UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Damjan Perenič

Praktična raba algoritmov za iskanje najkrajših poti pri upravljanju skladišč

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: prof. dr. Robnik Marko Šikonja

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00053/2010
Datum: 02.12.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **DAMJAN PERENIČ**

Naslov: **PRAKTIČNA RABA ALGORITMOV ZA ISKANJE NAJKRAJŠIH POTI PRI UPRAVLJANJU SKLADIŠČ**

PRACTICAL USE OF SHORTEST PATH SEARCH ALGORITHMS IN WAREHOUSE MANAGEMENT

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Algoritmi za iskanje najkrajših poti v usmerjenih grafih so praktično široko uporabni. Opišite njihovo rabo znotraj konkretnega programskega produkta za upravljanje skladišč. Pri tem analizirajte različne inačice teh algoritmov in obrazložite najboljšo izbiro. Opišite prilagoditve, ki so potrebne, da lahko splošen algoritem za preiskovanje v usmerjenih grafih uporabimo na problemu vodenja skladiščnih naprav. Na kratko opišite tudi celotno programsko rešitev, njeno arhitekturo in evolucijo.

Mentor:
prof. dr. Marko Robnik Šikonja

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Damjan Perenič,

z vpisno številko 24950401,

sem avtor diplomskega dela z naslovom:

Praktična raba algoritmov za iskanje najkrajših poti pri upravljanju skladišč

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  prof. dr. Marko Robnik-Šikonja
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 6.4.2011 Podpis avtorja:

Zahvala

Za strokovno vodenje, nasvete in pomoč pri izdelavi diplomske naloge se iskreno zahvaljujem mentorju prof. dr. Marku Robniku-Šikonji. Hvala Damjanu Širca in Epilogu, kjer si vsa ta leta nabiram dragocene izkušnje. Posebna zahvala pa gre Gregorju Jeromnu za konstruktivne kritike in debate.

Kazalo

Seznam uporabljenih kratic in simbolov

ERP *Enterprise Resource Planning* - poslovno informacijski sistem

WMS *Warehouse Management System* - sistem za upravljanje skladišč

RF Terminal *Radio Frequency Terminal* - radiofrekvenčni terminal

MFCS *Material Flow Control System* - sistem za upravljanje materialnega pretoka

PLC *Programmable Logic Controller* - programabilni logični krmilnik

TE Transport Unit - *transportna enota*

XML *Extensible Markup Language* - razširljiv označevalni jezik

SQL *Structured Query Language* - strukturiran povpraševalni jezik za delo s podatkovnimi bazami

Oracle AQ *Oracle Advanced Queueing* (ime produkta)

WKP *Workplace* - delovno mesto

PL/SQL *Procedural Language/Structured Query Language* - proceduralni jezik/strukturiran povpraševalni jezik

JSP *Java Stored Procedure* - javanska shranjena procedura

24/7 *24 hours a day, 7 days a week* - 24 ur na dan, 7 dni na teden, označuje neprestano razpoložljivost

JDBC *Java DataBase Connectivity* - javanska podatkovna povezljivost

JVM *Java Virtual Machine* - javanski virtualni stroj

JIT *Just-In-Time* - ravno ob pravem času

Povzetek

Diplomsko delo predstavlja logistiko za avtomatizirana skladišča. Podrobneje predstavljam uporabo Dijkstrovega algoritma za iskanje najkraše poti v usmerjenih in uteženih grafih, s katerim sem implementiral univerzalno rešitev za upravljanje premikov transportnih enot in materiala v avtomatiziranem skladišču. Za razvoj aplikacije sem uporabil programska jezika Java in PL/SQL ter podatkovno bazo Oracle. Predstavljena je tudi arhitektura za doseganje zanesljivega in robustnega 24/7 delovanja in vpliv izkušenj ter povratnih informacij na evolucijski razvoj te aplikacije. Danes program uspešno upravlja transport v več kot 50 skladiščih v zahodni Evropi.

Ključne besede:

algoritem Dijkstra, graf, iskalni algoritem, avtomatsko skladišče, evolucija programske opreme, logistika

Abstract

I present design and implementation of a logistic solution for automated warehouses. I modeled warehouses with weighted directed graphs, on which I use Dijkstra's search algorithm. This design enables implementation of an universal solution for moving transport units and goods in and out of the warehouse. I show balancing of the system activities by allocating work to various material sub-systems. The system supports dynamic rerouting based on host and other requirements and automatically finds diverting routes in case of sub-system failures. Software architecture to achieve reliability and robustness for 24/7 working environment and the value of experience and feedback on the software evolution are additionally demonstrated. The application is developed in Java and PL/SQL programming languages and runs on Oracle database. Today, this application runs in more then 50 automatic warehouses in western Europe.

Key words:

Dijkstra algorithm, graph, shortest path algorithm, automatic warehouse, program evolution, logistics

Poglavje 1

Uvod

1.1 Opis problema

Podjetje *Epilog d.o.o.* (`http://www.epilog.net`), v katerem delam, razvija sistem za upravljanje skladišč AtlasWMS, ki skrbi za informacijsko podporo procesom notranje logistike. V enem paketu sta združena WMS (Warehouse Management System) in MFCS (Material Flow Control System), zato lahko deluje kot MFCS, WMS ali oboje hkrati.

V funkciji upravljanja materialnega toka (MFCS) AtlasWMS koordinira avtomatizirane transportne naprave (regalno dvigalo, tekoči trak in druge). V funkciji sistema za upravljanje skladišča (WMS) upravlja s procesi materialnega toka – od prevzema do izdaje blaga iz skladišča.

AtlasWMS podpira tako avtomatska (material k človeku) kot ročna skladišča (človek k materialu). V prvem primeru koordinira avtomatske transportne naprave, da pripeljejo blago na komisionirno mesto, v drugem pa s pomočjo prenosnega RF-terminala ali izpisa na papir vodi skladiščnika komisionarja.

Verjetno najpomembnejši del sistema za nadzor materialnega toka je program, ki sprejema odločitve o premikih transportnih enot in materiala - logistika, oziroma kot smo ga poimenovali mi, Sokoban. Ime je dobil hkrati po znani igrici Sokoban (glej sliko 1.1 in po japonskem izrazu za skladiščnika (倉庫番).

Delovanje Sokobana si dejansko lahko predstavljamo, kot bi igrali igrico Sokoban z več skladiščniki hkrati (seveda so pravila premikanja v realnosti prilagojena naravi transportnih naprav). Vsako skladišče ima svojo shemo, ki jo v podjetju imenujemo “layout”, in, če nadaljujem s prispodobo, ustreza eni stopnji v igrici.

Prvi Sokoban je bil napisan precej togo (glej poglavje 6) in z naraščanjem

(a)

(b)

(c)

Slika 1.1: Računalniška igra Sokoban

števila "stopenj igrice" in pojavljanjem novih "pasti" je bilo kmalu treba razmisliti o univerzalnejši in prilagodljivejši rešitvi.

Od druge verzije naprej poganja razvoj jasna želja, da bi enak program znal "igrati" optimalno na poljubno oblikovanih "stopnjah" in da mora biti ustrezna razširitev, ko se pojavi nova oblika "pasti", kar se da enostavna, če ni podprta že kar z obstoječo kodo.

Dodaten izziv predstavljajo tudi zahteve okolja (visoka odzivnost, robustnost) in seveda poznavanje in izraba tehnologij, ki so na razpolago (podatkovna baza Oracle, okolje Java).

1.2 O rešitvi in diplomski nalogi

Izkazalo se je, da je dobra rešitev ponazoriti skladišče z grafom. Lokacije predstavljajo vozlišča, možni premiki pa povezave. Taka podatkovna predstavitev skladišča omogoča vrsto matematično definiranih/izvedljivih operacij in analiz. Grafi so že star in dodobra preizkušen formalizem, saj so se pojavili v znanosti že leta 1878 [3]. Od takrat je bilo na področju grafov odkritih, preizkušenih in analiziranih že ogromno algoritmov, tako da ima programer na tej podatkovni strukturi veliko izbiro. Struktura je enostavna, prilagodljiva, lahko razumljiva in nezahtevna.

Ena od osnovnih idej programske zasnove je bila, da vse skupaj zgradimo z nekaj elementarnimi operacijami. Prvi korak je bil napisati funkcijo za iskanje najkrajše poti med startom in ciljem. Na tej osnovi sem si zamislil funkcije za iskanje premika transportne enote in transportne enote v ciljno vozlišče. Na vse elementarne funkcije je pri moji zasnovi mogoče vplivati z uporabo vtičnikov.

Zaradi zahtev po visoki robustnosti, odzivnosti in odzivnosti v realnem času, sem v začetku zasnoval aplikacijo s čim manjšo kompleksnostjo, brez uporabniškega vmesnika in s strukturo podobno aplikaciji na mikrokontrolerju (inicializacija, zanka, stražar). Namesto stražarja se zanka prekine in program steče od začetka.

V diplomi sta problematika in reševanje problemov opisana po naslednjem vrstnem redu: teoretični temelji, preslikava teorije v prakso, zaključim pa z opisom rešitve, razvojem programa in najpomembnejšimi idejami za naprej.

V poglavju 2 sem zbral osnoven opis grafov in definicij. *Seznanjenost z grafi je nujna za razumevanje preostalega dela.*

V poglavju 3 je opisan Dijkstrov algoritem za iskanje najkrajše poti, ki ga uporablja Sokoban za iskanje poti od starta do cilja. Poznavanje delovanja algoritma ni potrebno za razumevanje preostalega dela naloge. Sledi podrobnejši opis zasnove. Opis je poenostavljen na tak nivo, da je lepo viden koncept uporabe iskalnega algoritma. V dejanski implementaciji Sokoban upošteva še dodatne entitete kot so naprave, skladiščna območja, poslovne pogoje naročil itd. Posamezna transportna naprava je obravnavana glede na svoje obnašanje, seznam premikov, ki jih lahko opravi, kam in kako se pozicionira itd. To v program vnese dodaten nivo kompleksnosti, ki je zaradi preglednosti izločen.

V poglavju 4 opišem, kako je skladišče modelirano z grafom, kako je bilo razbito na gradnike in opis elementarnih funkcij, s katerimi iščemo premike po skladišču. To je glavni del diplomskega dela. V poglavju 5 sledi opis arhitekture sistema: zakaj je program neinteraktiven, kako reagira na sporočila iz

baze, kako so implementirani vtičniki, dnevniki, konfiguracija itd. V poglavju 6 je opisano, kako se je program razvijal skozi čas, da je podpiral nove "nivoje" skladišč. Na koncu so opisane še ideje za nadaljni razvoj na področju uporabe iskalnih algoritmov.

Poglavje 2

Grafi

Grafi so v računalništvu pogosto prisotna podatkovna struktura in algoritmi za delo z njimi spadajo med temeljne na tem področju. Na stotine zanimivih računalniških problemov je opisanih prav z grafi. Uporabljamo jih za predstavitev komunikacijskih mrež, v lingvistiki, v umetni inteligenci, nadzoru pretoka, kemiji, fiziki, navigacijskih sistemih itd.

Problem, ki ga mora rešiti logistika v avtomatskem skladišču se v računalniški znanosti rešuje z modeliranjem skladišča na grafu. Nad grafom, ki predstavlja skladišče, nato izvajamo algoritme in ostale matematične operacije. Zato sledi kratek opis grafov in terminologije, ki je nujna za razumevanje kasnejših poglavij.

2.1 Opis grafov

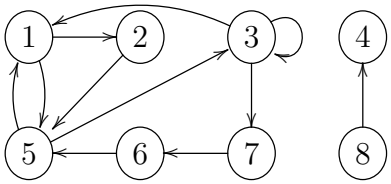
Graf je abstraktna podatkovna struktura, ki ponazarja množico točk (krajišč, vozlišč, vozlov), ki so med seboj povezane s povezavami (robovi, vejami).

To poglavje predstavlja dve vrsti grafov: usmerjene in neusmerjene.

Graf G je par množic (V, E), kjer je V končna množica in E dvojiška relacija V. Množico V imenujemo ***množica vozlišč*** grafa G in E ***množica povezav*** grafa G. Njeni elementi se imenujejo ***povezave***.

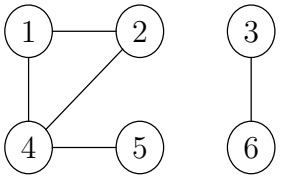
Če gre za ***usmerjen graf*** ali ***digraf***, je množica povezav sestavljena iz *urejenih parov*. Slika 2.1 je grafična predstavitev usmerjenega grafa z množico vozlišč $V = 1, 2, 3, 4, 5, 6$. Vozlišča predstavljajo krogi, povezave so predstavljene s puščicami. ***Zanke***, se pravi, povezave vozlišča s samim seboj so možne.

V ***neusmerjenem grafu*** $G = (V, E)$ je množica povezav E sestavljena iz *neurejenih* parov vozlišč. To pomeni, da je povezava par (u, v), kjer velja $u, v \in V$ in $u \neq v$. V neusmerjenem grafu so zanke (povezave vozlišča



Slika 2.1: Usmerjen graf $G = (V, E)$ kjer je $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ in $E = \{(1, 2), (1, 5), (2, 5), (3, 1), (3, 3), (3, 7), (5, 1), (5, 3), (6, 5), (7, 6), (8, 4)\}$

s seboj) prepovedane, zato vsaka povezava povezuje natanko dve različni vozlišči. Slika 2.2 je grafična predstavitev neusmerjenega grafa na množici vozlišč $\{1, 2, 3, 4, 5, 6, 7, 8\}$.



Slika 2.2: Neusmerjen graf $G = (V, E)$, kjer je $V = \{1, 2, 3, 4, 5, 6\}$ in $E = \{(1, 2), (1, 4), (2, 4), (3, 1), (3, 3), (3, 6), (4, 1), (6, 5)\}$

Precej definicij velja tako za usmerjene kot neusmerjene grafe, paziti pa je treba, ker imajo nekateri izrazi različne pomene v obeh kontekstih.

Če je (u, v) povezava v usmerjenem grafu $G = (V, E)$, rečemo da (u, v) ***izhaja iz*** ali ***zapušča*** vozlišče u in ***prihaja v*** ali ***vstopa*** v vozlišče v. Primer: vozlišče 3 na sliki 2.1 zapuščajo povezave $(3, 1), (3, 3), (3, 7)$, vanj pa vstopajo $(3, 3)$ in $(5, 3)$. Če je (u, v) povezava v neusmerjenem grafu $G = (V, E)$, rečemo, da (u, v) ***povezuje*** u in v. Na sliki 2.2 so povezave na 4 $(1, 4)$, $(2, 4)$ in $(4, 5)$.

Če je (u, v) povezava v grafu $G = (V, E)$, je vozlišče v ***sosednje*** vozlišču u. Kadar je graf neusmerjen, je relacija simetrična. Če je graf usmerjen, sosednost ni nujno simetrična. Če je vozlišče v sosednje u v usmerjenem grafu, to zapišemo kot $u \to v$. V obeh grafih 2.1 in 2.2 je vozlišče 2 sosednje vozlišču 1, ker je povezava $(1, 2)$ pripada obema grafoma. Vozlišče 1 ni sosednje vozlišču 2 na sliki 2.1, ker povezava $(2, 1)$ v grafu ne obstaja.

V usmerjenem grafu $G = (V, E)$ je vozlišče u ***direktno povezano*** z vsakim vozliščem, ki je sosedno v neusmerjeni različici grafa G. Torej u je direktno povezano z v, če je $(u, v) \in E$ ali $(v, u) \in E$. V neusmerjenem grafu, sta vozlišči u in v direktno povezani, če sta sosedni.

Stopnja $deg[v]$ vozlišča v neusmerjenem grafu je število njegovih povezav. Primer: povezava 2 na sliki 2.2 ima stopnjo 2. V usmerjenem grafu je ***izstopna stopnja*** $deg^+[v]$ vozlišča število povezav, ki ga zapuščajo in ***vstopna stopnja*** $deg^-[v]$ vozlišča število povezav, ki vanj vstopajo. ***Stopnja*** vozlišča v usmerjenem grafu je vsota vstopne in izstopne stopnje. Vozlišče 3 v sliki 2.1 ima vstopno stopnjo 2, izstopno stopnjo 3 in torej stopnjo 5.

Pot dolžine k od vozlišča v_0 do vozlišča v_k v grafu $G = (V, E)$ je zaporedje $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ vozlišč, $(v_{i-1}, v_i) \in E$ za $i = 1, 2, \ldots, k$. Dolžina poti je število povezav na njej. Pot ***vsebuje*** vozlišča $v_0, v_1, \ldots, v_k$ in povezave $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. Če obstaja pot p od u do u', rečemo, da je u' ***dosegljiv*** iz u po poti p, kar včasih zapišemo kor $u \overset{p}{\rightsquigarrow} u'$, če je G usmerjen graf. Pot je ***enostavna***, če so vsa vsebovana vozlišča različna. Na sliki 2.1, je pot $\langle 1, 2, 5, 3 \rangle$ enostavna pot z dolžino 3. Pot $\langle 5, 1, 2, 5, 3 \rangle$ ni enostavna, ker je vozlišče 5 obiskano dvakrat.

Del poti $p = \langle v_0, v_1, \ldots . v_k \rangle$ je nepretrgan del zapopredja vozlišč poti. To pomeni, da je za vsak $0 \leq i \leq j \leq k$ podzaporedje $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ dela poti p.

V usmerjenem grafu pot $\langle v_0, v_1, \ldots, v_k \rangle$ tvori ***cikel***, kadar velja $v_0 = v_k$, in pot vsebuje vsaj eno povezavo. Cikel je ***enostaven***, če velja, da so $v_1, v_2, \ldots, v_k$ različni. Zanka je cikel z dolžino 1. Dve poti $\langle v_o, v_1, v_2, \ldots, v_{k-1}, v_0 \rangle$ in $\langle v'_0, v'_1, v'_2, \ldots, v'_{k-1}, v'_0 \rangle$ tvorita isti cikel, če obstaja celo število j, tako da velja $v'_i = v_{(i+j) \bmod k}$ za $i = 0, 1, \ldots, k-1$. Na sliki 2.1, tvori pot $\langle 3, 7, 6, 5 \rangle$ isti cikel kot $\langle 7, 6, 5, 3, 7 \rangle$, $\langle 6, 5, 3, 7, 6 \rangle$ in $\langle 5, 3, 7, 6, 5 \rangle$. Ta cikel je enostaven. Cikel $\langle 1, 2, 5, 1, 5 \rangle$ pa ni enostaven. Cikel $\langle 3, 3 \rangle$, ki ga tvori povezava $(3, 3)$, je zanka. Usmerjen graf brez zank je ***enostaven***.

V povezanem grafu tvori pot $\langle v_0, v_1, \ldots, v_k \rangle$ ***cikel***, če velja $v_o = v_k$ in so vozlišča $v_1, v_2, \ldots, v_k$ različna. Primer na sliki 2.2: pot $\langle 1, 2, 4 \rangle$ je cikel. Graf brez ciklov je ***acikličen***.

Neusmerjen graf je ***povezan***, če je vsak par vozlišč povezan s potjo. ***Povezane komponente*** grafa so ekvivalenčni razredi vozlišč, ki so v relaciji "je dosegljivo iz". Graf na sliki 2.2 ima dve povezani komponenti: $\{1, 2, 4, 5\}$ in $\{3, 6\}$. Vsako od vozlišč $\{1, 2, 4, 5\}$ je dosegljivo iz vsakega od vozlišč $\{1, 2, 4, 5\}$. Neusmerjen graf je povezan, če ima natančno eno povezano komponento, to se pravi, da je vsako vozlišče dosegljivo iz vsakega vozlišča. Usmerjen graf je ***krepko povezan***, če je vsak par vozlišč medsebojno dosegljiv (obstaja pot v obeh smereh).

Krepko povezane komponente grafa so ekvivalenčni razredi vozlišč, ki so v relaciji "vzajemno dosegljiva".

Usmerjen graf je krepko povezan, če ima samo eno krepko povezano kompo-

nento. Graf na sliki 2.1 ima tri krepko povezane komponente: $\{1, 2, 3, 5, 6, 7\}$, $\{4\}$ in $\{8\}$. Vsi pari vozlišč iz množice $\{1, 2, 3, 5, 6, 7\}$ so medsebojno dosegljivi. Vozlišča $\{4, 8\}$ ne tvorijo krepko povezane komponente, ker vozlišče 8 ni dosegljivo iz vozlišča 4.

Graf $G' = (V', E')$ je ***podgraf*** $G = (V, E)$ če je $V' \subseteq V$ in $E' \subseteq E$. Podgraf grafa G zgrajen z množico $V' \subseteq V$ je graf $G' = (V', E')$, kjer:

$$E' = \{(u, v) \in E : u, v \in V'\}. \tag{2.1}$$

Če ima podgraf $G' = (V, E')$ isto množico vozlišč kot graf G, potem je to ***vpet podgraf***.

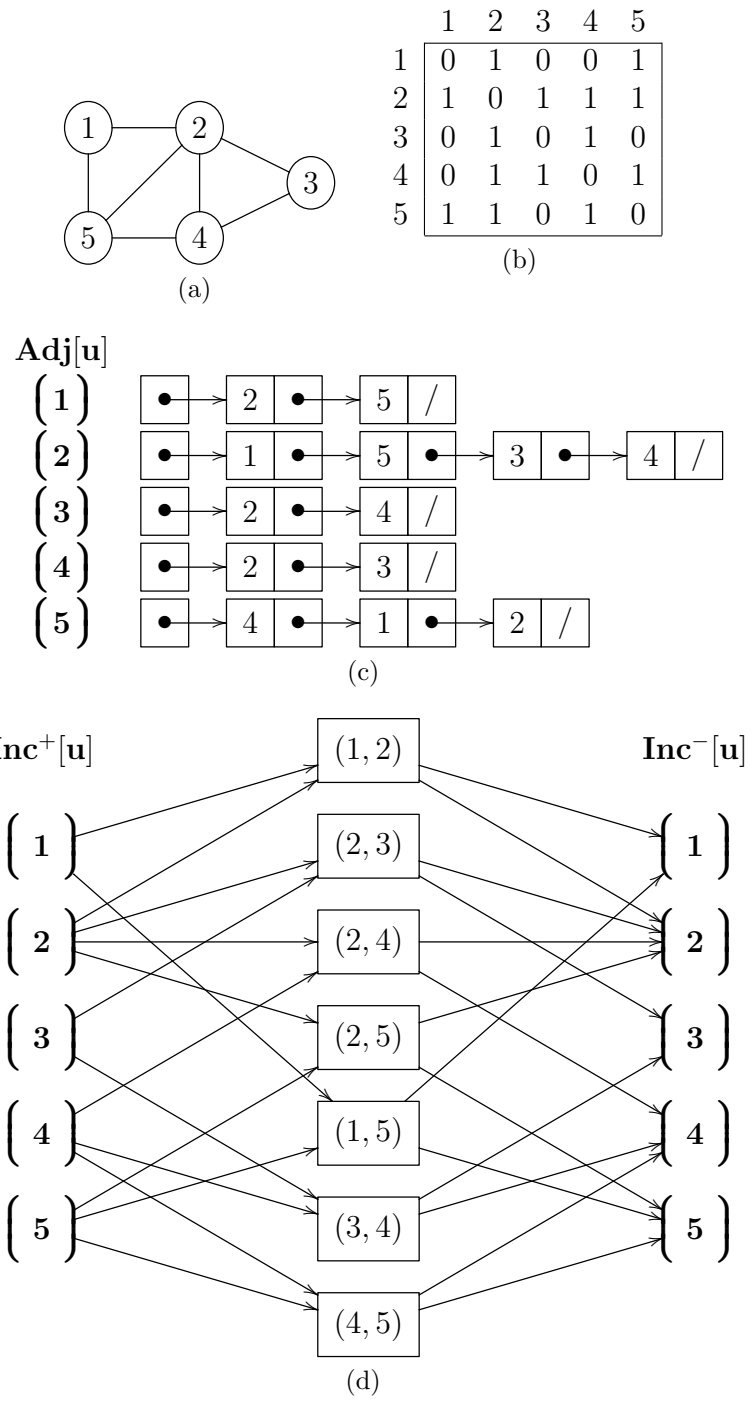
2.2 Predstavitev grafov

Obstaja več standardnih načinov za predstavitev grafa $G = (V, E)$: najbolj znani in razširjeni predstavitvi sta seznam sosednosti (Adjacency list) in matrika sosednosti (Adjacency Matrix). Običajno raje uporabljamo seznam sosednosti, ker je seznam bolj kompakten v primeru ***redkega grafa*** – to so tisti grafi, pri katerih je $|E|$ dosti manj kot $|V|^2$. Večina algoritmov na grafih predpostavlja, da je vhodni graf predstavljen s seznamom sosednosti. Matrika sosednosti je primernejša, kadar je graf ***gost*** ($|E|$ je blizu $|V|^2$) ali kadar želimo hitro priti do podatka o tem, ali sta dve vozlišči povezani ali ne. Vsi algoritmi v tem in naslednjem poglavju predpostavljajo, da so vhodni grafi predstavljeni v obliki seznamov sosednosti.

