VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2016 Bc. Martin Rajnoha

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

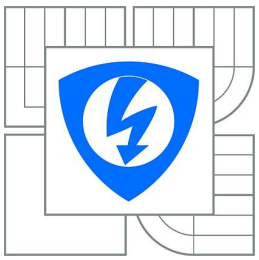
MODELOVÁNÍ SKLADŮ POMOCÍ GRAFICKÉHO ROZHRANÍ
WAREHOUSE MODELING USING GRAPHICAL USER INTERFACE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Martin Rajnoha**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Radim Burget, Ph.D.**
SUPERVISOR

BRNO 2016



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student:	Bc. Martin Rajnoha	***ID:***	136579
Ročník:	2	***Akademický rok:***	2015/2016

NÁZEV TÉMATU:

Modelování skladů pomocí grafického rozhraní

POKYNY PRO VYPRACOVÁNÍ:

Navrhněte algoritmus, který umožní rychlé vyhledávání délky cesty ve skladu. Aplikaci navrhněte tak, aby bylo možné podobu skladu nakreslit, následně převést na grafovou podobu a poté demonstrujte funkčnost na vybraných příkladech.

DOPORUČENÁ LITERATURA:

[1] PILLAC, V. "A review of dynamic vehicle routing problems." European Journal of Operational Research 225.1 (2013): 1-11.
[2] BEAMER, S., ASANOVIC, K., PATTERNSON, D. Direction-optimizing breadth-first search. Scientific Programming 21.3-4 (2013): 137-148.

Termín zadání:	1.2.2016	***Termín odevzdání:***	25.5.2016

Vedoucí práce: doc. Ing. Radim Burget, Ph.D.
Konzultanti diplomové práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Diplomová práca navrhuje nový algoritmus, ktorý umožňuje efektívnu konverziu grafickej reprezentácie skladu do grafovej reprezentácie a následne urýchľuje odhad ceny ciest. Navrhnutý algoritmus počíta vzdialenosti medzi ľubovoľnými miestami skladu na základe algoritmu BFS, „skeletonizácie“ z obrazového spracovania dát a Dijkstrovho algoritmu. Použitím navrhnutého algoritmu je možné vyhľadávať cesty v sklade efektívne a rýchlo pomocou predpočítanej smerovacej tabuľky. Čas vyhľadávania je menší ako milisekunda s použitím smerovacej tabuľky a nie je výrazne ovplyvnený veľkosťou skladu namiesto použitia Dijkstrovho algoritmu.

KĽÚČOVÉ SLOVÁ

sklad, konverzia, GUI, skeletonizácia, graf, vyhľadávanie, cesty, BFS, Dijkstrov algoritmus, smerovacia tabuľka

ABSTRACT

Master's thesis proposes a new algorithm which enables efficient conversion of graphical representation of warehouse into graph theory representation and consequently accelerates estimation for route costs. The proposed algorithm computes route distances between any places in warehouse based on Breadth first search, image processing „skeletonization“ and Dijkstra algorithm. Using the proposed algorithm it is possible to search routes in a warehouse effectively and fast using precomputed routing table. Searching time is less then milisecond using routing table and even size of warehouse doesn't affect it significantly instead of using Dijkstra algorithm.

KEYWORDS

warehouse, conversion, GUI, skeletonization, graph, searching, paths, BFS, Dijkstra algorithm, routing table

RAJNOHA, Martin *Modelování skladů pomocí grafického rozhraní*: diplomová práca. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2016. 56 s. Vedúci práce bol doc. Ing. Radim Burget, Ph.D.

PREHLÁSENIE

Prehlasujem, že som svoju diplomovoú prácu na tému „Modelování skladů pomocí grafického rozhraní“ vypracoval samostatne pod vedením vedúceho diplomovej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej diplomovej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/nebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona č. 121/2000 Sb., o právu autorském, o právoch súvisejúcich s právom autorským a o zmeně niektorých zákonov (autorský zákon), vo znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhé, hlavy VI. diel 4 Trestného zákoníka č. 40/2009 Sb.

Brno ....

(podpis autora)

POĎAKOVANIE

Rád by som poďakoval vedúcemu diplomovej práce pánovi doc. Ing. Radimovi Burgetovi, Ph.D. za odborné vedenie, konzultácie, ochotu a užitočné rady a návrhy k práci.

Brno ....

(podpis autora)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

POĎAKOVANIE

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .

(podpis autora)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj
pro inovace

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ÚVOD

Riadenie skladu je zložitý a časovo náročný proces, ktorý môže priniesť výrazné časové a finančné úspory. Vďaka tomu vznikli aj metódy, ktoré sa snažia optimalizovať činnosti v sklade automaticky. Neoddeliteľnou časťou všetkých optimalizačných metód je určenie vzdialenosti medzi ľubovoľnými miestami skladu. Táto práca navrhuje výpočetne efektívnu metódu, ktorá s pomocou teórie grafov predpočítava vzdialenosti v sklade a na základe toho výrazne skracuje čas vyhľadania cesty k ľubovoľnému miestu.

Aby však bolo umožnené vyhľadávanie ciest, je nutné najskôr vytvoriť dátový model skladu. Problémom je, ako prekonvertovať nakreslený sklad do dátového modelu a ako s ním vôbec pracovať, aby sa dosiahli očakávané výsledky v oblasti časovej náročnosti optimalizačných algoritmov. Dôležité je ale aj to, aby užívateľ pracoval pomocou grafického rozhrania – GUI (z angl. Graphical User Interface) so spoľahlivou, komplexnou a rýchlou aplikáciou, ktorá umožňuje všetky vyššie uvedené veci, bez ohľadu na užívateľove znalosti problematiky.

Diplomová práca opisuje spôsob, akým sa prevádza nakreslený sklad do dátového modelu a ako sa nad ním vytvorí dátová štruktúra grafu. Využíva pri tom prvok z oblasti obrazového spracovania dát – metódu skeletonizácie (nájdenie kostry obrazu). Z tejto kostry sa pomocou ďalších algoritmov identifikujú vrcholy a hrany grafu. Ďalej je v práci opísaný spôsob, akým sa vyhľadávajú jednotlivé cesty v sklade. Základom pre vyhľadaváníe ciest je grafová podoba skladu a Dijkstrov algoritmus, známy predovšetkým z počítačových sietí. Rýchle vyhľadávanie ciest umožňuje predpočítaná smerovacia tabuľka. Práca taktiež zahŕňa prípadné nehody a zablokovanie ciest, podobne opisuje aj spôsob ako uprednostniť niektoré cesty pred inými. Vyhľadávacie algoritmy počítajú s týmito možnosťami a sú navrhnuté tak, aby vypočítali najlepšiu možnú cestu, vzhľadom na zablokované miesta a uprednostnené cesty.

Hľavným prínosom práce je návrh algoritmu na prevod prakticky akéhokoľvek skladu v grafickej podobe do dátového modelu a na základe toho vytvorenie grafovej reprezentácie skladu. Ďaľej je navrhnutý algoritmus, ktorý umožňuje zistiť dobu prepravy a vyhľadávať najkratšie cesty medzi ľubovoľnými pozíciami skladu v rádoch stoviek mikrosekúnd (v prípade implementácie optimalizačných algoritmov, ktoré môžu využívať genetické algoritmy, je výpočetný čas kľúčový). V rámci diplomovej práce bola navrhnutá a vytvorená aplikácia, ktorá umožňuje sklad nakresliť, previesť do grafovej štruktúry, upravovať ceny ciest (uprednostňovanie ciest), pridávať a odoberať nehody a vyhľadávať cesty.

Obsah práce je organizovaný nasledovne: v prvej kapitole je opísaný teoretický úvod do problematiky skladového hospodárstva a teórie grafov. Najobsiahlejšou časťou je druhá kapitola, ktorá sa zaoberá návrhmi riešenia pri vývoji, opisuje jednotlivé

postupy pri prevode skladu do podoby grafu a problémy, ktoré vznikali popri vývoji aplikácie. Ďalej opisuje riešenie problému s nehodami a zablokovanými cestami a taktiež s uprednostňovaním jednotlivých ciest. Nakoniec je v druhej kapitole popísaný algoritmus na vyhľadávanie ciest. Táto kapitola taktiež okrajovo zachádza do implementácie. Poslednou časťou práce sú výsledky, ktoré poukazujú na správnu funkčnosť aplikácie z hľadiska prevodu do grafovej štruktúry a zobrazujú zaujímavé časové závislosti a porovnania pri vyhľadávaní ciest.

1 TEORETICKÝ ÚVOD

1.1 Skladové hospodárstvo

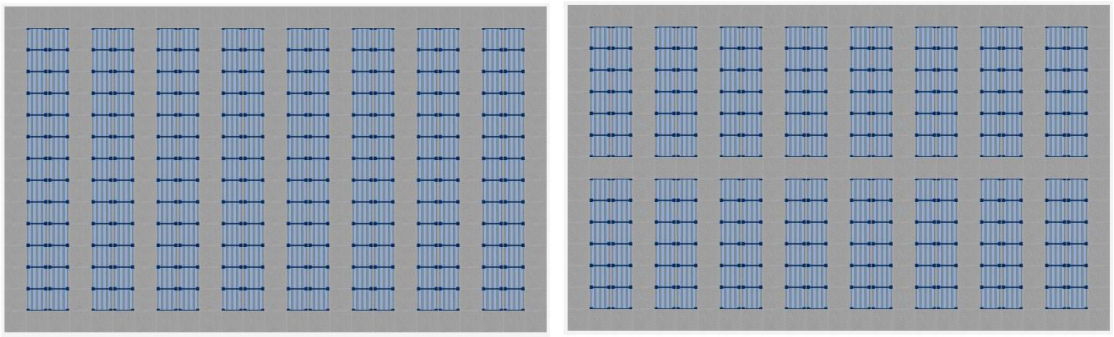
Spoločným cieľom každého obchodníka je snaha o čo najväčšiu redukciu pracovného času a naopak maximálne zvyšovať zisk. Nie je tomu inak ani u obchodov pracujúcich so skladovými priestormi alebo v distribučných centrách. V tomto prípade optimalizácia procesu naskladňovania tovaru a vybavovania objednávok môže viesť k výraznému zlepšeniu efektivity pracovníkov. 55-65 % zo všetkých nákladov na prevádzku skladu sú spojené práve s vybavovaním objednávok. Hlavné skladové úkony, ako sú naskladňovanie tovaru a vybavovanie objednávok, čelia spoločnému problému, ktorým je vyhľadanie najoptimálnejšej cesty. U vybavovania objednávok sa tento problém umocňuje tým, že objednávka obsahuje zvyčajnie viac položiek, ktoré je nutné nájsť a priniesť do určenej zóny. Tento proces je najviac časovo zaťažujúci pre prevádzku skladu a je ekvivalentný problému obchodného cestujúceho. Tento problém hľadá riešenie ako nájsť najkratšiu cestu medzi n mestami, ktoré navštívi práve jeden krát, začínajúc a končiac v rovnakom meste [13], [26].

Aby sa našlo optimálne riešenie a znížil sa výpočetný čas algoritmov na minimum, je nevyhnutné pracovať so systémom, ktorý umožní vyhľadávať cesty medzi ľubovolnými pozíciami skladu vo veľmi krátkom čase. Podobne je tomu aj pri optimalizácii procesu naskladňovania tovaru. Jedným z cieľov diplomovej práce je práve návrh algoritmu, ktorý umožní takéto vyhľadávanie ciest v rádoch milisekúnd.

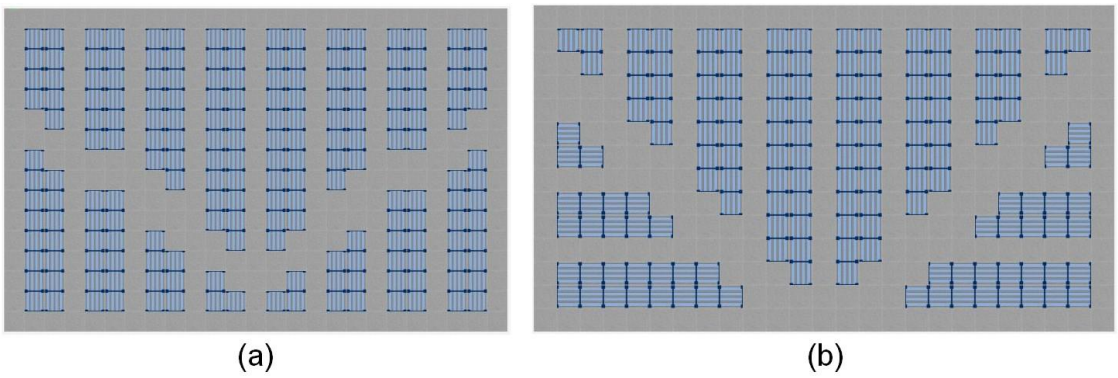
K dosiahnutiu tohto stavu je však nutné detailne poznať štruktúru skladu, v ktorom sa majú implementovať optimalizačné algoritmy. Každý skladový priestor je unikátny či už veľkosťou, počtom regálov, alebo ich rozložením, preto musí byť aplikácia nezávislá na štruktúre skladu a schopná pracovať s akýmkoľvek skladom. Z hľadiska rozmiestnenia regálov je možné rozdeliť hlavné skladové štruktúry na:

- Tradičný dizajn, obr. 1.1.
- „Lietajúce – V“ (z angl. flying – V) dizajn, obr. 1.2 (a).
- Dizajn „rybia kosť“ (z angl. fishbone), obr. 1.2 (b).
- „Obrátené – V“ (z angl. inverted – V) dizajn, obr. 1.3 [25].

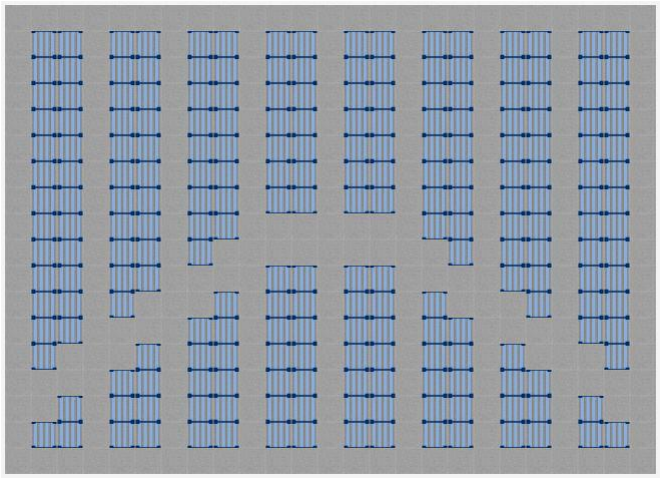
Aby mohli byť optimalizačné algoritmy nasadené, je nutné skladový priestor (jeho štruktúru) prekonvertovať do dátovej štruktúry. Prekonvertovanie skladu do tejto štruktúry preto nie je jednoduchá úloha a je to ďalší z cieľov tejto práce. Metóda na prekonvertovanie skladu do dátovej štruktúry je navrhnutá tak, aby ju bolo možné aplikovať na akúkoľvek podobu skladu a je plne automatická, tzn. bez potreby zásahu ľudského faktoru.



Obr. 1.1: Príklady tradičných dizajnov skladu.



Obr. 1.2: Príklady skladových dizajnov „lietajúce – V“ (a) a „rybia kosť“ (b).



Obr. 1.3: Príklad skladového dizajnu „obrátené – V“.

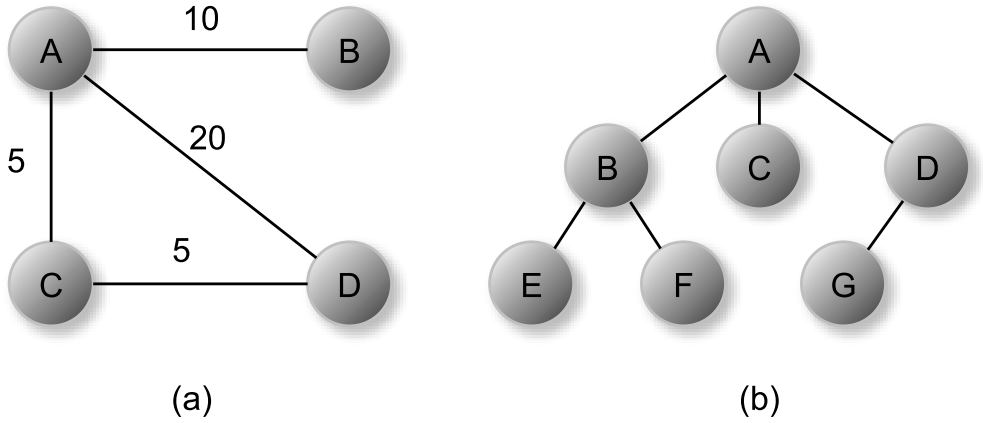
1.2 Teória grafov

Poznáme mnoho typov grafov, napr. využívané v štatistike (stĺpcový, koláčový ...), v matematike ako funkčné závislosti atď. Graf si môžeme predstaviť ako zjednodušenie reálneho problému, kde daný problém znázorňujeme pomocou bodov a čiar. V teórii grafov tieto body nazývame **vrcholy grafu** a čiary, ktoré ich spájajú, nazývame **hrany grafu** [24].

