost                                      19
for (NodeConnection connection : path.get(path.size()               20
- 1).connections) {                                                 21
if (!visitedNodesMap.contains(connection.destinationNode)           22
&& !connection.destinationNode.occupied) {                          23
if ((currentPathCost + connection.cost()) < lowestCost) {           24
lowestCost = currentPathCost + connection.cost();                   25
newBestPath = (ArrayList<StockNode>)path.clone();                   26
newBestPath.add(connection.destinationNode);                        27
// visitedNodesMap.add(connection.destinationNode);                 28
// newNodeFound = true;    // Set flag to true                     29
// ArrayList<StockNode> newPath = new ArrayList<StockNode>();      30
// newPath = (ArrayList<StockNode>)path.clone();                    31
// newPath.add(connection.destinationNode);                         32
// allPaths.add(newPath);                                           33
                        }                                           34
                    }                                               35
                }                                                   36
            }                                                       37
```

A.13 Shuttle pokračovanie deviate

Na výpisu A.13 vidíme triedu Shuttle pokračovanie deviate v jazyku java.

Výpis A.13: Príklad výpisu triedy Shuttle pokračovanie deviate

```
                                                                    1
        // No new node was found                                    2
             if (lowestCost == Double.MAX_VALUE) {                  3
                 break;                                             4
             }                                                      5
                                                                    6
        // Check if best path is destination to one of jobs         7
             int bestIndex = -1;                                    8
    for (int i = 0; i < jobs.size() - 1; i++) {                     9
        if (jobs.get(i) ==                                          10
newBestPath.get(newBestPath.size() - 1)) {                          11
                     bestIndex = i;                                 12
                     break;                                         13
                 }                                                  14
             }                                                      15
                                                                    16
             // If only last job is on list                         17
   if (jobs.size() == 1) {                                          18
     if (jobs.get(0) ==                                             19
newBestPath.get(newBestPath.size() - 1)) {                          20
                     bestIndex = 0;                                 21
                 }                                                  22
             }                                                      23
                                                                    24
             // If best job was not found                           25
             if (bestIndex >= 0) {                                  26
                 // Set nearest job node as current job             27
                 Collections.swap(jobs, 0, bestIndex);              28
                 return newBestPath;                                29
             }                                                      30
                                                                    31
                                                                    32
visitedNodesMap.add(newBestPath.get(newBestPath.size() - 1));       33
             allPaths.add(newBestPath);                             34
        }                                                           35
        return new ArrayList<>();                                   36
    }                                                               37
```

A.14 Shuttle pokračovanie desiate

Na výpisu A.14 vidíme triedu Shuttle pokračovanie desiate v jazyku java.

Výpis A.14: Príklad výpisu triedy Shuttle pokračovanie desiate

```
                                                                    1
    // Finds node optimal path to destination                       2
job using BFS0, BSF or Dijkstra                                     3
    private void findPathToCurrentJob() {                           4
        // Clear old path                                           5
        curentPath.clear();                                         6
        // Check if there is any other job                          7
        if (jobs.isEmpty()) {                                       8
            return;                                                 9
        }                                                           10
        boolean anyJobEmpty = false;                                11
// Check if any of jobs is empty                                    12
        for (StockNode job : jobs) {                                13
            if (!job.occupied) {                                    14
                anyJobEmpty = true;                                 15
                break;                                              16
            }                                                       17
        }                                                           18
        if (!anyJobEmpty) {                                         19
            // If no job is empty, do not find next job             20
        }                                                           21
        else                                                        22
        {                                                           23
            long startTime = System.nanoTime();                     24
            if (method == Method.BFS0) {                            25
                curentPath = findBFS0();                            26
// Without finding nearest job                                      27
            } else if (method == Method.BFS) {                      28
                curentPath = findBFS();                             29
// With sorting nearest job                                         30
            } else if (method == Method.DIJKSTRA) {                 31
                curentPath = findDijkstra();                        32
// Optimal for best job search                                      33
            } else if (method == Method.A_STAR) {                   34
                curentPath = findAStar();                           35
// Find best job using A* algorithm                                 36
            }                                                       37
```

A.15 Shuttle pokračovanie jedenáste

Na výpisu A.15 vidíme triedu Shuttle pokračovanie jedenáste v jazyku java.

Výpis A.15: Príklad výpisu triedy Shuttle pokračovanie jedenáste

```
 methodTimesNs.add(System.nanoTime() - startTime);
// Save method time
        }



        // If path was not found, set shuttle to wait state
        if (curentPath.isEmpty()) {
            Date date = new Date();
            if (waitTime == 0) {
                System.out.println("Shuttle: " + index
+ ": Could not found new path, starting waiting");
                state = State.WAITING;
                waitTime = date.getTime();
            }
            else if (date.getTime() - waitTime > 4000) {
                System.out.println("Shuttle: " + index
+ ": Moving shuttle in random direction");
                waitTime = 0;
                // Move randomly into first unoccupied node
for (NodeConnection connection: currentNode.connections) {
        if (!connection.destinationNode.occupied) {
             curentPath.add(currentNode);
             curentPath.add(connection.destinationNode);
             curentPath.get(0).occupied = true;
             curentPath.get(1).occupied = true;
                    }
                }
            }
        } else {
            curentPath.get(0).occupied = true;

            if (curentPath.size() > 1)
                curentPath.get(1).occupied = true;
            waitTime = 0;
System.out.println("Shuttle: " + index + ": New path found");
        }
    }
```

A.16 Shuttle pokračovanie dvanáste

Na výpisu A.16 vidíme triedu Shuttle pokračovanie dvanáste v jazyku java.

Výpis A.16: Príklad výpisu triedy Shuttle pokračovanie dvanáste

```
  // Update shuttle state and position
    public void update() {
        if (!inStock())
            return; // If shuttle is not in stock, do nothing
        else {
       // If all jobs are done, set shuttle state to finished
       if (jobs.isEmpty() && curentPath.isEmpty()) {
           System.out.println("Shuttle:␣"
+ index + ":␣All␣jobs␣finished");
                currentNode.occupied = false;
                state = State.FINISHED; }
            // If shuttle arrived to destination job position
            else if (position.getX() ==
jobs.get(0).position.getX() && position.getY() ==
jobs.get(0).position.getY() && position.getZ() ==
jobs.get(0).position.getZ()) { // Start loading procedure
                Date date = new Date();
       if (state == State.MOVING || state == State.WAITING) {
                    // Start of loading process
                    arriveTime = date.getTime();
                    if (curentPath.size() > 1)
                        curentPath.get(0).occupied = false;
                    state = State.LOADING;
                }
                else if (date.getTime() - arriveTime
> jobs.get(0).loadingTimeMSec) {
                    currentNode = jobs.get(0);
                    jobs.remove(0);
 findPathToCurrentJob(); // Find path to next job
                   if (!curentPath.isEmpty())
                        state = State.MOVING;
// If path was found, set shuttle state to moving
                    else
                        state = State.WAITING;
// If path was not found, set shuttle state to waiting
                }
            }
```

A.17 Shuttle pokračovanie trináste

Na výpisu A.17 vidíme triedu Shuttle pokračovanie trináste v jazyku java.

Výpis A.17: Príklad výpisu triedy Shuttle pokračovanie trináste