Seznam sosednosti grafa $G = (V, E)$ je sestavljen iz polja Adj z $|V|$ seznami, po en seznam za vsako vozlišče v V. Za vsako vozlišče $u \in V$, vsebuje seznam sosednosti $Adj[u]$ (kazalce na) vsa vozlišča v, ki imajo povezavo $(u, v) \in E$. To pomeni, da $Adj[u]$ sestavljajo vsa vozlišča sosednja u v G. Vozlišča v vsakem seznamu sosednosti so tipično shranjena v poljubnem vrstnem redu. Slika 2.3c je predstavitev s seznamom sosednosti neusmerjenega grafa s slike 2.3a. Na sliki 2.4c je z seznamom sosednosti predstavljen usmerjen graf s slike 2.4a.

Če je graf G usmerjen, je vsota dolžin vseh seznamov sosednosti enaka $|E|$, ker je povezava (u, v) predstavljena tako, da v nastopa v $Adj[u]$. Če je graf G neusmerjen, je vsota dolžin seznamov sosednosti $2|E|$, ker je neusmerjena povezava (u, v) predstavljena tako, da u nastopa v seznamu sosednosti v in obratno. Ne glede na to, ali je graf usmerjen ali ne zahteva seznam sosednosti le $O(max(V, E)) = O(V + E)$ pomnilnika.

Uteženi grafi so grafi, kjer vsaki povezavi pripada ***utež***. Ker seznam sosednosti nima prostora za shranjevanje dodatnih podatkov o povezavah, lahko



Slika 2.3: Dva načina predstavitve neusmerjenega grafa. **(a)** neusmerjen graf G ima pet vozlišč in sedem povezav. **(b)** matrika sosednosti za G **(c)** seznam sosednosti za graf G. **(c)** seznam pojavitev za graf G.

utežen graf predstavimo s seznamom sosednosti in z uvedbo dodatne ***utežne funkcije*** $w : E \to \mathbb{R}$.

Uvedba dodatne funkcije za vsak dodaten atribut povezave je nepraktična. Ta problem rešujemo s kombinacijo seznama sosednosti in objektnega programiranja, kar imenujemo ***seznam pojavitev*** (incidence list)[9]. Seznam pojavitev je sestavljen iz polja Inc^+ z $|V|$ seznami. Za vsako vozlišče $u \in V$, vsebuje seznam pojavitev $Inc^+[u]$ kazalce na vse izhodne povezave $(u, v) \in E$, ki so predstavljene z objektno strukturo. Objektna predstavitev povezave omogoča fleksibilno podatkovno strukturo, v katero lahko shranimo utež, oznako in ostale atribute povezave, ki jih bomo spoznali v nadaljevanju. Prav tako velja, da so povezave v seznamu pojavitev tipično shranjene v poljubnem vrstnem redu.

Ker Sokoban potrebuje poleg učinkovitega iskanja sosednjih vozlišč tudi učinkovito iskanje direktno povezanih vozlišč, je uvedeno dodatno polje Inc^- z $|V|$ seznami. Za vsako vozlišče $u \in V$, vsebuje seznam pojavitev $Inc^-[u]$ kazalce na vhodne povezave $(v, u) \in E$. Seznam vseh povezav vozlišča $u \in V$ v seznamu pojavitev dobimo z $Inc[u] = Inc^-[u] \cup Inc^+[u]$. Poraba pomnilnika za dodatno polje s seznamom povezav je neznatna, ker se podvojijo le reference do objektov, ki predstavljajo vozlišča in povezave. Poraba pomnilnika pri seznamu pojavitev je zaradi objektne predstavitve nekoliko večja od seznama sosednosti, vendar je še vedno $O(V + E)$.

Slika 2.3d je predstavitev s seznamom pojavitev neusmerjenega grafa s slike 2.3a. Na sliki 2.4d je z seznamom pojavitev predstavljen usmerjen graf s slike 2.4a.

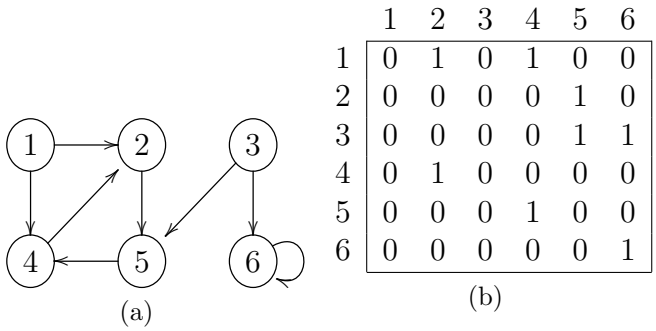
Slabost seznama sosednosti in seznama pojavitev je počasno ugotavljanje obstoja povezave (u, v). Potrebno je iskanje v v seznamu sosednosti $Adj[u]$. To slabost rešujemo z uporabo matrike sosednosti, za kar pa porabimo $O(|V|^2)$ pomnilnika.

Za ***matriko sosednosti*** grafa $G = (V, E)$ predpostavljamo, da so vozlišča oštevilčena: $1, 2, \ldots, |V|$. Matrika sosednosti grafa G je potem sestavljena iz $|V| \times |V|$ matrike $A = (a_{ij})$, tako da velja

$$a_{ij} = \begin{cases} 1 & \text{če } (i, j) \in E, \\ 0 & \text{če } (i, j) \notin E. \end{cases} \tag{2.2}$$

Sliki 2.3b in 2.4b sta matriki sosednosti neusmerjenega in usmerjenega grafa s slik 2.3a in 2.4a. Matrika sosednosti grafa zahteva $\theta(V^2)$ pomnilnika, ne glede na število povezav grafa.

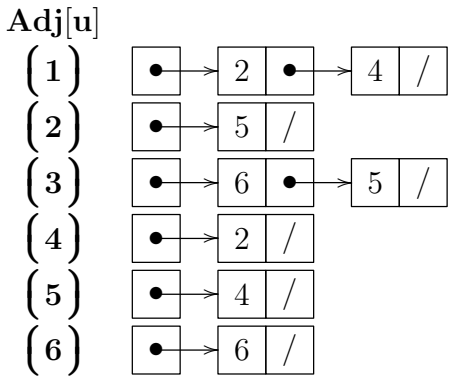
Oglejmo si simetrijo ob glavni diagonali matrike sosednosti v sliki 2.3b.



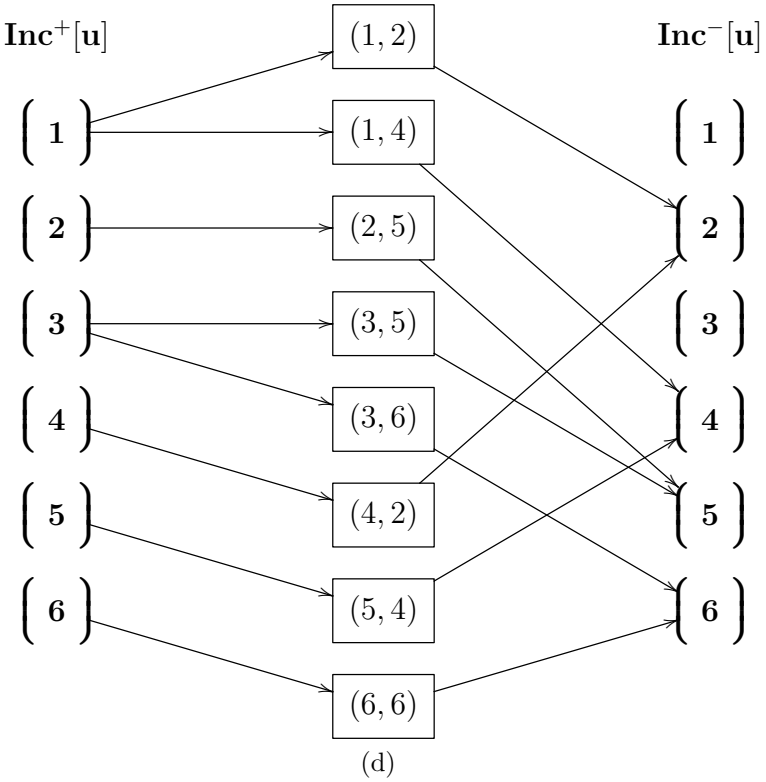
(a)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(b)



(c)



(d)

Slika 2.4: Dva načina predstavitve usmerjenega grafa. **(a)** usmerjen graf G s petimi vozlišči in sedmimi povezavami. **(b)** matrika sosednosti za graf G **(c)** seznam sosednosti za graf G. **(d)** seznam pojavitev za graf G

Definirajmo ***transpozicijo*** matrike $A = (a_{ij})$, ki naj bo matrika $A^T = (a^T_{ij})$, ki jo dobimo z $a^T_{ij} = a_{ji}$. Ker v neusmerjenemu grafu (u, v) in (v, u) predstavljata isto povezavo je matrika sosednosti A neusmerjenega grafa sama svoja transpozicija: $A = A^T$. V nekaterih situacijah lahko zato prihranimo skoraj polovico pomnilnika tako, da shranimo samo vrednosti na diagonali in nad njo.

Tudi predstavitev z matriko sosednosti lahko uporabimo za predstavitev uteženih grafov. Primer: če je $G = (V, E)$ utežen graf z utežno funkcijo w, je utež $w(u, v)$ povezave $(u, v) \in E$ shranjena v vrstici u in stolpcu v matrike sosednosti. Če povezava ne obstaja, lahko tja shranimo vrednost NIL. Za določene vrste problemov je lahko bolj pripravno uporabiti druge vrednosti, kot na primer 0 ali ∞.

Čeprav je seznam sosednosti učinkovitejši od matrike sosednosti v smislu porabe pomnilnika, za majhne grafe zaradi preprostosti raje uporabljamo matriko sosednosti.

2.3 Iskanje v širino

Iskanje v širino je eden izmed preprostejših algoritmov za preiskovanje grafov in podlaga za mnoge druge. Dijkstrov algoritem za iskanje najkrajše poti iz enega začetnega vozlišča, ki je opisan v poglavju 3.2, uporablja podobno idejo, kot iskanje v širino.

Graf $G = (V, E)$ in ***začetno*** vozlišče s iskanje v širino sistematično preiskuje povezave G, da "odkrije" vozlišča, ki so dosegljiva iz s. Pri tem izračuna oddaljenost $d[u]$ (najmanjše število povezav) od s za vsa dosegljiva vozlišča u. Hkrati zgradi tudi "drevo-najprej-v-širino", ki raste iz s in vsebuje vsa dosegljiva vozlišča. Za vsako vozlišče v, dosegljivo iz s, je v drevesu pot od s do v, ki ustreza "najkrajši poti" od s do v v grafu G, se pravi pot z najmanjšim številom povezav. Algoritem deluje na usmerjenih in neusmerjenih povezavah.

Iskanje v širino je poimenovano tako, ker širi mejo raziskanega področja enakomerno v vse strani po njenem obsegu. Z drugimi besedami: algoritem odkrije vsa vozlišča na oddaljenosti k od s, preden odkrije vozlišča na oddaljenosti $k + 1$.

Ob napredovanju algoritem iskanja v širino barva vozlišča belo, sivo ali črno. Vsa vozlišča so na začetku bela, nato so lahko pobarvana sivo ali črno. Vozlišče je ***odkrito***, ko algoritem nanj prvič naleti in ga pobarva sivo ali črno. Razliko med sivo in črno algoritem uporablja zato, da zagotavlja iskanje najprej v širino. Če je $(u, v) \in E$ in je vozlišče u črno, potem je vozlišče v sivo ali črno; to pomeni, da so vsa vozlišča sosednja črnim vozliščem odkrita. Siva vozlišča

imajo lahko nekaj sosedjnih belih vozlišč; ta vozlišča predstavljajo mejo med raziskanimi in neraziskanimi vozlišči.

Iskanje v širino zgradi drevo, ki na začetku vsebuje samo koren, ki je začetno vozlišče s. Kadarkoli je pri preiskovanju seznama sosednosti že odkritega vozlišča u odkrito novo vozlišče v, sta vozlišče v in povezava (u, v) dodana drevesu. Vozlišče u poimenujemo ***predhodnik*** ali ***starš*** vozlišča v v drevesu preiskovanja v širino. Ker je vsako vozlišče odkrito le enkrat, ima le enega starša. Prednik in naslednik v drevesu preiskovanja v širino sta definirana relativno glede na koren s: če je u na poti iz korena s do vozlišča v, potem je u prednik v in v je naslednik u.

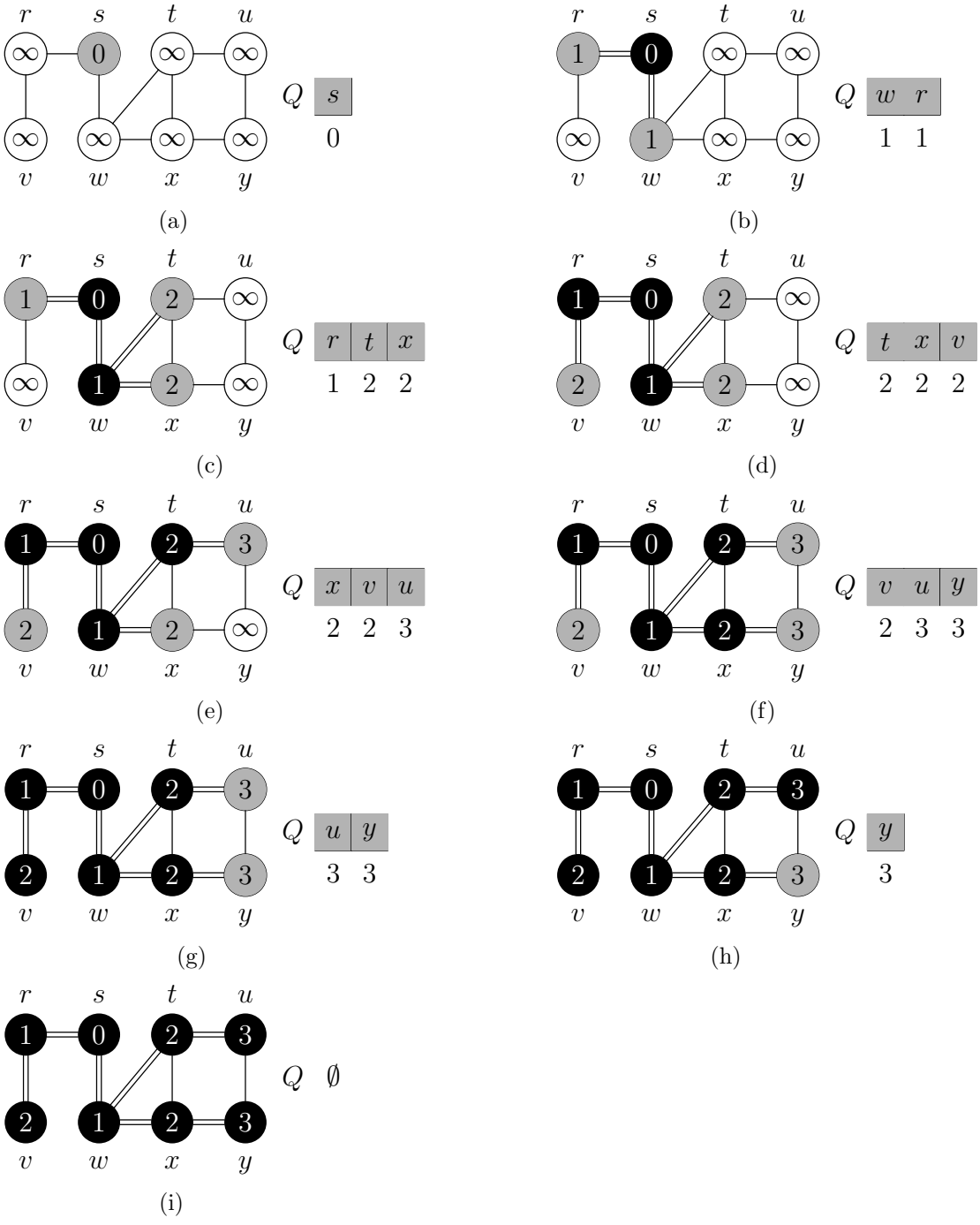
Izpis 1 Iskanje najprej v širino. Izračuna najkrajšo pot iz vozlišča s do vseh povezanih vozlišč na grafu G. Izvajanje algoritma je grafično prikazano na sliki 2.5

```
 1: procedure BFS(G, s)
 2:     for all u ∈ V[G] − {s} do          ▷ Inicializacija vseh vozlišč, razen s
 3:         barva[u] ← BELA                ▷ Obarvaj vozlišča belo
 4:         d[u] ← ∞                       ▷ Nastavi vozliščem neskončno razdaljo
 5:         π[u] ← NIL                     ▷ Vozlišča nimajo starša
 6:     end for
 7:     barva[s] ← SIVA                    ▷ Obarvaj začetno vozlišče s sivo
 8:     d[s] ← 0                           ▷ Razdalja od s do s je 0 povezav
 9:     π[s] ← NIL                         ▷ s nima starša
10:     Q ← {s}                            ▷ Inicializiraj vrsto Q z enim elementom s
11:     while Q ≠ ∅ do                     ▷ Obdelaj vrsto
12:         u ← PRVI(Q)                    ▷ Preberi prvo vozlišče iz vrste
13:         for all v ∈ Adj[u] do          ▷ Obdelaj vsa sosedna vozlišča
14:             if barva[v] = BELA then    ▷ Vozlišče v se neodkrito?
15:                 barva[v] ← SIVA        ▷ Odkrito; Obarvaj
16:                 d[v] ← d[u] + 1        ▷ Razdalja do v je razdalja do starša u + 1
17:                 π[v] ← u               ▷ Nastavi starša
18:                 UVRSTI(Q, v)           ▷ Vstavi vozlišče v na konec vrste Q
19:             end if
20:         end for
21:         IZLOČI(Q)                      ▷ Odstrani prvo vozlišče iz vrste
22:         barva[u] ← ČRNA                ▷ Označi vozlišče u kot obdelano
23:     end while
24: end procedure
```

Procedura za iskanje naprej v širino na izpisu 1 privzame, da je vhodni graf $G = (V, E)$ predstavljen s seznamom sosednosti. Definiranih je nekaj podatkovnih struktur, ki hranijo dodatne vrednosti za vsako vozlišče na grafu G. Barva vsakega vozlišča $u \in V$ je shranjena v spremenljivki $barva[u]$ in starš vozlišča u je shranjen v $\pi[u]$. Če vozlišče u nima starša, potem je $\pi[u] = \text{NIL}$. Izračunana razdalja med začetnim vozliščem s in vozliščem u se shranjuje v $d[u]$. Algoritem uporablja vrsto $Q[10]$, ki deluje po pravilu prvi noter, prvi ven (FIFO) za upravljanje množice sivih vozlišč. Nad vrsto imamo definirane tri operacije:

- UVRSTI(Q, s) - doda element s na konec vrste (enqueue)
- IZLOČI(Q) - izloči prvi element iz vrste (dequeue)
- PRVI(Q) - vrne prvi element v vrsti (first)

Iskanje v širino najde oddaljenost vsakega dosegljivega vozlišča v grafu $G = (V, E)$ od podanega začetnega vozlišča $s \in V$. ***Najkrajša pot*** $\delta(s, v)$ od s do v je najmanjše število povezav od vozlišča s do vozlišča v, oziroma ∞, če poti od s do v ni.



Slika 2.5: Delovanje iskanja v širino na neusmerjenem grafu. Povezave drevesa so osenčene, kot jih določi BFS. V vsakem vozlišču u je navedena oddaljenost $d[u]$. Vrsta Q je prikazana za začetek vsake iteracije zanke **while** v vrstici 12 na izpisu 1. Trenutna razdalja vozlišča v vrsti je navedena pod vozliščem v vrsti.

Poglavje 3

Iskalni algoritmi

Voznik želi najti najkrajšo možno pot od Sežane do Ljubljane. V rokah ima karto Slovenije - kako naj se loti iskanja najkrajše poti? En od načinov bi bil, da poišče vse možne poti iz Sežane do Ljubljane, izračuna dolžino vsake in izbere najkrajšo.

Tudi v primeru, da takoj izločimo poti, ki vsebujejo cikle, še vedno obstaja ogromno možnosti, od katerih jih je večina že na prvi pogled neuporabnih. Pot iz Sežane v Črnomelj in potem v Ljubljano je očitno slab kandidat, saj je Črnomelj na drugi strani Slovenije.

V tem poglavju bom predstavil način, kako tak problem učinkovito rešujemo.

Za reševanje ***problema najkrajše poti*** potrebujemo podatke s karte urejene v utežen usmerjen graf $G = (V, E)$, z utežno funkcijo $w = E \to \mathbb{R}$, ki prireja povezavam realna števila - uteži. ***Utež*** poti $p = \langle v_0, v_1, \ldots, v_k \rangle$ je vsota uteži vseh povezav, ki jo sestavljajo:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i). \tag{3.1}$$

Utež najkrajše poti od u do v definiramo kot

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} u\} & \text{če je obstaja pot od } u \text{ do } v, \\ \infty & \text{sicer.} \end{cases} \tag{3.2}$$

Najkrajša pot od vozlišča u do vozlišča v je potem definirana kot katerakoli pot p z utežjo $w(p) = \delta(u, v)$.

V primeru Sežana-Ljubljana lahko cestno karto predstavimo kot graf. Vozlišča predstavljajo križišča, povezave ceste med križišči in uteži predstavljajo cestno razdaljo med križišči. Naša naloga je najti najkrajšo pot med križiščem

v Sežani (recimo križišče Partizanske ceste in Ceste na Lenivec) do križišča v Ljubljani (recimo križišče Dunajske in Slovenske ceste).

Uteži povezav lahko predstavljajo tudi drugačne metrike, ne samo razdalj. Pogosto jih uporabljamo za predstavitev časa, zakasnitve, stroškov, izgub ali kakšne druge zanimive količine, ki se nabira vzdolž poti in jo želimo minimizirati.

V danem grafu $G = (V, E)$ želimo najti najkrajšo pot med danim ***začetnim*** vozliščem $s \in V$ do vsakega vozlišča $v \in V$. Z algoritmom za problem z enim začetnim vozliščem lahko rešujemo več inačic problema, na primer:

Problem najkrajše poti do enega cilja: najdi najkrajšo pot do ***ciljnega*** vozlišča t od vsakega vozlišča v. Z obratom smeri vsake poti v grafu lahko ta problem prevedemo na problem z enim izvornim vozliščem.

Problem najkrajše poti med danima vozliščema (Single-pair shortest-path problem): poišči najkrajšo pot med u in v za dani vozlišči u in v. Če rešimo problem enega začetnega vozlišča za začetno vozlišče u, smo rešili tudi ta problem. Še več: zaenkrat ni znan noben algoritem, ki bi bil v najslabšem primeru hitrejši kot najboljši algoritmi za eno začetno vozlišče.

Najkrajše poti med vsemi pari vozlišč (All-pairs shortest-paths problem): poišči najkrajšo pot med u in v za vsak par vozlišč u in v. Tudi ta problem lahko rešimo tako, da izvedemo algoritem za eno začetno vozlišče po enkrat za vsako vozlišče, vendar je ta problem največkrat mogoče rešiti tudi hitreje. Obstaja več algoritmov, ki tak problem rešujejo: Floyd-Warshall, Johnsonov algoritem in drugi.

Ker v skladiščih potrebujemo le *najkrajšo pot med danima vozliščema (Single-pair shortest-path)*, se bomo posvetili le temu.

Povezave z negativno utežjo

V nekaterih primerih problema najkrajše poti z enim začetnim vozliščem lahko obstajajo tudi povezave z negativno utežjo. Če graf $G = (V, E)$ ne vsebuje ciklov z negativno utežjo, dosegljivih iz začetnega vozlišča s, potem ostaja za vse $v \in V$ utež najkrajše poti $\delta(s, v)$ dobro definirana, tudi če ima negativno vrednost. Če obstaja negativen cikel, ki je dosegljiv iz s, utež najkrajše poti ni več dobro definirana in definiramo $\delta(s, v) = -\infty$.