Teória nedefinuje, čo presne vrcholy a hrany sú, ich interpretácia je definovaná až daným problémom, ktorý sa pomocou teórie grafov rieši. Z tohto dôvodu je práca s grafmi vo veľmi obecnej rovine a má veľké uplatnenie v informatike, kde sa grafy používajú ako dátová štruktúra modelujúca entity a vzťahy medzi nimi [23].

Do podmnožiny grafov spadajú všetky ostatné dátové štruktúry a majú najväčšiu vyjadrovaciu silu pre reprezentáciu dát. Ich nevýhodou oproti iným štruktúram je väčšia pamäťová a časová náročnosť.

Definícia: Graf G je usporiadaná dvojica $G = (V, E)$, kde V je konečná množina vrcholov a E je konečná množina hrán.
Definícia: Majme vrcholy $x, y \in V$. Tieto vrcholy nazývame susednými, pokiaľ platí $(x, y) \in E$ [5].



Obr. 1.4: Príklad neorientovaného grafu. Graf (a) je **ohodnotený** – možné použitie napr. v počítačovej sieti. Graf (b) je **neohodnotený**, táto štruktúra sa nazýva **strom** (špeciálny typ grafu, kde počet hrán je o jedna menší ako počet vrcholov).

Medzi dvoma vrcholmi však môže existovať viac hrán (vyššie uvedená definícia toto neumožňuje). K hranám sa často pridávajú číselné ohodnotenia, ktoré vyjadrujú napr. pravdepodobnosť udalosti, cenu linky... Výsledkom je reálny model siete, preto takýto graf nazývame ohodnotený, alebo sieť, viď obr.1.4 (a) [30].

1.3 Základné pojmy a rozdelenie grafov

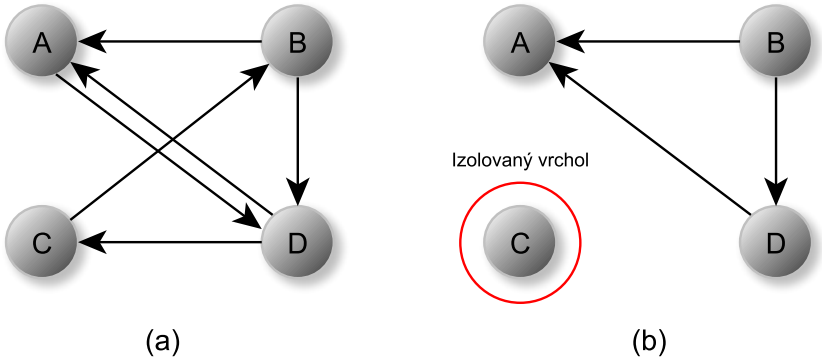
Kvôli nasledujúcemu textu, je potrebné popísať niektoré základné pojmy a typy grafov na základe [23] a [5]:

- **Stupeň vrcholu** je počet hrán vychádzajúcich z vrcholu.
- **Susedné vrcholy** tzv. susedia, sú akékoľvek dva vrcholy spojené hranou.
- **Slučka** je hrana vedúca z vrcholu V do toho istého vrcholu V.
- **Rovnobežné hrany** sú dve alebo viaceré hrany spájajúce rovnakú dvojicu vrcholov.
- **Izolovaný vrchol** je vrchol, ktorý neinciduje[1] so žiadnou hranou, viď obr. 1.5 (b).
- **Sled** je postupnosť vrcholov a hrán medzi počiatočným vrcholom V_s a koncovým vrcholom V_d. Pokiaľ $V_s = V_d$ nazývame sled uzavretý, inak sa jedná o otvorený sled. Dĺžka sledu je rovná počtu hrán, alebo súčtu ich ohodnotení.
- **Ťah** je sled, v ktorom sa neopakuje žiadna hrana.
- **Cesta** je taký ťah, v ktorom sa neopakuje žiadny vrchol. Každý vrchol môže incidovať maximálne s dvoma hranami tohto ťahu.
- **Vzdialenosť** medzi dvoma vrcholmi V_s a V_d je určená dĺžkou najkratšieho počtu hrán medzi nimi (dĺžka najkratšej cesty). Ak neexistuje sled medzi V_s a V_d, vzdialenosť je nekonečno.

Grafy môžeme klasifikovať podľa rôznych kritérií. Ako je uvedené vyššie, grafy môžu byť ohodnotené (obr. 1.4 (a)), alebo neohodnotené (obr. 1.4 (b)). Ďalej rozdeľujeme grafy:

- Podľa orientácie hrán na **orientované** (obr. 1.5) a **neorientované** (obr. 1.4).
- Podľa dostupnosti vrcholov na **súvislé** (existuje cesta medzi všetkými vrcholmi) (obr. 1.4 a obr. 1.5 (a)) a **nesúvislé** (neexistuje cesta medzi minimálne dvoma vrcholmi), viď obr. 1.5 (b).
- **Úplný** graf má hranu medzi každou dvojicou vrcholov. **Diskrétny** graf má naopak množinu hrán $E = 0$.

[1] Incidentný – vychádzajúci. Pre izolovaný vrchol neexistuje cesta minimálne do jedného vrcholu.



Obr. 1.5: Príklad orientovaného grafu, kde (a) zobrazuje **súvislý** graf a (b) **nesúvislý** graf s izolovaným vrcholom. (Graf by bol nesúvislý aj bez izolovaného uzlu, pretože neexistuje cesta z vrcholu A do žiadneho iného vrcholu).

1.4 Prehľadávanie grafu

Dátovou štruktúrou grafu dokážeme namodelovať problém, ale zvyčajne jeho neoddeliteľnou časťou je aj prehľadávanie (priechod) grafu. Algoritmy prechádzujúce stavovým priestorom[2] grafu sa rozdeľujú do dvoch základných skupín. Algoritmy bez žiadneho odhadu optimálnej cesty k nájdeniu cieľa – **slepé prehľadávanie** a presným opakom sú algoritmy využívajúce **informované metódy** [5].

1.4.1 Algoritmy využívajúce slepé prehľadávanie

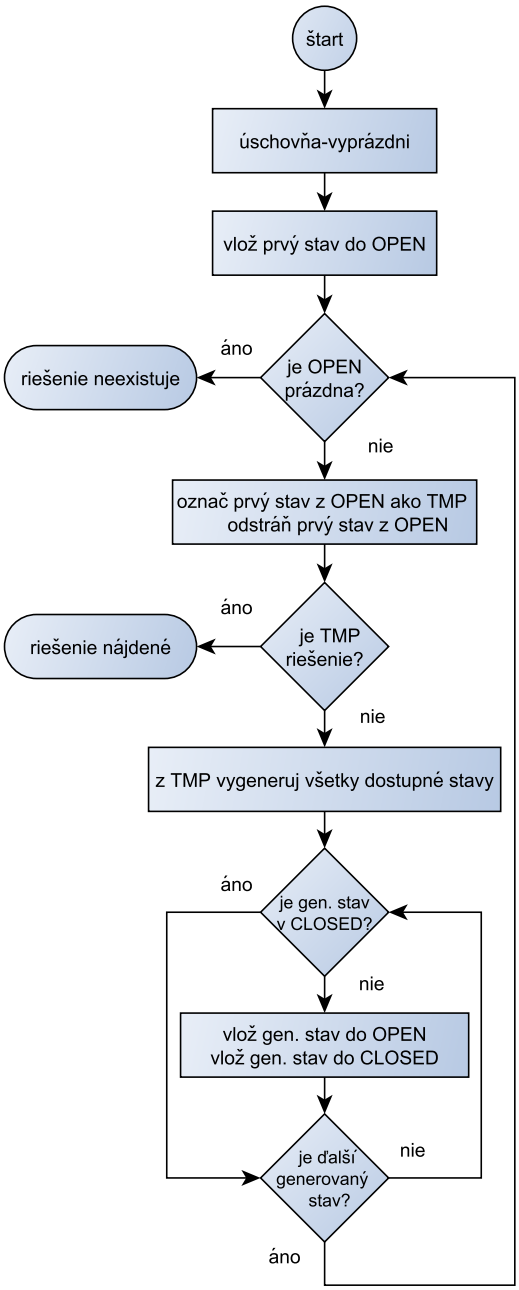
Pri slepom prehľadávaní sa prechádza stavový priestor postupne a všetky smery sa prehľadávajú s rovnakou prioritou. Základné algoritmy slepého prehľadávania využívajú rovnaký model, viď obr. 1.6, líšia sa však v spôsobe, ktorým implementujú „úschovňu“ pre dostupné stavy [21].

Prehľadávanie do hĺbky (Depth First Search – DFS)

DFS prechádza vždy cez prvého potomka, druhého atď., pokým nenarazí na list[3]. Potom sa vráti o krok späť a pokračuje druhým potomkom, rodiča listu atď., pokiaľ nie sú prehľadané všetky vrcholy. DFS ako úschovňu pre dostupné vrcholy implementuje zásobník typu LIFO (z angl. Last In First Out), viď obr. 1.6 [20].

[2]Stavový priestor – všetky možnosti.

[3]List je vrchol grafu, ktorý nemá potomkov [18]. Stupeň tohto vrcholu je 1 [8].



Obr. 1.6: Stavový diagram algoritmu pre slepé prehľadávanie grafu. Úschovňa obsahuje dve množiny: OPEN (možné stavy) a CLOSED (navštívené stavy). OPEN je implementovaná ako fronta pre **BFS**, ako zásobník pre **DFS**. **Dijkstra** vyberá z úschovne vždy vrchol najbližší počiatočnému vrcholu – **prioritná fronta** (popísané v 1.4.1) [5], [21], [28].

Prehľadávanie do šírky (Breadth First Search – BFS)

BFS narozdiel od DFS prechádza graf po vrstvách – prehľadá všetkých potomkov na rovnakej úrovni a ich potomkov pridáva do úschovne, implementovanej ako fronta

typu FIFO (z angl. First In First Out). To zaručí prehľadanie potomkov z ďalšej úrovne až po prehľadaní všetkých potomkov na danej úrovni, viď obr. 1.6 [20].

Dijkstrov algoritmus

Dijkstrov algoritmus je jedným zo základných algoritmov používaných v informatike k nájdeniu najkratšej cesty. Hľavnou myšlienkou je vložiť všetky vrcholy grafu do úschovne typu prioritnej[4] fronty, ktorá je kľúčovaná vzdialenosťou. Jej hodnota je dĺžka doposiaľ najkratšej známej cesty z počiatočného vrcholu do práve prehľadávaného vrcholu. V každom cykle je spracovávaný vrchol V s minimálnou vzdialenosťou (zatiaľ najkratšia cesta k počiatočnému vrcholu). Potom sa aktualizujú vzdialenosti k susedom V, pridajú sa do fronty a V sa pridá do množiny CLOSED. Tento proces sa opakuje, pokiaľ nie je nájdená najkratšia cesta (obr. 1.6). Podmienkou správnej funkčnosti Dijkstrovho algoritmu je, aby ohodnotenia hrán boli kladné [16], [5], [9].

1.4.2 Algoritmy využívajúce informované metody

Algoritmy sa snažia odhadnúť kadiaľ pokračovať, aby čo najskôr dosiahli cieľ. Uprednostňujú niektoré stavy oproti iným. To sa prejaví menším počtom prechádzaných stavov. Nevýhodou však je nutnosť fitness[5] funkcie, ktorá hodnotí každý stav. Potrebuje k tomu však čas navyše. Práca nevyužíva informované metódy, preto sú spomenuté len veľmi okrajovo a vymenované len základné algoritmy, ktoré ich využívajú:

- **Best FS** (First Search – FS) je optimalizovaný algoritmus BFS, expandujúci aktuálny stav, ale pokračuje v najvhodnejšom stave.
- **Hill Climbing** je opäť algoritmus vychádzajúci z BFS, ktorý však odstraňuje staré riešenia. Optimalizuje časovú zložitosť.
- **A*** je podobný algoritmus, ale pracuje navyše s kompletnou cestou, nie len s aktuálnym krokom [5].

[4]Prvky sa radia do prioritnej fronty podľa hodnoty (kľúča) bez ohľadu na poradie (to sa berie do úvahy len v prípade rovnakého kľúča) [15].

[5]Fitness funkcia – hodnotiacia funkcia, ktorá určuje ako dobre bol daný problém vyriešený [17].

2 NÁVRH RIEŠENIA

2.1 Úvod

Aby bolo možné vyhľadávať cesty v sklade, je nutné nakreslený sklad najskôr prekonvertovať do podoby grafu. Nasledujúce kapitoly popisujú detailnejšie najdôležitejšie časti algoritmu, ktorý pracuje následovne. Ako prvá sa vytvorí dátová štruktúra skladu – **dvojrozmerné pole**, ktorého prvkami sú priechodné objekty (chodba) a nepriechodné objekty (regál). Aby bolo možné vytvoriť šturktúru grafu, pole sa ďalej prekonvertuje do obrázku typu **byteProcessor** – každý pixel je prezentovaný 8 bitmi (1 byte) a následne do **binaryProcessor** – obrázok, ktorý má len dve hodnoty (zvyčajne 0 a 1). V takejto reprezentácii obrázku sa nájde kostra grafu pomocou metódy **skeletonizácie** [4], [2].

Kostra sa ďalej optimalizuje kvôli bezchybnému vyhľadaniu smerovačov a ciest, ktoré budú tvoriť vrcholy a hrany grafu. V optimalizovanej kostre sa identifikujú miesta, kde majú byť smerovače (indexy pola skladu a binaryProcessoru sú rovnaké). Medzi smerovačmi sa pomocou kostry vyhľadajú cesty (hrany grafu). Pri hľadaní ciest sa zároveň počíta ich cena (ohodnotenie hrany). Táto časť je pravdepodobne najzložitejšia v algoritme. Pomocou vyhľadaných ciest a ich dvojíc smerovačov sa vytvorí grafová štruktúra. V tomto momente už je možné hľadať cesty pomocou niektorých z metód prehľadávania grafu. Vyhľadávanie je však možné zatiaľ iba z miest, kde ležia smerovače. Preto sa následne všetkým bunkám skladu predpočítajú najbližšie smerovače a vzdialenosti k nim, aby bolo umožnené vyhľadávanie z ktorejkoľvek pozície skladu. Pretože je potrebné vyhľadávanie ciest rádovo v milisekundách, predpočíta sa pomocou Dijkstrovho algoritmu **smerovacia tabuľka**. Pri samotnom vyhľadávaní algoritmus pozerá na jednotlivé ceny ciest (z dôvodu prípadného uprednostnenia cesty podľa užívateľa) a taktiež počíta s možnosťou nehody v sklade a pokiaľ to je možné, snaží sa im vyhnúť.

2.2 Štruktúra skladu

V práci je pre účely skladu použitá dátová štruktúra dvojrozmerného poľa. Jeho nevýhodou je nemenná dĺžka (to ale nevadí, pretože rozmery skladu sa nemenia). Výhodou je však rýchlosť získania prvku zo znalosti indexu[1]. Jeho prvky sú v tejto práci pomenované ako bunky.

Pole je typu abstraktných buniek – abstraktná[2] trieda AbstractStockCell. Táto trieda má atribúty index v poli X a Y, vektory[3] najbližších smerovačov a vzdialeností. Dedia[4] z nej jednotlivé triedy na kreslenie skladu:

- **regál** – RackCell (nepriechodný objekt)
- **chodba** – CoridorCell (môže byť použitá aj ako nakladacia rampa – priechodný objekt)
- **hranica** – BorderCell (ohraničenie skladu kvôli lepšiemu vykresleniu do užívateľského rozhrania a hlavne kvôli chybe pri skeletonizácii, viď obr. 2.5, hranica je taktiež použitá v prípade nepriechodného objektu, ktorý nie je regál)
- **smerovač** – RouterCell (potrebný na vyhľadávanie ciest v sklade, bližšie popísané v 2.3.2 a 2.4.2)

Dedičnosť a atribúty jednotlivých tried sú zobrazené na UML (z angl. Unified Modeling Language) diagrame na obr. 2.1.