```
// If current path is not set, find new one
        else if (!jobs.isEmpty() && curentPath.isEmpty()) {
                findPathToCurrentJob();
            } // If current path is finished, find new one
            else if (curentPath.size() == 1) {
                currentNode = curentPath.get(0);
                findPathToCurrentJob();
            }// If shuttle arrived to one of path nodes,
remove it from path and continue to next one
            else if (position.getX() ==
curentPath.get(1).position.getX() && position.getY() ==
curentPath.get(1).position.getY() && position.getZ() ==
curentPath.get(1).position.getZ()) {
                currentNode = curentPath.get(1);
                // If shuttle arrived into path node
                // Check if next node is empty
                if (curentPath.size() > 2 &&
curentPath.get(2).occupied) {
                    if (state != State.WAITING)
                        curentPath.get(0).occupied =
false; // Release occupation of previous node
                    state = State.WAITING;
                    Date date = new Date();
                    if (waitTime == 0) {
                        waitTime = date.getTime();
                    }
                    else if (date.getTime()
- waitTime > 4000) {
                        currentNode = curentPath.get(1);
                        findPathToCurrentJob();
                    }
                }
                else { // Release previous node only if
shuttle was not waiting and node was not already released
                  if (state != State.WAITING)
                      curentPath.get(0).occupied = false;
                    curentPath.remove(0);
```

A.18 Shuttle pokračovanie štrnáste

Na výpisu A.18 vidíme triedu Shuttle pokračovanie štrnáste v jazyku java.

Výpis A.18: Príklad výpisu triedy Shuttle pokračovanie štrnáste

```
curentPath.get(0).occupied = true;                                    1
                    if (curentPath.size() > 1)                        2
                        curentPath.get(1).occupied = true;            3
                    else { state = State.WAITING;                     4
                        currentNode = curentPath.get(0);              5
                        currentNode.occupied = true;                  6
                        curentPath.clear();                           7
                        findPathToCurrentJob();                       8
                    }                                                 9
                }                                                     10
            }                                                         11
        else { // Move shuttle towards the nearest path node          12
              double speed = currentConnectionSpeed();                13
              state = State.MOVING;                                   14
if (position.distance(curentPath.get(1).position) < 5) {              15
                    position.set(curentPath.get(1).position);         16
               } else {                                               17
Point3d direction = dir(position, curentPath.get(1).position);        18
                    // Move shuttle                                   19
                    position.set(position.getX() +                    20
(direction.getX() * speed), position.getY() +                         21
(direction.getY() * speed),position.getZ() +                          22
(direction.getZ() * speed));                                          23
                    Point3d newDirection =                            24
dir(position, curentPath.get(1).position) ;                           25
// If shuttle overshoot, set its position to desired position        26
                    if (sign(direction.getX()) !=                     27
                    sign(newDirection.getX()) ||                      28
sign(direction.getY()) != sign(newDirection.getY()) ||                29
sign(direction.getZ()) != sign(newDirection.getZ())) {                30
                    position.set(curentPath.get(1).position);         31
                    }                                                 32
                }                                                     33
            }                                                         34
        }                                                             35
    }                                                                 36
}                                                                     37
```

A.19 Shuttle3D

Na výpisu A.19 vidíme triedu Shuttle3D v jazyku java.

Výpis A.19: Príklad výpisu triedy Shuttle3D

```
import com.sun.j3d.utils.geometry.Sphere;
import javax.media.j3d.*;
import javax.vecmath.Color3f;
import javax.vecmath.Point3f;
import javax.vecmath.Vector3d;
import java.awt.*;
public class Shuttle3D extends Shuttle {
    public TransformGroup shuttleTG;
    private Sphere sphere;
    private Appearance sphereAppearance = new Appearance();
    private TransparencyAttributes transparencyAttributes =
new TransparencyAttributes();
    private Material sphereMaterial = new Material();
    public Shuttle3D() {
        sphereMaterial.setDiffuseColor
        (new Color3f(0.7f, 0.7f, 0.7f));
        sphereMaterial.setCapability
        (Material.ALLOW_COMPONENT_WRITE);
        sphereAppearance.setMaterial(sphereMaterial);
        // Create a TransparencyAttributes
object and set it to fully transparent
        transparencyAttributes.setTransparencyMode
        (TransparencyAttributes.FASTEST);
        transparencyAttributes.setTransparency(0.5f);
        transparencyAttributes.setCapability
        (TransparencyAttributes.ALLOW_VALUE_WRITE);
        sphereAppearance.setTransparencyAttributes
        (transparencyAttributes);
        // Shuttle transform
        Transform3D sphereTransform = new Transform3D();
        sphereTransform.setTranslation
        (new Vector3d(-100, 0, 0.0));
        // position the sphere at the
        shuttleTG = new TransformGroup(sphereTransform);
```

A.20 Shuttle3D pokračovanie prvé

Na výpisu A.20 vidíme triedu Shuttle3D pokračovanie prvé v jazyku java.

Výpis A.20: Príklad výpisu triedy Shuttle3D pokračovanie prvé

```
  // Enable object moving                                          1
        shuttleTG.setCapability                                    2
        (TransformGroup.ALLOW_TRANSFORM_WRITE);                    3
        shuttleTG.setCapability                                    4
        (TransformGroup.ALLOW_TRANSFORM_READ);                     5
                                                                   6
        // Create shuttle                                          7
        sphere = new Sphere                                        8
(10.f, Sphere.GENERATE_NORMALS, 32, sphereAppearance);             9
        sphere.setCapability                                       10
(Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);                  11
                                                                   12
        shuttleTG.addChild(sphere);                                13
                                                                   14
        // Create a new Font3D object                              15
        Font3D font3D = new Font3D                                 16
(new Font("Helvetica", Font.PLAIN, 20), new FontExtrusion());      17
                                                                   18
        // Create a new Text3D object                              19
        Text3D text3D = new Text3D                                 20
(font3D, Integer.toString(index),                                  21
new Point3f(-10.0f, -10.0f, 15.0f));                               22
                                                                   23
        // Create a new Shape3D object                             24
        Shape3D textShape = new Shape3D(text3D);                   25
                                                                   26
        shuttleTG.addChild(textShape);                             27
    }                                                              28
                                                                   29
    public void setColor(Color3f color) {                          30
        sphereMaterial.setDiffuseColor(color);                     31
    }                                                              32
    public void setVisible(boolean visibility) {                   33
        transparencyAttributes.setTransparency                     34
        (visibility ? 0.0f : 1.0f);                                35
    }                                                              36
}                                                                  37
```

A.21 Stock

Na výpisu A.21 vidíme Stock triedu v jazyku java.

Výpis A.21: Príklad výpisu triedy Stock