Najkrajšo pot v grafih s povezavami z negativnimi utežmi lahko iščemo z algoritmi, kot je Bellman-Ford, ki potrebuje za izvedbo $O(VE)$ časa.

3.1 Najkrajše poti in sproščanje

Da bi lažje razumeli algoritme za iskanje najkrajše poti z enega začetnega vozlišča, je dobro poznati tehnike, ki jih uporabljajo, in lastnosti najkrajših poti, ki jih izkoriščajo. Glavna tehnika, ki jo uporablja algoritem v tem poglavju, je sproščanje, ki iterativno zmanjšuje zgornjo mejo uteži najkrajše poti in išče dalje. V tem poglavju si bomo ogledali, kako sproščanje deluje.

Sproščanje

Skoraj vsi poznani algoritmi za iskanje najkrajše poti z enim izvornim vozliščem uporabljajo tehniko ***sproščanja***. Za vsako vozlišče $v \in V$ vzdržujejo atribut $d[v]$, ki pomeni zgornjo mejo uteži najkrajše poti od začetka s do v. $d[v]$ imenujemo ***ocena najkrajše poti***. Začetne vrednosti ocene poti in predhodnikov nastavimo s proceduro v izpisu 2.

Izpis 2 Začetna nastavitev algoritma z enim začetnim vozliščem

```
1: procedure INICIALIZIRAJ-SPREMENLJIVKE(G, s)
2:     for all v ∈ V[G] do
3:         d[v] ← ∞
4:         π[v] ← NIL
5:     end for
6:     d[s] ← 0
7: end procedure
```

Po začetni nastavitvi velja $\pi[v] = \text{NIL}$ za vse $v \in V$, $d[v] = 0$ za $v = s$ in $d[v] = \infty$ za $v \in V - \{s\}$.

Postopek ***sproščanja*** povezave (u, v) je sestavljen iz poskusa izboljšati najkrajšo pot do v, ki smo jo dosedaj našli, tako, da gremo skozi u in če je taka pot boljša, popravimo $d[v]$ in $\pi[v]$. Sprostitev lahko zmanjša vrednost ocene najkrajše poti $d[v]$ in spremeni vrednost predhodnika vozlišča v, $\pi[v]$. Koda algoritma 3 prikazuje sproščanje povezave (u, v).

Slika 3.1 prikazuje dva primera sproščanja povezave. V enem se ocena najkrajše poti zmanjša, v drugem pa spremembe ni.

Dijkstrov algoritem, opisan kasneje v tem poglavju, kliče INITIALIZE-SINGLE-SOURCE in za tem v zanki sprošča povezave. Še več - sproščanje je edini način, s katerim se zmanjšujejo ocene najkrajših poti in spreminjajo predniki. V

Izpis 3 Sproščanje povezave

```
1: procedure SPROSTI(u, v, w)
2:     if d[v] > d[u] + w(u, v) then
3:         d[v] ← d[u] + w(u, v)
4:         π[v] ← u
5:     end if
6: end procedure
```

Slika 3.1: Sproščanje povezave (u, v). Ocena najkrajše poti je za vsako vozlišče prikazano v vozlišču. **(a)** ker pred sproščanjem velja $d[v] > d[u] + w(u, v)$ se vrednost $d[v]$ zmanjša. **(b)** v tem primeru je pred sprostitvijo $d[v] \leq d[u] + w(u, v)$, zato ostane vrednost $d[v]$ nespremenjena.

Dijkstrovem algoritmu je vsaka povezava sproščana natanko enkrat, v drugih algoritmih je ista povezava lahko sproščana večkrat.

3.2 Dijkstrov algoritem

Dijkstrov algoritem rešuje problem najkrajše poti z enim začetnim vozliščem na uteženem usmerjenem grafu $G = (V, E)$ za primer, ko so vse uteži povezav pozitivne. V tem poglavju zato predpostavljamo, da velja $w(u, v) \geq 0$ za vsako povezavo $(u, v) \in E$.

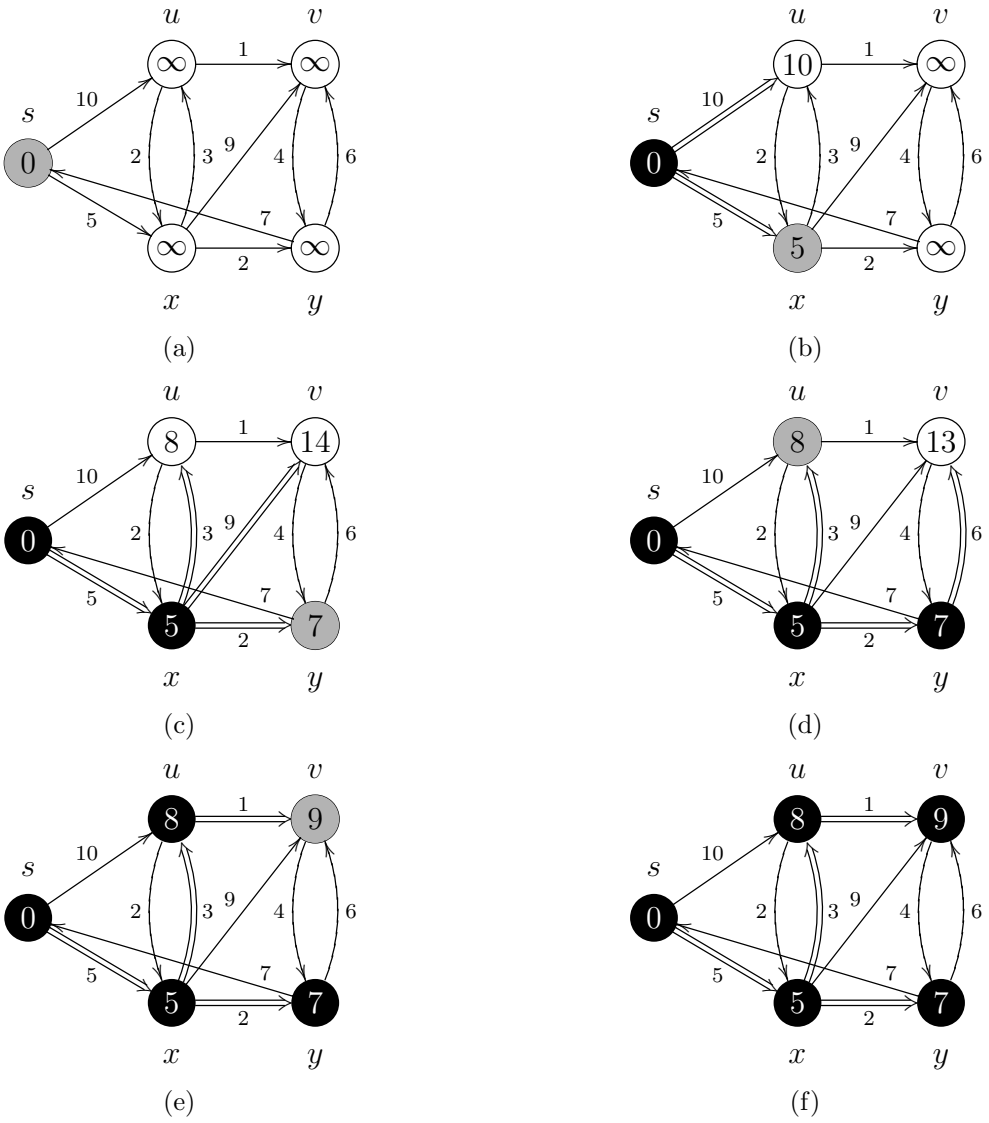
Dijkstrov algoritem vzdržuje množico vozlišč S, katerih zadnje uteži najkrajših poti od izhodišča s do v so že izračunane. To pomeni, da za vsa vozlišča $v \in S$ velja $\delta[v] = \delta(s, v)$. Algoritem v zanki izbira vozlišče $u \in \{V - S\}$ z najmanjšo oceno najkrajše poti, doda u v S, in sprosti vse povezave, ki zapuščajo u. V implementaciji algoritma, ki sledi, vzdržujemo prioritetno vrsto Q, ki vsebuje vsa vozlišča v $V - S$, urejena po njihovih vrednostih d. Implemetacija predpostavlja, da je graf G predstavljen s seznamom sosednosti.

Funkcija Izvleči-Min(Q) briše vozel z najmanjšim ključem iz kopice Q in vrne kazalec nanj. Implementacija take funkcije je odvisna od implementacije kopice. Za Dijkstrov je najprimernejša (najhitrejša) Fibonaccijeva kopica.

Izpis 4 Dijkstrov algoritem. Izračuna najkrajšo pot iz vozlišča s do vseh povezanih vozlišč na grafu G. Izvajanje algoritma je grafično prikazano na sliki 3.2

```
 1: procedure Dijkstra(G, w, s)
 2:     Inicializiraj-Spremenljivke(G, s)
 3:     S ← ∅
 4:     Q ← V[G]
 5:     while Q ≠ ∅ do
 6:         u ← Izvleči-Min(Q)
 7:         S ← S ∪ u
 8:         for all v ∈ Adj[u] do
 9:             Sprosti(u, v, w)
10:         end for
11:     end while
12: end procedure
```

Dijkstrov algoritem prikazan na izpisu 4 sprošča povezave, kot je prikazano na sliki 3.2. Vrstica 2 nastavi običajne začetne vrednosti d in π in vrstica 3 nastavi množico S prazno. Vrstica 4 zatem nastavi prioritetno vrsto Q tako, da vsebuje vsa vozlišča, ki so v $V - S = V - \emptyset = V$. Ob vsakem izvajanju zanke **while** na vrsticah 5-11, je iz $Q = V - S$ vzeto eno vozlišče u in dodano v množico S. (V prvi iteraciji zanke je $u = s$). Vozlišče u ima zato



Slika 3.2: Izvajanje Dijkstrovega algoritma. Začetno vozlišče je čisto na levi. Ocene najkrajše poti so prikazane v vozliščih in osenčene povezave prikazujejo vrednosti predhodnikov: če je povezava (u, v) senčena, potem je $\pi[v] = u$. Črna vozlišča so v množici S in bela vozlišča so v prioritetni vrsti $Q = V - S$. **(a)** stanje tik pred prvo iteracijo zanke **while** (vrstice 5-11). Osenčeno vozlišče ima najmanjšo vrednost d in je izbrano kot vozlišče u v vrstici 6. **(b)-(f)** stanje po vsaki zaporedni iteraciji zanke **while**. Senčeno vozlišče je v vsakem koraku izbrano kot u v vrstici 6 naslednje iteracije. Vrednosti d in π, prikazane v stanju (f) so končne vrednosti.

najnižjo oceno najkrajše poti izmed vseh vozlišč v $V - S$. Vrstici 8 in 9 zatem sprostita vse povezave (u, v), ki zapuščajo u in s tem nastavijo novo oceno $d[v]$ in predhodnika $\pi[v]$, če je najkrajša pot do v skrajšana in če gre skozi vozlišče u. Naj opozorim, da po vrstici 4 ni v Q dodano nobeno vozlišče več. Vozlišče je natanko enkrat vzeto iz Q in dodano v S. Zaradi tega se zanka **while** na vrsticah 5-11 izvede natanko $|V|$/krat.

Ker Dijkstrov algoritem vedno izbere "najlažje" oziroma "najbližje" vozlišče v $V - S$ za dodajanje v S, rečemo, da uporablja požrešno strategijo. Požrešne strategije v splošnem ne dajejo vedno najboljšega rezultata, vendar Dijkstrov algoritem zanesljivo izračuna najkrajšo pot! [12]

Kako hiter je Dijkstrov algoritem? Kot prvi primer analizirajmo situacijo v kateri vzdržujemo prioritetno vrsto $Q = V - S$ kot linearno tabelo (linear array). Za takšno implementacijo vzame vsaka IZVLEČI-MIN operacija $O(V)$ časa. Skupaj imamo $|V|$ takšnih operacij, kar pomeni skupno porabo časa za IZVLEČI-MIN operacije $O(V^2)$. Vsako vozlišče $v \in V$ je dodano v množico S natanko enkrat, tako, da je vsako vozlišče v seznamu sosednosti $Adj[v]$ v zanki **while** na vrsticah 5-11 preiskano natanko enkrat v celotni izvedbi algoritma. Ker je skupno število povezav v vseh seznamih sosednosti $|E|$ enako tudi skupnemu številu iteracij zanke **while**, od katerih vsaka traja $O(1)$ časa. Čas izvajanja celotnega algoritma je potemtakem $O(V^2 + E) = O(V^2)$.

Če je graf redek, je prioritetno vrsto Q praktično implementirati kot binarno kopico. Algoritem, ki s tem nastane, včasih imenujemo ***modificiran Dijkstrov algoritem***. Vsaka IZVLEČI-MIN operacija v tem primeru traja $O(\lg V)$. Enako kot prej jih je $|V|$. Čas, potreben za gradnjo binarne kopice je $O(V)$. Prireditev $d[v] \leftarrow d[u] + w(u, v)$ v RELAX je izvedena s klicem DECREASE-KEY $(Q, v, d[u] + w(u, v))$, ki rabi $O(\lg V)$ časa in takih operacij je največ $|E|$. Skupen čas izvajanja algoritma je torej $O((V + E) \lg V)$, kar je $O(E \lg V)$, če so iz začetnega dostopna vsa vozlišča.

Dejansko lahko dosežemo čas izvajanja $O(V \lg V + E)$ s tem, da implementiramo prioritetno vrsto Q s ***Fibonaccijevo kopico*** [4]. Povprečen strošek vsake od $|V|$ IZVLEČI-MIN operacij je $O(\lg V)$ in vsak od $|E|$ klicev DECREASE-KEY vzame povprečno le $O(1)$ časa.

Dijkstrov algoritem je podoben iskanju v širino, ker množica S ustreza množici črnih vozlišč pri iskanju v širino. Enako kot vsebujejo vozlišča v S svoje končne uteži najkrajše poti, imajo črna vozlišča pri iskanju v širino svoje prave oddaljenosti pri iskanju v širino.

Poglavje 4

Uporaba iskalnega algoritma v praksi

Avtomatsko skladišče je skladišče tipa blago k človeku, v katerem, namesto da bi človek hodil med policami in ročno nabiral blago, tega pripeljejo avtomatizirane transportne naprave.

Glavni deli takega skladišča so skladiščni prostor (največkrat so to visokoregalna skladišča z nekaj tisoč do nekaj desettisoč skladiščnimi mesti) in sestav avtomatiziranih transportnih naprav različnih izvedb in lastnosti, ki premikajo transportne enote ("škatle"raznih izvedb, v katerih je blago) ali blago samo.

Za mikropremike znotraj lokacij in iz ene lokacije na drugo skrbi podrejeni sistem, običajno realiziran v programabilnem industrijskem kontrolerju (PLC), na primer Siemens Simatic.

Ta podsistem preko podatkovne povezave dobiva od nadzornega sistema (MFCS, Material Flow Control System) naloge za premike, ki jih izvaja in vrača potek in rezultat izvedbe. Prav tako javlja stanje posameznih naprav, tako, da je MFCS vedno na voljo aktualna informacija o stanju in razpoložljivosti naprav.

Program, opisan v tej diplomski nalogi, se imenuje Sokoban. Teče na nivoju MFCS in skrbi za optimalno notranjo logistiko: izrabo avtomatskih transportnih naprav in pretok transportnih enot in blaga.

4.1 Modeliranje skladišča

Tako kot smo v poglavju 3 modelirali cestno karto kot graf, modeliramo tukaj avtomatsko skladišče kot graf G. V primeru, da imamo opravka z več avtomatskimi skladišči je G podgraf nekega večjega grafa G_a, ki vsebuje po en

podgraf za vsako avtomatsko skladišče. Vse kar je opisano v nadaljevanju kot graf G, velja v primeru enega skladišča, oziroma kot nepovezan podgraf enega skladišča v grafu večih skladišč.

Zaradi potreb po natančnem sledenju transportov po skladišču potrebujemo poleg množice vozlišč V in povezav E še množico lokacij L sestavljeno iz elementov lokacij l. Množica L je končna množica.

Lokacije ne vplivajo na izračun poti, so le natančna prostorska matrika za sledenje po skladišču, zato preslikava med lokacijami in vozlšči ni 1:1. "Svoje" vozlišče imajo le lokacije, kjer je potrebna odločitev o nadaljevanju transporta, ostale lokacije pa upoštevamo kot podaljšek poti in povečamo utež posamezne povezave med vozliščema. Velja torej da je $V \subseteq L$.

Utež predstavlja strošek premika med dvemi vozlišči. Sokoban ne išče najkrajše poti, ampak najoptimalnejšo pot, zato moramo pri nastavljanju uteži upoštevati več dejavnikov, ki vplivajo na odločitev logistike:

- upoštevati je potrebno dolžino poti (število lokacij, ki jih je treba prevoziti),
- hitrost delovanja naprave,
- preferenčne smeri potovanja transportnih enot (glavna pot, rezervna pot),
- v primeru več sočasnih naprav na enem segmentu je pomembna tudi bližina naprave

Povezavam med vozlišči je poleg utežne funkcije w dodana še funkcija $\varrho : E \to \mathbf{L}$, ki za vsako povezavo $e \in E$ vrne pot povezave kot zaporedje lokacij $l \in L$. Pot povezave $\varrho(u, v)$ je sekvenca $\langle l_0, l_1, \ldots, l_k \rangle$ lokacij, kjer je $l_0 = u$ in $l_k = v$.

Avtomatsko skladišče je sestavljeno iz mehanskih naprav, na katerih se nahajajo lokacije, od katerih so nekatere vozlišča. Vsako vozlišče ima zato atribut, ki pove na kateri napravi se nahaja.

Kadar so naprave v okvari, so izklopljene oziroma se opravlja vzdrževanje in ne morejo izvajati transportov, se jih označi kot neuporabne. Seznam vozlišč na delujočih napravah hranimo v množici V'.

Vsako skladišče modeliramo z dvema grafoma $G = (V, E)$ in njegovim vpetim podgrafom $G' = (V, E')$.

Graf $G = (V, E)$ vsebuje vsa vozlišča in vse povezave v skladišču. Graf $G' = (V, E')$ pa vsebuje ***vsa vozlišča***, vendar le trenutno delujoče povezave.

Trenutno delujoče povezave dobimo, če izločimo vse povezave, ki se povezujejo na nedelujoče povezave:

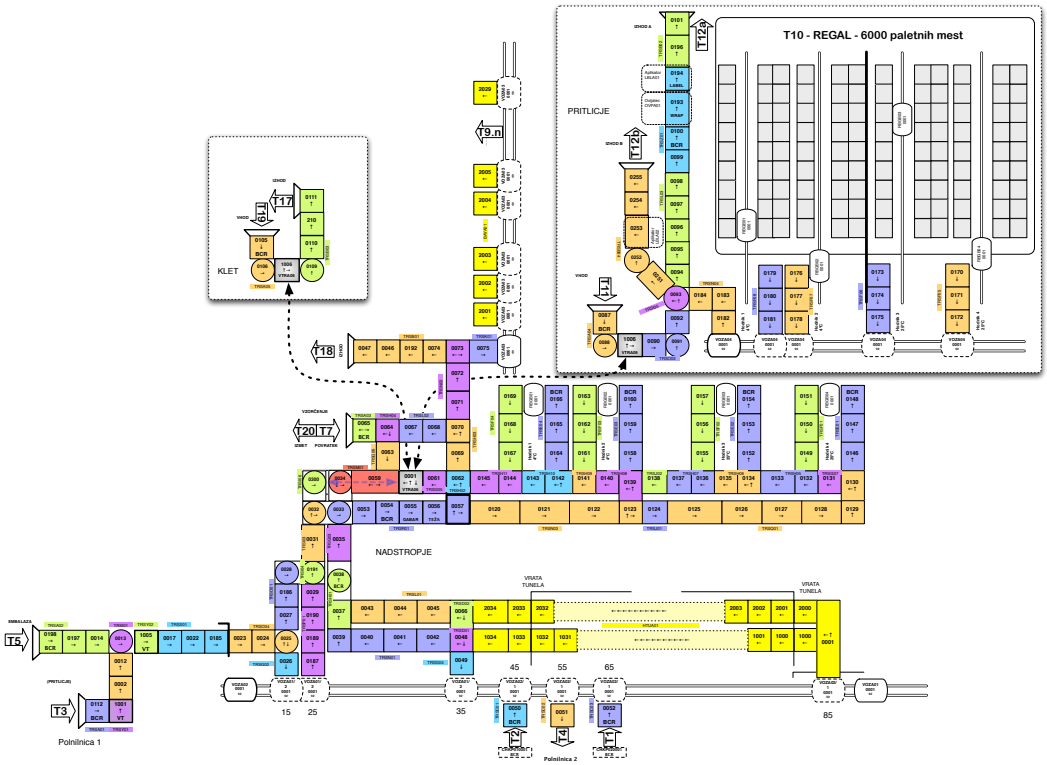
$$E' = \{(u, v) \in E : u, v \in V'\} \tag{4.1}$$

Torej $E' \subseteq E$. Množica vozlišč V je v obeh grafih skupna, množica E' pa je implementirana z razširitvijo seznama sosednosti. Na tak način imamo dva grafa, ki zasedata pomnilniški prostor enega. Zakaj potrebujemo sočasno oba grafa, bomo videli v razdelku 4.4.2.

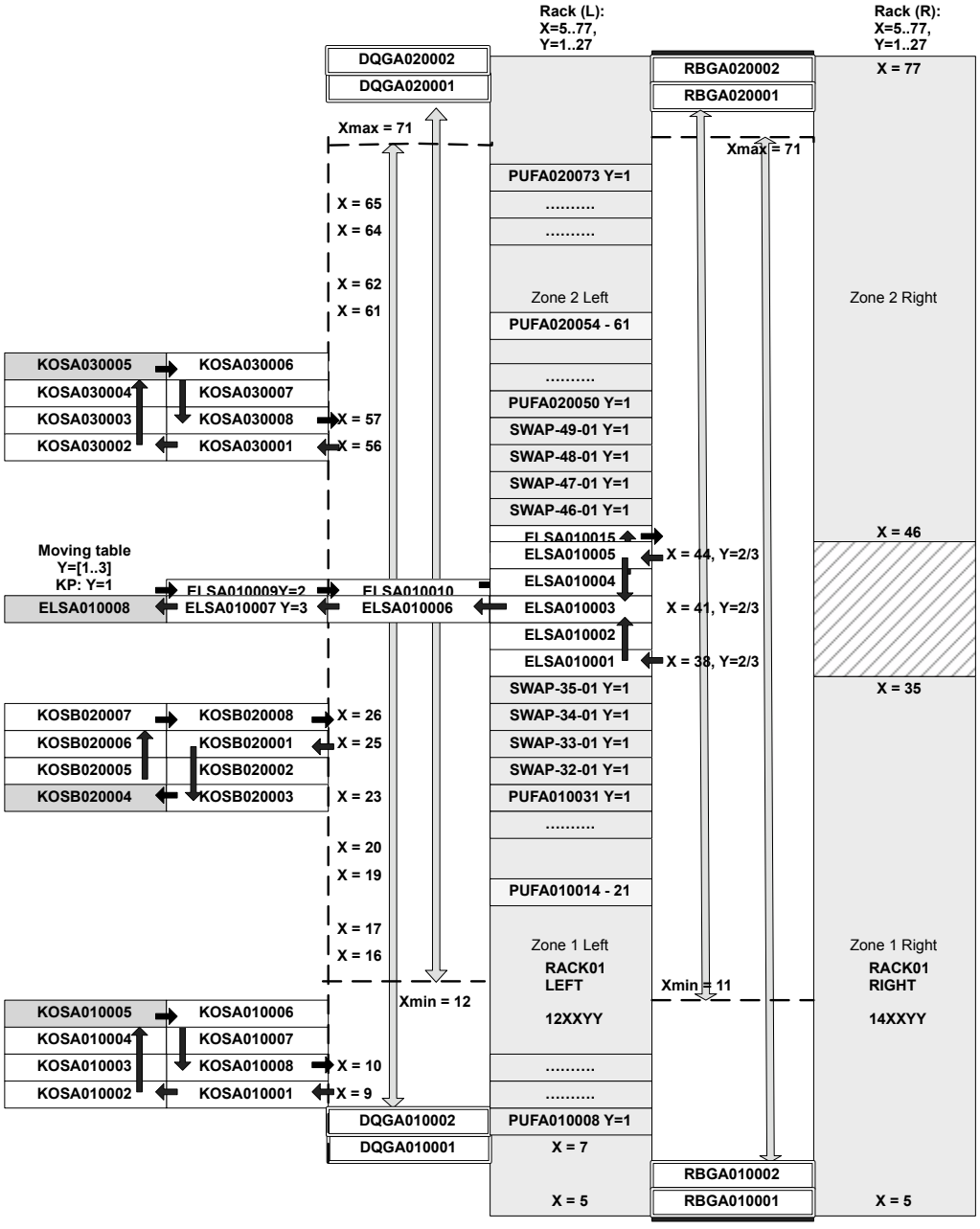
S stališča toka transportnih enot poznamo dva tipa skladišč: odprti sistem in zaprti sistem. V ***odprtem sistemu*** (Slika 4.1) transportne enote prihajajo v sistem na vhodih in sistem zapuščajo na izhodih.