Pri tvorbe skladu v aplikácii sa jednotlivé prvky poľa napĺňajú už konkrétnymi typmi buniek. Tým, že je pole typu triedy AbstractStockCell, môžu byť jeho prvkami všetky triedy, ktoré z nej dedia. Rozmery poľa sú odvodené z dĺžky a šírky skladu a hodnoty oddeľovača. Oddeľovač (delimiter) určuje, po koľkých metroch sa má sklad deliť na bunky, resp. aký je rozmer najmenšieho objektu v sklade. Týmto spôsobom sa minimalizuje veľkosť poľa pri zachovaní rozmerov skladu, viď obr. 2.2. Po zadaní týchto údajov užívateľom je okrem iného kontrolovaný hlavne zvyšok po delení rozmerov skladu oddeľovačom (musí byť nula). Ukážka kódu pre vytvorenie pola skladu:

```
AbstractStockCell[][] pole = new AbstractStockCell[x][y];
```

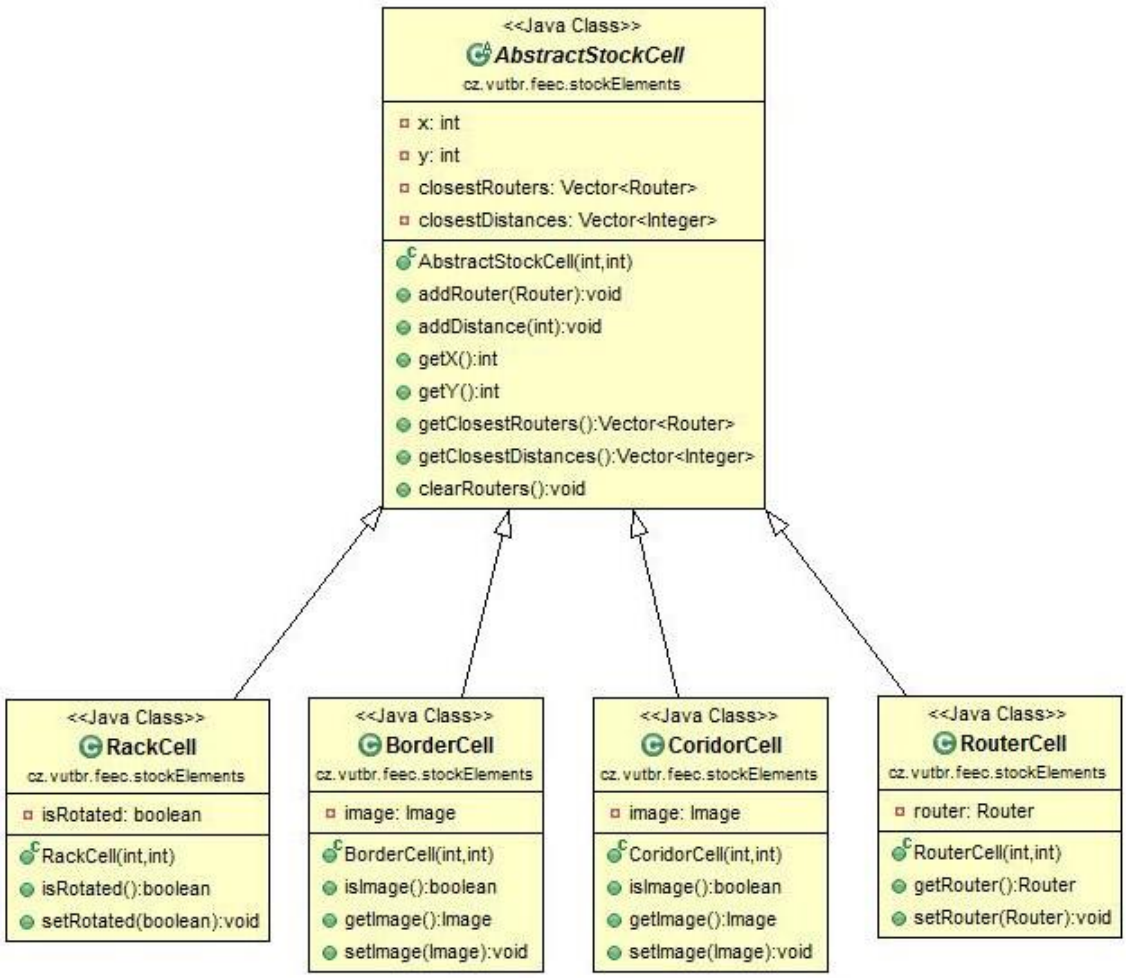
kde $x = dĺžka/delimiter + 2$ a $y = šírka/delimiter + 2$. Pridávajú sa 2 extra riadky kvôli vyššie uvedeným hraničným bunkám a problémom pri skeletonizácii. Po zadaní rozmerov skladu sa pole naplní bunkami chodby (vznikne prázdna plocha skladu).

[1] Index určuje poradie prvku v poli.

[2] Z abstraktnej triedy nie je možné vytvárať objekty, zvyčajne sa z nej dedí [3].

[3] Vektor je štruktúra implementujúca rastúce pole objektov o premennej dĺžke [7].

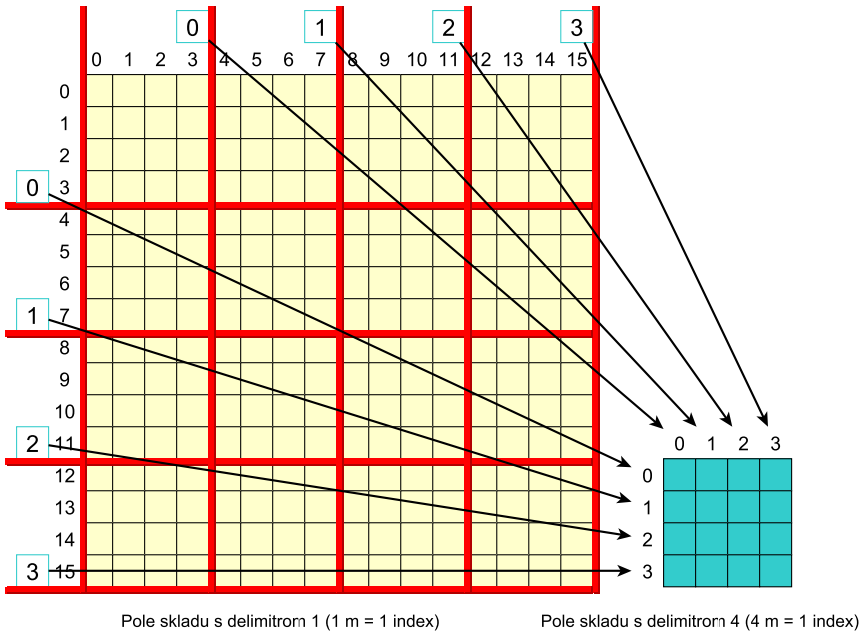
[4] Dedičnosť – preberanie vlastností.



Obr. 2.1: UML diagram tried buniek skladu, zobrazujúci vzťah dedičnosti a atribúty jednotlivých tried.

2.3 Prevod do podoby grafu

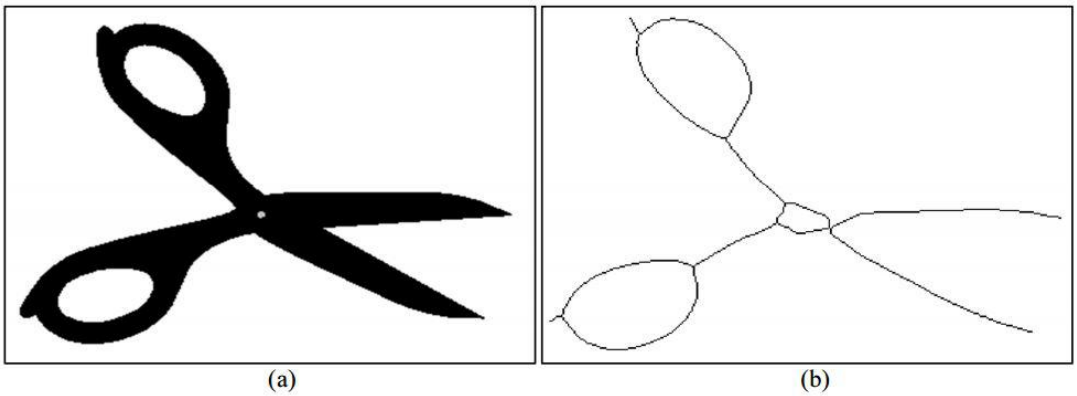
Aby bolo možné rýchle vyhľadávanie ciest v sklade, je nutné vytvoriť graf s čo najmenším počtom vrcholov. Štruktúra grafu je podobná ako v počítačových sieťach. Vrcholy grafu sú smerovače a hrany, ktoré ich spájajú, cesty. Problémom je, ako v poli skladu identifikovať miesta, kde majú byť smerovače a vyhľadať cesty medzi nimi. Sklad nemusí byť vždy nakreslený ideálne, môže obsahovať veľa nadbytočných buniek chodby, prípadne môže mať chyby. Vlastný algoritmus vyhľadávania smerovačov by bol preto zdĺhavý a nemusí byť vždy správny. Kvôli týmto dôvodom sa využíva metóda skeletonizácie.



Obr. 2.2: Dvojrozmerné pole skladu. Porovnanie veľkostí polí skladov bez minimalizácie a s použitím delimitra o hodnote 4.

2.3.1 Skeletonizácia

Skeletonizácia je transformácia digitálneho obrazu do podmnožiny pôvodného obrazu, s cieľom znížiť množstvo dát, potrebného na reprezentáciu obrazu, alebo zjednodušiť tvar obrazu [27]. Príklad skeletonizácie je zobrazený na obr.2.3.



Obr. 2.3: Príklad skeletonizácie, kde (a) zobrazuje pôvodný obraz a (b) kostru vytvorenú skeletonizáciou. Prebrané z [1].

Ako je spomenuté v úvode, je potrebné najskôr prekonvertovať pole skladu do podoby, na ktorej môže byť vykonaná skeletonizácia. Pole obsahuje viacero buniek (chodba, regál. . .), ale v princípe sú rozdelené na 2 typy – priechodné a nepriechodné

objekty. Pole je cyklom prekonvertované do obrázku, typu byteProcessor, kde každá bunka poľa reprezentuje jeden pixel v obrázku.

```
int length = stock.getLength() / stock.getDelimiter();
int width = stock.getWidth() / stock.getDelimiter();
AbstractStockCell[][] field = stock.getField();
ByteProcessor bp = new ByteProcessor(length, width);

for (int i = 0; i < length; i++) {
   for (int j = 0; j < width; j++) {
   if (field[i][j].getClass() == RackCell.class) {
      bp.set(i, j, RACK); // hodnota 255
   } else if (field[i][j].getClass() == BorderCell.class) {
      bp.set(i, j, BORDER); // hodnota 70
   }
   // ostatné bunky zostávajú čierne
```

Aby s ním bolo možné ďalej pracovať je nutné vytvoriť obrázok typu binaryProcessor, ktorého pixely naberajú len dve hodnoty, zvyčajne vyjadrované ako 0 a 1, v obrázku však ako 0 a 255 (čierna a biela).

```
BinaryProcessor binp = new BinaryProcessor(
      (ByteProcessor) bp.duplicate());
```

Ak sa pri vytváraní binaryProcessoru narazí na pixel s inou hodnotou, je takýto pixel podľa prahovej hodnoty[5] nastavený na 0 alebo 255. V našom prípade majú regále hodnotu 255, chodby 0 a hraničné bunky 70, ktoré sú prevedené na 0 v obrázku binárneho typu. Vznikne čierne pozadie s bielymi regálmi, viď obr. 2.4 (b). Nad takouto reprezentáciou už je možné vykonať skeletonizáciu [2].

```
binp.skeletonize();
```

Obrázky typu ByteProcessor a BinaryProcessor sú vytvorené pomocou knižnice **ImageJ**, ktorá taktiež umožňuje vykonať skeletonizáciu.

Nájdenie kostry

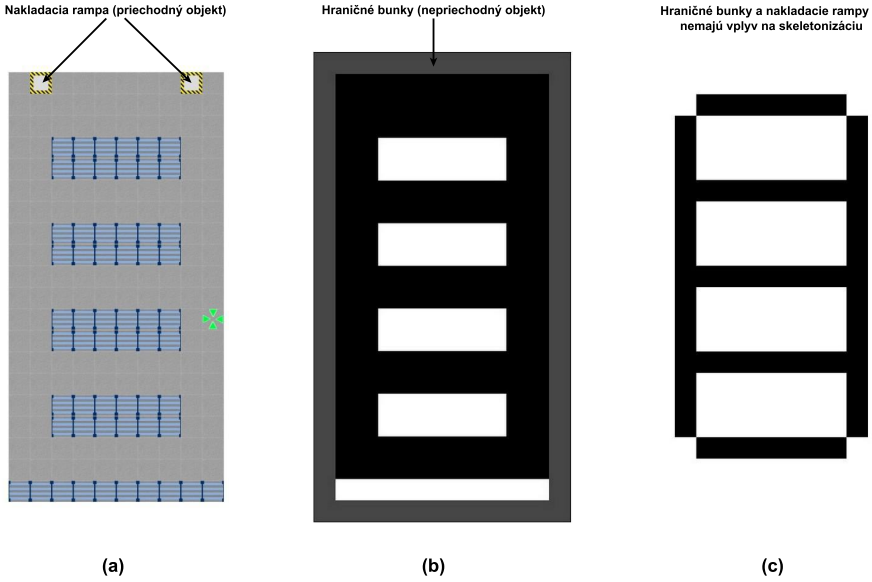
Poznáme viacero druhov skeletonizačných metód. V tejto práci sa využíva metóda iteratívneho „stenšovania“ (iterative thinning) k nájdeniu kostry objektu (obrazu). Kostra je súbor tenkých čiar, oblúkov a kriviek (zvyčajne široká jeden pixel), ktoré sú spojené tak, aby zachovali pôvodné topologické a geometrické vlastnosti objektu.

[5]Pixely s odtieňom šede nad prahovou hodnotou sú nastavené na hodnotu 255 a ostatné na 0.

Stenšovací algoritmus by mal splňovať nasledujúce podmienky:

- Výsledný súbor by mal byť ideálne tenký (jeden pixel).
- Výsledný súbor by mal aproximovať k strednej osi[6].
- Koncové body a spojitosť objektu a pozadia musia byť zachované.

Kostra je definovaná transformáciou na strednú os (Medial Axis Transform – MAT) objektu. MAT objektu (obrazu) R je vypočítaná nasledovne: Pre každý bod p v R vyhľadaj najbližšieho suseda N, ak bod p má viac susedov ako N, tak patrí do strednej osi R. Cieľom skeletonizácie je redukovať objekt R tak, aby vygenerovaná kostra bola široká jeden pixel a maximálne uchovala rozsah a spojitosť originálneho objektu. Aj malé nezrovnalosti v originálnom objekte môžu viesť k chybám v kostre, viď obr. 2.6 a 2.5.

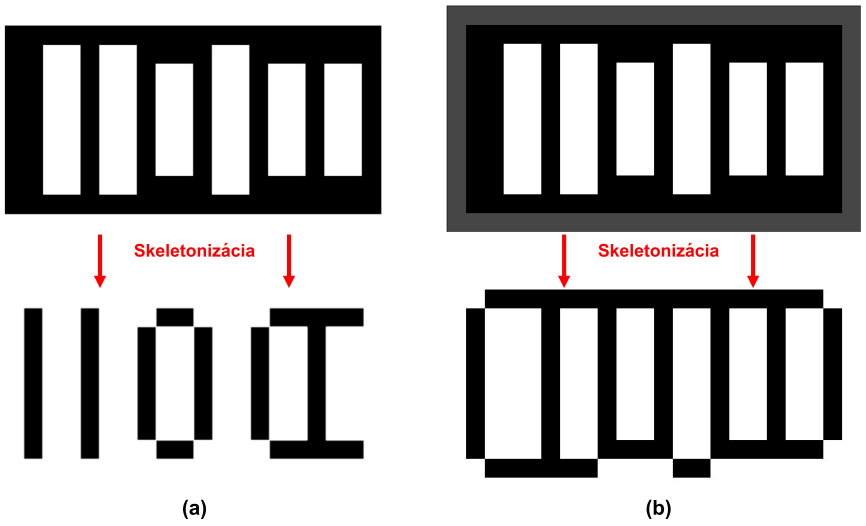


Obr. 2.4: Proces skeletonizácie. (a) zobrazuje nakreslený sklad (hraničné bunky sú v GUI skryté). (b) znázorňuje prekonvertovaný sklad do podoby byteProcessoru, kde bunka regálu reprezentuje biely pixel, chodba čierny. Po prekonvertovaní do binaryProcessoru hraničné bunky reprezentujú taktiež biele pixely. (c) zobrazuje výslednú kostru vypočítanú skeletonizáciou.

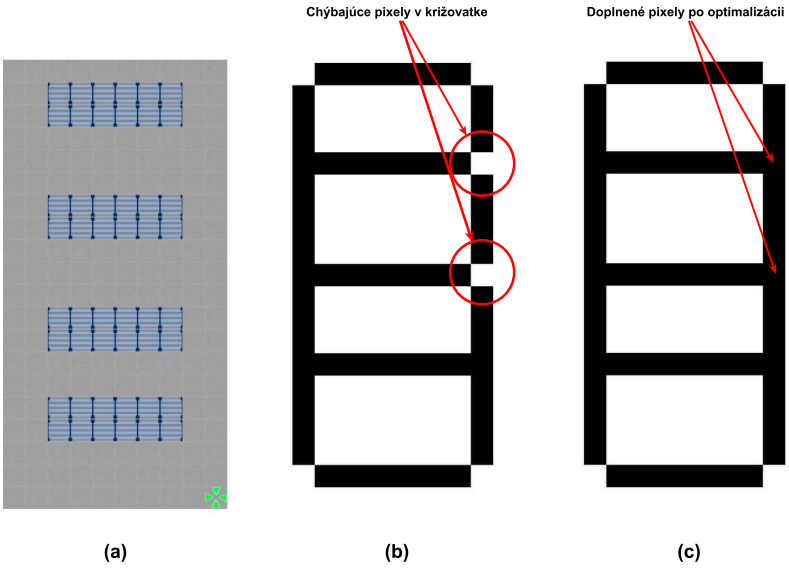
Z kostry dokážeme určiť rôzne útvary na základe počtu „trojitých bodov“ (miesto, kde sa stretávajú minimálne tri vetvy kostry). Takýto bod tvorí v podstate **križovatku**. Po vykonaní skeletonizácie nad binaryProcessorom skladu, získame kostru chodieb o šírke jeden pixel tj. jedna bunka v poli, čím sa zredukovali všetky

[6]Stredná os objektu je množina všetkých bodov, ktoré majú minimálne jeden bližší bod k hranici objektu [12].

nadbytočné bunky. Proces skeletonizácie je zobrazený na obr. 2.4. Každá vzniknutá križovatka vytvára smerovač (vrchol grafu) a vetvy kostry zasa cesty (hrany grafu) [31].