```
import javax.vecmath.Point3d;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
public class Stock {
//    ArrayList<StockNode> nodes =
new ArrayList<StockNode>();
// List of all nodes in stock
    ArrayList<StockNode3D> nodes =
    new ArrayList<StockNode3D>();
    // List of all nodes in stock
//    ArrayList<Shuttle> shuttles =
new ArrayList<>();
// List of all shuttles available
    ArrayList<Shuttle3D> shuttles =
    new ArrayList<>();
    // List of all shuttles available
    int maximalShuttlesInStock = 100;
// Maximal count of shuttles in stock at once
    // Add new node into stock graph
    void addNode(Point3d position,
    long loadingTimeMSec) {
//        StockNode node =
new StockNode(position, nodes.size());
        StockNode3D node =
new StockNode3D(position, nodes.size());
        node.loadingTimeMSec = loadingTimeMSec;
        nodes.add(node);
    }
    // Add connection between two nodes
    void connectNodes(int sourceNodeIndex,
int destinationNodeIndex, double maxSpeed,
boolean isBothDirectional) {
// Check if source index is valid
        if (sourceNodeIndex < 0 ||
        sourceNodeIndex >= nodes.size())
```

A.22 Stock pokračovanie prvé

Na výpisu A.22 vidíme Stock triedu pokračovanie prvé v jazyku java.

Výpis A.22: Príklad výpisu triedy Stock pokračovanie prvé

```
    {                                                                    1
            System.out.println                                           2
("First␣index␣of␣connection␣is␣out␣of␣range");                           3
            return;                                                      4
        }                                                                5
        // Check if destination index is valid                          6
        else if (destinationNodeIndex                                    7
< 0 || destinationNodeIndex >= nodes.size()) {                           8
            System.out.println                                           9
("Second␣index␣of␣connection␣is␣out␣of␣range");                          10
            return;                                                      11
        }                                                                12
        double distance = nodes.get(sourceNodeIndex)                     13
.position.distance(nodes.get(destinationNodeIndex).position);            14
        // Create connection from source to destination node             15
        NodeConnection con = new NodeConnection();                       16
        con.distance = distance;                                         17
        con.maxSpeed = maxSpeed;                                         18
        con.destinationNode = nodes.get(destinationNodeIndex);           19
        nodes.get(sourceNodeIndex).connections.add(con);                 20
                                                                         21
        // If connection is not oneway,                                  22
add connection from destination to source node                           23
   if (isBothDirectional) {                                              24
       NodeConnection dCon = new NodeConnection();                       25
       dCon.distance = distance;                                         26
       dCon.maxSpeed = maxSpeed;                                         27
       dCon.destinationNode = nodes.get(sourceNodeIndex);                28
       nodes.get(destinationNodeIndex).connections.add(dCon);            29
        }                                                                30
    }                                                                    31
    // Return true if all nodes are connected                           32
    public boolean allConnected() {                                      33
        Set<StockNode> visited = new HashSet<>();                        34
        Set<StockNode> newNodes = new HashSet<>();                       35
        newNodes.add(nodes.get(0));                                      36
```

A.23 Stock pokračovanie druhé

Na výpisu A.23 vidíme Stock triedu pokračovanie druhé v jazyku java.

Výpis A.23: Príklad výpisu triedy Stock pokračovanie druhé

```
  while (newNodes.size() != visited.size()) {
    Set<StockNode> newNodesOld = new HashSet<>(newNodes);
    for (StockNode node : newNodesOld) {
    for (NodeConnection con : node.connections) {
        if (!visited.contains(con.destinationNode)) {
newNodes.add(con.destinationNode);  // Add new node
                }
            }
            visited.add(node);
        }
    }
// If all nodes were visited, all nodes are connected
    return visited.size() == nodes.size();
  }
  // Check if all shuttles are have finished their job
  public boolean stockEmpty() {
    for (Shuttle shuttle: shuttles) {
        if (shuttle.state != Shuttle.State.FINISHED) {
            return false;
        }
    }
    return true;
  }
  // Returns current count of shuttles in stock
  private int shuttlesInStock() {
    int count = 0;
    for (Shuttle shuttle: shuttles) {
        if (shuttle.inStock()) {
            count++;
        }
    }
    return count;
  }
```

A.24 Stock pokračovanie tretie

Na výpisu A.24 vidíme triedu Stock pokračovanie tretie v jazyku java.

Výpis A.24: Príklad výpisu triedy Stock pokračovanie tretie

```
 // Returns count of available shuttles
    private int readyShuttles() {
        int count = 0;
        for (Shuttle shuttle: shuttles) {
            if (shuttle.state == Shuttle.State.READY) {
                count++;
            }
        }
        return count;
    }

    // Updates whole stock states
    public void update()
    {
        // Add shuttle into simulation
if needed and if start node is not occupied
        while (shuttlesInStock() < maximalShuttlesInStock
&& readyShuttles() > 0 && !nodes.get(0).occupied) {
            for (Shuttle shuttle: shuttles) {
                if (shuttle.state == Shuttle.State.READY) {
                    shuttle.setStartNode(nodes.get(0));
                    nodes.get(0).occupied = true;
                    shuttle.state = Shuttle.State.WAITING;
// Set shuttles state to waiting
                    break;
                }
            }
        }

        // Update states of all shuttles
        for (Shuttle shuttle: shuttles) {
            shuttle.update();
        }
    }
}
```

A.25 StockNode

Na výpisu A.25 vidíme triedu StockNode v jazyku java.

Výpis A.25: Príklad výpisu triedy StockNode