V ***zaprtem sistemu*** (Slika 4.2) ima skladišče fiksno število transportnih enot in le-te nikoli ne zapustijo skladišča. V zaprtem sistemu se skladišči le blago, ki se vlaga in jemlje iz transportnih enot. Pogoj za zaprti sistem je, da je graf krepko povezan, pri odprtem sistemu te zahteve ni.

Predstavimo dva primera skladišč, nad katerima bom opisal delovanje in implementacijo logističnega programa Sokoban.



Slika 4.1: Odprti sistem. Transportne enote prihajajo v skladišče na vhodih kot so *TRSA020198, TRSA010112, TRSC010050, ...* Njihovi cilji so lahko ali visoko regalna skladišča *(Hodnik 1, 2, 3, 4)* ali eden ali več izhodov kot so *TRSC020051, TRSA030065, TRSB030111, ...* $|V| = 6064, |E| = 12557$



Slika 4.2: Zaprti sistem ima fiksno število transportnih enot, ki nikoli ne zapustijo sistema. Prazne transportne enote se zapeljejo na delovno mesto npr. *ELSA010008*, kjer se naloži blago. Ko je blago naloženo, se kaseta zapelje v skladišče. Za izskladiščenje blaga, se transportna enota zopet zapelje na delovno npr. *KOSA010005*, kjer se blago izloži ven. $|V| = 6634, |E| = 31415$

4.2 Uporaba iskalnega algoritma

Osnova za vse premike v skladišču je nalog za premik transportne enote, ki vsebuje izvorno mesto, ciljno mesto, ter identifikacijo transportne enote.

Ko Sokoban dobi nalog za premik, poišče pot do cilja z uporabo algoritma za iskanje najkrajše poti. Pri tem uporablja iskalni algoritem Dijkstra, ki je implementiran z uporabo načrtovalnega vzorca Strategija (Design Pattern: Strategy), kar je razvidno na izpisu 5. To omogoča zelo enostavno menjavo algoritma, če se izkaže, da potrebujemo drugačnega.

Izpis 5 Implementacija algoritma po načrtovalskem vzorcu *Strategija*

```
1 public class PathManager
2 {
3     PathFinder pf = new Dijkstra();
4 }
```

Algoritem mora implementirati vmesnik PathFinder (izpis 6), ki definira vse potrebne metode za iskanje poti, ter poizvedovanje nad vozlišči in potmi, ki jih Sokoban potrebuje za svoje delovanje.

Podgraf G' je implementiran z razširitvijo predstavitve seznama sosednosti. Predstavitev seznama sosednosti je razširjena še za atribut, ki pove ali je povezava trenutno na voljo ali ne. Definicija je razvidna v izpisu 6 v vrsticah 17 in 18. Če je parameter *availableOnly=**true***, potem vrneta funkciji le delujoče vhodne oziroma izhodne povezave vozlišča na grafu G'. Če je parameter *availableOnly=**false***, potem funkciji vrneta vse povezave vozlišča na grafu G.

4.2.1 Ključne funkcije

Za lažje razumevanje preglejmo še enkrat vse spremenljivke in obliko zapisov, ki smo jih definirali do sedaj:

$\emptyset$ Prazna množica

$\mathbb{N}$ Množica naravnih števil

$\mathbb{R}$ Množica realnih števil

$\mathbf{B}$ Množica vrednosti $\{true, false\}$

$\mathbf{L}$ Množica vseh lokacij v skladišču

Izpis 6 Univerzalni vmesnik do algoritma

```
1  public interface PathFinder
2  {
3     public PathList findShortestPath(Vertex source,
4                                      Vertex destination);
5
6     public interface Vertex extends Comparable
7     {
8        public Edge getEdgeEntering(Vertex towards);
9        public Edge getEdgeLeaving(Vertex from);
10       public Location getLocat();
11       public int getDeviceId();
12       public EdgeList leavingEdges(boolean availableOnly);
13       public EdgeList enteringEdges(boolean availableOnly);
14       public String getLocatType();
15       public boolean isRackLocat();
16       public boolean isStorageLocat();
17       public boolean isSwapLocat();
18       public boolean isLocatChaotic();
19       public boolean equals(Object o);
20    }
21
22    public interface Edge
23    {
24       public int getId();
25       public Vertex getDest();
26       public Vertex getSource();
27       public int getCost();
28       public String getTag();
29       public boolean isQueueing();
30    }
31 }
```

$\mathbf{V}$ Množica vseh vozlišč v skladišču

$\mathbf{E}$ Množica vseh povezav v skladišču

$\mathbf{E'}$ Množica delujočih povezav v skladišču

$\mathbf{G} = (\mathbf{V}, \mathbf{E})$ Graf celotnega skladišča

$\mathbf{G'} = (\mathbf{V}, \mathbf{E'})$ Podgraf G, ki nima povezav ki bi šle preko izklopljenih naprav

$\mathbf{l} \in \mathbf{L}$ Lokacija v skladišču

$\mathbf{u}, \mathbf{v} \in \mathbf{V}$ Vozlišča na grafu G, uporabljamo za začasne spremenljivke

$\mathbf{s}, \mathbf{d} \in \mathbf{V}$ Vozlišča na grafu G, uporabljamo za označitev začetnega in ciljnega vozlišča.

$(\mathbf{u}, \mathbf{v}) \in \mathbf{E}$ Povezava med vozlišči u in v

$\varrho(\mathbf{u}, \mathbf{v})$ Funkcija, ki vrne zaporedje lokacij, ki sestavljajo povezavo (u, v)

$\langle \mathbf{u}, \mathbf{v} \rangle$ Pot sestavljena iz povezav $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$

$\langle \mathbf{u}, \mathbf{v} \rangle [\mathbf{i}]$ Povezava na i-tem mestu v poti $\langle u, v \rangle$

$\mathbf{Adj_G}[\mathbf{u}]$ Množica vozlišč sosednjih vozlišču u na grafu G

$\mathbf{Inc_G}[\mathbf{u}]$ Množica povezav povezanih z vozliščem u na grafu G.

$\mathbf{Inc^+_G}[\mathbf{u}]$ Množica izhodnih povezav iz vozlišča u na grafu G.

$\mathbf{Inc^-_G}[\mathbf{u}]$ Množica vhodnih povezav v vozlišče u na grafu G.

Za lažje iskanje povezanih vozlišč uvajamo novo funkcijo $\lambda : E \times V \to \mathbf{V}$, ki iz množice povezav vrne vozlišča, ki so povezana z danim vozliščem V, torej:

$$\lambda(E, u) = \{v : (u, v) \in E \vee (v, u) \in E\} \tag{4.2}$$

To nam omogoča enostavno iskanje povezanih vozlišč:

- vozlišča direktno povezana vozlišču u: $\lambda(Inc[u], u)$,
- vozlišča vhodno povezana vozlišču u: $\lambda(Inc^-[u], u)$,
- sosedna vozlišča (izhodno povezana) vozlišču u: $\lambda(Inc^+[u], u)$.

Zunanje funkcije: Psevdokoda vsebuje tudi klice sledečih zunanjih funkcij:

Lokacija-Prosta$(l \in L)$ Funkcija vrne true, če je lokacija l prosta in lahko sprejme novo transportno enoto

Premik-Dovoljen$((u, v) \in E)$ Funkcija vrne true, če je premik na povezavi (u, v) dovoljen. Implementacija je prikazana na izpisu 18 v poglavju 5.1.5.

Preberi-Cilj-TE$(l \in L)$ Funkcija prebere končni cilj za transportno enoto, ki se nahaja na lokaciji l.

Določi-Prioriteto$((u, v) \in E)$ Funkcija ovrednoti vsak premik na povezavi glede na pravila o pretoku, poslovna pravila, ter parametre transporta in mu določi prioriteto.

Ustvari-Premik$(\langle u, v \rangle)$ Procedura dobi pot kot parameter in ustvari premike za transportno enoto. Implementacija je prikazana v poglavju 4.6.1 na strani 55.

Poišči pot

Zaradi zahtev po visoki odzivnosti se vsi izračuni poti $\langle s, d \rangle$ med začetnim vozliščem s in ciljnim vozliščem d shranijo z ***razpršilno funkcijo*** $h : V \times V \to \mathbb{N}$, ki vsakem paru vozlišč $u, v \in V$ določi naravno število $n \in \mathbb{N}$, kot indeks razpršene tabele $H[h(s, d)]$. Za izračun razpršenega ključa potrebujemo še funkcijo $r : V \to \mathbb{N}$, ki nam preslika vozlišče $v \in V$ v unikatno naravno število $n \in \mathbb{N}$, predstavljeno z 32-bitno besedo. Razpršilna funkcija h izračuna ključ za par vozlišč $s \in V, d \in V$, ki se nato preslika v predal v razpršeni tabeli $H[0 \ldots m-1]$:

$$h(s, d) = (r(s) \ll 16) | r(d) \tag{4.3}$$

Simbol $r(s) \ll 16$ pomakne vrednost za 16 bitov, operator $|$ je bitni operator za operacijo *ali*. Razpršilna funkcija je preprosta a deluje, saj se je izkazalo, da v praksi izračuna unikatno vrednost ključa, ker ima skladišče redko več kot 65535 vozlišč. Tudi v primeru, ko ima skladišče več kot 65535 vozlišč, je funkcija še vedno zelo učinkovita, saj je v tem primeru večina vozlišč skladiščnih lokacij in kot bomo videli kasneje, se skladiščne lokacije v glavnem naslavlja preko skupnega virtualnega vozlišča.

V praksi se je pokazalo, da je v skladišču vedno najbolj optimalna pot do sosednjega vozlišča preko direktne povezave. Zato privzamemo, da če obstaja

direktna povezava, funkcija POIŠČI-POT vrne pot, ki je le ena direktna povezava med vozliščema. V tem primeru se izpusti klic algoritma za iskanje najkrajše poti. Implementacija funkcije POIŠČI-POT je prikazana v izpisu 7.

Izpis 7 Funkcija POIŠČI-POT je skladišču prilagojena implementacija funkcije za iskanje najkrajše poti

```
1: function POIŠČI-POT(s, d)
2:     if s = d then
3:         return ∅                                  ▷ Zanke niso dovoljene
4:     end if
5:     ⟨s, d⟩ ← H[h(s, d)]
6:     if ⟨s, d⟩ = NIL then                          ▷ Pot še ni izračunana
7:         ⟨s, d⟩ ← FIND-SHORTEST-PATH(G, s, d)      ▷ Poišči pot na grafu G
8:         if ⟨s, d⟩ = ∅ then                        ▷ Ni poti
9:             ⟨s, d⟩ ← FIND-SHORTEST-PATH(G′, s, d)  ▷ Poskusi G′
10:        end if
11:        H[h(s, d)] ← ⟨s, d⟩                       ▷ Shrani pot
12:    end if
13:    return ⟨s, d⟩
14: end function
```

Iskanje poti za transportne enote

Vse funkcije za iskanje najkrajše poti so prilagojene situaciji v skladišču in upoštevajo prisotnost transportnih enot in stanje transportnih naprav. Zato npr. ne vračajo poti, če ni transportne enote, ki bi lahko pot prevozila, ali če je cilj zaseden. Dejansko iščejo le izvedljive premike in ne poti. Funkcije vračajo pot od trenutnega vozlišča do cilja, da lahko v primeru tranzitov (zaporednih lokacij brez križišč - glej razdelek 4.3.4) Sokoban uporabi kar shranjeno pot.

Osnovna funkcija za iskanje poti je **POT-TE-IZ-VOZLIŠČA**(v), ki vrne nadaljnjo pot za transportno enoto, ki se nahaja na vozlišču v. Če je vozlišče v prazno, vrne prazno množico. Prazno množico vrne tudi, če je naslednja lokacija na premiku iz vozlišča zasedena, oziroma če premik iz nekega razloga v tistem trenutku ni dovoljen.

Vhodni parameter je izvorno vozlišče. Funkcija sama zbere vse ostale podatke potrebne za izračun poti, ter ugotovi ali je premik dovoljen ali ne (izpis8).

Funkcija **POT-TE-NA-VOZLIŠČE**(v) najprej preveri ali je vozlišče v prazno in lahko sprejme novo transportno enoto. V koliko je vozlišče prazno, pregleda

Izpis 8 Funkcija POT-TE-IZ-VOZLIŠČA vrne nadaljnjo pot za TE ***iz*** vozlišča

```
 1: function POT-TE-IZ-VOZLIŠČA(s)
 2:     d ← PREBERI-CILJ-TE(s)            ▷ Preberi cilj za transportno enoto
 3:     if d = NIL then                                    ▷ TE ni na vozlišču
 4:         return ∅                      ▷ Vrni prazno pot. Premik ni mogoč
 5:     end if
 6:     ⟨s, d⟩ ← POIŠČI-POT(s, d)                          ▷ Poišči pot do cilja
 7:     if PREMIK-DOVOLJEN(⟨s, d⟩[0]) = true then ▷ Naslednja lokacija na
    poti prosta?
 8:         return ⟨s, d⟩                                  ▷ Da. Premik možen
 9:     else
10:         return ∅                  ▷ Ne. Premik ni možen. Vrni prazno pot
11:     end if
12: end function
```

vozlišča sosednja vozlišču v, če imajo na sebi transportno enoto in če gre lahko nadaljnja pot TE preko vozlišča v. V kolikor najde možne premike preko vozlišča v, jih ovrednoti in vrne pot za tistega, ki je zbral najvišje število točk.

Vhodni parameter je ciljno vozlišče. Funkcija sama zbere vse ostale podatke potrebne za izračun poti ter ali je premik dovoljen ali ne; prikazana v izpisu 9.

Funkcija **POT-TE-PREKO-VOZLIŠČA**(v) išče ali obstaja transportna enota na vozliščih sosednjih v. Deluje podobno kot funkcija POT-TE-NA-VOZLIŠČE(v) s tem, da dodatno preveri ali je izhodna lokacija na voljo, da ne pride do zastojev. Za to funkcijo je premik na vozlišče sprejemljiv le, če je dovoljen tudi premik po izhodni povezavi. To se vidi na izpisu 10 vrstica 10, kjer se preverja izvedljivost premika iz prehodnega vozlišča.

Vhodni parameter je prehodno vozlišče. Funkcija sama zbere vse ostale podatke potrebne za izračun poti ter ugotovi, ali je premik dovoljen ali ne.

Izpis 9 Funkcija POT-TE-NA-VOZLIŠČE vrne pot za TE, ki gre ***na*** vozlišče. V kolikor je več možnih premikov, izbere le najoptimalnejšega

```
1: function POT-TE-NA-VOZLIŠČE(v)
2:     p_h ← 0                                   ▷ Najvišja prioriteta do sedaj
3:     ⟨s_b, d_b⟩ ← ∅
4:     if LOKACIJA-PROSTA(d) = false then        ▷ Vozlišče zasedeno?
5:         return ∅                              ▷ Da. Premik ni mogoč
6:     end if
7:     for all u ∈ λ(Inc⁻[v], v) do ▷ Preglej vsa vozlišča vhodno povezana v
8:         ⟨s_t, d_t⟩ ←POT-TE-IZ-VOZLIŠČA(u)
9:         if ⟨s_t, d_t⟩[0] = (s_t, v) then      ▷ Gre pot TE iz vozlišča u preko v?
10:            if PREMIK-DOVOLJEN(⟨s_t, d_t⟩[0]) = true then
11:                p_t ←DOLOČI-PRIORITETO(⟨s_t, d_t⟩[0])
12:                if p_t > p_h then             ▷ Višja od trenutno najvišje?
13:                    p_h = p_t                 ▷ Da. Shrani najboljše stanje
14:                    (s_b, d_b) ← (s_t, d_t)
15:                end if
16:            end if
17:        end if
18:    end for
19:    return ⟨s_b, d_b⟩          ▷ Premik TE možen na to vozlišče po tej poti
20: end function
```

Izpis 10 Funkcija POT-TE-PREKO-VOZLIŠČA vrne pot za TE, ki gre ***preko*** vozlišča. Funkcija tudi zagotovi, da je izstopna lokacija iz vozlišča prosta. V kolikor je možnih več premikov, vrne le najoptimalnejšega.

```
1:  function POT-TE-PREKO-VOZLIŠČA(v)
2:      p_h ← 0                                   ▷ Trenutno najvišja prioriteta
3:      ⟨s_b, d_b⟩ ← ∅
4:      if LOKACIJA-PROSTA(v) = false then        ▷ Vozlišče v zasedeno?
5:          return ∅                              ▷ Da. Premik ni mogoč
6:      end if
7:      for all u ∈ λ(Inc⁻[v], v) do ▷ Preglej vsa vozlišča vhodno povezana v
8:          ⟨s_t, d_t⟩ ←POT-TE-IZ-VOZLIŠČA(u)
9:          if ⟨s_t, d_t⟩[0] = (s_t, v) then      ▷ Gre pot TE iz vozlišča u preko v?
10:             if PREMIK-DOVOLJEN(⟨s_t, d_t⟩[1]) = true then
        ▷ Izhodna lokacija je prosta, torej ne bo blokirala prehodnega vozlišča
11:                 p_t ←DOLOČI-PRIORITETO(⟨s_t, d_t⟩[0])
12:                 if p_t > p_h then             ▷ Višja od trenutno najvišje?
13:                     p_h = p_t                 ▷ Da. Shrani najboljše stanje
14:                     (s_b, d_b) ← (s_t, d_t)
15:                 end if
16:             end if
17:         end if
18:     end for
19:     return ⟨s_b, d_b⟩        ▷ Premik TE možen preko vozlišča v po tej poti
20: end function
```

4.3 Elementi mreže logističnih poti

Na tem mestu bi rad opozoril na razliko med ***sosednjim vozliščem*** in ***direktnim povezanim vozliščem***, katerih definicija je opisana v poglavju 2.1 na strani 8. V tem poglavju je razlika pomembna in izraza uporabljam po definiciji.

Sokoban naloži vsa vozlišča ob inicializaciji in jih klasificira na več načinov. Ena klasifikacija je glede na stopnjo vozlišča, ker stopnja vozlišča določa, kako skozi vozlišče poteka promet. Na podlagi tega so vozlišča razdeljena na naslednje kategorije:

- Križišče (Junction) - množica vozlišč $v \in C$
- Zlitje (Joint) - množica vozlišč $v \in J$
- Razcep (Split) - množica vozlišč $v \in S$
- Tranzit (Transit) - množica vozlišč $v \in T$

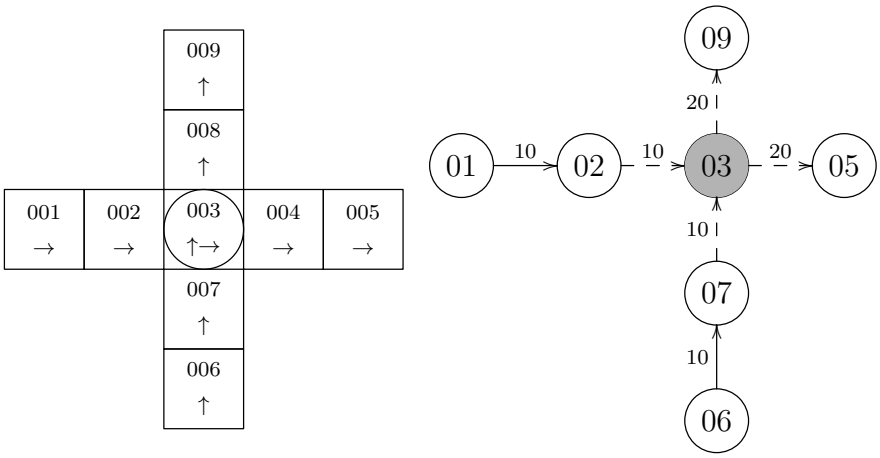
Vozlišče v lahko pripada le eni naštetih množic, ni pa nujno, da pripada katerikoli od njih, kot bomo videli kasneje. Zaporedje obdelave poteka po vrstnem redu, kot so naštete kategorije. V nadaljevanju sledi razlaga kako se obdela posamezno vozlišče. Kako se obdela množica vozlišč je prikazano na izpisu 20 v poglavju 5.1.7.

4.3.1 Križišča

Vozlišče $v \in V$ je klasificirano kot ***križišče***, kadar je njegova vhodnja stopnja $deg^-[v] \geq 1$ in izhodnja stopnja $deg^+[v] \geq 1$. To pomeni, da je vozlišče enako cestnemu. Za učinkovito delovanje mora voznik preveriti, ali bo križišče lahko zapustil preden zapelje vanj, sicer bo prišlo do zastoja, tudi če je katera od izhodnih poti prosta.

Pomembno je, da Sokoban zagotavlja, da je vozlišče-križišče zasedeno le za čas, ki ga potrebuje transportna enota, da ga prevozi in zapusti. Primer na sliki4.3: če se transportna enota premika od vozlišča 07 do vozlišča 09, Sokoban ne sme dovoliti premika na vozlišče 03, razen če je lokacija 08 prosta.

Ta problem je rešen tako, da je križiščno vozlišče v odgovorno za premik transportnih enot iz vseh sosednjih vozlišč (glej sliko 4.3) na v. Za vozlišče $v \in C$ kličemo Pot-TE-Preko-Vozlišča. Le kadar ni nobene transportne enote, ki bi prečkala v, se pokliče Pot-TE-Na-Vozlišče, da pridobi pot za transportno enoto na križiščno vozlišče v. Ta premik se izvaja le, kadar je



Slika 4.3: Vozlišče v je križišče, ker ima vhodno stopnjo $deg^-[v] \geq 1$ in izhodno stopnjo $deg^+[v] \geq 1$

križiščno vozlišče končna destinacija za transportno enoto. S tem Sokoban zagotavlja, da bodo vse transportne enote, ki morajo skozi to križišče najprej prečkale križišče, in šele ko na sosednjih vozliščih ni več transportnih enot, bo enota, ki ima za cilj križiščno vozlišče v lahko zapeljala nanj in ga zasedla. Implementacija je vidna v izpisu 11.

Za učinkovito kontrolo pretoka transportnih enot morajo biti vozlišča poleg križišča samega tudi vse sosednje lokacije. Podobno kot sta stop znak in semafor postavljena tik pred kržiščem in ne 100 m pred njim, mora biti vmesna destinacija vozlišče tik pred križiščem.

4.3.2 Zlitja

Vozlišče $v \in V$ je razvrščeno v ***zlitja***, kadar je njegova vhodna stopnja $deg^-[v] \geq 1$ in njegova izhodna stopnja $deg^+[v] = 1$. To je križišče, kjer se dve enosmerni ulici združita v eno, ali kjer se dve stezi avtoceste združita v eno samo.

Glavna naloga Sokobana na takem vozlišču je zagotavljati gladek pretok prometa. Pretok na obeh (ali več) vhodnih vozliščih mora biti uravnotežen, tako, da čakalni vrsti ne bosta predolgi. Sokoban mora upoštevati tudi prioritete in druge zahteve poslovnega procesa, ki vplivajo na pretok transportnih enot.