Obr. 2.5: Porovnanie výsledných kostier vytvorených z polí bez hraničných buniek (a), kde vznikla chyba a s pridanými hraničnými bunkami (b), vďaka ktorým je táto chyba eliminovaná.



Obr. 2.6: Chyba v skeletonizácii, zle vyhodnotená križovatka. (a) zobrazuje nakreslený sklad, (b) kostru, vypočítanú skeletonizáciou s dvoma chybami v križovatkách a (c) zobrazuje výslednú opravenú kostru po optimalizačnom algoritme, ktorý detekuje chýbajúce pixely a doplní ich.

2.3.2 Vytvorenie grafu

Aby bolo možné vytvoriť graf, je nutné definovať vrcholy a hrany. Tie sú v tejto práci reprezentované ako smerovače a cesty. Po vykonaní skeletonizácie a získaní kostry je nadväzujúcim krokom priamo v binaryProcessoru identifikovanie smerovačov a následne ciest. Predtým, ako začne samotné vyhľadávanie smerovačov, je potrebné optimalizovať kostru, kvôli prípadným chybám v skeletonizácii, viď obr. 2.6. Algoritmus identifikuje pixely, ktoré majú minimálne z troch susedných strán pixel čiernej farby (chodbu), a v prípade, že taký nájde, zmení jeho farbu na čiernu. V takto optimalizovanej kostre už je možné identifikovať smerovače a cesty a následne ich pridať do príslušných vektorov skladu. Trieda skladu obsahuje niekoľko atribútov a pretože je unikátna v celej aplikácii, je vytvorená podľa návrhového vzoru jedináčik (Singleton)[7].

```
public class Stock {

private static final Stock STOCK = new Stock(); // jedináčik
private float length = 1;
private float width = 1;
private short delimiter = 1;
private AbstractStockCell[][] field;
private Vector<Router> routers = new Vector<Router>();
private Vector<Path> paths = new Vector<Path>();
private Dijkstra dijkstra = null;
private Graph graph = new Graph();
private RoutingTable routingTable;
private AbstractStockCell isCurrent;
private Vector<AbstractStockCell> ramps = new Vector<AbstractStockCell>();
private Vector<AbstractStockCell> accidentCells = new Vector<AbstractStockCell>()

private Stock() {
// privátny konštruktor na zamedzenie vytvorenia instancie
}
public static Stock getInstance() {
// vráti jedinečnú instanciu
return STOCK;
}
//ďalšie metody ....
```

[7]Návrhové vzory reprezentujú riešenie problému (tried problémov), ktoré môžu byť vložené do vlastného riešenia (kódu) [22].

Identifikovanie smerovačov

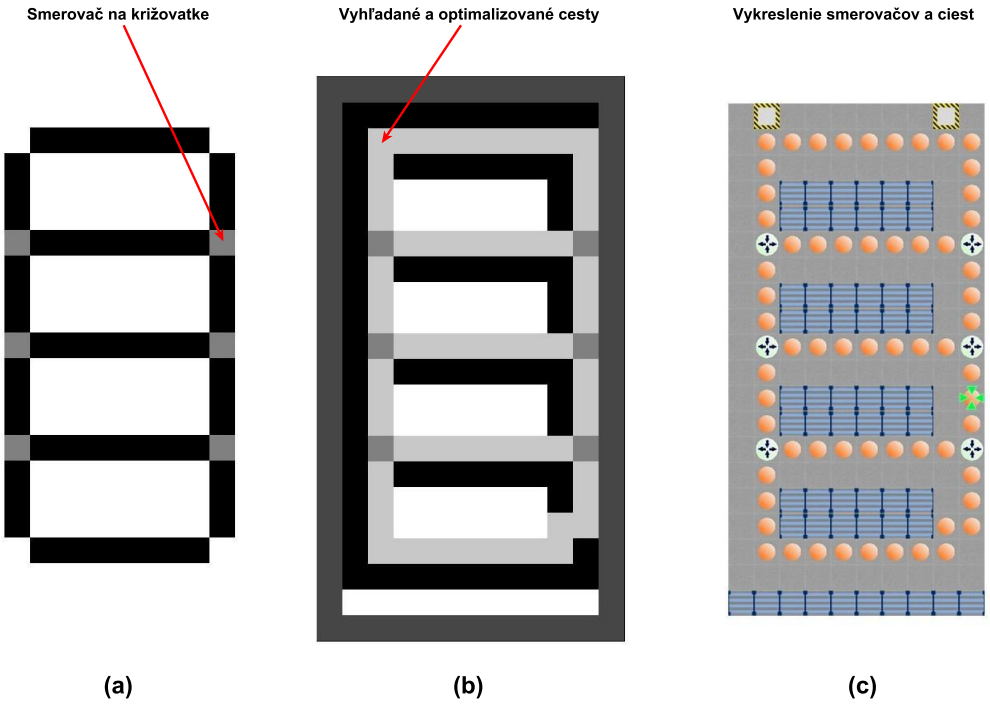
Na detekciu pixela, ktorý je križovatkou – miestom, kde má byť smerovač, musia platiť nasledujúce podmienky:

- Pixel musí ležať na kostre (jeho hodnota musí byť 0 – čierna farba).
- Pixel musí mať dva susedné čierne pixely, ktoré sú navzájom kolmé.

Pseudokód druhej podmienky, ktorá sa v obmenách opakuje pre všetky 4 smery, je:

```
if(pixel(x-1, y)==0) && ((pixel(x, y-1)==0) || pixel(x, y + 1)==0)
```

Cyklom sa prehľadá každý pixel binaryProcessoru a pokiaľ spĺňa tieto podmienky, je to miesto, kde má byť smerovač. Dva a viac smerovačov nikdy nesmú byť priamo vedľa seba. Preto sa pred pridaním smerovača skontroluje všetkých 8 okolitých buniek a v prípade, že ďalší smerovač sa nenachádza v jeho okolí, vytvorí sa nový smerovač, pridá sa do skladu, pixel v binaryProcessoru sa prefarbí a konkrétna bunka v poli skladu sa zmení na typ smerovač – RouterCell (kvôli účelom prehľadávania, viď obr. 2.15). Vyhľadanie smerovačov je znázornené na obr. 2.7 (a).



Obr. 2.7: Vyhľadané smerovače a cesty. (a) zobrazuje kostru so zvýraznenými miestami, kde je smerovač (na každej križovatke). (b) ukazuje vyhľadané cesty spoločne s optimalizáciou chýbajúcich buniek. (c) je vykreslenie nájdených ciest a smerovačov do GUI.

Identifikovanie ciest

Cesta musí vždy začínať v niektorom zo smerovačov. Vyhľadávanie ciest preto začína cyklom, v ktorom sa vyhľadajú bunky ciest okolo každého smerovača (čierne pixely). Akonáhle je takáto bunka nájdená, vytvorí sa nová cesta a pridá do skladu. Trieda cesty má nasledujúce atribúty:

- cena (`int cost`),
- prvý smerovač (`Router firstRouter`),
- druhý smerovač (`Router secondRouter`),
- vektor buniek tvoriacich cestu (`Vector<AbstractStockCell> path`).

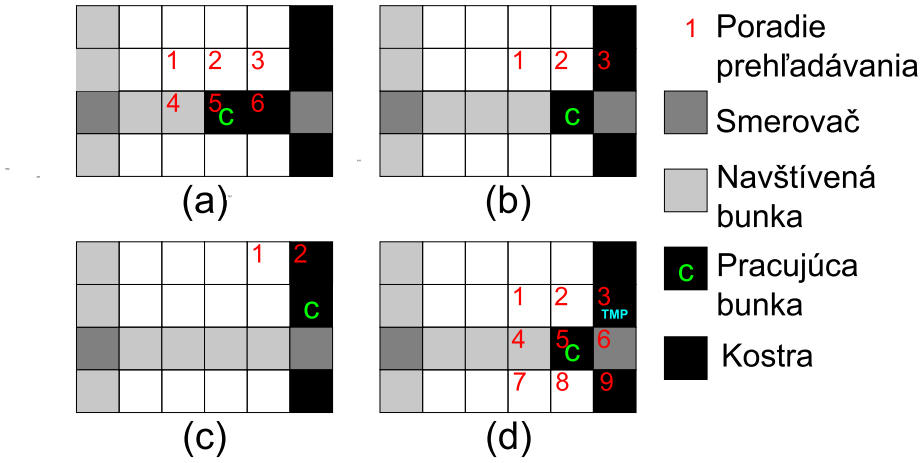
Prehľadávaný smerovač a aktuálna bunka sa pridajú do vektorov. Pomocou BFS algoritmu sa začne vyhľadávať cesta. Je potreba odlíšiť už navštívené bunky (v BFS množina CLOSED), preto aktuálna bunka zmení farbu.

Takisto je potrebné prefarbiť bunky iných ciest vychádzajúce zo smerovača, aby v prvom kroku našiel algoritmus správne pokračovanie cesty a nepridával do cesty bunky, ktoré do nej nepatria. Všetkých 8 susedných buniek okolo smerovača sa prefarbí a uložia sa ich indexy, aby bolo možné ich po nájdení cesty znovu vrátiť do pôvodného stavu.

Z aktuálnej bunky sa generujú dostupné stavy – môže pokračovať len jednou bunkou, kvôli skeletonizácii, pretože v kostre nemôžu byť dve bunky cesty vedľa seba (inak by tu bol smerovač), viď kapitola 2.3.2. Táto bunka je nájdená cyklom, ktorý prehľadáva všetky okolité bunky (obr. 2.8 (a),(d)), z ktorých nájde práve jednu dostupnú, tú pridá do vektoru, prefarbí a pokračuje prehľadávanie z tejto bunky. Proces sa opakuje, pokiaľ sa nenarazí na druhý smerovač (pridá sa do vektoru), alebo nebude možné pokračovať (slepá ulička v sklade). V tomto prípade sa vyčerpajú všetky dostupné stavy a cesta obsahuje iba jeden smerovač. Cyklus hľadania dostupných buniek by normálne mohol končiť v momente nájdenia bunky, pretože v ďalších krokoch sa už na dostupnú bunku nenarazí (cesta nemôže pokračovať dvoma smermi).

Problémom je, ak cyklus natrafí na dostupnú bunku, ktorá však náleží inej ceste. Tento prípad môže nastať len na konci cyklu, keď má byť objavený druhý smerovač, viď obr. 2.8 (b). Keby hľadanie pokračovalo ďalej týmto spôsobom (obr. 2.8 (c)), nájdené cesty by boli úplne nepoužiteľné.

Kvôli zabráneniu tomuto nežiadúcemu stavu sa v jednom cykle generovania dostupných buniek skontroluje všetkých 8 okolitých buniek. Pokiaľ cyklus narazí na bunku cesty, pridá ju do pomocnej premennej TMP (z angl. Temporary – dočasný) a dokončí prehľadávanie okolia bunky, viď obr. 2.8 (d). Ak narazí na smerovač, cesta tu končí, TMP sa zruší a smerovač sa pridá do vektoru smerovačov danej cesty. V opačnom prípade sa TMP pridá do cesty, zmení farbu a pokračuje sa v hľadaní.



Obr. 2.8: Vyhľadávanie ciest v kostre. Obrázky (a),(b),(c) zobrazujú postupné hľadanie cesty cyklom, ktorý po nájdení dostupnej bunky končí. Obrázok (d) ukazuje cyklus, ktorý kontroluje aj okolité bunky. V (a) je správne nájdená dostupná bunka č. 6, cyklus končí a prehľadávanie pokračuje v (b), kde je nájdená dostupná bunka č. 3 avšak inej cesty. Smerovač je preskočený a prehľadávanie pokračuje v (c), kde už nie je možné naraziť na smerovač. V (d) sa v 3. kroku narazí na dostupnú bunku a uloží sa do TMP. Cyklus pokračuje ďalej a narazí na smerovač v 6. kroku, čo znamená, že tu cesta končí.

Počas vyhľadávania ciest sa zároveň počíta aj ich cena – ohodnotenie hrany:

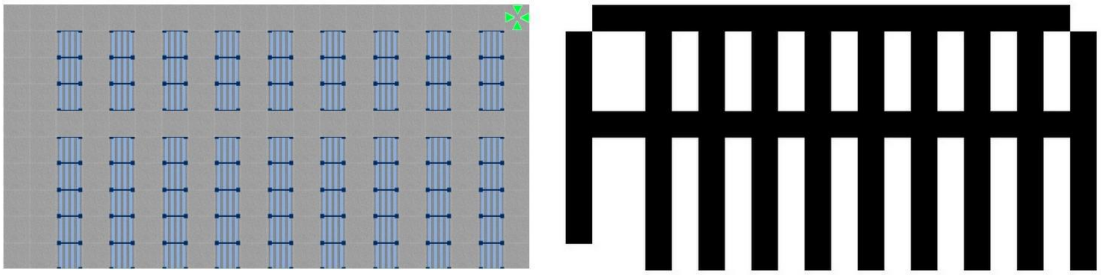
$$c = \frac{d \cdot n}{v} \tag{2.1}$$

kde c je cena, d oddeľovač, n počet buniek v ceste a v rýchlosť v ms^{-1}. Cesty je potrebné ešte optimalizovať (dopočítať chýbajúce bunky), viď obr. 2.7 (b), aby cena cesty vyjadrovala reálnu hodnotu. Rýchlosť je možné v aplikácii meniť. Cena cesty vyjadruje čas, za ktorý je možné danú cestu prejsť.

Pridanie do grafu

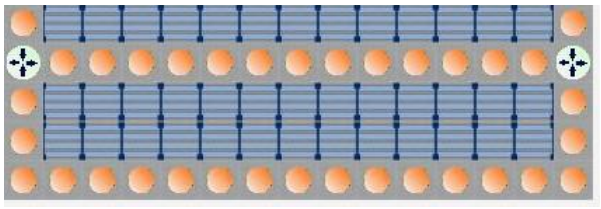
Všetky doteraz nájdené smerovače a cesty boli pridané zatiaľ iba do skladu, nie do grafu. Je to z dôvodu možnej prítomnosti „slepej uličky“ v sklade (cesta má v tomto prípade len jeden smerovač), viď obr. 2.9.

Na danom vrchole by sa vytvorila slučka. Tieto cesty nemá zmysel pridávať do grafu, pretože nikam nevedú (sú nepotrebné pre vyhľadávanie medzi smerovačmi). Musia byť však uchované pre potreby rýchleho vyhľadávania ciest (popísané v 2.4.2).



Obr. 2.9: Slepé uličky v sklade. Po skeletonizácii môžeme vidieť, že v dolnej časti skladu sú slepé uličky, cesty nekončia križovatkami (v smerovačoch).

Do grafu sa pridajú iba cesty s dvojicou smerovačov – vytvoria sa susedia (susedné vrcholy spojené hranou). V mnohých prípadoch môžu medzi vrcholmi vzniknúť rovnobežné hrany, viď obr. 2.10. Po pridaní všetkých ciest, vzniká konečná podoba grafu – **sieť** skladu.



Obr. 2.10: Príklad vzniknutia rovnobežných hrán grafu.

2.4 Vyhľadávanie ciest v sklade

2.4.1 Hľadanie z každého miesta skladu

Vo vytvorenej štruktúre grafu, je už možné vyhľadávať, zatiaľ ale len medzi jeho vrcholmi (z jedného smerovača do niektorého iného). Pre potreby skladu je nutné umožniť vyhľadávanie z ktorejkoľvek pozície (nie len buniek, kde sú smerovače). Kvôli tomuto dôvodu má každá bunka atribúty vektorov najbližších smerovačov a najbližších vzdialeností im odpovedajúcim. Pre rýchlejšie vyhľadávanie sa každej bunke tieto atribúty predpočítajú.

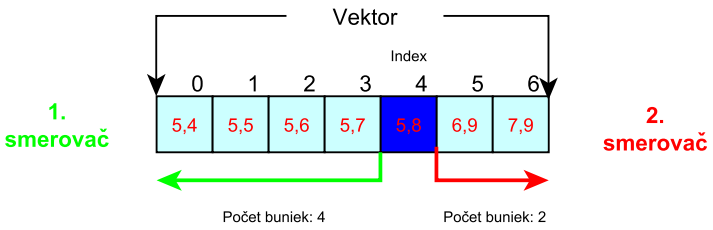
Bunky v cestách

Najjednoduchšie je spracovať bunky zahrnuté v cestách. Pri vyhľadávaní ciest v kostre sa do ich vektorov buniek vkladala každá nájdená bunka, preto je jednoduché

získať poradie bunky v danej ceste a vypočítať vzdialenosť k smerovačom, pretože vektor udržuje poradie vkladaných prvkov, viď obr.2.11.