```
import javax.vecmath.Point3d;
import java.util.ArrayList;

public class StockNode {
    Point3d position;   // Metric position of node
    double loadingTimeMSec = 1000;
    // Time of loading material
    ArrayList<NodeConnection> connections =
    new ArrayList<NodeConnection>();
    static int counter = 0;
    // Counter of all created stock nodes
    int index;  // Index of current node
    boolean occupied = false;
// Flag if node is currently occupied by a shuttle
    public StockNode(Point3d nodePosition, int nodeIndex) {
        position = nodePosition;
        index = counter++;
    }

    // Returns distance of two nodes
    public double nodeDistance(StockNode node){
        return position.distance(node.position);
    }

    // Returns node with the lowest cost to current node
    public StockNode getLowestCostConnection() {
        double lowestCost = Double.MAX_VALUE;
        StockNode best = null;
        for (NodeConnection node : connections) {
            if (node.cost() < lowestCost) {
                lowestCost = node.cost();
                best = node.destinationNode;
            }
        }
        return best;
    }
}
```

A.26 StockNode3D

Na výpisu A.26 vidíme triedu StockNode3D v jazyku java.

Výpis A.26: Príklad výpisu triedy StockNode3D

```
import com.sun.j3d.utils.geometry.Box;
import javax.media.j3d.*;
import javax.vecmath.Color3f;
import javax.vecmath.Point3d;
import javax.vecmath.Point3f;
import javax.vecmath.Vector3d;
import java.awt.*;
public class StockNode3D extends StockNode{
    public TransformGroup nodeTG;
    LineAttributes lineAttributes = new LineAttributes();
    public StockNode3D(Point3d nodePosition, int nodeIndex) {
      super(nodePosition, nodeIndex);
      // Node transform
      Transform3D nodeTransform = new Transform3D();
      nodeTransform.setTranslation(new Vector3d
(position.getX(), position.getY(),
position.getZ())); // position the sphere at the
     nodeTG = new TransformGroup(nodeTransform);
     // create a new polygon attributes
object and set the polygon mode to LINE
     PolygonAttributes polyAttribs = new PolygonAttributes();
     polyAttribs.setPolygonMode
    (PolygonAttributes.POLYGON_LINE);
     // create a new appearance object
    and set the polygon attributes
    Appearance polyAppearance = new Appearance();
    polyAppearance.setPolygonAttributes(polyAttribs);
    ColoringAttributes colorAttrib = new ColoringAttributes
(new Color3f(0.0f, 0.0f, 0.0f),
ColoringAttributes.SHADE_FLAT);
     polyAppearance.setColoringAttributes(colorAttrib);
     // Create the line attributes object
     lineAttributes.setLineWidth(1.0f);
     lineAttributes.setLineAntialiasingEnable(true);
     lineAttributes.setCapability
     (LineAttributes.ALLOW_WIDTH_WRITE);
```

A.27 StockNode3D pokračovanie prvé

Na výpisu A.27 vidíme triedu StockNode3D pokračovanie prvé v jazyku java.

Výpis A.27: Príklad výpisu triedy StockNode3D pokračovanie prvé

```
 polyAppearance.setLineAttributes(lineAttributes);

        // Create node box
        Box box = new Box(10.f, 10.f, 10.f,
Box.GENERATE_NORMALS | Box.GENERATE_TEXTURE_COORDS,
polyAppearance);
        nodeTG.addChild(box);

        // Create a new Font3D object
        Font3D font3D = new Font3D(new Font
("Helvetica", Font.PLAIN, 7), new FontExtrusion());

        // Create a new Text3D object
        Text3D text3D = new Text3D(font3D,
Integer.toString(index), new Point3f(-10.0f, -10.0f, 10.0f));

        // Create a new Shape3D object
        Shape3D textShape = new Shape3D(text3D);

        nodeTG.addChild(textShape);
    }

    public void updateRender() {
        lineAttributes.setLineWidth(occupied ? 3.0f : 1.0f);
    }
}
```

A.28 StockSimulation

Na výpisu A.28 vidíme triedu StockSimulation v jazyku java.

Výpis A.28: Príklad výpisu triedy StockSimulation

```
import javax.swing.*;
import javax.vecmath.Point3d;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.AffineTransform;
import java.awt.geom.Point2D;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

import static java.lang.Math.round;

public class StockSimulation extends JPanel {
  Stock stock = new Stock();
  static long randomSeed; // Random seed for stock generation
  long simulationStart;   // Time of start of simulation
  Timer updateTimer;
  public StockSimulation() {
    System.out.println("Stock simulation started");
      // Create stock
       setPreferredSize(new Dimension(900,900));

       Random random = new Random();
       random.setSeed(randomSeed);

      // Generate stock graph
       int gridSize = 8;
      int nodeDistance = 100;
      for (int i = 0; i < gridSize; i++) {
          for (int j = 0; j < gridSize; j++) {
             stock.addNode(new Point3d(50 +
(nodeDistance * j), 50 + (nodeDistance * i),
 0), round(random.nextDouble() * 2000) + 500);
           }
       }
```

A.29 StockSimulation pokračovanie prvé

Na výpisu A.29 vidíme triedu StockSimulation pokračovanie prvé v jazyku java.

Výpis A.29: Príklad výpisu triedy StockSimulation pokračovanie prvé

```
   for (int i = 0; i < gridSize; i++) {
            for (int j = 0; j < gridSize; j++) {
                if (random.nextDouble() > 0.5) {
                    if (i < gridSize - 1) {
                        stock.connectNodes(i + (gridSize * j)
,i + (gridSize * j) + 1,(random.nextDouble() * 10) + 1,true);
                    }
                    if (j < gridSize - 1) {
                     stock.connectNodes(i + (gridSize * j),i +
(gridSize * j) + gridSize,(random.nextDouble() * 10) + 1,true);
                    }
                }
            }
        }
        // Generate shuttles
        int numberOfShuttles = 10;
        for (int i = 0; i < numberOfShuttles; i++) {
            stock.shuttles.add(new Shuttle3D());
        }
        // Set random jobs to shuttles
        int jobNumber = 5;
        for (int i = 0; i < stock.shuttles.size(); i++) {
            for (int j = 0; j < jobNumber; j++) {
                stock.shuttles.get(i).jobs.add(stock
.nodes.get((int)round(random.nextDouble() *
(stock.nodes.size()-1))));
            }
        } // Timer for updating of visualisation
        updateTimer = new Timer(10, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent arg0) {
                update();
                if (stock.stockEmpty()) {
                  long totalTime = System.currentTimeMillis()
                  - simulationStart;
                  System.out.println("Simulation␣finished␣in␣"
                  + totalTime + "␣milliseconds");
```

A.30 StockSimulation pokračovanie druhé

Na výpisu A.30 vidíme triedu StockSimulation pokračovanie druhé v jazyku java.

Výpis A.30: Príklad výpisu triedy StockSimulation pokračovanie druhé