Kot v primeru križišča, morajo biti vozlišča za učinkovito kontrolo prometa tudi vse lokacije tik pred zlitjem. Če se na vozlišču v že nahaja transportna

Izpis 11 Obravnava križišča

```
 1: procedure Obdelaj-Križišče(v)
 2:     ⟨s, d⟩ ← Pot-TE-Preko-Vozlišča(v)
 3:     if ⟨u, v⟩ ≠ ∅ then                          ▷ Poišči premik preko v
 4:         Ustvari-Premik(⟨s, d⟩)
 5:     else                          ▷ Noben premik na voljo za preko križišča
 6:         ⟨s, d⟩ ← Pot-TE-Na-Vozlišče(v)
 7:         g ← Preberi-Cilj-TE(s)
 8:         if v = g then          ▷ Le če je končni cilj TE križišče v sme naprej
 9:             Ustvari-Premik(⟨s, d⟩)
10:         end if
11:     end if
12: end procedure
```

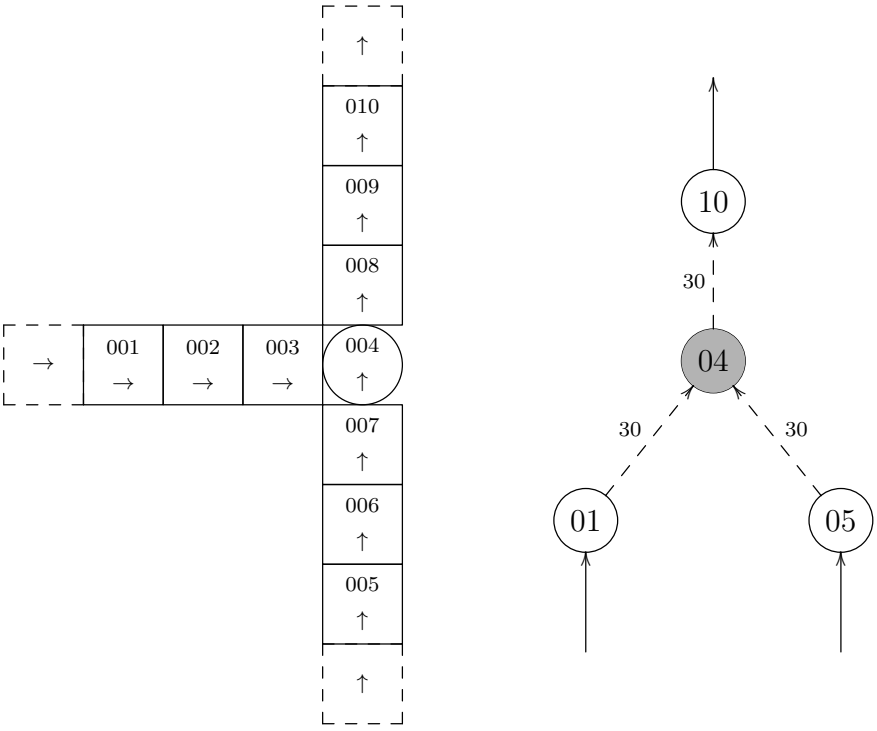
Izpis 12 Obravnava zlitja

```
 1: procedure Obdelaj-Zlitje(v)
 2:     ⟨s, d⟩ ← Pot-TE-Iz-Vozlišča(v)
 3:     if ⟨s, d⟩ ≠ ∅ then
 4:         Ustvari-Premik(⟨s, d⟩)
 5:     else
 6:         ⟨s, d⟩ ← Pot-TE-Na-Vozlišče(v)
 7:         if ⟨s, d⟩ ≠ ∅ then
 8:             Ustvari-Premik(⟨s, d⟩)
 9:         end if
10:     end if
11: end procedure
```

enota, mora Sokoban poskrbeti, da se ta odpelje naprej in ga sprosti. V nasprotnem primeru bodo nove transportne enote blokirane in Sokoban bo izgubljal sredstva za klicanje Pot-TE-Na-Vozlišče, ki bo vračala $\emptyset$, ker je ciljno vozlišče zasedeno. Implementacija je vidna v izpisu 12.

4.3.3 Razcepi

Vozlišče $v \in V$ je ***razcep***, kadar je njegova vhodna stopnja $deg^-[v] \leq 1$ in izhodna stopnja $deg^+[v] > 1$. Edina logistična odločitev v razcepnem vozlišču je usmeritev transportne enote na pravi izhod. Implementacija algoritma je vidna v izpisu 13.



Slika 4.4: Vozlišče v je kategorizirano kot zlitje, ker ima $deg^-[v] \geq 1$ in $deg^+[v] = 1$

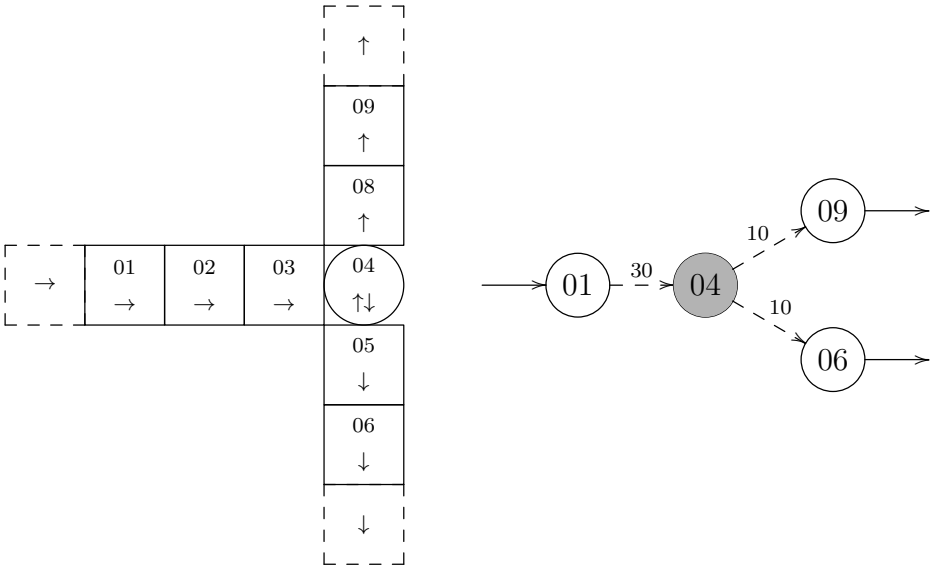
4.3.4 Tranzit

Vozlišče $v \in V$ je klasificirano kot ***tranzit***, kadar je njegova vhodna stopnja $deg^-[v] \leq 1$ in izhodna stopnja $deg^+[v] = 1$. Obstaja več različic tranzita, kot so prikazane na sliki 4.6

Na sliki 4.6 vidimo več različic transportnih vozlišč, ki ustrezajo pravilu o vhodnih in izhodnih stopnjah. Vendar so pri tem tudi nekatere izjeme. Kadar je eno od sosednjih vozlišč "tranzita" križišče, zlitje ali razcep, tranzitno vozlišče ne bo dodano v množico tranzitov T. Razlog je v tem, da imajo druge vrste vozlišč prednost in same poskrbijo za premike transportnih enot sosednjih vozlišč. Tak primer je vozlišče 01 na sliki 4.6d.

Tudi končno vozlišče 02 na sliki 4.6c ni dodano v množico tranzitov T. Vozlišče v s stopnjo $deg^-[v] = 1, deg^+[v] = 0$ ni dodano v nobeno kategorijo, ker vse premike nanj kontrolirajo sosednja vozlišča.

Vozlišče 02 na sliki 4.6d je tudi končno, vendar se lahko transportna enota



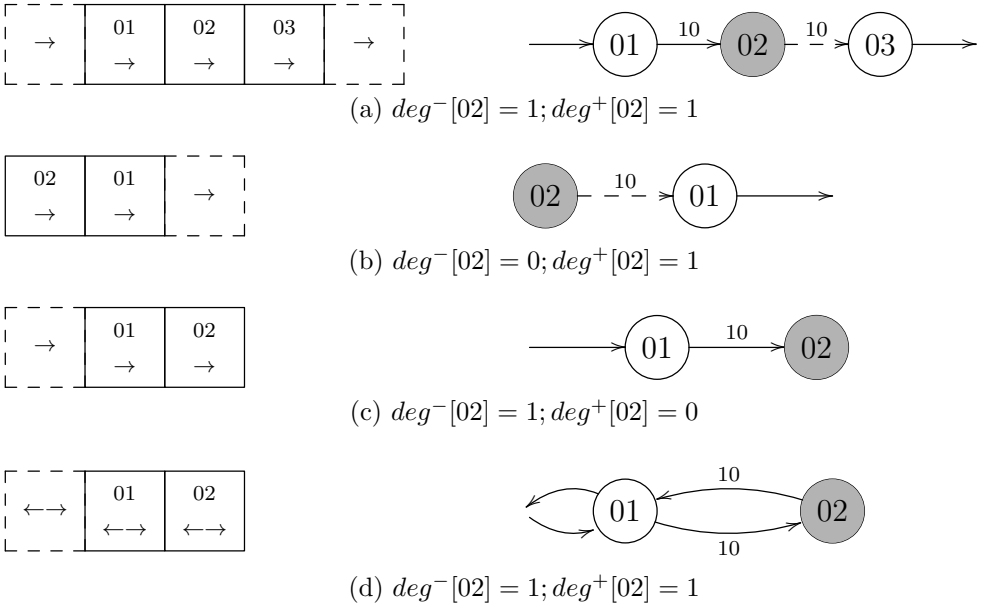
Slika 4.5: Vozlišče v je razcep, ker ima $deg^-[v] \leq 1$ in $deg^+[v] > 1$

Izpis 13 Obravnava razcepa

```
1: procedure Obdelaj-Razcep(v)
2:     ⟨s, d⟩ ← Pot-TE-Iz-Vozlišča(v)   ▷ je potrebno TE na v odpeljati?
3:     if ⟨s, d⟩ = ∅ then               ▷ Ne. Poglej za premik TE na razcep v?
4:         ⟨s, d⟩ ← Pot-TE-Na-Vozlišče(v)
5:     end if
6:     if ⟨s, d⟩ ≠ ∅ then               ▷ Obstaja izvedljiva pot, torej ustvari premik
7:         Ustvari-Premik(⟨s, d⟩)
8:     end if
9: end procedure
```

po istih vozliščih tudi vrača. Tudi v tem primeru je sosednje vozlišče 01 križišče in bo "urejalo" tudi vse premike za 02.

S tem, ko iz množice tranzitov $v \in T$ izpustimo končna vozlišča ($deg^-[v] = 1, deg^+[v] = 0$) in dvosmerna vozlišča, lahko opustimo klicanje funkcije Pot-TE-Na-Vozlišče, ki je precej počasnejša in manj učinkovita kot Pot-TE-Iz-Vozlišča. Ker je tranzitno vozlišče v skladišču najpogostejše s tem veliko prihranimo. Implementacija je vidna v izpisu 14.



Slika 4.6: Vozlišče 02 je tranzit, ker ima $deg^-[v] \leq 1$ in $deg^+[v] \leq 1$

Izpis 14 Obravnava tranzitov

```
1: procedure Obdelaj-Tranzit(v)
2:     ⟨s, d⟩ ← Pot-TE-Iz-Vozlišča(v)
3:     if ⟨s, d⟩ ≠ ∅ then
4:         Ustvari-Premik(⟨s, d⟩)
5:     end if
6: end procedure
```

4.3.5 Dvosmerna pot

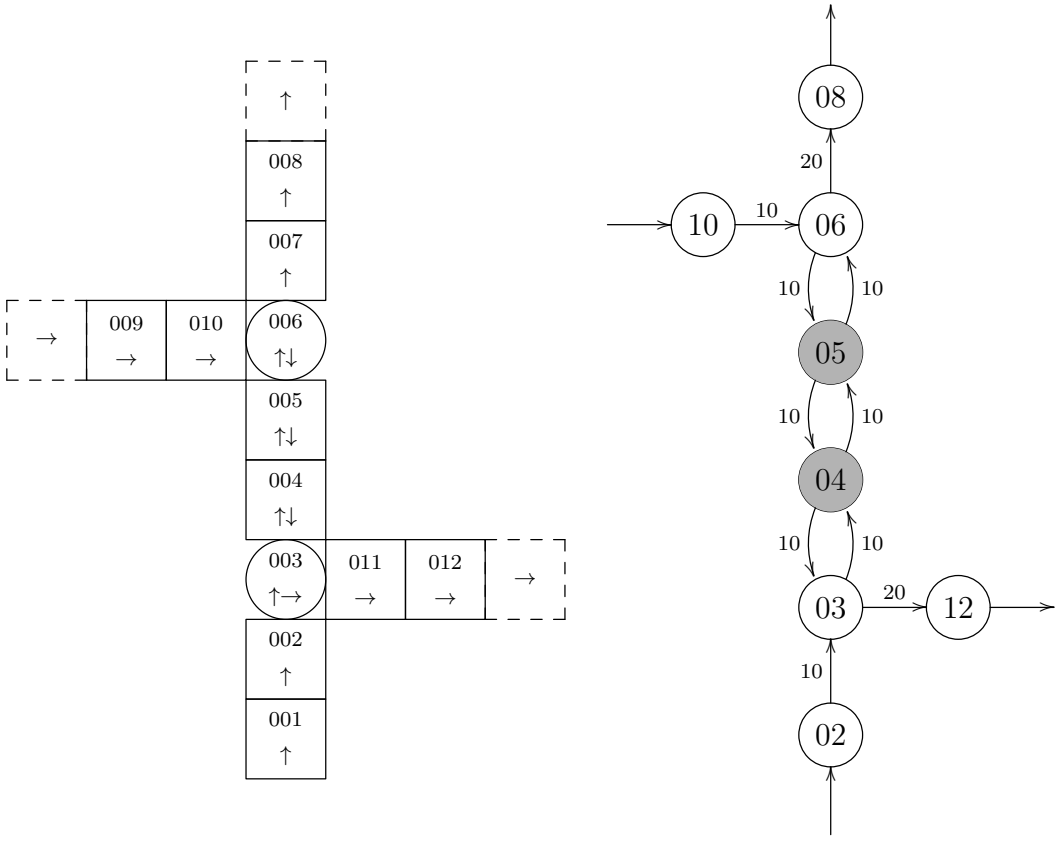
Pot, sestavljena iz dvosmernih vozlišč, je podobna situaciji z delom na cesti, kjer je en pas zaprt in promet poteka izmenično enosmerno. Običajno v tem primeru na obeh straneh namestijo semaforje, ki upravljajo promet tako, da ga v primernih intervalih spuščajo zdaj z ene, zdaj z druge strani.

Situacija na dvosmerni transportni poti je zelo podobna. Nobena še tako sofisticirana logistika ne more spremeniti dejstva, da je dvosmerna pot ozko grlo (bottleneck) v transportnem sistemu. Zato se uporablja le na območjih z majhnim stalnim pretokom transportnih enot ali majhnim številom preklopov smeri delovanja.

Dvosmerna vozlišča so križiščna vozlišča in delujejo kot eno veliko križišče.

Pogoj, da lahko transportna enota prevozi dvosmerni del, je:

- izhodna povezava mora biti prosta in na voljo (prazna, naprava mora delovati),
- znotraj dvosmernega dela ne sme biti nobene transportne enote, ki se giba v nasprotno smer.



Slika 4.7: Dvosmerna pot

Ker so vsa vozlišča na dvosmerni poti dejansko križišča, jih upravlja enak algoritem kot križišča 11.

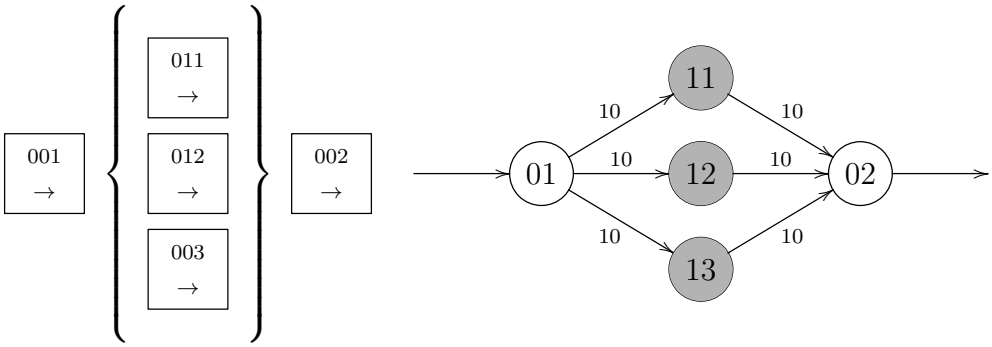
4.3.6 Čakalnica (buffer)

V skladišču je precej primerov, ko ena premikajoča se naprava streže več delovnih mest s spreminjajočim se pretokom transportnih enot. Kadar so zahtevane drobne materialne operacije, je treba v kratkem času dostaviti več transportnih enot, ko pa je v obdelavi velika količina materiala naenkrat, zadošča manjše število. Podoben problem predstavlja uporabnik, ki večkrat zapored potrebuje isto transportno enoto.

Da bi se sistem lahko hitro odzval na nenadno povečanje prometa in uporabniku hitro postregel z isto transportno enoto, so v bližini delovnega mesta na voljo prostori rezervirani za čakajoče transportne enote.

Vozlišča, ki sestavljajo čakalnico, ne pripadajo nobeni prej naštetih kategorij. Teoretično gledano so to tranziti, vendar, ker imajo za sosede križišča, zlitja ali razcepe, vse premike zanje upravljajo ta vozlišča.

Postavljeno je pravilo, da imajo vse povezave, ki vstopajo ali zapuščajo čakalna vozlišča, enako utež. Še več - utež poti preko čakalnice mora biti enaka, ne glede na to, preko katerega čakalnega vozlišča bo transportna enota prepeljana. Sokoban prepozna vozlišče kot čakalnico le če je pot preko nje enaka najkrajši možni poti. V kolikor bi imela povezava med 01 in 11 na sliki 4.8 utež 15 namesto 10, Sokoban vozlišča 11 ne bi prepoznal kot čakalnico, saj pot preko nje ne bi bila najkrajša.



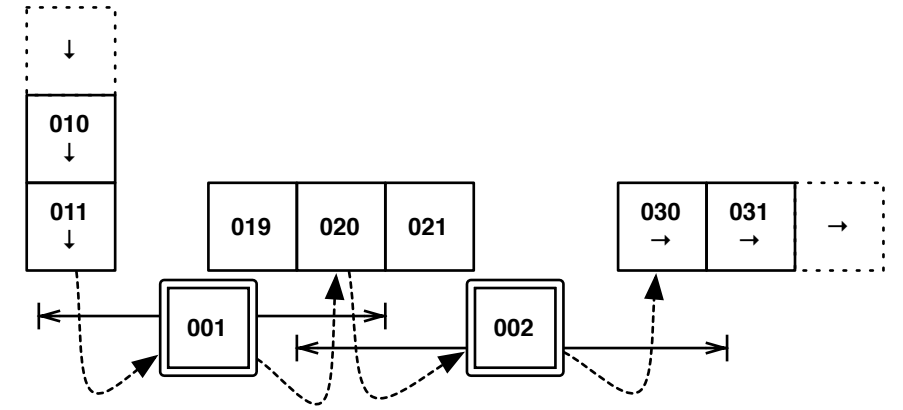
Slika 4.8: Čakalnica (Buffers)

4.3.7 Izmenjalno vozlišče (Swap)

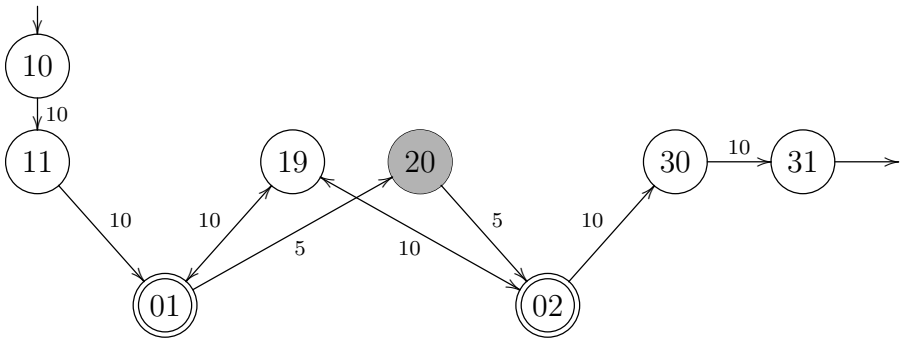
V delu skladišča s premikajočimi se napravami obstajajo situacije, ko se dve napravi gibata po isti tračnici. S tem se poveča propustnost sistema, nastopijo

pa novi robni primeri. Ker se napravi gibata po isti tračnici, se ne moreta gibati po celotni dolžini, zato je doseg posamezne omejen in ne pokriva vseh lokacij.

Za rešitev problema porabimo nekaj lokacij, ki so dostopne obema napravama in jih uporabimo za izmenjevalna mesta. Te lokacije so posebej rezervirane in označene kot "swap", čeprav so to običajno navadne skladiščne lokacije, ki se nahajajo v bližini sredine tračnice. Sokoban to upošteva in na te lokacije transportnih enot nikoli ne shranjuje.



(a)



(b)

Slika 4.9: 001 in 002 sta premični napravi, ki se gibljeta na skupni osi. Vsaka naprava ima svoje lastno območje delovanja na osi. Srednji del osi je skupen. Na tem skupnem območju se postavi izmenjalna vozlišča preko katere si lahko napravi predata transportno enoto. Na sliki je 020 izmenjalna lokacija. 019 in 021 sta skladiščni lokaciji. Tam se palete skladiščijo.

Nastavljanje uteži povezav, ki vstopajo in zapuščajo izmenjalno vozlišče, zahteva nekaj načrtovanja. Upoštevati je treba naslednja pravila:

- Utež direktne poti transportne enote mora biti vedno nižja kot utež poti z uporabo izmenjalnega vozlišča (s tem zagotovimo, da izmenjava ne bo prišla v poštev, če obstaja direktna pot).
- Utež poti preko izmenjalnega vozlišča mora biti vedno nižja od poti preko ostalih lokacij. (s tem zagotovimo, da ne bi kakšna druga lokacija postala izmenjalna).

Če so uteži pravilno nastavljene, bo funkcija FIND-SHORTEST-PATH:

- vrnila direktno povezavo, kadar ta obstaja,
- sicer vrnila povezavo preko izmenjalne lokacije.

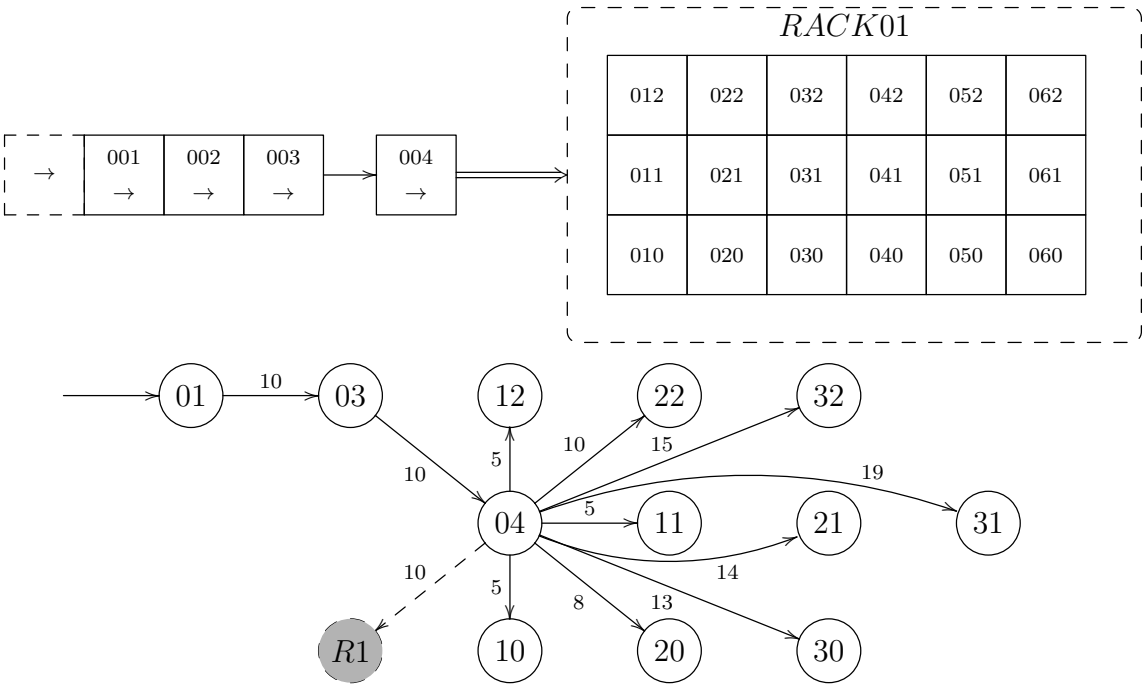
4.3.8 Skupinsko vozlišče

Skupinsko vozlišče je navidezno vozlišče, s katerim predstavimo množico vozlišč, ki ustrezajo nekemu cilju. Tipičen primer skupinskega vozlišča v skladišču so visokoregalne skladiščne lokacije.