Algoritmus pre jednu bunku v ceste je nasledujúci:

- Najskôr sa musí identifikovať cesta, do ktorej bunka patrí.
- Ďalej sa musí určiť jej index vo vektore (poradie vloženia).
- Z indexu danej bunky sa určí vzdialenosť k prvému smerovaču a bunka si túto vzdialenosť a smerovač uloží.
- Ak cesta obsahuje aj druhý smerovač, obdobne sa opakuje predchádzajúci krok.
- V prípade slepej uličky algoritmus končí a bunka má iba jeden smerovač.



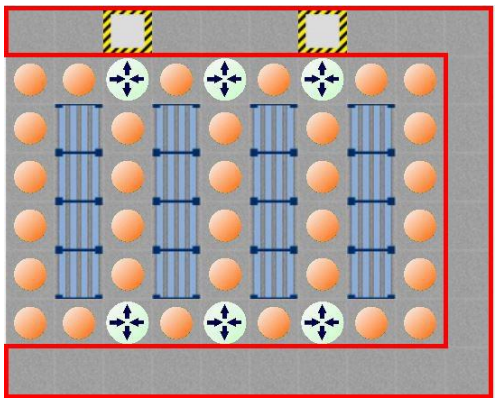
Obr. 2.11: Pracovanie s bunkami vo vektore. Bunka na 4. indexe má k dosiahnutiu 1. smerovača 4 bunky a 2. smerovača 2 bunky. Z tohto počtu buniek sa určí vzdialenosť podľa vzorca (2.1).

Z tohto dôvodu sa nevymazávali cesty s jedným smerovačom, ale používajú sa pre predpočítavanie buniek, ktoré sú v kostre (takýto proces je omnoho jednoduchší ako u nezaradených buniek, viď nasledujúca časť).

Ostatné bunky v sklade

Pomocou metódy skeletonizácie sa našla minimálna kostra, v ktorej sa vyhľadali smerovače a cesty s jednotlivými bunkami. Stále ale zostávajú miesta mimo kostry – bunky, ktoré nie sú zahrnuté v žiadnej ceste, viď obr. 2.12. Tento prípad by nemal nastať v ideálne nakreslenom sklade, je však nutné s ním počítať a z týchto buniek musí byť taktiež umožnené vyhľadávanie ciest.

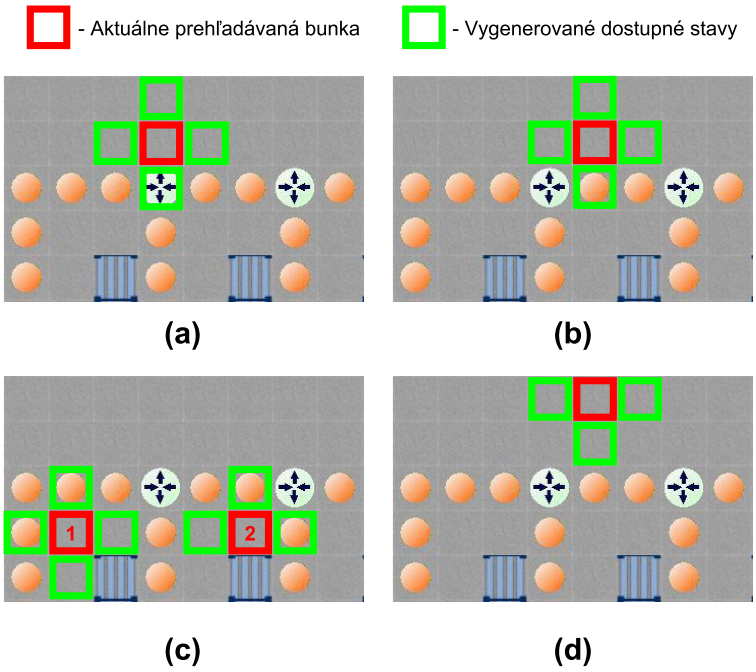
Po ukončení predpočítavania smerovačov a vzdialeností pre bunky ležiace v kostre nasleduje algoritmus, ktorý nastaví tieto parametre aj pre ostatné bunky. Keďže bunky neležia na žiadnej ceste, ktorá má v atribútoch príslušné smerovače, je nutné ich najskôr nájsť. Každá bunka prehľadáva svoje okolie, z dôvodu nájdenia najbližšej cesty alebo smerovača. Vhodný prehľadávací algoritmus je BFS, ktorý expanduje všetky stavy s rovnakou prioritou. Nastáva však podobný problém pri prehľadávaní



Obr. 2.12: Bunky nezahrnuté v cestách (ohraničené červenou farbou), miesta z ktorých taktiež musí byť umožnené vyhľadávanie ciest.

okolia bunky ako u situácie s vyhľadávaním ciest v kostre (obr. 2.8). Je preto nevyhnuté po každom vygenerovaní dostupných buniek skontrolovať všetky tieto bunky, a nie len aktuálne spracovávanú. Generujú sa vždy 4 susedné bunky, aby bolo zaručené, že sú navzájom dosiahnuteľné (pri okolných 8 bunkách by mohlo nastať, že dostupný stav leží uhlopriečne medzi regálmi a tým nie je priamo dosiahnuteľný). Situácia je komplikovanejšia tým, že môže nastať viacero prípadov, kde sa nachádzajú nezahrnuté bunky:

- Priamo nad smerovačom (v dostupných stavoch sa nachádza jeden smerovač a chodby bez cesty). V tomto prípade má bunka iba jeden smerovač a jednu vzdialenosť – neleží medzi dvoma smerovačmi. Obr. 2.13 (a).
- Vedľa smerovača a cesty (príp. dvoch ciest). Môže nastať pri nesprávne nakreslenom sklade – chybnej skeletonizácii (v dostupných stavoch sa nachádza jeden smerovač a jedna cesta). Je nutné iba dohľadať z bunky cesty druhý smerovač a príslušnú vzdialenosť. Ošetrený je samozrejme aj prípad so slepou uličkou.
- Vedľa cesty (v dostupných stavoch sa nachádza len jedna cesta a žiadny smerovač). Bunke môžeme priamo priradiť smerovače a príslušné vzdialenosti danej cesty. Obr. 2.13 (b).
- Medzi dvoma cestami (v dostupných stavoch sa nachádzajú dve cesty a žiadny smerovač). Ak sa jedná o rovnakú cestu, priradí sa kratšia vzdialenosť k jednotlivým smerovačom, na základe vzdialeností zo susedných buniek cesty. Pokiaľ sú to dve rozdielne cesty, musí uhlopriečne ležať smerovač. V tomto prípade sa pridá iba tento jeden smerovač (dosiahnuteľný určite je, keďže sú v jeho okolí cesty). Obr. 2.13 (c).
- V okolí chodieb (v dostupných stavoch sa nenachádza žiadna cesta ani smerovač). Všetky tieto bunky sa pridajú do OPEN fronty a pokračuje sa obdobne v prehľadávaní okolia. Obr. 2.13 (d).

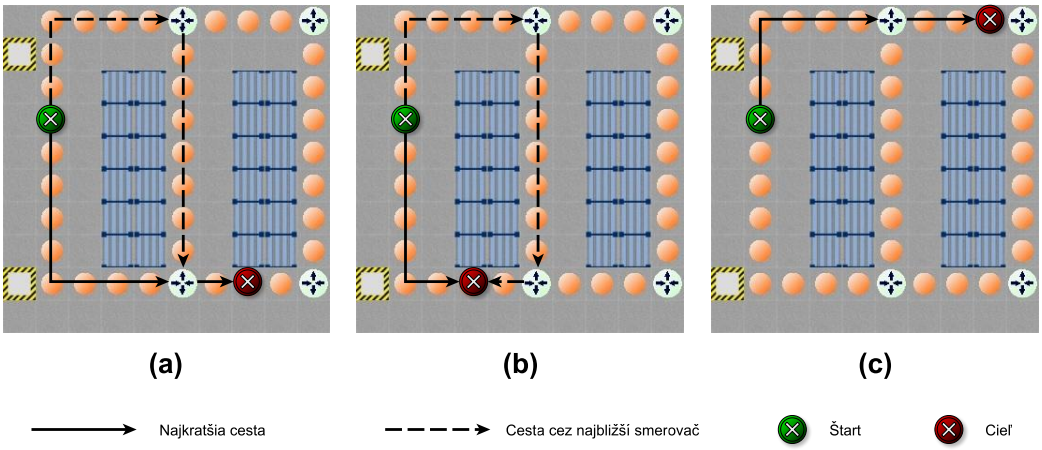


Obr. 2.13: Hľadanie smerovačov bunkám ležiacim mimo ciest pomocou algoritmu BFS, kde (a) zobrazuje prípad, keď bunka susedí so smerovačom, (b) zobrazuje bunku ležiacu priamo vedľa cesty. Obrázok (c) ukazuje situáciu keď vo vygenerovaných dostupných stavoch sú dve bunky cesty. V prípade 1 sa jedná o rovnakú cestu a v prípade 2 o dve rôzne cesty. Obrázok (d) poukazuje na prípad, keď sa bunka nachádza iba v okolí chodieb a algoritmus pokračuje ďalej v prehľadávaní.

Každej priradenej vzdialenosti je nutné pripočítať ešte vzdialenosť od prehľadávanej bunky na základe počtu krokov BFS algoritmu, podľa vzorca (2.1). Po prehľadaní všetkých buniek a priradení najbližších smerovačov je sklad pripravený na vyhľadávanie ciest z ktorejkoľvek pozície.

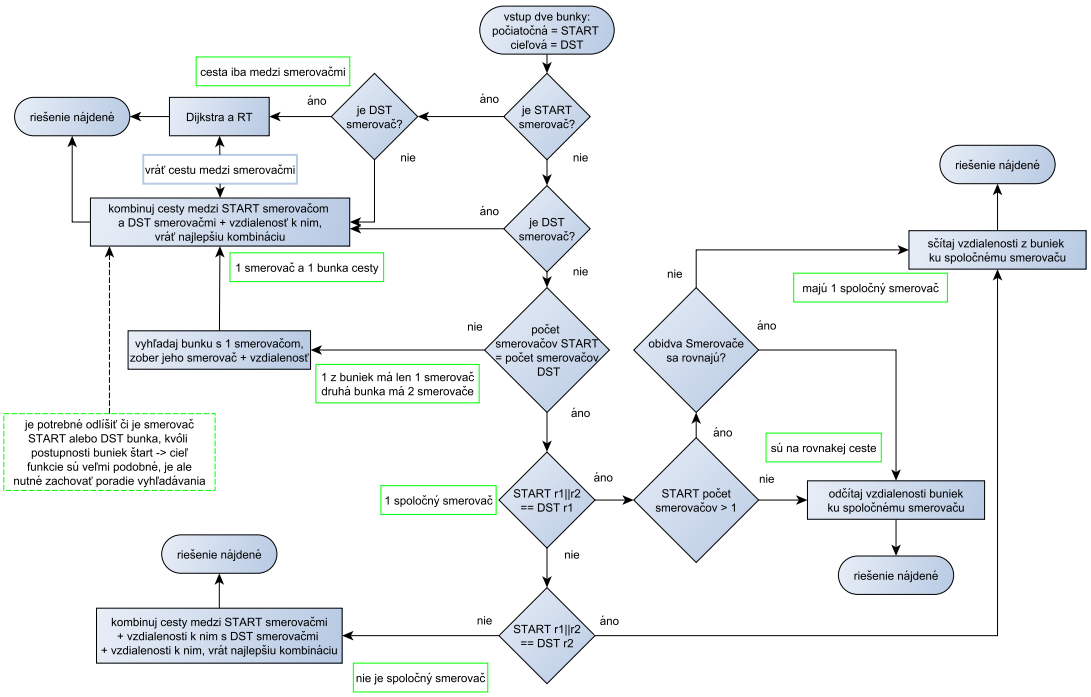
2.4.2 Možnosti vyhľadávania

Bunky majú predpočítané informácie o najbližších smerovačoch a vzdialenostiach k nim. Najbližší smerovač však nemusí vždy viesť k najlepšej ceste, viď obr. 2.14 (a). Preto je nutné, aby každá bunka mala predpočítané obidva smerovače (výnimka – slepá ulička) a prehľadávajú sa všetky kombinácie cez obidva smerovače štart a cieľovej bunky a použije sa najkratšia cesta. Nie všetky cesty musia viesť vždy cez smerovače, viď obr. 2.14 (b), kde štart a cieľ ležia na rovnakej ceste. Taktiež nie vždy k nájdeniu cesty musí byť použitý graf, viď obr. 2.14 (c), kde bunky majú jeden spoločný smerovač (neexistuje kratšia cesta ako cez práve tento smerovač).



Obr. 2.14: Možnosti hľadania ciest v sklade. (a) porovnáva vyhľadanú cestu použitím najbližšieho smerovača a kratšiu cestu cez druhý smerovač. (b) zobrazuje najkratšiu cestu bez použitia smerovačov – cieľ a štart ležia na spoločnej ceste. (c) poukazuje na možnosť vyhľadavánia bez použitia grafu, štart a cieľ majú spoločný smerovač – neexistuje kratšia cesta.

Tieto prípady je nutné odlíšiť v algoritme pre vyhľadávanie ciest v grafe, do ktorého môžu vstupovať rôzne metódy hľadania (z indexu poľa skladu, súradníc, alebo priamo buniek). V telách jednotlivých metód sa všetky typy upravia na jeden tvar – vyhľadávanie z konkrétnych buniek. Algoritmus je znázornený na obr. 2.15.



Obr. 2.15: Algoritmus vyhľadávania ciest v sklade. Kvôli rýchlej identifikácii, či sa vyhľadáva zo smerovača, sa menila bunka na RouterCell pri nájdení smerovača po skeletonizácii.

2.4.3 Smerovacia tabuľka

V prípade, že bunky, medzi ktorými sa hľadá cesta, neležia na rovnakej ceste, alebo nemajú jeden spoločný smerovač, je nutné vyhľadať cestu v štruktúre grafu. Graf sa prehľadáva pomocou Dijkstrovho algoritmu, ktorý nájde najlepšie riešenie. Rýchlosť vyhľadávania však závisí na počte vrcholov grafu (viď graf 3.5), preto je nežiadúce počítať Dijkstrov algoritmus pri každom vyhľadávaní cesty. Problém sa dá vyriešiť pomocou predpočítanej smerovacej tabuľky.

Tá vypočíta cesty pre každú dvojicu smerovačov jedenkrát (po prevode do grafu) na základe Dijkstrovho algoritmu. Cesty sú uložené v hešovacej mape[8]. Pri vyhľadávaní ciest sa iba vyhľadá daný počiatočný smerovač a jeho najlepšia predpočítaná cesta k cieľovému smerovaču. Rýchlosť vyhľadávania ciest je tak výrazne lepšia (viď graf 3.7).

[8]Hešovacia mapa používa hešový kód na celočíselnú identifikáciu objektu a použije ho k rýchlemu vyhľadaniu požadovaného kľúča. Výsledkom je zlepšenie výkonu [14].

2.5 Nepriechodné a obmedzené cesty

2.5.1 Nehody a zablokované cesty.

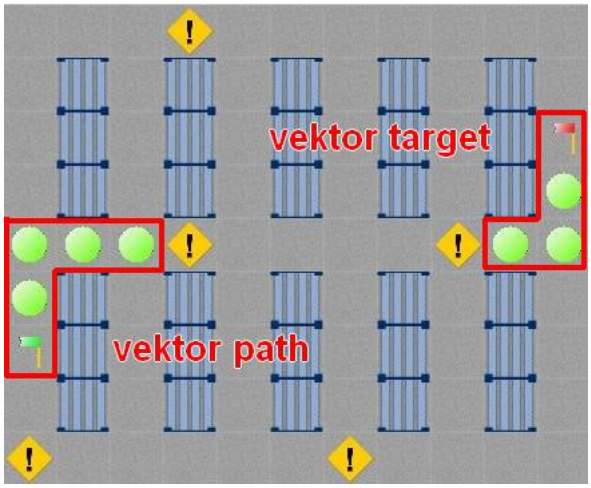
Počas prevádzky skladu je pravdepodobné, že občas nastane nehoda, ktorá môže zablokovať niektorú z ciest. V prípade, že nie je odstránená hneď, je nutné ju zahrnúť do algoritmu pre hľadanie cesty. Podobne môže v sklade prebiehať údržba, opravy, alebo iné elementy, ktoré na určitú dobu zablokujú niektoré z ciest, alebo z iných dôvodov je nutné zmeniť naskladňovacie cesty. Vyhľadaná cesta, ktorou nie je možné prepraviť tovar, je absolútne nežiadúca.

Metóda na pridávanie nehody bola pôvodne vymyslená tak, že po jej pridaní do skladu sa znovu prepočíta smerovacia tabuľka, ktorá sa začne používať na vyhľadávanie ciest. V prípade implementácie optimalizačných algoritmov, ktoré využívajú genetické algoritmy, to môže znamenať značné časové spomalenie, keďže by bolo treba už stávajúci výsledný súbor optimalizácií (ciest a úloh) znovu prepočítať. Navyše nehody môžu nastávať často, prípadne niektoré nehody môžu byť aj rýchlo odstránené, čo taktiež zhoršuje vlastnosti danej metódy. Nehoda by sa do skladu pridala ako nepriechodný objekt, čo by mohlo vytvárať pri skeletonizácii a následnom vytvorení grafu značné problémy, ako je možné vidieť na obrázkoch 2.5 a 2.6.