```
File file = new File("simulation.txt");
  try {
   FileWriter writer = new FileWriter(file, true);
               writer.write("Grid: ");
               writer.write(Integer.toString(gridSize));
               writer.write("x");
               writer.write(Integer.toString(gridSize));
               writer.write("\tSeed: ");
               writer.write(Long.toString(randomSeed));
               writer.write("\tMode: ");
               writer.write(Shuttle.method.name());
               writer.write("\tShuttles: ");
               writer.write(Long.toString(numberOfShuttles));
               writer.write("\tJobs: ");
               writer.write(Long.toString(jobNumber));
               writer.write("\tMilliseconds: ");
               writer.write(Long.toString(totalTime));

               writer.write("\n");
               writer.close();
       System.out.println("Simulation data written to file.");
                     } catch (IOException e) {
                         System.out.println
("An error occurred during writing to file.");
                         e.printStackTrace();
                     }

                     stopApp();
                 }
             }
         });
         updateTimer.start();
         simulationStart = System.currentTimeMillis();
    }
```

A.31 StockSimulation pokračovanie tretie

Na výpisu A.31 vidíme triedu StockSimulation pokračovanie tretie v jazyku java.

Výpis A.31: Príklad výpisu triedy StockSimulation pokračovanie tretie

```
    private void stopApp() {
        updateTimer.stop();
        if (SwingUtilities.getWindowAncestor(this) != null)
            SwingUtilities.getWindowAncestor(this)
            .setVisible(false);
        else
            System.out.println("Not␣valid␣GUI␣pointer");
    }
    private void update()
    {
        // Update stock states
        stock.update();
        // Repaint stock scene
        repaint();
    } //Source: https://stackoverflow.com/a/4112875/8142658
void drawArrow(Graphics g1, int x1, int y1, int x2, int y2) {
        Graphics2D g = (Graphics2D) g1.create();
        int ARR_SIZE = 10;
        double dx = x2 - x1, dy = y2 - y1;
        double angle = Math.atan2(dy, dx);
        int len = (int) Math.sqrt(dx*dx + dy*dy);
        AffineTransform at =
        AffineTransform.getTranslateInstance(x1, y1);
        at.concatenate
        (AffineTransform.getRotateInstance(angle));
        g.transform(at);
        // Draw horizontal arrow starting in (0, 0)
        g.drawLine(0, 0, len, 0);
        g.fillPolygon(new int[]
        {len, len-ARR_SIZE, len-ARR_SIZE, len},
                new int[] {0, -ARR_SIZE, ARR_SIZE, 0}, 4);
    }
    private static Point2D sum(Point3d p1, Point2D p2) {
        return new Point2D.Double(p1.getX() +
        p2.getX(), p1.getY() + p2.getY());
    }
```

A.32 StockSimulation pokračovanie štvrté

Na výpisu A.32 vidíme triedu StockSimulation pokračovanie štvrté v jazyku java.

Výpis A.32: Príklad výpisu triedy StockSimulation pokračovanie štvrté

```
     private static Point2D multiply                                    1
    (Point2D p1, double scale) {                                        2
        return new Point2D.Double(p1.getX()                             3
        * scale, p1.getY() * scale);                                    4
    }                                                                   5
    private static Point2D lineParametrisation                          6
    (Point3d p1, Point3d p2, double t) {                                7
        Point2D sum = sum(p2, new Point2D.Double                        8
        (-p1.getX(), -p1.getY()));                                      9
        Point2D mul =  multiply(sum, t);                                10
        return sum(p1, mul);                                            11
    }                                                                   12
    @Override                                                           13
    public void paintComponent(Graphics g) {                            14
        super.paintComponent(g);                                        15
        int nodeSize = 60;                                              16
        int shuttleSize = 50;                                           17
        // Draw nodes connections                                       18
        for (StockNode node:stock.nodes) {                              19
            for (NodeConnection                                         20
            connection:node.connections) {                              21
                double distance = node.position.distance                22
                (connection.destinationNode.position);                  23
                double tStart = (nodeSize / 2) / distance;              24
                double tEnd = 1 - tStart;                               25
                Point2D start = lineParametrisation                     26
(node.position, connection.destinationNode.position,tStart);            27
                Point2D end = lineParametrisation                       28
(node.position, connection.destinationNode.position,tEnd);              29
                drawArrow(g, (int)round(start.getX()),                  30
(int)round(start.getY()), (int)round(end.getX()),                       31
(int)round(end.getY()));                                                32
            }                                                           33
        } // Draw stock positions                                       34
        for (StockNode node:stock.nodes) {                              35
            if (!node.occupied) {                                       36
                g.setColor(new Color(255,255,255));}                    37
```

A.33 StockSimulation pokračovanie piate

Na výpisu **??** vidíme triedu StockSimulation pokračovanie piate v jazyku java.

Výpis A.33: Príklad výpisu triedy StockSimulation pokračovanie piaté

```
else {                                                                  1
                g.setColor(new Color(150,150,150));                     2
            }                                                           3
g.fillOval((int)round(node.position.getX() -                            4
(nodeSize / 2)),(int)round(node.position.getY() -                       5
(nodeSize / 2)), nodeSize, nodeSize);                                   6
            g.setColor(new Color(0,0,0));                               7
            g.drawOval((int)round(node.position.getX()                  8
- (nodeSize / 2)),(int)round(node.position.getY() -                     9
(nodeSize / 2)), nodeSize, nodeSize);                                   10
            g.drawString(Integer.toString(node.index),                  11
(int)round(node.position.getX() + (nodeSize / 2)),                      12
(int)round(node.position.getY() + (nodeSize / 2)));                     13
        }                                                               14
                                                                        15
        // Draw shuttles                                                16
        for (Shuttle shuttle:stock.shuttles) {                          17
            if (shuttle.inStock()) {                                    18
                g.setColor(new Color(0,255,0));                         19
                int angle = (int)Math.round                             20
((shuttle.jobCount - (shuttle.jobs.size() - 1))                         21
/ (double)shuttle.jobCount * 360.0);                                    22
                g.fillArc((int)round                                    23
                (shuttle.position.getX() -                              24
(shuttleSize / 2)) - 2, (int)round                                      25
(shuttle.position.getY() - (shuttleSize / 2)) -                         26
2, shuttleSize + 4, shuttleSize + 4,   90, -angle);                     27
                                                                        28
                g.setColor(new Color(0,0,0));                           29
                g.fillOval((int)round                                   30
(shuttle.position.getX() - (shuttleSize / 2)),                          31
(int)round(shuttle.position.getY() -                                    32
(shuttleSize / 2)), shuttleSize, shuttleSize);                          33
                                                                        34
                g.setColor(new Color(255,255,255));                     35
                g.drawString(Integer.toString                           36
```

A.34 StockSimulation pokračovanie šieste

Na výpisu A.34 vidíme triedu StockSimulation pokračovanie šieste v jazyku java.

Výpis A.34: Príklad výpisu triedy StockSimulation pokračovanie šieste