ERP sistem ne ve, ali je določena skladiščna lokacija prazna, prav tako ne ve, ali bo še prazna, ko pride transportna enota do skladišča. Zato skladiščenje transportnih enot prevzame Sokoban. ERP sistem tudi običajno nima podatka, kje se nahaja transportna enota, zato se skupno vozlišče uporabi kot začetna lokacija v transportnem nalogu.

Če je cilj transportne naloge skupinsko vozlišče, potem za Sokoban ni razlik do trenutka, ko pride na vrsto premik po povezavi proti skupnemu vozlišču. V tem trenutku Sokoban pokliče funkcijo, ki vrne ustrezno vozlišče za naslednji premik. V transportnem nalogu se potem ažurira cilj z dejanskim vozliščem. Na sliki 4.10 je prikazan primer. Če je cilj naloge skladiščne lokacije predstavljene z skupinskim vozliščem $R1$, potem Sokoban namesto da ustvari premik po povezavi $04 \rightarrow R1$, pokliče funkcijo, ki vrne ustrezno vozlišče, glede na tip in zasedenost skladišča (kaotično, dvojna globina, ...).

Sokoban začne z izvajanjem *nove* transportne naloge le če se transportna enota nahaja na začetni lokaciji. Skupinsko vozlišče nam omogoča, da lahko kot začetno mesto podamo množico ustreznih vozlišč. Na samo izvajanje začetno skupinsko vozlišče nima vpliva, saj Sokoban vedno dela premik od trenutne lokacije transportne enote.



Slika 4.10: Skupinski cilji

4.4 Alternativne poti

Transportne enote se v skladišču običajno premikajo proti cilju po najbolj optimalni poti. Do cilja obstajajo večkrat tudi alternativne poti, preko katerih se preusmeri transportne enote v primeru izpada segmenta skladišča, zasičenja na določenem segmentu skladišča ali poslovnih pravil kot so kontrola kvalitete.

V Sokobanu obstajata dva mehanizma za preusmeritev po alternativni poti. Prvi mehanizem je ***Obvoz***. Obvoz je avtomatski standardni mehanizem za preusmeritev transportnih enot na alternativne poti v primeru delnih izpadov skladišča. Drugi mehanizem je ***Dinamična izbira poti*** s katero rešujemo vse ostale probleme, ki pa so odvisni od posameznega skladišča in se zato vedno implementirajo z vtičnikom. Sledi podrobnejši opis obeh mehanizmov.

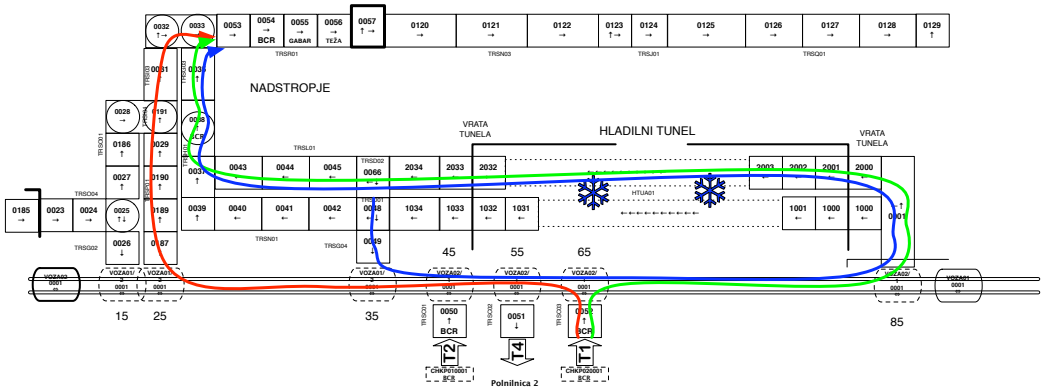
4.4.1 Dinamična izbira poti

Transportni nalog lahko vsebuje parametre, zaradi katerih blago ne sme na cilj po najbolj optimalni poti ampak mora narediti obvoz preko neke točke.

Razlogov za dinamični obvoz je veliko. Razlog je lahko poslovne narave, kot na primer naključna kontrola kvalitete iz proizvodnje, kjer se transportna

enota zapelje proti kontrolnemu mestu, lahko je povsem pretočne narave, kjer zaznamo zasičenje na določenem delu skladišča in preusmerimo promet preko alternativne poti, da razbremenimo ozko grlo.

Na sliki 4.11 vidimo primer iz realnega sveta. Palete z mlečnimi izdelki prihajajo iz produkcije na vhod *TRSC030052*. Ciljno mesto teh palet je vedno visoko regalno skladišče *T10*. A če ima transportni nalog parameter, ki pove, da se morajo mlečni izdelki na paleti najprej ohladiti na 4°C, jo Sokoban namesto po optimalni poti, ki je označena z rdečo barvo, zapelje v skladišče preko enega izmed dveh hladilnih tunelov. Pot je označena z zeleno barvo. Ob izhodu iz hladilnega tunela, se parameter za hlajenje odstrani in paleta se zapelje v skladišče po optimalni poti. V kolikor se mlečni izdelki v prehodu skozi hladilni tunel niso dovolj ohladili, se parameter za hlajenje ne spremeni. V tem primeru Sokoban preusmeri paleto znova skozi hladilni tunel. Pot je na sliki označena s modro barvo.



Slika 4.11: Na vhod *TRSC030052* prihajajo palete z mlečnimi izdelki iz produkcije, ki so namenjene v visoko regalno skladišče. V kolikor ima transportni nalog za paleto vključen parameter za hlajenje, se odpelje skozi tunel, ki je označen z zeleno barvo. Če hlajenje ni zahtevno, se odpelje v skladišče po optimalni poti označeni z rdečo barvo.

Vse dinamične poti se implementira glede na specifične zahteve preko vtičnika opisanega v poglavju 5.1.5.

4.4.2 Obvoz

Druga oblika alternativnih poti so obvozi. Skladišča imajo med glavnimi povezavami zaradi zahtev po redundanci in robustnosti več možnih alternativnih

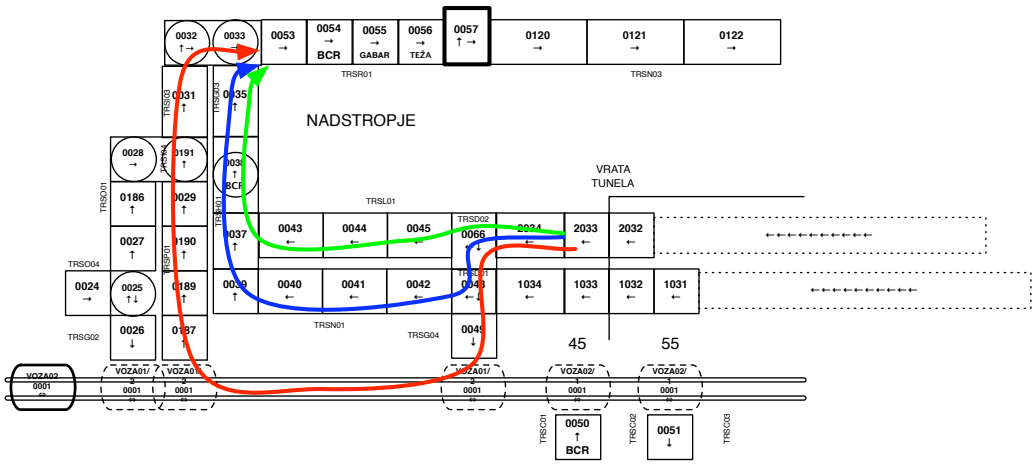
poti. V cestnem prometu si to predstavljamo kot večja opravila v cestnem prometu ali popravilo vodovoda, kjer določene ulice zaprejo za promet. Ves promet se usmeri preko alternativnih poti, ki so označene z prometnim znakom "Obvoz".

V avtomatskem skladišču to poteka tako, da v primeru okvare posamezne naprave, ali kadar je potrebno vzdrževanje, operater to napravo izklopi. Sokoban dobi obvestilo o izklopu naprave in na novo zgradi graf G', ki vsebuje le delujoče povezave med vozlišči. Ker Sokoban ustvarja naloge povezavo po povezavo ima novi graf takojšen vpliv na vse premike transportnih enot v skladišču, kar je pomembna odlika sistema.

Obvozi obstajajo

Ker ima Sokoban generični iskalni algoritem in ker poteka iskanje najprej po grafu z delujočimi povezavami, najde najoptimalnejšo pot med alternativami.

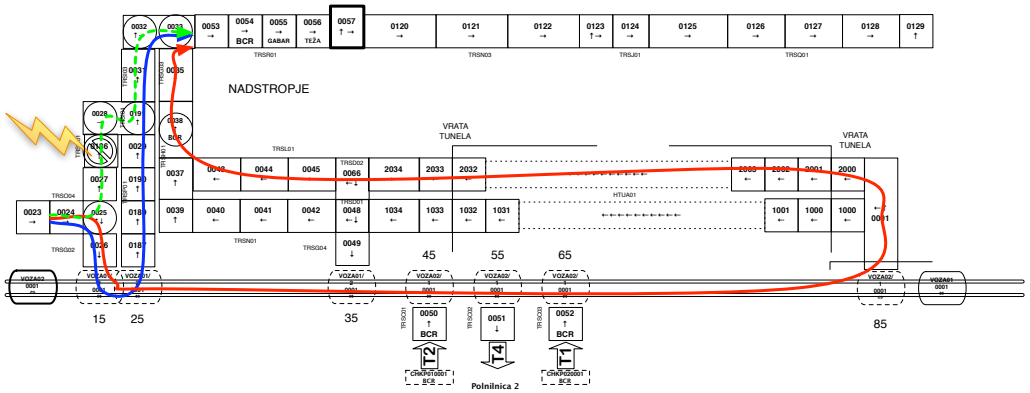
Na slikah 4.12, 4.13 in 4.14 je prikazanih nekaj primerov obvozov.



Slika 4.12: Obvoz iz hladilnega tunela. Z modro barvo je označena optimalna pot. Z rdečo barvo je označena pot, če pride do izpada naprave *TRSL01*. Z zeleno barvo je označena pot, če pride do izpada naprave *TRSH01* ali *TRSG03*, oziroma do sočasnega izpada naprav *TRSL01* in *TRSN01*.

Obvozi ne obstajajo

Ogledali smo si kako rešujemo obvoze, ko alternativne poti obstajajo. Obstajajo tudi primeri, ko alternativnih poti ni. V tem primeru ločujemo primer, ko



Slika 4.13: Slika prikazuje optimalno pot z zeleno barvo. V kolikor pride do izpada naprave *TRSO01*, obstajata še dve alternativni poti označeni z rdečo in modro barvo. Pot preko hladilnega tunela je seveda le izhod v sili.

se je transportni nalog že začel izvajati in se transportna enota že premika po skladišču in ko se ne.

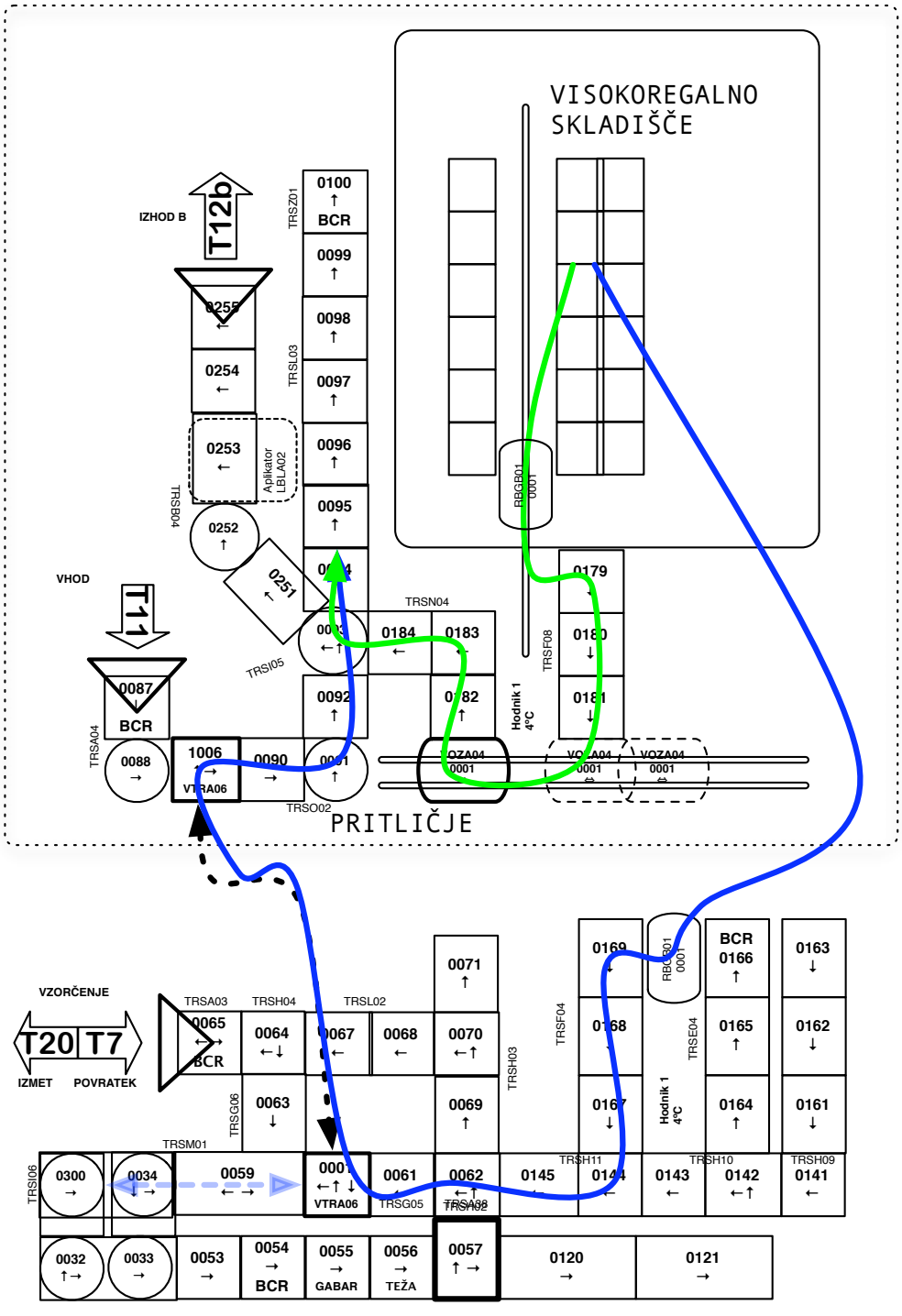
V primeru, da se transportni nalog še ni začel izvajati, se izvajanje zadrži, saj Sokoban ne najde celotne poti do cilja. Če je transportna enota še v visokoregalnem skladišču, čaka tam. Če je še na vhodu, obstoji tam, saj je bolje, da obstoji na vhodu, kot da zablokira dele skladišča, ki delujejo.

Drugače je s transportnimi enotami, ki so že na poti. V tem primeru potrebujemo poleg grafa z delujočimi povezami G' tudi graf G z vsemi povezavami. Če alternativna pot ne obstaja, Sokoban preklopi iskanje poti za aktivne transportne naloge na graf G, sicer bi transportna enota obtičala, kjer bi se v tistem trenutku nahajala.

Oglejmo si primer na sliki 4.1. Imamo transportno enoto na vozlišču *TRSS010017* ki je namenjena na izhod *T12a* v pritličju *TRSB020101*. Predpostavljamo, da je prišlo do izpada naprave *TRSB02*. Če ne bi imeli grafa G, bi transportna enota obtičala na *TRSS010017*, saj iskalni algoritem ne bi našel poti. Taka transportna enota bi povsem zablokirala vhod iz polnilnice *T3*, iz katere gredo palete v visoko regalno skladišče, prav tako bi zablokirala dovoz embalaže iz vhoda T5, ki ima vedno cilj *T4*. Tak učinek je nezaželen. Zato v primeru, ko ni alternativne poti, Sokoban pelje transportno enoto po optimalni poti po grafu G, dokler le-ta ne pride do izklopljene naprave. V tem primeru bi transportna enota obtičala na vozlišču *TRSZ01*. Vedno lahko pride do zasičenja, če obstaja zadostno število transportnih enot, ki potujejo na nedosegljiv cilj. V tem primeru je potreben poseg operaterja, ki preusmeri

transportne enote na drug, dosegljiv cilj.

Obnašanje obvozov v primeru izpadov je možno prilagoditi specifikam skladišča preko vtičnika, kjer dopišemo dinamične obvoze v primeru izpada naprave.



Slika 4.14: Na sliki je z zeleno barvo označena optimalna pot iz visoko regalnega skladišča v klet. V primeru izpada ene izmed naprav na zeleni poti je možna še alternativna pot, označena z rdečo barvo. Segment, po katerem se nahaja večina poti označene z rdečo barvo, se nahaja eno nadstropje višje v pritličju. Palete prispejo v klet preko vertikalnega dvigala *VTRA06*.

4.5 Naprave in seznam premikov, ki jih naprava lahko opravi

Avtomatsko skladišče je sestavljeno iz lokacij, ki se nahajajo na aktivnih in pasivnih napravah. Aktivne naprave so tiste, ki so sposobne izvajati premike transportnih enot. Pasivne naprave so tiste, ki transportne enote le "nosijo", niso pa jih sposobne premikati in v glavnem služijo za grupiranje lokacij. Skladiščna vozlišča v regalu se npr. nahajajo na pasivni psevdo-napravi "RACK". Poleg tega, da je nadrejena lokacijam, ima naprava status delovanja, ki pove ali je na razpolago ali ne. Če naprava ni na razpolago, ni na razpolago nobena lokacija na njej. Vsako vozlišče v avtomatskem skladišču se prav tako nahaja na napravi.

4.5.1 Doziranje premikov napravam

Nekatere naprave zaradi svoje narave (odzivnost, mehanska izvedba) zahtevajo določeno časovno zaporedje nalog, če jih želimo optimalno izkoristiti. Naprave, ki se premikajo in prekladajo transportne enote, so pogosto ozko grlo sistema. Zato jim običajno serviramo eno nalogo vnaprej, še preden so končale s prvo. Tako naprava nikoli ne čaka na nadzorni sistem in čas med dvema premikoma zmanjšamo na minimum.

Druge naprave upravljajo z urejeno vrsto transportnih enot in tem lahko za vsako transportno enoto kar takoj izdamo nalog za premik do konca vrste, naprava pa sama poskrbi, da se transportne enote peljejo ena za drugo.

4.6 Logistične izboljšave

4.6.1 Enosmerke (queueing)

V avtomatskem skladišču se prav lahko zgodi, da je neka logično ravna transportna linija razdrobljena na več naprav, kar tudi pomeni več vozlišč. Sokoban mora za vsak premik do nadaljnjega vozlišča izračunati celotno pot do ciljnega mesta, poleg tega mora ob vsakem takem dogodku analizirati stanje na napravah, kar seveda zahteva svoje vire.

Ko je določen del poti sestavljen iz samih tranzitnih vozlišč (in morebitnega začetnega zlitja) in na tem segmentu naprave zmorejo samostojno opravljati "kontrolo pretoka", lahko povezavam med tranziti nastavimo atribut uvrščanje (queueing), kar je razvidno v vrstici 29 na izpisu 6.

Potrebujemo novo funkcijo $\kappa : E \to \mathbf{B}$, ki vrne vrednost atributa "uvrščanje" na povezavi. V izpisu 15, je prikazano kako ta algoritem deluje. Za delovanje potrebujemo še spremenljivki f in q. S prvo si zagotovimo prvi vstop v zanko, saj je premik po povezavi do direktno povezanega vozlišča nadzorovan s strani Sokobana. Vsi nadaljnji premiki po poti bodo ustvarjeni le, če je to dovoljeno v konfiguraciji. Zunanja procedura USTVARI-PREMIK-NAPRAVI ustvari premik med vozliščema u in v in ga pošlje napravi. Implementacija je prikazana v poglavju 5.1.5 na strani 64.

Izpis 15 Procedura za ustvarjanje nalogov za premik

```
1: procedure USTVARI-PREMIK(⟨s, d⟩)
2:     f ← true          ▷ S spremenljivko si zagotovimo prvi vstop v zanko
3:     q ← false                    ▷ Definicija spremenljivke za uvrščanje
4:     while ⟨s, d⟩ ≠ ∅ in (q = true ali f = true) do
           ▷ Zanka se izvede le, če pot ⟨s, d⟩ ni prazna množica povezav, in je
       uvrščanje(q) ali vstop(v) = true
5:         f ← false                          ▷ Prvo izvajanje se je izvedlo
6:         (u, v) ← ⟨s, d⟩[0]        ▷ Pridobi prvo povezavo iz seznama v poti
7:         POŠLJI-PREMIK-NAPRAVI((u, v))          ▷ Ustvari premik napravi
8:         ⟨s, d⟩ ← ⟨s, d⟩ − {(u, v)}          ▷ Odstrani prvo povezavo s poti
9:         q ← κ(⟨s, d⟩[0])          ▷ Preveri uvrščanje naslednje povezave
10:    end while
11: end procedure
```

4.6.2 Napredne naprave

Do sedaj smo si ogledali, kako transportne enote premikamo po preprostih napravah. V avtomatskem skladišču pa so tudi naprave, ki zahtevajo izredno visoko odzivnost. Pri njih ni sprejemljivo, da po tem, ko opravijo trenutno nalogo, sistem potrebuje celo sekundo za pripravo nove.

Take naprave so recimo regalna dvigala, veliki večosni roboti, ki uskladiščujejo in izskladiščujejo transportne enote v prostranih regalih s tisoči regalnih mest. Na sliki 4.2 so to označeni z imeni naprav *RBGA01* in *RBGA02*, na 4.1 pa *RBGC01, RBGC02, RBGC03* in *RBGC04*. Vozlišča na teh napravah imajo stopnjo $deg[v]$ lahko tudi 10.000 in več. Imajo precej več logike kot jo ima *Transporter* in je hiter odzivni čas (manj kot 500ms) težko zagotoviti. Daljši odzivni čas pa se pri 3000 in več premikih na mesec že precej pozna pri kapaciteti.

To rešujemo tako, da te naprave sprejmejo po dve in več nalog sočasno, ki jih potem naprava sama izvaja zaporedno. Izključene so vse generične kontrole ali je vozlišče prazno, ali se bo možno premakniti preko križišča, saj ta stanja niso pomembna pri ustvarjanju nalog za naprej.

Te naprave "simulirajo" kakšno bo v prihodnosti stanje transportnih naprav glede na izdane naloge za premik po grafu G. Naprava mora to upoštevati in nikoli narediti premika, ki bi lahko povzročil zastoj. Gre za dokaj kompleksno obdelovanje, ki je zunaj obsega te diplomske naloge.

Poglavje 5

Arhitektura

V poglavju 4 smo si ogledali, kako uporabljamo iskalne algoritme v skladišču. V tem poglavju je predstavljena arhitektura Sokobana: pojasnil bom, kako je zgrajen, kakšne so bile zahteve, ki jim mora ustrezati, prikazal primere, kako deluje in opisal, kakšne tehnologije so bile uporabljene za dosego cilja.

5.1 Zahteve in koncepti programa

Avtomatizirana skladišča zahtevajo zelo veliko stopnjo zanesljivosti delovanja. Le enourni izpad v produkcijskem avtomatskem skladišču lahko povzroči ustavitev celotne proizvodnje. To je povezano z velikim stroški, saj je lahko ponoven zagon proizvodnje večuren proces. Izpad v distribucijski skladiščih prav tako povzroči velike stroške in nevšečnosti, saj blago strankam ni dostavljeno pravočasno.

5.1.1 Neinteraktivnost

Sokoban ni interaktivni program in kot tak nima uporabniškega vmesnika. Zaradi tega je manj kompleksen, manj kompleksni sistemi pa imajo po definiciji manj napak in višjo zanesljivost. Sokoban je del sistema, ki deluje samodejno, za uporabniški vmesnik skrbi modul Vizualizacija. Komunikacija med njima poteka preko podatkovne baze, v bazi je tudi konfiguracija in vsi podatki, ki jih logistika potrebuje.

5.1.2 Robustnost (24/7)

Posebna pozornost je posvečena robustnosti. Za zagotavljanje visoke stopnje razpoložljivosti sem implementiral naslednje mehanizme:

- v primeru težje napake, na primer izgube povezave z bazo, se Sokoban odklopi, počaka 30 sekund, ponovno vzpostavi začetno stanje in poskusi znova.
- generične rešitve v samem Sokobanu: iz prakse se je izkazalo, da večina problemov nastane kot posledica specifičnega programiranja preko vtičnikov.
- če se Sokoban ne javi več kot eno minuto, stražar (watchdog) pošlje opozorilo preko elektronske pošte lokalnemu operaterju in nadzornemu centru.