Metóda na pridávanie nehody je preto implementovaná bez zasahovania do grafu a smerovacej tabuľky. Každá vypočítaná cesta obsahuje v konečnom dôsledku zoznam buniek, po ktorých sa dá dostať z jedného bodu do druhého. V sklade sa vytvorí zoznam buniek, ktoré obsahujú nehodu a po vyhľadaní cesty sa vždy prehľadá tento zoznam, či výsledná cesta neobsaje nehodu (či je cesta priechodná). Algoritmus začne v poradí prehľadávať bunky a porovnáva ich so zoznamom nehôd. Ak neobsahuje bunka nehodu, premiestni sa do vektoru výslednej cesty – **path** a zároveň sa vymaže zo zoznamu buniek vypočítanej cesty. Cyklus pokračuje pokiaľ nie je zoznam prázdny. Ak sa počas cyklu narazí na nehodu, je potrebné cestu prepočítať.

Algoritmus pokračuje vytvorením vektora cieľu cesty – **target** a obdobne premiestňuje bunky do vektoru smerom od cieľa až pokiaľ nenarazí na nehodu. V tomto kroku je nepodstatné, či je to tá istá nehoda ako z predchádzajúceho kroku, alebo akákoľvek iná nehoda, ktorých môže byť v sklade viac. Dôležitý je vektor buniek target, čo je cesta do cieľa bez nehody. V prípade viacnásobných nehôd na ceste sú ostatné bunky nepodstatné (obojstranne vedú k nehode) a sú zo zoznamu vymazané, viď obr. 2.16.

V tomto kroku sú pripravené vektory buniek path a target, medzi ktorými je potrebné nájsť cestu. Z poslednej bunky vektoru path sa začne sklad prehľadávať

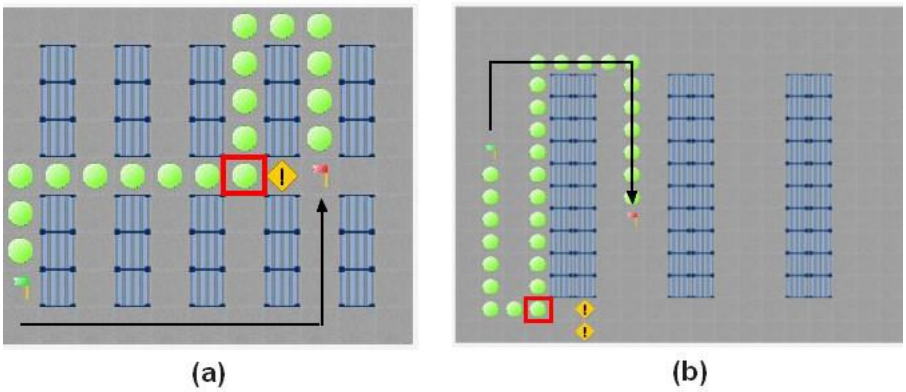


Obr. 2.16: Vektor buniek cesty a target. Zvyšné bunky pôvodnej cesty by boli pre výpočet novej cesty nepodstatné, keďže sú ohraničené nehodami.

BFS algoritmom, pričom každý dostupný stav (bunka chodby bez nehody) si uchováva informáciu o predchádzajúcom stave, kvôli spätnému dohľadaniu cesty (bunka **parent**). Agoritmus BFS pokračuje, pokiaľ nenarazí na niektorú bunku z target vektoru, prípadne sa vyčerpajú stavy a tým pádom neexistuje žiadna cesta, pokiaľ sa neodstráni nehoda. Po narazení na niektorú bunku z target vektora je nutné odstrániť všetky ďalšie bunky v poradí v target vektore (bunky vedú do nehody). Pomocou uloženej informácie o predchádzajúcom stave sa spätne dohľadajú bunky do vektoru cesty, z ktorého je takisto nutné vymazať prípadné bunky vedúce smerom k nehode.

Algoritmus nevyhľadáva cestu priamo zo štart bunky do cieľovej bunky z dôvodu časovej náročnosti, ktorá s rastúcim počtom buniek taktiež rastie. Zníženie tejto časovej náročnosti na nevyhnutné minimum je dosiahnuté práve vektorom buniek target a vyhľadávania od poslednej priechodnej bunky z path vektoru. Táto pôvodná metóda však nepracuje správne pre dva prípady, u ktorých je nevýhodou práve začiatok prehľadávania od poslednej priechodnej bunky path vektora. Vyhľadaná cesta kvôli tomu nie je najlepšia možná. Tieto prípady sú zobrazené a popísané na obr. 2.17.

Aby sa zamedzilo nesprávnemu prepočítaniu cesty, pôvodná metóda bola upravená. Vektory buniek path a target stále zostávajú, pokiaľ ale cesta obsahuje nehodu, vektor buniek path sa vymaže, vloží sa iba štart bunka a algoritmus BFS začne prehľadávať novú cestu z tejto bunky. Ostatné časti metódy zostávajú nemenné.



Obr. 2.17: Nesprávne prepočítané cesty s nehodou. Červeno ohraničené bunky znázorňujú poslednú priechodnú bunku path vektora, z ktorej začal algoritmus BFS. Čiernou je naznačená najkratšia cesta. V prípade (b) je navyše možné vidieť nevýhodu path vektora, ktorý zahŕňa do cesty aj bunky z pôvodnej časti priechodnej cesty.

Táto metóda má ďalšiu výhodu, ktorou je znalosť pozície pracovníkov skladu, ktorí svojím spôsobom taktiež tvoria prekážku v ceste – nepriechodný objekt pre iného pracovníka skladu. S touto metódou je možné v optimalizačných algoritmoch predikovať pozíciu pracovníkov skladu a zahrnúť ich do výpočtu ciest. Ukážka tejto funkcionality je v sekcii výsledkov na obr.3.9.

2.5.2 Obmedzené cesty

Nehoda znamená nepriechodnosť určitou časťou skladu. Môže však nastať situácia, kedy je priechodnosť určitého miesta v sklade nejakým spôsobom obmedzená, ale nie nemožná (napr. miesto, kde sa kompletizujú objednávky, miesto, kadiaľ sa presúvajú zamestnanci k výrobným linkám, alebo aj časť pri regáli, kde môže byť dočasne umiestnený niektorý tovar). V tomto prípade sa nemôže miesto označiť nehodou, ktorá by úplne vylúčila prechod týmto miestom pri vyhľadávaní cesty. Nastáva situácia, kedy je prechod nežiadúci, ale v krajných prípadoch možný. Z tohto dôvodu je v aplikácii možná zmena ceny jednotlivých ciest. Týmto spôsobom je taktiež možné zvýhodniť niektoré cesty, alebo jednoducho užívateľsky nadefinovať základné pravidlá pre vyhľadávanie ciest. Je však nevyhnutá úprava grafu a následný prepočet smerovacej tabuľky. V grafe sa musí zmeniť cena pre obidva smerovače danej cesty na novú hodnotu. Ukážka tejto funkcionality je v sekcii výsledkov na obr. 3.8.

```
r1.setCost(r2, cost);
r2.setCost(r1, cost);
```

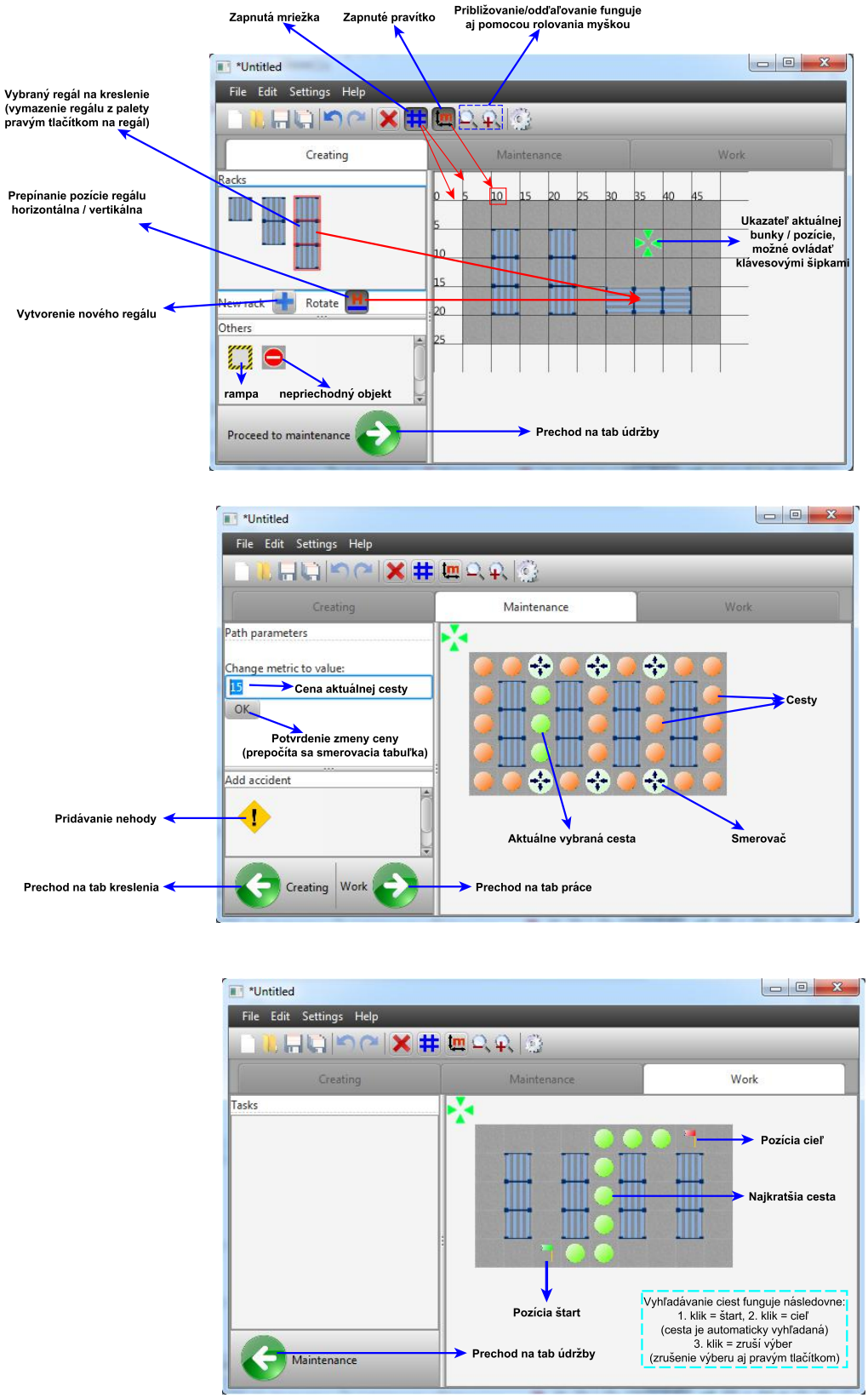
3 VÝSLEDKY PRÁCE

V práci bola vytvorená aplikácia zobrazená na obrázku 3.1, kde je znázornený aj jednoduchý popis ovládania. Aplikácia umožňuje nakreslenie skladu, jeho prevod do dátového modelu, identifikovanie miest, kde majú byť smerovače (križovatky), a ciest medzi nimi a následne vytvorenie grafovej štruktúry, v ktorej je možné rýchlo vyhľadávať cesty z ľubovoľných pozícií skladu. Aplikácia je navrhnutá pomocou návrhového vzoru MVC (z angl. Model View Controller), podľa ktorého je architektúra aplikácie rozdelená do nezávislých vrstiev, čo umožňuje jednoduchú modulárnosť a škálovateľnosť [29].

GUI je tzv. udalosťami riadené(z angl. event driven[1]), ale celá riadiaca logika je pre užívateľa skrytá. Užívateľ je schopný sklad iba nakresliť, obsluhovať (meniť ceny ciest) a vyhľadávať v ňom. GUI obsahuje základné ovládacie prvky, akými sú uloženie/otvorenie projektu, úpravy krok späť/dopredu (fungujúce aj pomocou klávesových skratiek), validačné hlášky (napr. ukončenie aplikácie s neuloženým projektom) a iné.

Aplikácia bola vytvorená na platforme **JavaFX** (pomocou nástroja JavaFX Scene Builder), v ktorej je možné navrhovať, vytvárať, testovať a nasadzovať interaktívne aplikácie s jednoduchou prístupnosťou a použiteľnosťou z hľadiska užívateľa. Aplikácie vytvorené v JavaFX sa chovajú rovnako na všetkých bežne dostupných operačných systémoch [11], [32], [6].

[1]Event driven GUI – Užívateľ môže kedykoľvek interaktívne ovládať GUI a tým je činnosť programu riadená sériami udalostí [19].

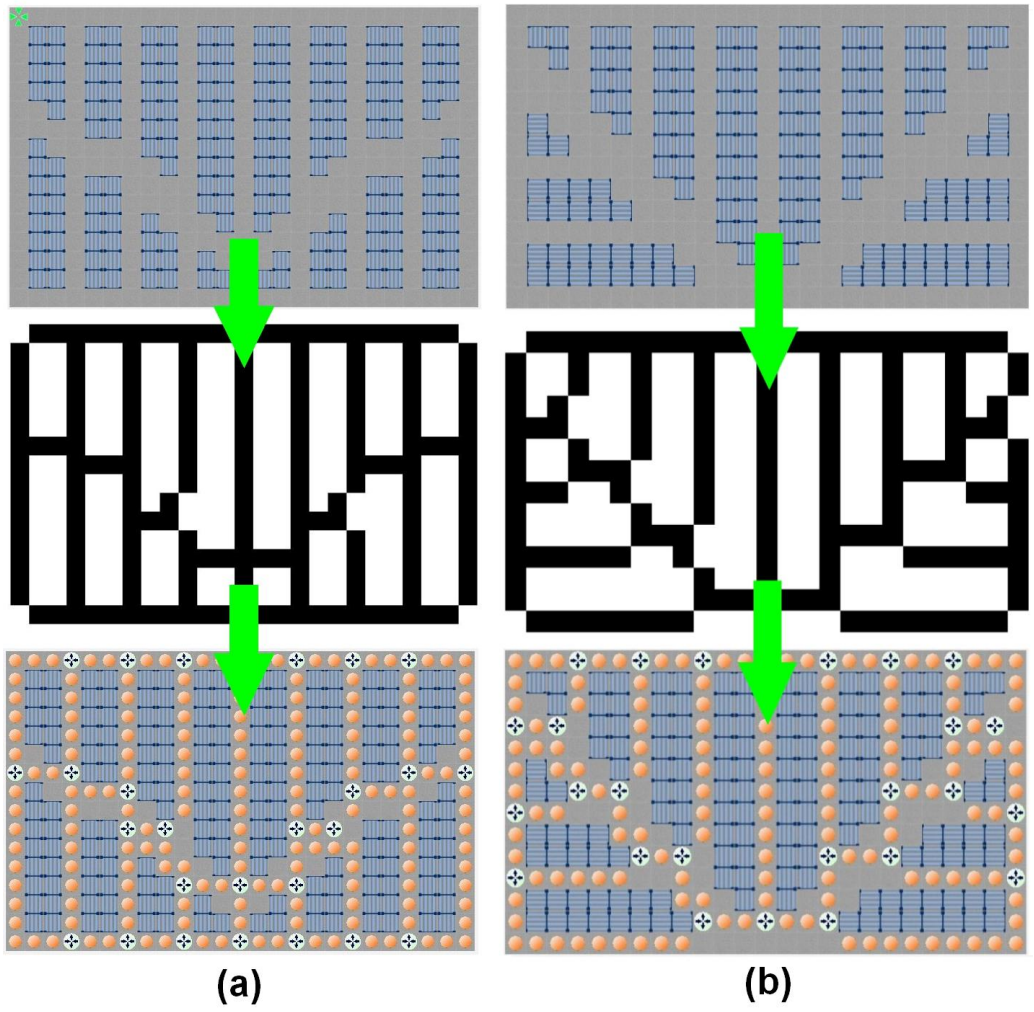


Obr. 3.1: Vytvorená aplikácia so základným popisom GUI.