```
 (shuttle.index), (int)round(shuttle.position.getX()
- 3), (int)round(shuttle.position.getY() - 10));
                if (!shuttle.jobs.isEmpty())
                    g.drawString("Pos:" +
Integer.toString(shuttle.jobs.get(0).index),
(int)round(shuttle.position.getX() - 12),
(int)round(shuttle.position.getY() + 15));
                String state = "";
                if (shuttle.state == Shuttle.State.LOADING) {
                    g.setColor(new Color(0,255,0));
                    state = "LOADING";
                }
            else if (shuttle.state == Shuttle.State.WAITING) {
                    g.setColor(new Color(255,0,0));
                    state = "WAITING";
                }
            else if (shuttle.state == Shuttle.State.READY)
                    state = "READY";
            else if (shuttle.state == Shuttle.State.FINISHED)
                    state = "FINISHED";
              else if (shuttle.state == Shuttle.State.MOVING)
                    state = "MOVING";
                g.drawString(state,
(int)round(shuttle.position.getX() - 24),
(int)round(shuttle.position.getY() + 4));
             }
        }
    }
}
```

A.35 StockSimulation3D

Na výpisu A.35 vidíme triedu StockSimulation3D v jazyku java.

Výpis A.35: Príklad výpisu triedy StockSimulation3D

```
import com.jogamp.nativewindow.util.Point;
import com.sun.j3d.utils.universe.SimpleUniverse;
import javax.media.j3d.*;
import javax.swing.*;
import javax.vecmath.*;
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.Random;

import static java.lang.Math.round;

public class StockSimulation3D extends Canvas3D
implements ActionListener, KeyListener,
MouseListener, MouseMotionListener {

 Stock stock = new Stock();
 static long randomSeed; // Random seed for stock generation
 private long simulationStart;
// Time of start of simulation
   Timer updateTimer;
    private static final long serialVersionUID = 1L;
    private SimpleUniverse universe;
//    private Canvas3D canvas;
    private TransformGroup viewTG;
    private Transform3D viewTransform = new Transform3D();
    private Vector3f cameraTranslation;
    private Vector2f cameraRotation;
    private Point lastMousePos = new Point(-1, -1);
    private long startTime;
    private JLabel timeLabel = new JLabel("0 s");
    private int gridSize = 8;
    private HashMap<Shuttle, TransformGroup> shuttleTexts =
    new HashMap<>();
```

A.36 StockSimulation3D pokračovanie prvé

Na výpisu A.36 vidíme triedu StockSimulation3D pokračovanie prvé v jazyku java.

Výpis A.36: Príklad výpisu triedy StockSimulation3D pokračovanie prvé

```
 public StockSimulation3D() {
super(SimpleUniverse.getPreferredConfiguration());
        System.out.println("Stock␣simulation␣started");
        // Create stock
        setPreferredSize(new Dimension(1500,900));
        Random random = new Random();
        random.setSeed(randomSeed);
        // Generate stock graph
        int gridSize = 10;
        int gridLayers = 5;
        int nodeDistance = 100;
        for (int k = 0; k < gridLayers; k++) {
            for (int j = 0; j < gridSize; j++) {
                for (int i = 0; i < gridSize; i++) {
                    stock.addNode(new Point3d(nodeDistance *
i, nodeDistance * j, -nodeDistance * k),
round(random.nextDouble() * 2000) + 500);
                }  }
        } for (int k = 0; k < gridLayers; k++) {
            for (int j = 0; j < gridSize; j++) {
                for (int i = 0; i < gridSize; i++) {
                    if (random.nextDouble() <= 0.9)
                    { if (i < gridSize - 1) {
                            stock.connectNodes(i +
(gridSize * j) + (gridSize * gridSize * k),i +
(gridSize * j) + (gridSize * gridSize * k) + 1,
(random.nextDouble() * 10) + 1,true);
                        } if (j < gridSize - 1) {
                            stock.connectNodes(i +
(gridSize * j) + (gridSize * gridSize * k),i +
(gridSize * j) + (gridSize * gridSize * k) +
gridSize,(random.nextDouble() * 10) + 1,true);
                        } if (k < gridLayers - 1) {
stock.connectNodes(i + (gridSize * j) +
(gridSize * gridSize * k),i + (gridSize * j) +
(gridSize * gridSize * k) + (gridSize * gridSize),
(random.nextDouble() * 10) + 1,true);
```

A.37 StockSimulation3D pokračovanie druhé

Na výpisu A.37 vidíme triedu StockSimulation3D pokračovanie druhé v jazyku java.

Výpis A.37: Príklad výpisu triedy StockSimulation3D pokračovanie druhé

```
   }                                                                    1
                    }                                                   2
                }                                                       3
                                                                        4
            }                                                           5
        }                                                               6
        // Generate shuttles                                            7
        int numberOfShuttles = 10;                                      8
        for (int i = 0; i < numberOfShuttles; i++) {                    9
            stock.shuttles.add(new Shuttle3D());                        10
        }                                                               11
                                                                        12
        // Set random jobs to shuttles                                  13
        int jobNumber = 20;                                             14
        for (int i = 0; i < stock.shuttles.size(); i++) {               15
            for (int j = 0; j < jobNumber; j++) {                       16
                stock.shuttles.get(i).jobs.                             17
add(stock.nodes.get((int)round(random.nextDouble() *                    18
(stock.nodes.size()-1))));                                              19
            }                                                           20
        }                                                               21
        addMouseListener(this);                                         22
        addMouseMotionListener(this);                                   23
        addKeyListener(this);                                           24
        universe = new SimpleUniverse(this);                           25
        BranchGroup scene = createScene();                              26
        universe.getViewingPlatform()                                   27
.setNominalViewingTransform();                                          28
     //Transform3D viewTransform = new Transform3D();                   29
     viewTransform.lookAt(new Point3d(5, 5, 20),                        30
new Point3d(5, 5, 0.0), new Vector3d(0.0, 1.0, 0.0));                   31
     viewTransform.invert();                                            32
     viewTG = universe.getViewingPlatform()                             33
    .getViewPlatformTransform();                                        34
     viewTG.setTransform(viewTransform);                                35
      getView().setBackClipDistance(10000);                             36
```

A.38 StockSimulation3D pokračovanie tretie

Na výpisu A.38 vidíme triedu StockSimulation3D pokračovanie tretie v jazyku java.

Výpis A.38: Príklad výpisu triedy StockSimulation3D pokračovanie tretie