5.1.3 Visoka odzivnost

Za tekoč pretok v skladišču je potrebna zelo visoka odzivnost: cilj je, da se Sokoban odzove v manj kot 500 milisekundah. Za doseganje tega cilja sem uporabil naslednje prijeme:

- natančno analiziranje dogodkov in obdelava le tistih vozlišč, na katerega bi dogodek lahko imel vpliv.
- segmentacija in paralelna obdelava večjega sistema: s tem lahko izkoristimo večprocesorske in večjedrne sisteme.
- v primeru zahtevnejših naprav in v vozliščih z veliko stopnjo, se uporabljajo naprave, ki sprejmejo dve in več nalog vnaprej. Ker izvajanje posamezne naloge (dejanski premik transportne enote) ne traja manj kot 10 sekund, ima s tem Sokoban na voljo precej več časa za izračun naslednje naloge.

5.1.4 Konfiguracija (xml)

Skladišče je sestavljeno iz ogromnega števila vozlišč in povezav ter ostalih parametrov, potrebnih za delovanje, vizualizacijo, ipd. Zato za konfiguracijo skladišča uporabljamo XML strukturo, kar močno olajša konfiguracijo, saj povprečna konfiguracija obsega dve do tri strani XML-ja.

Podatki podani v XML obliki so vhodni parametri za program "whsconfigurator". Ta razpozna vhodne podatke in pripravi SQL stavke za konfiguracijo skladišča v podatkovni bazi.

Okrnjen primer (zaradi preglednosti) XML konfiguracije je prikazan na izpisu 16. XML je sestavljen iz treh osnovnih gradnikov. To so:

- naprave (devices)
- skladiščna mesta (rack)
- povezave (path)

Definicijo naprave, brez povezav, lahko vidimo na izpisu 16 v vrstici 3, kjer je definirana naprava *RBGC01*. Ima eno vozlišče, to je *RBGC010001*. Blok med vrstico 3 in 5 bo ustvaril

- napravo *RBGC01*,
- vozlišče *RBGC010001*.

V vrstici 9 lahko vidimo definicijo naprave *TRSA01*, ki ima eno vozlišče ***TRSA010112***, ki je sosedno vozlišču ***TRSY011001***. Ta definicija bo ustvarila:

- napravo *TRSA01*,
- vozlišče *TRSA010112*,
- povezava med *TRSA010112* in *TRSY011001*.

Bolj zanimiv je blok med vrstico 19 in 27, ki ustvari:

- napravo *TRSN03*,
- vozlišči *TRSN030120* in *TRSN030123*,
- lokaciji *TRSN030121* in *TRSN030122*,
- povezavo na napravi med vozliščema *TRSN030120* in *TRSN030123*. Index pri lokaciji je sekvenčna številka lokacije na povezavi,
- ter povezave med vozlišči *TRSN030123* in sosednimi vozlišči *TRSJ010123* na napravi *TRSJ01* in *TRSH080139* na napravi *TRSH08*.

Izpis 16 Konfiguracija skladišča poteka preko XML-ja

```
<warehouse>
  <devices>
    <device tag="RBGC01" type="RBG" variant="C" descr="Regalno dvigalo 01">
      <place tag="RBGC010001" index="1" />
    </device>
    <device tag="RBGC02" type="RBG" variant="C" descr="Regalno dvigalo 02">
      <place tag="RBGC020001" index="1" />
    </device>
    <device tag="TRSA01" type="TRS" variant="A" descr="Transporter A01">
        <place tag="TRSA010112" index="1">
           <neighbour name="TRSY011001" />
        </place>
    </device>
    <device tag="TRSY01" type="TRS" variant="A" descr="Vertikalc Y01">
        <place tag="TRSY011001" index="1">
            <neighbour name="TRST010002" />
        </place>
    </device>
    <device tag="TRSN03" type="TRS" variant="A" descr="Transporter N03">
        <place tag="TRSN030120" index="1" />
        <place tag="TRSN030121" index="2" />
        <place tag="TRSN030122" index="3" />
        <place tag="TRSN030123" index="4">
            <neighbour name="TRSJ010124" />
            <neighbour name="TRSH080139" />
        </place>
    </device>
  </devices>

  <racks>
    <rack name="RACK01" descr="rack 1" parent="WHS01">
      <block aisle="01" side="R" xfrom="1" xto="50" yfrom="1" yto="8" />
      <block aisle="01" side="L" xfrom="1" xto="50" yfrom="1" yto="8" />
      <block aisle="02" side="R" xfrom="1" xto="50" yfrom="1" yto="8" />
      <block aisle="02" side="L" xfrom="1" xto="50" yfrom="1" yto="8" />
    </rack>
  </racks>

  <paths>
    <path name="RACK01->RBG1" from="RACK01" to="RBGC010001" seq="false"
expandsource="true" />
    <path name="RBG1->RACK01" from="RBGC010001" to="RACK01" seq="false"
expandsource="true" />
    <path name="RBG->TRSF" from="RBGC010001" to="TRSF040169" seq="false" />
    <path name="RBG->TRSE" from="TRSE040166" to="RBGC010001" seq="false" />
    <path name="RBG->TRSF" from="RBGC010001" to="TRSF080179" seq="false" />
  </paths>
</warehouse>
```

Gradnik za opis naprave nam omogoča zapis vsega, kar je povezano z napravo, t.j. lokacije na napravi, vozlišča na napravi, ter sosednjih vozlišč naprave.

Drugi gradnik so skladiščne lokacije. Tipično so organizirane v regale ali podobne skupine in jih je lahko zelo veliko, zato hkrati definiramo cel blok. Med vrsticama 31 in 36 imamo definirano skladišče *RACK01*, ki ima dva hodnika. Vsak hodnik ima na levi in desni strani 400 skladiščnih lokacij, ki so tudi vozlišča. Skupno torej 1600 lokacij. Te lokacije dobijo oznako sestavljeno po standardnem pravilu, možna je pa tudi prilagoditev preko oblikovalca.

Zadnji gradnik so povezave. To so povezave, ki jih ni praktično opisati posredno hkrati s konfiguracijo naprav in prostorov. V vrstici 40 in 41 je prikazano, kako narediti povezave med vozliščem in množico skladiščnih lokacij. Vrstica 40 naredi povezavo med *RBGC010001* in "skupnim vozliščem" *RACK01* ter 1600 povezav med vozliščem *RBGC010001* ter vozlišči znotraj *RACK01*. V vrstici 41 se naredi enako število povezav v nasprotni smeri.

5.1.5 Vtičniki za prilagoditev

Sokoban je eden za vsa skladišča, ne glede na konfiguracijo. Občasno (v 20% primerov) se pojavijo zahteve po prilagoditvah. Prilagoditve so lahko poslovne narave ali pa imamo primer strukture, ki jo generična implementacije ne podpira.

Za pokrivanje teh dodatnih zahtev ima Sokoban razmeroma preprost vtičnik, preko katerega lahko implementiramo praktično vse dodatne zahteve. Vtičnik je zaradi lažje implementacije in vzdrževanja spisan v programskem jeziku PL/SQL in je viden na izpisu 17.

Funkcija **GET_DETOUR_LOCAT_ID** skrbi za obvoze in dinamično izbiro poti glede na parametre transporta, ki smo si jih ogledali v poglavju 4.4.1. Kot vhodni parameter dobi identifikacijsko številko transporta (p_cmo_id) in izbrani cilj.

Preko ***identifikacijske številke transporta*** lahko pridobimo vse ostale podatke, kot je identifikacija transportne enote, trenutna lokacija transportne enote, začetna lokacija transportne enote (start), ciljna lokacija, poslovno naročilo pripeto na ta transport itd. Ker imamo na tem mestu tudi direkten dostop do baze, upoštevamo tudi stanje naprav. Na podlagi vseh teh informacij lahko vtičnik izračuna, kdaj je potrebno transportno enoto dinamično preusmeriti preko alternativne poti. Funkcija vrne vozlišče $v \in V$, preko katerega bo transportna enota preusmerjena.

Preko funkcije **CHECK_IF_MOVE_ALLOWED** lahko dopolnimo funkcionalnost funkcije PREMIK-DOVOLJEN, iz poglavja 4.2.1. Funkcija v vtičniku vrača

Izpis 17 Programski vmesnik (API) od Sokobanovega vtičnika

```
1  CREATE OR REPLACE PACKAGE sokoban_plugin_pkg AS
2
3  FUNCTION get_detour_locat_id
4           (p_cmo_id  IN cmo.cmo_id%TYPE,
5            p_dest_id IN locat.locat_id%TYPE)
6    RETURN locat.locat_id%TYPE;
7
8  FUNCTION check_if_move_is_allowed
9           (p_ctrl_dev_id IN mfs_device.mfs_device_id%TYPE,
10           p_mfs_path_id IN mfs_path.mfs_path_id%TYPE,
11           p_cmo         IN t_cmo_path_record)
12   RETURN NUMBER;
13
14 FUNCTION mfs_mission_create
15          (p_mfs_id       OUT mfs_mission.mfs_mission_id%TYPE,
16           p_cont_id       IN mfs_mission.cont_id%TYPE,
17           p_src_id        IN mfs_mission.source%TYPE,
18           p_dest_id       IN mfs_mission.dest%TYPE,
19           p_cmo_id        IN mfs_mission.cmo_id%TYPE,
20           p_mfs_path_id   IN mfs_mission.mfs_path_id%TYPE)
21   RETURN NUMBER;
22
23 PROCEDURE mfs_mission_created
24           (p_mfs_id       IN mfs_mission.mfs_mission_id%TYPE,
25            p_cont_id      IN mfs_mission.cont_id%TYPE,
26            p_src_id       IN mfs_mission.source%TYPE,
27            p_dest_id      IN mfs_mission.dest%TYPE,
28            p_cmo_id       IN mfs_mission.cmo_id%TYPE,
29            p_mfs_path_id  IN mfs_mission.mfs_path_id%TYPE)
30
31 FUNCTION delete_finished_cmo
32          (p_cmo_id IN cmo.cmo_id%TYPE)
33   RETURN NUMBER;
34
35 FUNCTION filter_paths
36   RETURN idArray;
37
38 FUNCTION get_diversion_all_bins_full
39          (p_rbg_id IN mfs_device.mfs_device_id%TYPE,
40           p_cmo_id IN cmo.cmo_id%TYPE)
41   RETURN t_cmo_path_record;
42
43 PROCEDURE device_status_has_changed
44           (p_device_id IN mfs_device.mfs_device_id%TYPE,
45            p_status    IN NUMBER);
46
47 PROCEDURE device_idle
48           (p_device_id IN mfs_device.mfs_device_id%TYPE);
49
50 END sokoban_plugin_pkg;
```

tri možne vrednosti:

- Sokoban uporabi privzete kontrolo ali je premik dovoljen ali ne
- premik ni dovoljen,
- premik je dovoljen.

Izpis 18 Prikaz delovanja vtičnika v funkciji PREMIK-DOVOLJEN

```
1: function PREMIK-DOVOLJEN((u, v))
2:     a ← VTIČNIK-PREMIK-DOVOLJEN((u, v))
3:     if a = premik_dovoljen then
4:         return PREVERI-IZVEDLJIVOST((u, v))
5:     else
6:         return a
7:     end if
8: end function
```

Funkcija je prikazana v izpisu 18, kjer vidimo kako je uporaba vtičnika implementirana. V kolikor vrne metoda privzeto vrednost, se izvede standardna kontrola.

V praksi se je pokazalo, da je večkrat koristno, da se določen premik zaustavi. Dovoliti premik mimo standardne kontrole pa je potrebno le v res izjemnih primerih. Treba je biti tudi zelo pazljiv, saj nepopolna implementacija zlahka povzroči zastoj oziroma nedelovanje večjega dela sistema.

Sorodni sta funkciji **MFS_MISSION_CREATE** in **MFS_MISSION_CREATED**, katerih uporaba je vidna v izpisu 19. Funkcija se izvede, preden se ustvari nalog za napravo. Parametri so identifikacijska številka transporta in povezava, preko katere se bo naredil nalog za premik. Številka transportne enote ter začetno in ciljno vozlišče sta tehnično gledano redundantna podatka, a jih zaradi hitrosti in praktičnosti prenašamo preko parametrov, da se izognemo klicem v bazo.

Funkcija vrača tri možne vrednosti:

- OK - nadaljuj s premikom,
- Not ok - premika ne naredi in končaj,
- Vtičnik naredil premik - vtičnik je sam naredil premik. Identifikacijo narejenega premika vrne v **p_mfs_id**.

Izpis 19 Prikaz delovanja vtičnika v funkciji USTVARI-PREMIK-NAPRAVI

```
1: procedure USTVARI-PREMIK-NAPRAVI((u, v))
2:     a ← VTIČNIK-USTVARI-PREMIK((u, v))
3:     if a = prekini then
4:         return
5:     else if a = standard then
6:         USTVARI-PREMIK-ZA-NAPRAVO((u, v))
7:     end if
8:     VRTIČNIK-PREMIK-USTVARJEN((u, v))
9: end procedure
```

V metodi je lahko nekaj dodatnih ukazov, ki denimo ob ustvarjanju določene naloge, pošljejo nalogo zunanji napravi, kot je recimo ovijalec palet ali aplikator nalepk, nalog za pozicioniranje vozička, itd.

Procedura MFS_MISSION_CREATED se izvede le če je bil premik narejen. Najpogosteje se uporablja za zapis projektno specifičnih parametrov v nalogo, preden bo poslana napravi.

Funkcija **CMO_STARTED** se izvede, ko se je transportni nalog začel izvajati. To se zgodi, ko transportna enota dobi prvi nalog za premik. Uporabno, če je ob začetku premikanje potrebna dodatna akcija, ali pa je obvestilo nadrejenemu sistemu, da se je transport dejansko začel.

Funkcija **DELETE_FINISHED_CMO** se izvede, ko je transportna enota prispela na končni cilj. Preden se pobriše izveden transportni nalog, se pokliče še metoda vtičnika, kar je uporabno recimo za pošiljanje obvestila nadrejenemu sistemu, da je transportna enota prispela na cilj. Če funkcija vrne $false$, ostane transportni nalog aktiven do naslednje iteracije.

Funkcija **FILTER_PATHS** se uporablja za izločanje poljubnih povezav iz grafa G', kar je uporabno, ko je potrebno v primeru servisiranja zaradi varnosti serviserjev dodatno omejiti območje delovanje posamezne naprave.

Funkcija **GET_DIVERSION_ALL_BINS_FULL** je zelo uporabna za odprte sisteme. Če pride do situacije, da se regalno skladišče $RACK01$ zapolni in ni več prostora za novo transportno enoto, se pokliče ta metoda in v njej se preusmeri transportno enoto v skladišče $RACK02$. Na tak način se izognemo zastoju.

Procedura **DEVICE_STATUS_CHANGED** se izvede ko posamezna naprava preide v stanje avtomatskega režima in obratno. Potrebno je zaznati dogodek, ko naprava preide v avtomatsko delovanje in jo je potrebno postaviti na določeno mesto.

Procedure **DEVICE_IDLE** se izvede, ko naprava nima nobenega aktivnega

transporta in Sokoban ni našel nove naloge. Takrat naprava miruje. Ker je naprava v tem času brez opravil, iz stanja v skladišču pa vemo, da se bo naslednji premik zgodil v bližini naprave $KAR01$, pošljemo napravo v bližino $KAR01$, da bo naslednji premik kar se da hiter.

5.1.6 Paralelna in parcialna obdelava/segmentacija večjega sistema

Sokoban nima teoretične omejitve kako velika skladišča in koliko skladišč lahko ena izvedba podpira. Glede na potrebe lahko Sokoban nastavimo, da podpira več skladišč z eno izvedbo, kar pride prav v primeru manjših skladišč in z računalnikom, ki ima omejen pomnilniški prostor.

Veliko bolj uporabna in večkrat uporabljena lastnost je segmentacija, kjer uporabimo eno izvedbo na skladišče. To zagotavlja večjo robustnost in večjo odzivnost, sploh v primeru večprocesorskih sistemov, seveda na račun večje porabe delovnega pomnilnika.

V primeru večjih skladišč imamo možnost, da eno skladišče razbijemo na več segmentov in vsak segment dobi svojo izvedbo Sokobana.

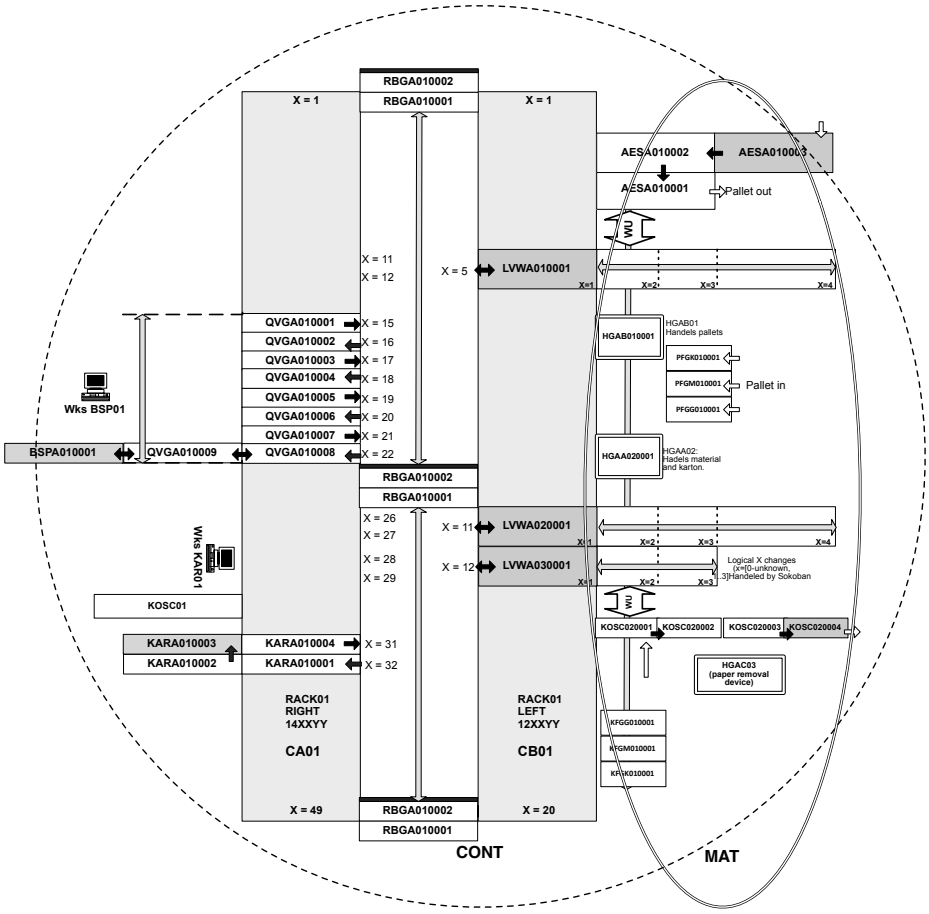
Segmentacijo naredimo tako, da ima vsaka izvedba Sokobana svoje unikatno ime s. V konfiguraciji določimo vozliščem kateri izvedbi pripadajo. Vsaka izvedba Sokobana naloži množico vozlišč $V_s \subseteq V$ za katero je ta izvedba odgovorna. Kako se ta množica obravnava, bomo videli v naslednjem poglavju. Poglejmo si posamezne implementacije.

Eno skladišče, en Sokoban V primeru da imamo le eno skladišče je podgraf $G_w = (V_w, E_w)$ enak $G = (V, E)$. Torej $V_w = V$ in $E_w = E$. Instanca Sokobana bo obdelovala vsa vozlišča in povezave na grafu, torej je $V_s = V$.

Več skladišč, en Sokoban V primeru več skladišč imamo celotnih graf $G = (V, E)$ sestavljen iz več nepovezanih podgrafov $G_w = (V_w, E_w)$, vsak podgraf predstavlja eno skladišče. Instanca Sokobana bo obdelovala vsa vozlišča in povezave na grafu G, torej je $V_s = V$.

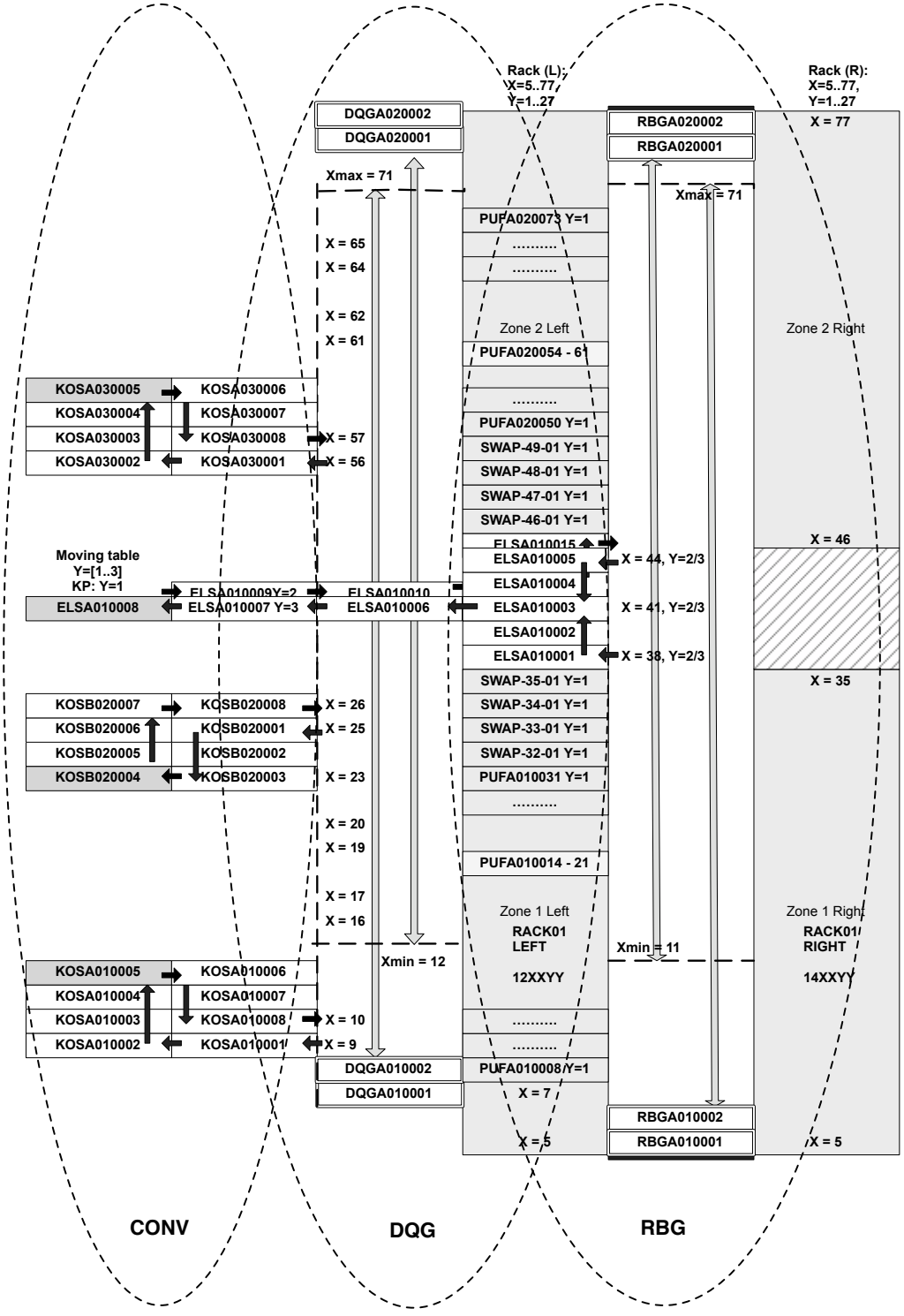
Eno skladišče, več Sokobanov Ko imamo eno večje skladišče, ki ga želimo razdeliti na posamezne dele, imamo grafom $G = (V, E)$ ki predstavlja celotno skladišče. Razlika nastane pri množici vozlišč, za katere je izvedba Sokobana zadolžena. Za delovanje potrebuje graf celotnega skladišča, saj v nasprotnem primeru ne bi našel poti. Vsaka izvedba dobi le vozlišča ki imajo oznako

izvedbe in ta vozlišča se naložijo v množico V_s. Tak primer je prikazan na sliki 5.1, kjer je instanca z imenom $s = CONT$ upravlja levo polovico, instanca z imenom $s = MAT$ pa naprave za upravljanje z materiali.