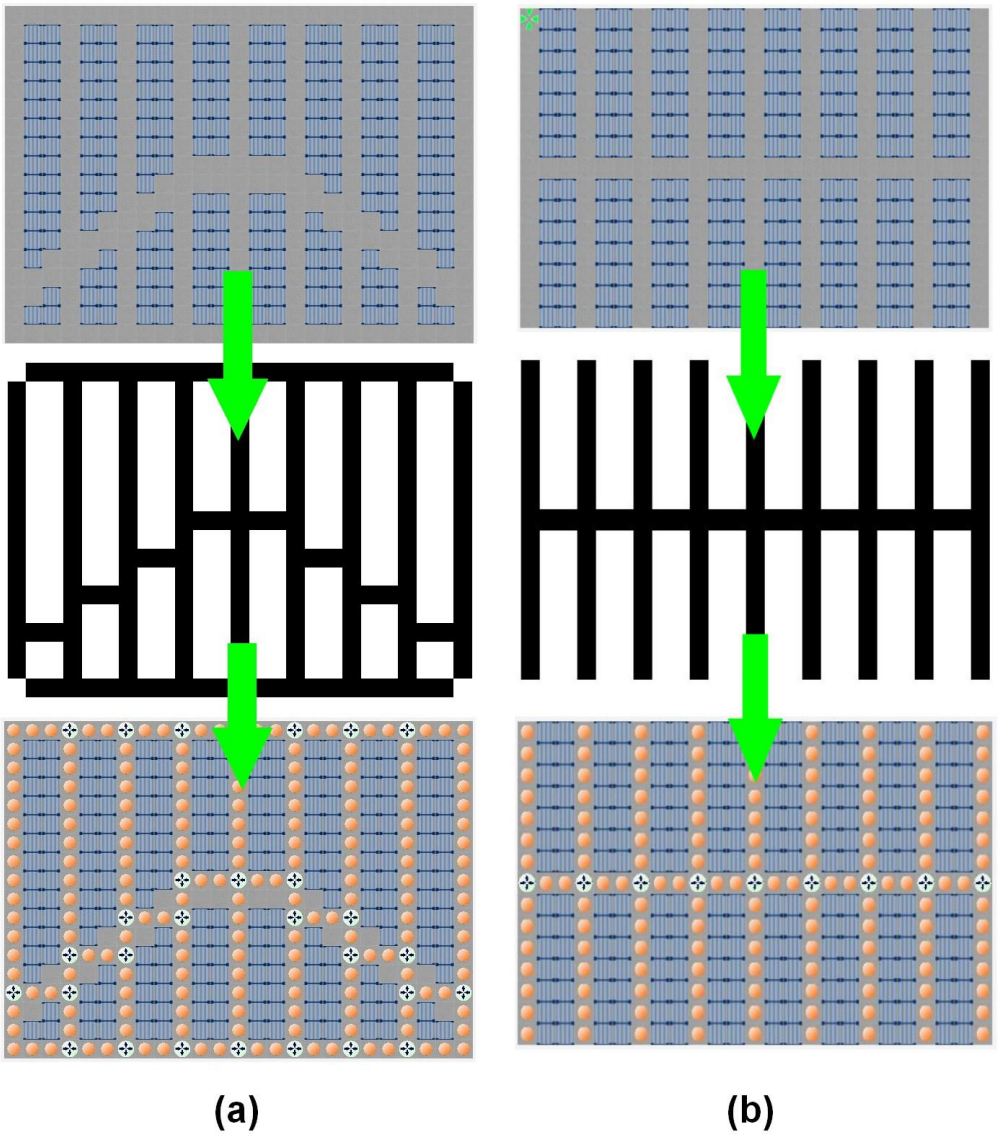
3.1 Prevod skladu do podoby grafu

Kroky procesu prekonvertovania skladu do podoby grafu (overenie správnosti procesu) je možné zapnúť v nastaveniach aplikácie (Settings). Kroky zobrazujú jednotlivé podoby byte/binary processoru spolu so skeletonizáciou.

Správne vytvorenie štruktúry grafu bolo overené počas samotného vývoja aplikácie, pri každom meraní vyhľadávania ciest a testovania konkrétne tejto funkcionality. Ukážka funknčnosti pre hlavné skladové štruktúry (mimo tradičného dizajnu) je zobrazená na obrázkoch 3.2 a 3.3.

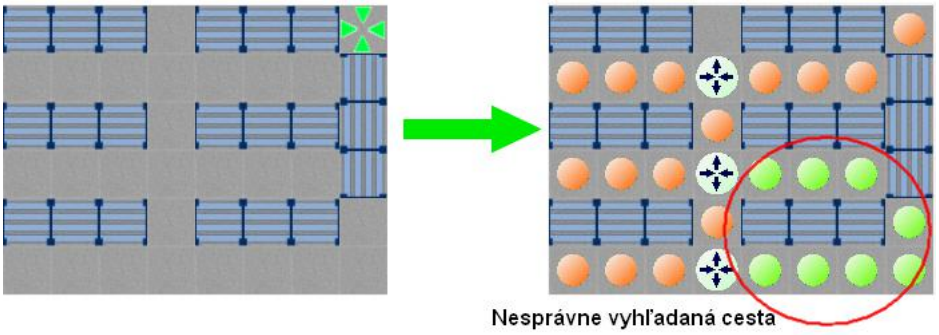


Obr. 3.2: Prekonvertovanie skladov typu „lietajúce – V“ (a) a „rybia kosť“ (b) do grafovej štruktúry.



Obr. 3.3: Prekonvertovanie skladu typu „obrátené – V“ (a) a skladu so slepými uličkami (b) do grafovej štruktúry.

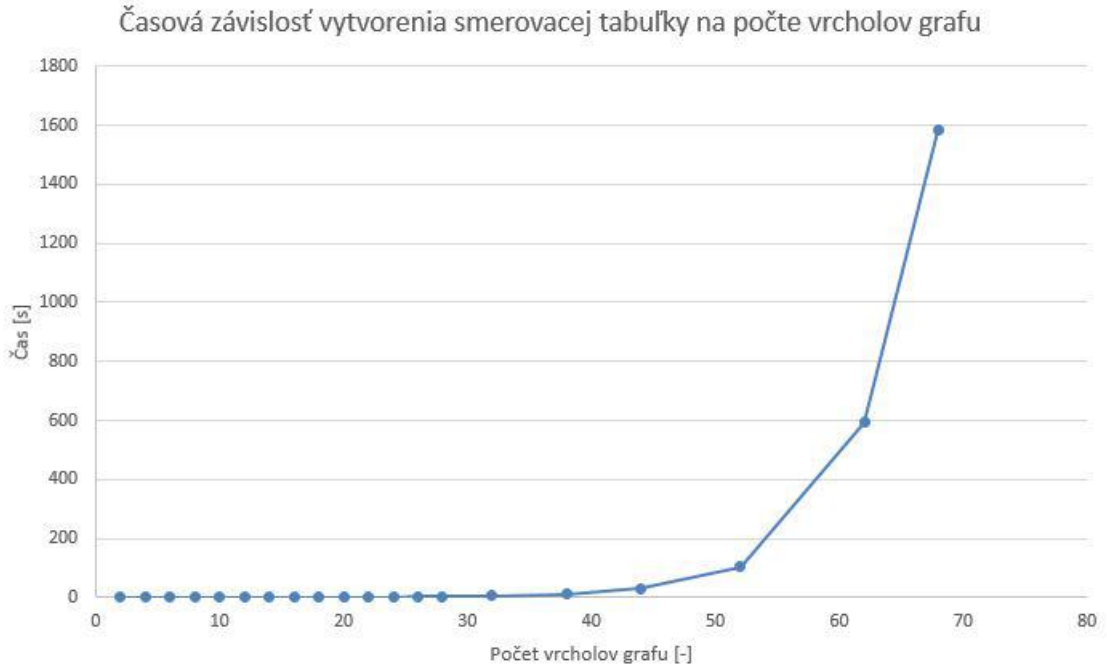
Zo všetkých testov (cca 1000) nastala iba jedna chyba a to konkrétne v jednom prípade rozloženia regálov v sklade, ktoré pravdepodobne v reálnom prostredí nenastane, viď obr. 3.4. Pre hlavné skladové štruktúry a prakticky pre všetky rôzne nakreslené sklady, ktoré by sa mohli v reálnom prostredí vyskytnúť, je chybovosť **nulová**.



Obr. 3.4: Chyba pri identifikovaní cesty. Nesprávne identifikovaná cesta, ktorá prechádza uhlopriečne medzi dvoma regálmi, čo má za následok nesprávne vytvorenie grafu.

3.2 Vyhľadávanie ciest

Z dôvodu rýchleho vyhľadávania ciest v sklade, je nutné najskôr predpočítať smerovaciu tabuľku. Na jej základe môžu prebiehať ďalšie merania vyhľadávania ciest. Merania boli vykonané na počítači s procesorom `AMD A6-3420M APU with Radeon(tm) HD Graphics 1,50 GHz`. Časová závislosť vytvorenia smerovacej tabuľky na veľkosti grafu – počte vrcholov, je znázornená v grafe 3.5.

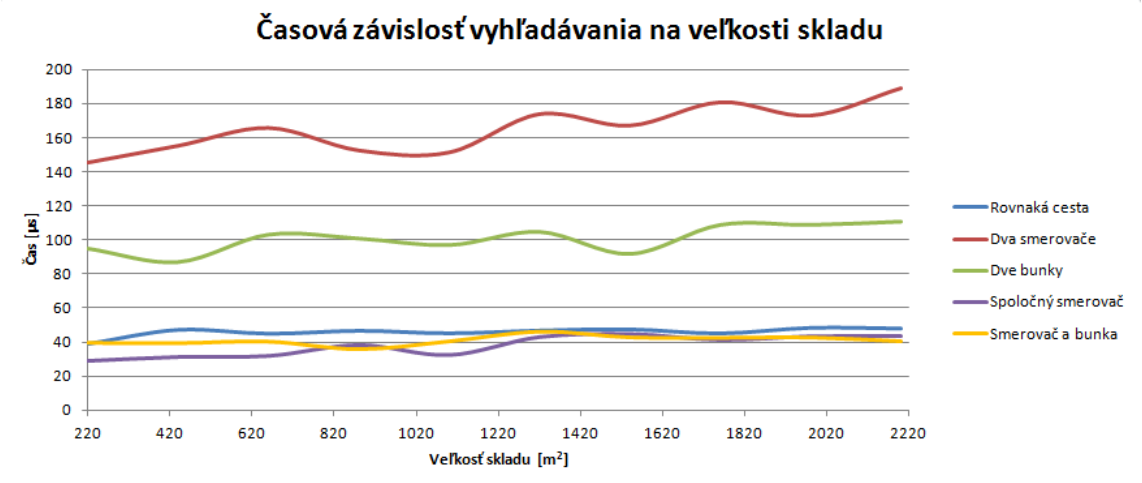


Obr. 3.5: Časová závislost vytvorenia smerovacej tabuľky na počte vrcholov grafu.

Ako je spomenuté v časti 2.4.2, rozlišujú sa rôzne typy vyhľadávania ciest v sklade:

- na **rovnakej ceste**,
- cez **spoločný smerovač**,
- štart a cieľ sú **dva smerovače**,
- štart a cieľ sú **dve bunky**, ktoré ležia na rôznych cestách,
- jedna bunka je **smerovač** a druhá **bunka** leží na ceste.

Vyhľadávanie na rovnakej ceste a cez spoločný smerovač nevyužívajú smerovaciu tabuľku. Tieto prípady sú včas detekované algoritmom na vyhľadávanie ciest a majú vytvorené osobitné metódy pre vyhľadávanie ciest. Po predpočítaní smerovacej tabuľky boli vykonané ďalšie merania, ktoré ju pre svoje fungovanie využívajú. Výsledok merania dôb vyhľadávania ciest v závislosti na veľkosti skladu (pri rovnakom počte vrcholov) zobrazuje graf 3.6. V grafe sú vynesené aj závislosti pre typ vyhľadávania na rovnakej ceste a cez spoločný smerovač, kvôli porovnaniu s typmi, ktoré využívajú smerovaciu tabuľku.



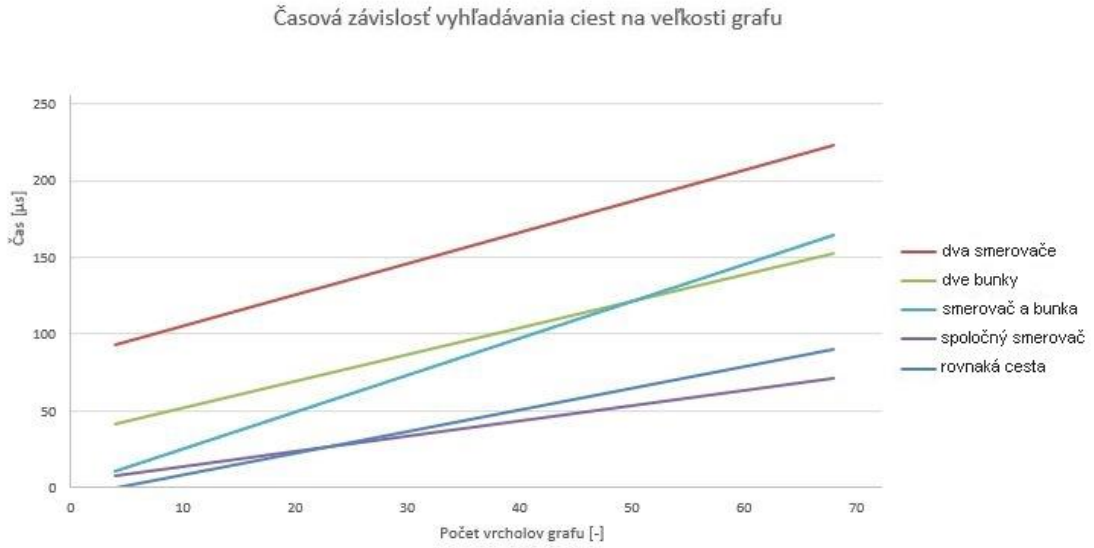
Obr. 3.6: Časová závislost vyhľadávania na veľkosti skladu pri rovnakom počte vrcholov v grafe (8 vrcholov). Pre každý typ vyhľadávania vo všetkých veľkostiach skladu bolo použitých 500 testovacích vzoriek a v grafe použitý ich priemer. Závislosti sú takmer konštantné, takže veľkosť skladu nemá vplyv na dobu vyhľadania cesty.

Porovnanie časov vyhľadávania pre všetky typy je taktiež zobrazené v tabuľke 3.1.

	rovnaká cesta	dva smerovače	dve bunky	spoločný smerovač	smerovač a bunka
max. (µs)	133,956	871,200	4579,911	101,200	198,000
min. (µs)	4,889	19,556	13,689	4,888	4,400
avg. (µs)	13,702	100,021	91,312	91,312	31,856
med. (µs)	10,756	84,333	37,155	12,222	26,889

Tab. 3.1: Porovnanie rýchlosti jednotlivých typov vyhľadávania. V každom jednotlivom meraní bolo 500 vzorkov. Táto tabuľka meraní pripadá na graf s počtom vrcholov 20.

Dôležitá závislosť však je vzhľadom na veľkosť grafu a nie skladu samotného. Veľkosť grafu závisí hlavne od počtu jeho vrcholov. Počet vrcholov v grafe sa rovná počtu križovatiek v sklade. Výsledok merania dôb vyhľadávania ciest v závislosti na veľkosti grafu (počte vrcholov) zobrazuje graf 3.7.



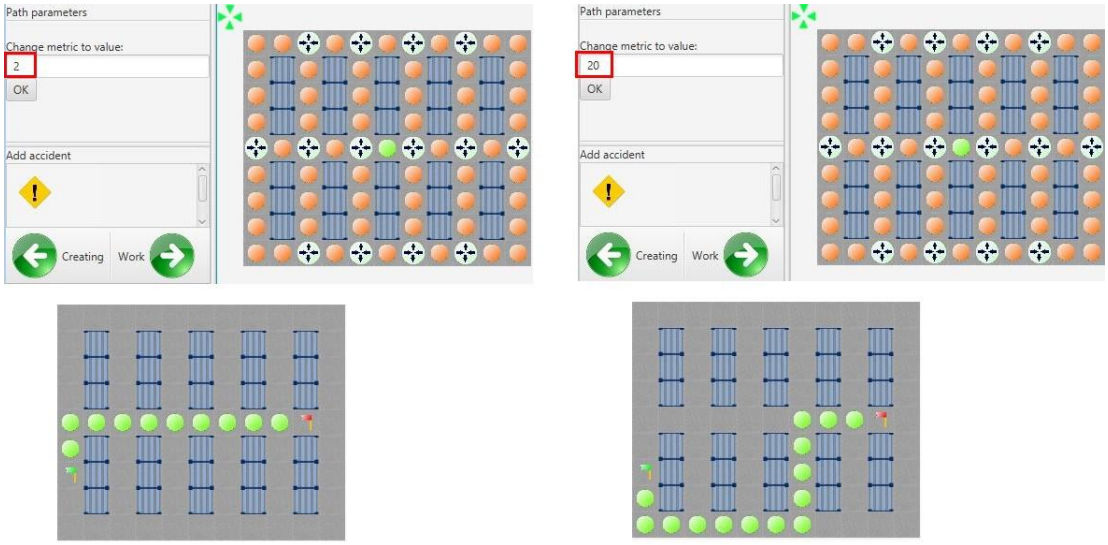
Obr. 3.7: Časová závislosť vyhľadávania ciest na veľkosti grafu. Pre každý typ vyhľadávania pre všetky počty vrcholov bolo použitých 500 testovacích vzoriek a v grafe použitý ich priemer. Závislosti sú preložené lineárne.

Z grafu je zreteľné, že závislosť je lineárna, s narastajúcim počtom vrcholov rastie aj čas vyhľadávania. Čas vyhľadávania sa ale pohybuje v rádoch stoviek mikrosekúnd (ovplyvnené procesorom), čo je stále o celý rad lepšie, ako požadované milisekundy. V tabuľke 3.2 sú porovnavné časy vyhľadávania ciest, pre jednotlivé typy skladov.