```
  cameraTranslation =  new Vector3f                                    1
  (gridSize / 2 * 100, gridSize / 2 * 100, 1000);                      2
        cameraRotation =  new Vector2f(0, 0);                          3
        universe.addBranchGraph(scene);                                4
        startTime = System.currentTimeMillis();                        5
        new Timer(100, this).start();                                  6
        // Timer for updating of visualisation                         7
        updateTimer = new Timer(10, new ActionListener() {             8
            @Override                                                  9
            public void actionPerformed(ActionEvent arg0) {            10
                stock.update();                                        11
                updateView();                                          12
                if (stock.stockEmpty()) {                              13
                    long totalTime =                                   14
System.currentTimeMillis() - simulationStart;                          15
                    System.out.println                                 16
("Simulation␣finished␣in␣" + totalTime + "␣milliseconds");             17
                    File file = new File("simulation.txt");            18
  try {                                                                19
     FileWriter writer = new FileWriter(file, true);                   20
                writer.write("Grid:␣");                                21
                writer.write(Integer.toString(gridSize));              22
                writer.write("x");                                     23
                writer.write(Integer.toString(gridSize));              24
                writer.write("x");                                     25
                writer.write(Integer.toString(gridLayers));            26
                writer.write("\tSeed:␣");                              27
                writer.write(Long.toString(randomSeed));               28
                writer.write("\tMode:␣");                              29
                writer.write(Shuttle.method.name());                   30
                writer.write("\tShuttles:␣");                          31
                writer.write(Long.toString(numberOfShuttles));         32
                writer.write("\tJobs:␣");                              33
```

A.39 StockSimulation3D pokračovanie štvrté

Na výpisu A.39 vidíme triedu StockSimulation3D pokračovanie štvrté v jazyku java.

Výpis A.39: Príklad výpisu triedy StockSimulation3D pokračovanie štvrté

```
     writer.write(Long.toString(jobNumber));
     writer.write("\tMilliseconds: ");
     writer.write(Long.toString(totalTime));

    // Calculate average of the method times
              long sum = 0;
         for (long time : Shuttle.methodTimesNs) {
              sum += time;
               }
          double average = (double) sum /
          Shuttle.methodTimesNs.size();
          writer.write("\tMethod milliseconds: ");
          writer.write(Double.toString(average / 1e6));

          writer.write("\n");
          writer.close();
 System.out.println("Simulation data written to file.");
             } catch (IOException e) {
System.out.println("An error occurred
during writing to file.");
                   e.printStackTrace();
                    }

                    stopApp();
                }
            }
        });
        updateTimer.start();
        simulationStart = System.currentTimeMillis();
    }

    private BranchGroup createScene() {
        BranchGroup scene = new BranchGroup();
```

A.40 StockSimulation3D pokračovanie piate

Na výpisu A.40 vidíme triedu StockSimulation3D pokračovanie piate v jazyku java.

Výpis A.40: Príklad výpisu triedy StockSimulation3D pokračovanie piate

```
// Background color                                                    1
        Background background = new Background                         2
        (new Color3f(1.0f,1.0f,1.0f));                                 3
        background.setApplicationBounds                                4
        (new BoundingSphere(new Point3d(0,0,0), 10000));               5
        scene.addChild(background);                                    6
                                                                       7
        // Add ambient light                                           8
        Color3f ambientColor =                                         9
        new Color3f(1.0f, 1.0f, 1.0f);                                 10
        AmbientLight ambientLight =                                    11
        new AmbientLight(ambientColor);                                12
        ambientLight.setInfluencingBounds                              13
        (new BoundingSphere(new Point3d(0,0,0), 10000));               14
        scene.addChild(ambientLight);                                  15
                                                                       16
        // Add directional light                                       17
        Color3f lightColor =                                           18
        new Color3f(0.3f, 0.3f, 0.3f);                                 19
        Vector3f lightDirection =                                      20
        new Vector3f(-1.0f, -1.0f, -1.0f);                             21
        DirectionalLight light =                                       22
        new DirectionalLight(lightColor, lightDirection);              23
        light.setInfluencingBounds                                     24
        (new BoundingSphere(new Point3d(0,0,0), 10000));               25
        scene.addChild(light);                                         26
        // Add stock nodes                                             27
        for (StockNode3D node: stock.nodes) {                          28
            scene.addChild(node.nodeTG);                               29
        }                                                              30
        for (Shuttle3D shuttle : stock.shuttles) {                     31
            // Add shuttle to scene                                    32
            scene.addChild(shuttle.shuttleTG);                         33
        }                                                              34
        for (StockNode3D node: stock.nodes) {                          35
            // Add node connections                                    36
           for (NodeConnection con : node.connections) {               37
```

A.41 StockSimulation3D pokračovanie šieste

Na výpisu A.41 vidíme triedu StockSimulation3D pokračovanie šieste v jazyku java.

Výpis A.41: Príklad výpisu triedy StockSimulation3D pokračovanie šieste

```
// Create the line array                                          1
                LineArray lineArray =                             2
                new LineArray(2, LineArray.COORDINATES);          3
                lineArray.setCoordinate                           4
(0, new Point3f(node.position));                                  5
                lineArray.setCoordinate                           6
(1, new Point3f(con.destinationNode.position));                   7
                Appearance appearance = new Appearance();         8
                Shape3D shape3D =                                 9
                new Shape3D(lineArray, appearance);               10
                // Combine line and arrowhead                     11
                TransformGroup arrowTG = new TransformGroup();    12
                arrowTG.addChild(shape3D);                        13
                scene.addChild(arrowTG);                          14
            }                                                     15
        }                                                         16
                                                                  17
        return scene;                                             18
    }                                                             19
    private void stopApp() {                                      20
        System.out.println("Closing simulation window.");         21
        updateTimer.stop();                                       22
        if (SwingUtilities.getWindowAncestor(this) != null)       23
            SwingUtilities.getWindowAncestor(this)                24
            .setVisible(false);                                   25
        else                                                      26
            System.out.println("Not valid GUI pointer");          27
        System.out.println("Simulation window closed.");          28
    }                                                             29
    private void updateView() {                                   30
        for (Shuttle3D shuttle : stock.shuttles) {                31
            shuttle.setVisible(shuttle.inStock());                32
            if (shuttle.state == Shuttle.State.LOADING) {         33
                shuttle.setColor(new Color3f                      34
                (0.0f, 1.0f, 0.0f));                              35
            }                                                     36
```

A.42 StockSimulation3D pokračovanie siedme

Na výpisu A.42 vidíme triedu StockSimulation3D pokračovanie siedme v jazyku java.

Výpis A.42: Príklad výpisu triedy StockSimulation3D pokračovanie siedme