Typ skladu	Maximum (µs)	Minimum (µs)	Priemer (µs)	Medián (µs)	Počet vrcholov	Výpočet s. tab. (ms)
Tradičný	7167,112	10,755	160,133	128,578	28	2,325
Lietajúce V	1546,844	8,800	148,610	118,311	27	4,557
Rybia kosť	1570,800	11,733	127,027	117,333	26	3,645
Obrátené V	1955,067	12,222	156,114	124,910	29	3,650

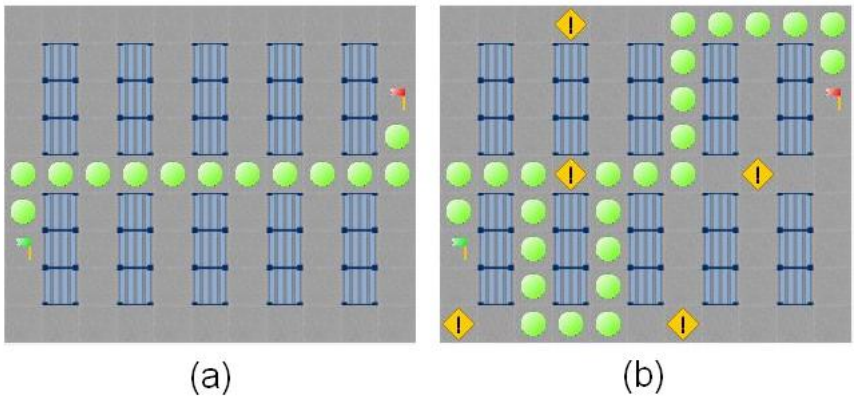
Tab. 3.2: Porovnanie rýchlosti vyhľadávania v jednotlivých typoch skladov. Boli použité dizajny z obrázkov 3.2 a 3.3. V každom jednotlivom meraní bolo použitých 500 náhodne vygenerovaných vzoriek ciest.

V práci bola taktiež overená funkčnosť menenia ceny ciest (popis v sekcii 2.5.2), ktorá pracuje bez chýb, ako napr. na obr. 3.8. Nevýhodou tejto metódy je však nutnosť prepočítavania smerovacej tabuľky pri každej zmene ceny cesty.

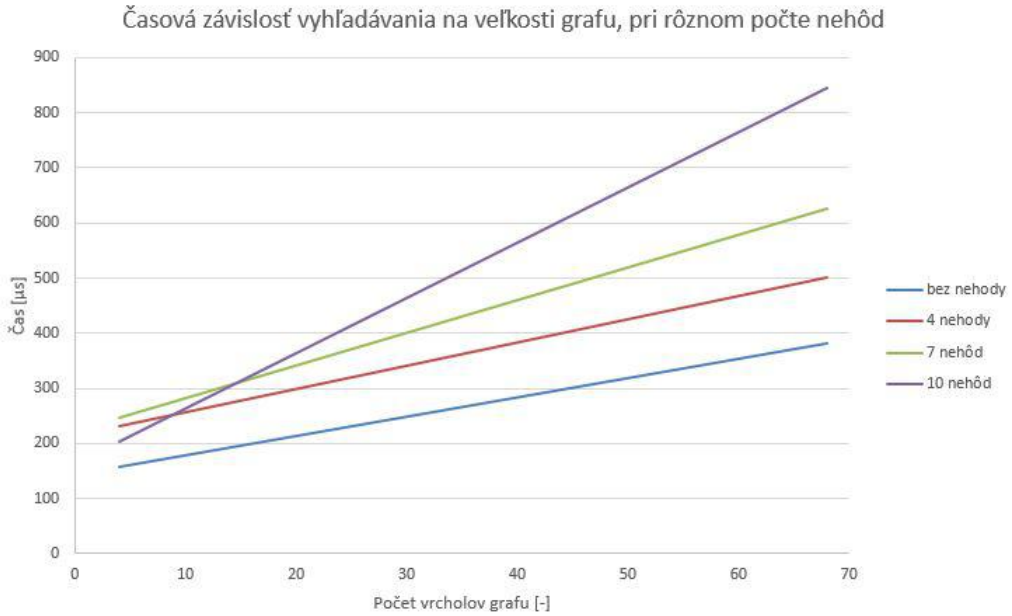


Obr. 3.8: Zmena ceny cesty v sklade. Obrázok (a) hore znázorňuje originálnu cenu cesty a dole príslušnú vyhľadanú cestu. Obrázok (b) hore ukazuje zmenenú cenu cesty a dole vyhľadanú cestu po zmene ceny.

Vyhľadávanie ciest v sklade obsahujúcom nehody, funguje taktiež správne pre každý prípad (príklad obr. 3.9). Pokiaľ nie je možné sa dostať z jedného miesta skladu do druhého kvôli nehode, cesta končí na najbližšej nehode. Časová závislosť vyhľadávania s rôznymi počtami nehôd je zobrazená na obr. 3.10.



Obr. 3.9: Vyhľadanie cesty v sklade s nehodami. Obrázok (a) znázorňuje vyhľadanú cestu bez nehôd. Obrázok (b) ukazuje prepočítanú cestu medzi nehodami.



Obr. 3.10: Časová závislost vyhľadávania ciest na veľkosti grafu, pri rôznom počte nehôd v sklade. Pre každý typ vyhľadávania pre všetky počty vrcholov bolo použitých 500 testovacích vzoriek a v grafe použitý ich priemer. Závislosti sú preložené lineárne.

4 ZÁVER

Optimalizačné metódy potrebujú pre svoju funkčnosť určovať vzdialenosti a cesty medzi ľubovoľnými pozíciami skladu. Tieto metódy môžu implementovať genetické algoritmy, u ktorých je výpočetný čas kľúčový. Aby optimalizácie skladových činností boli efektívne, je nutné umožniť toto vyhľadávanie v čo najkratšom čase.

V diplomovej práci bola vytvorená aplikácia, pomocou ktorej je možné nakresliť akýkoľvek sklad, následne ho previesť do grafovej reprezentácie bez nutnosti zásahu ľudského faktora. Aplikácia ďalej umožňuje meniť ceny jednotlivých ciest, čím sa vo vyhľadávaní môžu zvýhodniť, alebo znevýhodniť oproti ostatným. Podobne je možnosť pridávania nehôd, čím sa cesta úplne zablokuje. Aplikácia ďalej umožňuje rýchle vyhľadávanie najlepších ciest medzi ľubovoľnými pozíciami v sklade. Hlavným prínosom práce je samotný proces prekonvertovania nakresleného skladu do podoby grafu, ktorý využíva metódu skeletonizácie z oblasti obrazového spracovania dát. Táto metóda zjednoduší obraz skladu a vytvorí jeho kostru, v ktorej sa pomocou ďalších algoritmov identifikujú vrcholy a hrany grafu. Nad štruktúrou grafu sa vytvorí smerovacia tabuľka na základe Dijkstrovho algoritmu, ktorá predpočíta cesty medzi všetkými dvojicami vrcholov. Pomocou týchto predpočítaných ciest je možné vyhľadávať v sklade v rádoch stoviek mikrosekúnd.

V práci boli vykonané testy a merania, popísané v sekcii výsledkov, ktoré dokazujú funkčnosť prevodu nakresleného skladu do grafu, rýchle vyhľadávanie ciest, zvýhodňovania niektorých ciest oproti iným a vyhľadávanie alternatívnych ciest v prípade zablokovanej cesty.

Pri vyhľadávaní ciest nie je podstatná samotná rýchlosť vyhľadávania, ktorá je ovplyvnená procesorom, ale či má vplyv veľkosť grafu na čas vyhľadania cesty. Dôležitým prínosom a výsledkom tejto práce je preto aj časová závislosť rýchlosti vyhľadávania ciest na veľkosti grafu, ktorá je lineárna.

LITERATÚRA

[1] ABDULLAH S. N. H. S., ABU-AIN W., KHAIRUDDIN O., *A simple iterative thinning algorithm for text and shape binary images.* Journal of Theoretical and Applied Information Technology, Vol. 63 No. 2, 20. 5. 2014 ISSN: 1992-8645. Dostupné z URL: <http://www.jatit.org/volumes/Vol63No2/5Vol63No2.pdf>.

[2] *Binary Images* [online]. 1997 [cit. 1. 11. 2015]. Dostupné z URL: <http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT2/node3.html>.

[3] BOROVANSKY, P. *Triedy a objekty (na príkladoch)* [online]. Univerzita Komenského v Bratislave. Katedra aplikovanej informatiky. [cit. 28. 10. 2015]. Dostupné z URL: <http://dai.fmph.uniba.sk/courses/JAVA/Prednasky/03_java.pdf>.

[4] BURGER, W., BURGE M. J. *Digital Image Processing. An Algorithmic Introduction using Java.* First Edition 2008 ISBN 978-1-84628-379-6, e-ISBN 978-3-540-30941-3. Dostupné z URL: <https://books.google.bg/books?id=jCEi9MVfxD8C&printsec=frontcover&hl=sk#v=onepage&q&f=false>.

[5] BURGET, R. *Teoretická informatika.* Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, 2013. ISBN: 978-80-214-4897-1

[6] CASTILLO C. *JavaFX Scene Builder Getting Started with JavaFX Scene Builder Release 2.0.* April 2014. E51278-01. Dostupné z URL: <https://docs.oracle.com/javase/8/scene-builder-2/JSBGS.pdf>.

[7] *Class Vector<E>* [online]. 2014 [cit. 1. 11. 2015]. Dostupné z URL: <http://docs.oracle.com/javase/7/docs/api/java/util/Vector.html>.

[8] *CME 305: Discrete Mathematics and Algorithms* [online]. [cit. 1. 11. 2015]. Dostupné z URL: <http://web.stanford.edu/class/cme305/Notes/2.pdf>.

[9] COCHRAN W., O. *Dijkstra's Algorithm.* 20. 11. 2014. Dostupné z URL: <http://ezekiel.vancouver.wsu.edu/~cs223/lectures/graphs/search/dijkstra/dijkstra.pdf>.

[10] ČADA, R., KAISER, T., RYJÁČEK, Z. *Diskrétní matematika.* Západočeské univerzita v Plzni. Katedra matematiky FAV, 2004. Dostupné z URL: <http://www.cam.zcu.cz/~ryjacek/students/DMA/skripta/dma.pdf>.

[11] ČÍKA, P., ZUKAL M. *Multimediální služby – počítačová cvičení.* Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, 2013. ISBN: 978-80-214-4721-9

[12] DEY, T. K., ZHAO, W. *Approximating the Medial Axis from the Voronoi Diagram with a Convergence Guarantee.* Department of CIS, Ohio State University, Columbus, OH 43210, USA. Dostupné z URL: <http://web.cse.ohio-state.edu/~tamaldey/paper/medial.pdf>.

[13] Dharmapriya, U. S. S., Kulatunga, A. K. *New Strategy for Warehouse Optimization – Lean warehousing.* Proceedings of the 2011 International Conference on Industrial Engineering and Operations Management. Kuala Lumpur, Malaysia, 20. – 22. 1. 2011. Dostupné z URL: <http://ieomsociety.org/ieom2011/pdfs/IEOM076.pdf>.

[14] ECKEL, B. *Myslíme v jazyku Java.* edice: Knihovna programátora.

[15] ELDER, J. *Priority Queues & Heaps.* York university. CSE 2011. Dostupné z URL: <http://www.eecs.yorku.ca/course_archive/2011-12/W/2011/lectures/07%20Priority%20Queues%20and%20Heaps.pdf>.

[16] FELNER, A. *Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm.* Information Systems Engineering, Ben-Gurion University, Be'er-Sheva, Israel 85104. Proceedings, The Fourth International Symposium on Combinatorial Search (SoCS-2011). Dostupné z URL: <https://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/viewFile/4017/4357>.

[17] *geneticprogramming.com* [online]. 2013 [cit. 25. 10. 2015]. The GP tutorial. Dostupné z URL: <http://www.geneticprogramming.com/Tutorial/>.

[18] *Graph Theory* [online]. Mathematics learning centre. [cit. 1. 11. 2015]. Dostupné z URL: <http://www3.ul.ie/~mlc/support/CompMaths2/files/GraphTrees.pdf>.

[19] *Graphical User Interfaces* [online]. Chapter 14. 2. 10. 2010. Dostupné z URL: <http://www.buildingjavaprograms.com/samples/3ed/ch14-2ed-gui-sample.pdf>.

[20] GURAL, A. *Basic Graph Theory (BFS, DFS, and Floodfill)* [online]. 10. 2011 [cit. 1. 11. 2015]. Dostupné z URL: <https://activities.tjhsst.edu/sct/lectures/1112/bfsdfs100711.pdf>.

[21] HLINENY, P. *Teorie Grafů (FI: MA010).* Masarykova univerzita. Fakulta informatiky, Brno, 14. 8. 2012. Dostupné z URL: <http://www.fi.muni.cz/~hlineny/Vyuka/GT/Grafy-text07.pdf>.

[22] HOLZNER S. *Design Patterns For Dummies®.* 2006. ISBN-13: 978-0-471-79854-5. Dostupné z URL: <http://www.cs.uah.edu/~rcoleman/CS307/Announcements/Design_Patterns_For_Dummies.pdf>.

[23] Hordějčuk, V. *Teorie grafů* [online]. [cit. 21. 10. 2015]. Dostupné z URL: <http://voho.cz/wiki/graf/>.

[24] JIROVSKÝ, L. *Teorie grafů* [online]. [cit. 21. 10. 2015]. Dostupné z URL: <http://teorie-grafu.cz/uvod/vyuziti-grafu.php>.

[25] KARÁSEK, J. *An overview of warehouse optimization.* International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems. 22. 11. 2013. Dostupné z URL: <http://ijates.org/index.php/ijates/article/viewFile/61/60>.

[26] KEY, R., DASGUPTA A. *Warehouse Pick Path Optimization Algorithm Analysis.* Int'l Conf. Foundations of Computer Science 2015, s.63-69.

[27] KLETTE, G. *Skeletons in Digital Image Processing.* ITR Tamaki, University of Auckland Tamaki Campus, Building 731, Auckland, New Zealand. 29. 7. 2002. Dostupné z URL: <http://www.citr.auckland.ac.nz/techreports/2002/CITR-TR-112.pdf>.

[28] MUTLU, M. *(AI) Sliding Puzzle Solution Analyzer* [online]. 17. 4. 2012 [cit. 28. 10. 2015]. Dostupné z URL: <http://www.codeproject.com/Articles/368188/AI-Sliding-Puzzle-Solution-Analyzer>.

[29] NAKOV, O., TRIFONOV R., MANOLOV S., POPOV G. *Computer security.* Technical University Sofia. 2012 ISBN: 978-954-323-973-3.

[30] ŠEDA, M. *Teorie grafů.* Vysoké učení technické v Brně. Fakulta strojního inženýrství, 2003. Dostupné z URL: <http://www.uai.fme.vutbr.cz/~mseda/TG03_MS.pdf>.

[31] VANAJAKSHI, B., SUJATHA B., KRISHNA S. R. *An Analysis of Thinning & Skeletonization for Shape Representation.* International Journal of Computer Communication and Information System (IJCCIS) – Vol 2. No1. ISSN: 0976–1349 7 – 12 2010. Dostupné z URL: <http://www.ijcns.com/pdf/53.pdf>.

[32] *What Is JavaFX?* [online]. 2013 [cit. 17. 11. 2015]. Dostupné z URL: <http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>.

ZOZNAM SYMBOLOV, VELIČÍN A SKRATIEK

A*	algoritmus A star
Best FS	Best First Search
BFS	Breadth First Search – Prehľadávanie do šírky
CLOSED	množina navštívených stavov
c	cost – cena (ohodnotenie hrany)
d	delimiter – oddeľovač
DFS	Depth First Search – Prehľadávanie do hĺbky
E	Edge – Hrana
Fishbone	dizajn rybia kosť
FIFO	First In First Out – fronta (prvý dnu, prvý von)
Flying – V	Lietajúce – V dizajn
G	Graph – Graf
GUI	Graphical User Interface – Uživateľské rozhranie
Inverted – V	Obrátené – V dizajn
LIFO	Last In First Out – zásobník (posledný dnu, prvý von)
MAT	Medial Axis Transform – Transformácia na strednú osu
MVC	Model View Controller (návrhový vzor)
n	number – počet (všeobecne)
N	Neighbor – sused
p	point – bod (pixel)
R	Region – Oblasť (objekt)
Singleton	Jedináčik (návrhový vzor)
TMP	Temporary – dočasný
UML	Unified Modeling Language

v	velocity – rýchlosť
V	Vertex – Vrchol
V_d	Vertex destination – Koncový vrchol
V_s	Vertex start – Počiatočný vrchol
x	rozmer poľa x
X	index X v poli
y	rozmer poľa y
Y	index Y v poli

ZOZNAM PRÍLOH

A OBSAH PRILOŽENÉHO CD

Priložený CD disk obsahuje nasledujúce položky:

- Elektronická verzia diplomovej práce.
- Exportovaný Java projekt so zdrojovými kódmi.
- Vytvorená aplikácia v podobe spustiteľného JAR súboru.