```
else if (shuttle.state == Shuttle.State.WAITING) {
             shuttle.setColor(new Color3f(1.0f, 0.0f, 0.0f));
            }
            else {
             shuttle.setColor(new Color3f(0.7f, 0.7f, 0.7f));
            }

            Transform3D tf = new Transform3D();
            tf.setTranslation(new Vector3d
            (shuttle.position.getX(),
shuttle.position.getY(), shuttle.position.getZ()));
// position the sphere at the
            shuttle.shuttleTG.setTransform(tf);
        }

        for (StockNode3D node : stock.nodes) {
            node.updateRender();
        }

        viewTransform = new Transform3D();
        viewTransform.setTranslation(cameraTranslation);

        Transform3D rotX = new Transform3D();
        rotX.rotX(cameraRotation.getX());
        Transform3D rotY = new Transform3D();
        rotY.rotY(cameraRotation.getY());

        viewTransform.mul(rotX);
        viewTransform.mul(rotY);

        viewTG.setTransform(viewTransform);
    }
```

A.43 StockSimulation3D pokračovanie ôsme

Na výpisu A.43 vidíme triedu StockSimulation3D pokračovanie ôsme v jazyku java.

Výpis A.43: Príklad výpisu triedy StockSimulation3D pokračovanie ôsme

```
public void actionPerformed(ActionEvent e) {                          1
  long elapsedTime = System.currentTimeMillis() - startTime;          2
  timeLabel.setText(elapsedTime / 1000 + "␣s");                       3
    }                                                                 4
                                                                      5
    @Override                                                         6
public void keyTyped(KeyEvent e) {                                    7
                                                                      8
    }                                                                 9
                                                                      10
public void keyPressed(KeyEvent e) {                                  11
  int key = e.getKeyCode();                                           12
  switch (key) {                                                      13
    case KeyEvent.VK_LEFT:                                            14
      cameraTranslation.setX(cameraTranslation.getX() - 50.f);        15
      updateView();                                                   16
      break;                                                          17
    case KeyEvent.VK_RIGHT:                                           18
      cameraTranslation.setX(cameraTranslation.getX() + 50.f);        19
      updateView();                                                   20
      break;                                                          21
    case KeyEvent.VK_UP:                                              22
      cameraTranslation.setZ(cameraTranslation.getZ() - 50.f);        23
      updateView();                                                   24
      break;                                                          25
    case KeyEvent.VK_DOWN:                                            26
      cameraTranslation.setZ(cameraTranslation.getZ() + 50.f);        27
      updateView();                                                   28
      break;                                                          29
        }                                                             30
    }                                                                 31
    @Override                                                         32
    public void keyReleased(KeyEvent e) {                             33
```

A.44 StockSimulation3D pokračovanie deviate

Na výpisu A.44 vidíme triedu StockSimulation3D pokračovanie deviate v jazyku java.

Výpis A.44: Príklad výpisu triedy StockSimulation3D pokračovanie deviate

```
  }
public void mousePressed(MouseEvent e) {
        lastMousePos.setX(e.getX());
        lastMousePos.setY(e.getY());
    }
public void mouseDragged(MouseEvent e) {
 if (e.getModifiersEx() == MouseEvent.BUTTON1_DOWN_MASK){
 if (lastMousePos.getX() != -1 &&
 lastMousePos.getY() != -1) {
    int dx = e.getX() - lastMousePos.getX();
    int dy = e.getY() - lastMousePos.getY();
cameraRotation.setX(cameraRotation.getX() + (dy / 1000.0f));
cameraRotation.setY(cameraRotation.getY() + (dx / 1000.0f));
                updateView();
            }
        }
 if (e.getModifiersEx() == MouseEvent.BUTTON3_DOWN_MASK) {
     if (lastMousePos.getX() != -1 &&
    lastMousePos.getY() != -1) {
         int dx = e.getX() - lastMousePos.getX();
         int dy = e.getY() - lastMousePos.getY();
      cameraTranslation.setX(cameraTranslation.getX() - dx);
      cameraTranslation.setY(cameraTranslation.getY() + dy);
      updateView();
            }
        }
        lastMousePos.setX(e.getX());
        lastMousePos.setY(e.getY());
    }

    public void mouseReleased(MouseEvent e) {
        lastMousePos.setX(-1);
        lastMousePos.setY(-1);
```

A.45 StockSimulation3D pokračovanie desiate

Na výpisu A.45 vidíme triedu StockSimulation3D pokračovanie desiate v jazyku java.

Výpis A.45: Príklad výpisu triedy StockSimulation3D pokračovanie desiate

```
  }

    public void mouseMoved(MouseEvent e) {
    }

    public void mouseClicked(MouseEvent e) {
    }

    public void mouseEntered(MouseEvent e) {
    }

    public void mouseExited(MouseEvent e) {
    }
}
```

B Obsah elektronickej prílohy

Obsahom tejto elektronickej prílohy je kód pre simmuláciu skladu v jave.

- / ..... koreňový adresár priloženého archívu java
  - Shuttle System ..... simmuláciu skladu.zip
    - .idea.ẋml
      - libraries
      - gitignoreġitignore
      - miscẋml
      - modulesẋml
      - workspaceẋml
    - .libs
    - .out
      - production
        - StockSim
          - Main.class
          - NodeConnection.class
          - ShuttleDijkstraPair.class
          - ShuttleMethod.class
          - ShuttleState.class
          - Shuttle.class
          - Shuttle3D.class
          - Stock.class
          - StockNode.class
          - StockNode3D.class
          - StockSimulation1.class
          - StockSimulation.class
          - StockSimulation3D1.class
          - StockSimulation3D.class
    - .src
      - Main.java ..... Trieda zabezpečuje našu simuláciu
      - NodeConnection.java ..... Trieda programu
      - Shuttle.java ..... Trieda programu
      - Shuttle3D.java ..... Trieda programu
      - Stock.java ..... Trieda programu
      - StockNode.java ..... Trieda programu
      - StockNode3D.java ..... Trieda programu
      - StockSimulation.java ..... Trieda a hlavná logika programu pre 2D
      - StockSimulation3D.java ..... Trieda a hlavná logika programu pre 3D
    - .StockSimiml