Slika 5.1: Segmentacija enega skladišča

Več skladišč, več Sokobanov Ta možnost je zelo fleksibilna, saj je možna poljubna kombinacija zgornjih konfiguracij. Če ena izvedba podpira le eno skladišče oziroma en segment v skladišču, lahko Sokoban naloži podgraf za svoj osnovni graf in na tak način prihrani pri porabi pomnilnika. Izvedba, ki krmili več kot eno skladišče (npr. dve od skupno sedem), naloži graf za vseh sedem skladišč.



Slika 5.2: Segmentacija večjega sistema

5.1.7 Optimizacija odziva na podlagi podrobnejših podatkov o dogodku

V prvi verziji je Sokoban izpraševal (pooling) bazo vsake pol sekunde. Vsakič je prebral celotno stanje naprav, transportnih enot, premike v izvajanju, ..., in na podlagi tega opravil analizo in naredil premike.

V verziji 2.0 je Sokoban opravljal "sproženo" izpraševanje. Iz baze je dobil obvestilo, da se je nekaj spremenilo. Ob vsakem obvestilu je analiziral celotno stanje v skladišču, saj ni bilo nobenih dodatnih atributov, ki bi nosili dodatne informacije. Prihranek je bil v manjših skladiščih, kjer ni bilo veliko dogajanj, saj je prišel ta dogodek v povprečju na nekaj sekund.

V velikih skladiščih to ni zadostovalo, saj se je poraba resursev večala z $O(n^2)$. Več kot je bilo vozlišč in povezav, več obvestil je Sokoban prejel in večji sistem je bilo potrebno analizirati.

V verziji 4.0 je bil uveden sistem natančnega obveščanja. Na ključnih tabelah v bazi so prožila, ki ob vsaki spremembi statusa naprave, premika transportne enote, sprememba statusa naloga za premik ..., zapišejo dogodek v vrsto (queue).

Sokoban je odjemalec te vrste. Vsakokrat, ko pride dogodek v vrsto, Sokoban prebere vrsto, analizira na katera vozlišča bi to lahko imelo vpliv in si doda vozlišča v začasno množico V_π, za katero velja $V_\pi \subseteq V$.

Sokoban vedno analizira vse dogodke v vrsti preden gre v obdelavo in se tako izogne temu, da bi bilo vozlišče večkratno obdelano brez razloga. Sokoban pridobi vozlišča, na katere bi dogodek lahko imel vpliv, preko grafa G in seznama pojavitev. Na sliki 5.3 imamo seznam dogodkov, ki se zgodijo, ko preide naprava v avtomatski režim delovanja.

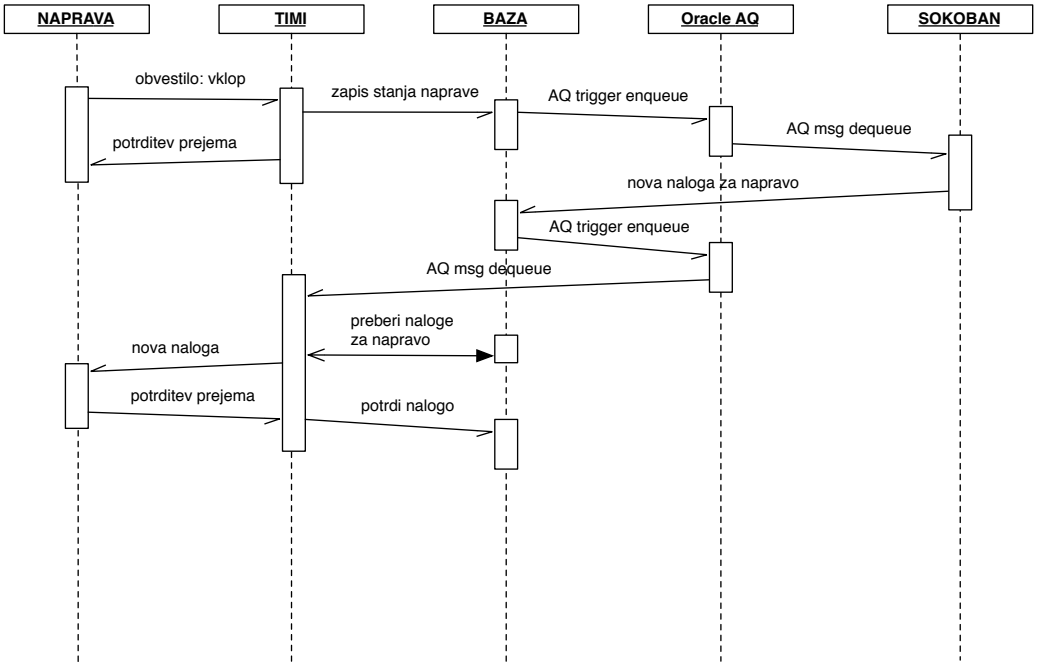
Ko Sokoban prejme dogodek, dobi tudi informacijo, katera naprava je prešla v avtomatski režim. Vklop naprave ima vpliv na vsa vozlišča ki se nahajajo na napravi V_n, in vsa direktno povezana vozlišča.

$$V_\pi = \{v \in V_n\} \cup \{u \in \lambda(Inc[v], v)\} \tag{5.1}$$

Ko je množica V_π zbrana, se vozlišča procesirajo glede na klasifikacijo po stopnji, kar je opisano v poglavju 4.3 na strani 38. Procesiranje vozlišč v množici V_π je prikazano v izpisu 20.

Ob premiku transportne enote, ki je prikazan na sliki 5.4, je zgodba podobna. Zopet je potrebno izgraditi množico vozlišč, na katere bi zaključen premik oziroma obvestilo o posameznem ključu, lahko imel vpliv.

Pri obvestilu o kmalu opravljeni nalogi je množica vozlišč enaka, gre le za časoven zamik. Določene naprave zmorejo predpomniti naslednji premik (glej



Slika 5.3: Grafična predstavitev dogodkov in sporočil, ki se izvedejo ob vklopu naprave

poglavje 4.6.2), zato mora Sokoban zanje pripraviti novo nalogo, še preden se trenutna zaključi.

Pri obvestilu o končanem premiku, dobimo podatek preko katere povezave (u, v) se bo/je premik zaključil. Torej vemo, da bo vozlišče u prazno in bo lahko sprejelo novo transportno enoto, volišče v bo pa imelo na sebi transportno enoto za odvoz. V množico V_π dodamo vozlišči u in v, ter vozlišča, ki so

$$V_\pi = \{u\} \cup \{v\} \cup \lambda(Inc^-[u], u) \cup \lambda(Inc^+[v], v). \tag{5.2}$$

Množico vozlišč V_π zopet obdelamo z proceduro Obdelaj-Vozlišča, prikazano v izpisu 20.

5.1.8 Upravljanje z viri

Ključna funkcija Sokobana je upravljanje učinkovitega nadzora pretoka blaga ter učinkovita izraba virov. Cilj je, da je paleta vedno tam, kjer je trenutno potrebna. V skladiščih uporabljamo sistem čakalnic za izravnavo razlik. Vsak

Izpis 20 Funkcija Obdelaj-Vozlišča prejme kot vhod množico vozlišč in jih obdela glede na kategorizacijo

```
1: procedure Obdelaj-Vozlišča(V_π)
2:     for all v ∈ (V_π ∩ V_s ∩ C) do
3:         Obdelaj-Križišče(v)
4:     end for
5:     for all v ∈ (V_π ∩ V_s ∩ J) do
6:         Obdelaj-Zlitje(v)
7:     end for
8:     for all v ∈ (V_π ∩ V_s ∩ S) do
9:         Obdelaj-Razcep(v)
10:    end for
11:    for all v ∈ (V_π ∩ V_s ∩ T) do
12:        Obdelaj-Tranzit(v)
13:    end for
14: end procedure
```

premik na poti ovrednotimo s funkcijo Določi-Prioriteto. Ta ima standardno implementacijo za znane naprave. V primerih, ko želimo standardni implementaciji dodati skladiščno specifične zahteve to storimo preko prepisane metode (method overriding). Vhodni podatek sta nalog transportne enote ter premik, ki mu je potrebno določiti prioriteto.

Funkcija Določi-Prioriteto ima vpliv le, ko ima križišče ali zlitje na voljo dva ali več možnih premikov iz vhodnih vozlišč. Glavno vlogo ima ovrednotenje premika iz skladišča na dvigalo, saj ima dvigalo nekaj tisoč vhodnih vozlišč in nekaj 100 nalogov za premik.

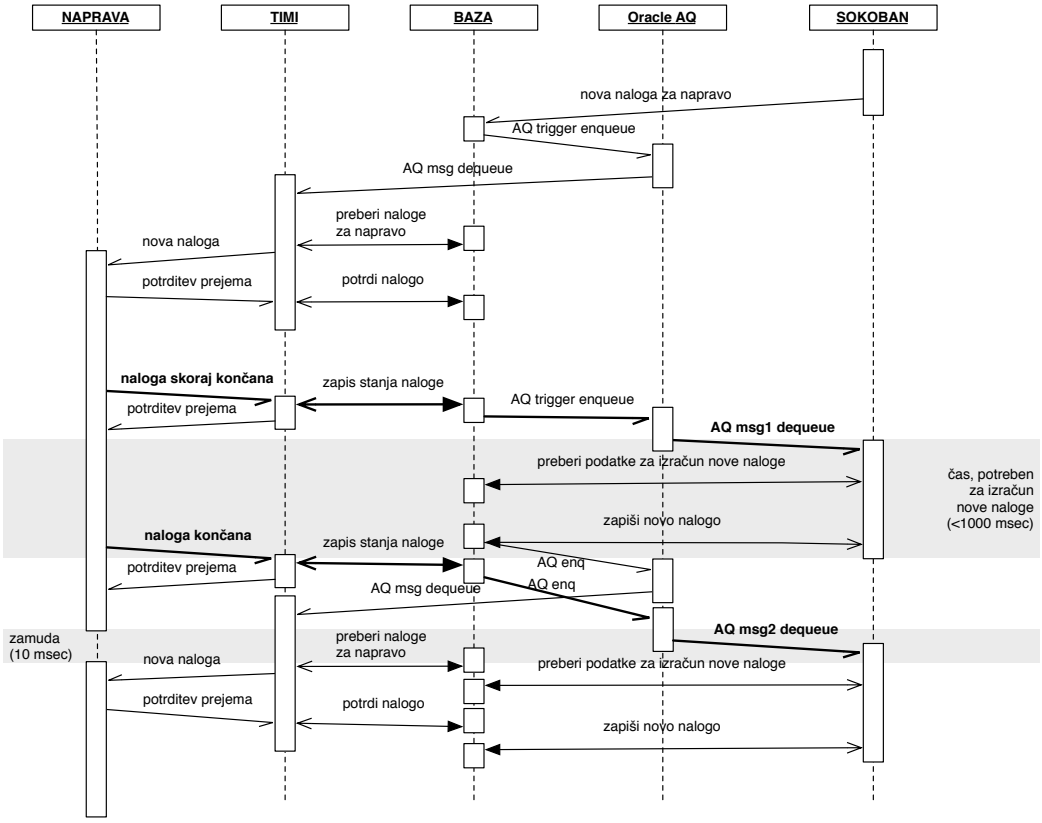
Na sliki 5.5 so prikazane tri situacije, kako se računa prioriteta za posamezno delovno mesto v skladišču, ki je prikazano na sliki 4.2.

Posamezni kriteriji so:

čas od zadnje dostave transportne enote na delovno mesto. Zaželjeno je, da je ta čim manjši, zato se s porastom časa veča prioriteta.

čas od zadnjega odvoza transportne enote iz delovnega mesta. Če je delovno mesto zaradi počasnega odvoza blokirano, temu primerno močno dvignemo prioriteto.

poknjiženo; če je transakcija poknjižena, pomeni, da bo delovno mesto kmalu potrebovalo novo transportno enoto, zato se mu dvigne prioriteta.



Slika 5.4: Grafična predstavitev dogodkov in sporočil, ki se izvedejo ob premiku transportne enote

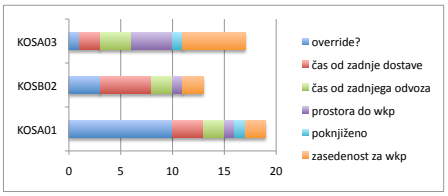
zasedenost do delovnega mesta (WKP); število zasedenih mest s transpornimi enotami v čakalnici do delovnega mesta. Bolj je čakalnica zasedena, manjšo prioriteto dobijo naloge na poti proti temu delovnemu mestu.

prostor do delovnega mesta (WKP) število prostih čakalnih mest na poti do delovnega mesta; večja je številka, večja je prioriteta.

Override zahtevajo posamezne situacije, kjer je potrebno dvigniti ali znižati prioriteto glede na specifiko skladišča oziroma zahteve stranke.

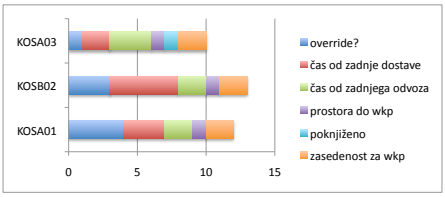
SITUACIJA 1

	KOSA01	KOSB02	KOSA03
override?	10	3	1
čas od zadnje dostave	3	5	2
čas od zadnjega odvoza	2	2	3
prostora do wkp	1	1	4
poknjiženo	1	0	1
zasedenost za wkp	2	2	6
	19	**13**	**17**



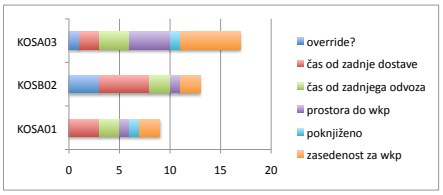
SITUACIJA 2

	KOSA01	KOSB02	KOSA03
override?	4	3	1
čas od zadnje dostave	3	5	2
čas od zadnjega odvoza	2	2	3
prostora do wkp	1	1	1
poknjiženo	0	0	1
zasedenost za wkp	2	2	2
	12	**13**	**10**



SITUACIJA 3

	KOSA01	KOSB02	KOSA03
override?	0	3	1
čas od zadnje dostave	3	5	2
čas od zadnjega odvoza	2	2	3
prostora do wkp	1	1	4
poknjiženo	1	0	1
zasedenost za wkp	2	2	6
	9	**13**	**17**



Slika 5.5: Računanje prioritet

5.2 Tehnologije

Sokoban je spisan v kombinaciji dveh programskih jezikov. Iskalni algoritmi, dnevniki, naprave in predvsem napredne naprave opisane v poglavju 4.6.2 so spisani v programskem jeziku ***Java***.

V programskem jeziku ***Oracle PL/SQL*** so napisane poizvedbe SQL ter procedure, ki obdelujejo podatke.

Javanski del ne kliče SQL poizvedb direktno, ampak kliče PL/SQL proceduro, ki opravi vse poknjižbe, oziroma PL/SQL funkcijo, če potrebuje podatek iz baze. Funkcije vrnejo že obdelane podatke, zaradi tega je malo obširnih komunikacij med podatkovno bazo in Javo.

5.2.1 Java

Funkcije, ki uporabljajo grafe in iskalne algoritme v Javi, so napisane po načrtovalskem vzorcu Edinec (Singleton). Definirane so v statičnem razredu PathManager.

Razred in vse metode morajo biti statične, da jih lahko uvozimo v Oracle podatkovno bazo kot "Java Stored Procedures" (v nadaljevanju JSP). To nam omogoča, da imamo vse funkcije na strukturi grafov na voljo tudi znotraj PL/SQL in celo znotraj SQL poizvedb.

5.2.2 Oracle PL/SQL

Za zbiranje in obdelavo podatkov Sokoban uporablja programski jezik Oracle PL/SQL. PL/SQL je tesno integriran s poizvedbenim jezikom SQL in se izvaja ***v podatkovni bazi*** sami. Obdelava podatkov je v PL/SQL posledično zelo učinkovita. Sintaksa PL/SQL omogoča tudi zelo hitro implementacijo programske kode za obdelavo podatkov.

Ker so funkcije javanskega razreda PathManager uvožene v Oracle podatkovno bazo kot JSP, jih lahko uporabljamo znotraj SQL-a, kot da bi bile del SQL poizvedbenega jezika.

Izpis 21 Primer uporabe iskalnih metod znotraj SQL poizvedb

```
1  SQL> SELECT seq_nr, mp.name, cost
2  FROM mfs_path mp,
3       (SELECT column_value AS edge_id, ROWNUM AS seq_nr
4        FROM TABLE(path_manager_pkg.find_shortest_path('TRSC030052, 'RACK01'))
5       ) p
6  WHERE mp.mfs_path_id = p.edge_id
7  ORDER BY seq_nr;
8
9  SEQ_NR NAME                           COST
10 ------ ------------------------------ ----
11      1 TRSC030052->VOZA020001           20
12      2 VOZA020001->TRSP010187           20
13      3 TRSP010187->TRSP010029           40
14      4 TRSP010029->TRSI040191           20
15      5 TRSI040191->TRSI030031           20
16      6 TRSI030031->TRSI030032           20
17      7 TRSI030032->TRSR010033           20
18      8 TRSR010033->TRSR010057           60
19      9 TRSR010057->TRSN030120           20
20     10 TRSN030120->TRSN030123           40
21     11 TRSN030123->TRSH080139           20
22     12 TRSH080139->TRSH080140           20
23     13 TRSH080140->TRSH090141           20
24     14 TRSH090141->TRSH100142           20
25     15 TRSH100142->TRSE040164           20
26     16 TRSE040164->TRSE040166           30
27     17 TRSE040166->RBGC010001          100
28     18 RBGC010001->RACK01              100
29
30 18 rows selected.
```

Na izpisu 21 vidimo primer uporabe JSP znotraj poizvedbe SQL. Tabela

mfs_path vsebuje seznam in atribute vseh povezav. Povezave so identificirane preko primarnega ključa *mfs_path_id*. V vrstici 4 poiščemo pot med dvema vozliščema z uporabo JSP PATH_MANAGER_PKG.FIND_SHORTEST_PATH, ki vrne pot sestavljeno iz primarnih ključev povezav. Pot povežemo s tabelo povezav v vrstici 6 in nato v vrstici 7 uredimo izpis povezav v vrstnem redu poti.

5.2.3 Oracle Advanced Queuing za zanesljivo in realnočasovno obveščanje o spremembah stanja

Oracle Advanced Queuing (AQ) je zmogljiv transakcijski mehanizem, preko katerega si lahko aplikacije asinhrono in preko vrste izmenjujejo sporočila. Sistem zagotavlja, da bo vsako sporočilo dobil odjemalec enkrat in samo enkrat. Vsa pisanja in branja iz vrste so transakcijska, torej je potrebno po vnosu oziroma branju sporočila transakcijo zaključiti z *COMMIT* oziroma *ROLLBACK*.

Vsaka izvedba Sokobana ima svoj odjemalec na različnih vrstah za prejemanje sporočil.

Prednost AQ sistema je, da ni potrebno baze spraševati po intervalu, ampak se Sokoban prijavi na vrsto in čaka dokler ne dobi obvestila.

Izpis 22 Mehanizem za sprejem dogodkov v skladišču

```
1  BEGIN
2    -- sortiran po vrstem redu glede na pomembnost
3    l_agent_list(1) := sys.aq$_agent(p_recipient, 'app_process_queue', NULL);
4    l_agent_list(2) := sys.aq$_agent(p_recipient, 'mfs_device_queue', NULL);
5    l_agent_list(3) := sys.aq$_agent(p_recipient, 'device_event_queue', NULL);
6    l_agent_list(4) := sys.aq$_agent(p_recipient, 'cont_queue', NULL);
7    l_agent_list(5) := sys.aq$_agent(p_recipient, 'locat_queue', NULL);
8    l_agent_list(6) := sys.aq$_agent(p_recipient, 'cmo_queue', NULL);
9    l_agent_list(7) := sys.aq$_agent(p_recipient, 'mfs_mission_queue', NULL);
10   BEGIN
11     DBMS_AQ.LISTEN(
12       agent_list   =>   l_agent_list,
13       wait         =>   p_timeout,
14       agent        =>   l_agent);
15     p_queue_name := l_agent.address;
16   EXCEPTION WHEN l_exception_timeout
17   THEN
18     RETURN 0;
19   END;
20 END;
```

V izpisu 22 je prikazano, kako Sokoban čaka na sporočila. Spremenljivka *p_recipient* vsebuje ime instance Sokobana. Med vrsticama 3 in 9 si v seznam

shrani vrsto, na katere bo čakal. Nato pride v vrstico 11, kjer začne s poslušanjem. V kolikor ima neka vrsta že sporočilo, se bo ta klic takoj vrnil z imenom vrste, kjer čaka Sokoban sporočilo.

Sokoban bo ta sporočila obdelal, kot je opisano v poglavju 5.1.7 in nato pobrisal sporočila iz vrste. Obdelava in brisanje iz vrste poteka znotraj ene transakcije.

Nato bo spet poslušal vrsto. Če ni dogodka, čaka na poslušanju do 60 sekund. Če znotraj tega časa pride novo sporočilo, se klic takoj vrne z imenom vrste, v katero je prišlo sporočilo. Tako zagotovimo odzivnost v realnem času.

Če skladišče miruje in 60 sekund ni nobenega sporočila, se klic funkcije DBMS_AQ.LISTEN vseeno zaključi. S tem zagotovimo, da se Sokoban pravočasno prijavi stražarju in da se ugotovijo neregularnosti, kot je prekinitev mrežne povezave do strežnika s podatkovno bazo. V primeru prekinitve Sokoban poskuša s ponovnim zagonom vsakih 30 sekund.

5.3 Diagnostika in metrike

Pomembna lastnost sistemov z visoko razpoložljivostjo je, da so na voljo podatki, ki omogočajo temeljito analizo in odkrivanje vzrokov za incidente.

Sistem mora premišljeno beležiti potek procesov in vse ključne podatke, ki vplivajo na odločitve in delovanje. Prav tako je pomembno, da je sistem vedno odziven.

V nadaljevanju si bomo pogledali, kakšni podatki so na voljo za lažjo diagnostiko.

5.3.1 Dnevniki za odpravljanje napak

Do zastoja v skladišču lahko pride ob nepredvidljivem času. Za nemoteno delovanje in hiter ponoven zagon v primeru incidenta, skrbi oddelek podpore strankam 24/7. Njihova naloga je spraviti skladišče čimprej znova v obratovanje ter zabeležiti osnovne podatke o incidentu za kasnejšo analizo.

Analiza napake se opravlja po tem, ko je incident odpravljen in je skladišče znova v obratovanju. Zaradi nujnosti obratovanja skladišča se razhroščevalnika ne uporablja za analizo, tudi če se incident zgodi med delovnim časom, saj je za analizo potrebno dovolj časa. Poleg tega je običajno že prepozno, ko do napake pride, saj lahko pride do težav v zelo specifičnih kombinacijah in Sokoban ustvari napačen premik.

Dnevniki v Sokobanu zato beležijo natančno vse ključne podatke in procese. Napake se običajno zgodijo, ko so transportne enote na določenih mestih

in se zgodi dogodek, ki povzroči napačno odločitev. V samem dnevniku se največkrat že vidi razlog napake. Kadar razlog ni razpoznaven, je možno iz dnevnika ter podatkov o incidentu rekonstruirati situacijo v testnem okolju.

Dnevniki so ključnega pomena in se pišejo hkrati s programsko kodo. Knjižnica za zapisovanje dnevnika sama zabeleži ime paketa, ime razreda, ime funkcije in vrstico, kjer je bil zapis zahtevan. Parameter beležni funkciji je kratek opis vsebine zapisa ter vse ključne spremenljivke.

Beleženje klicev v bazo je v precejšnji meri avtomatizirano. Gonilnik JDBC je prestrežen in vrača *LoggablePreparedStatement* in *LoggableCallableStatement* namesto standardnih javanskih *PreparedStatement* in *CallableStatement*. *LoggablePreparedStatement* ima nadomeščeno metodo *toString*, ki vrne klic z vhodnimi in izhodnimi podatki v strukturi *String*.

V izpisu 23 je prikazen kratek klic shranjene procedure iz Javanske programske kode. Vrstica 22 zabeleži klic bazne procedure. Sprejme en parameter tipa String. Sestavljen je iz vhodnih parametrov *device* in *via* ter $CallableStatementcsGetMoveVia$. Izpis tega klica je viden na izpisu 24 v vrstici 1. Za *CallableStatement* se zabeleži ime bazne procedure, vhodni parametri, izhodni parametri ter čas izvajanja.

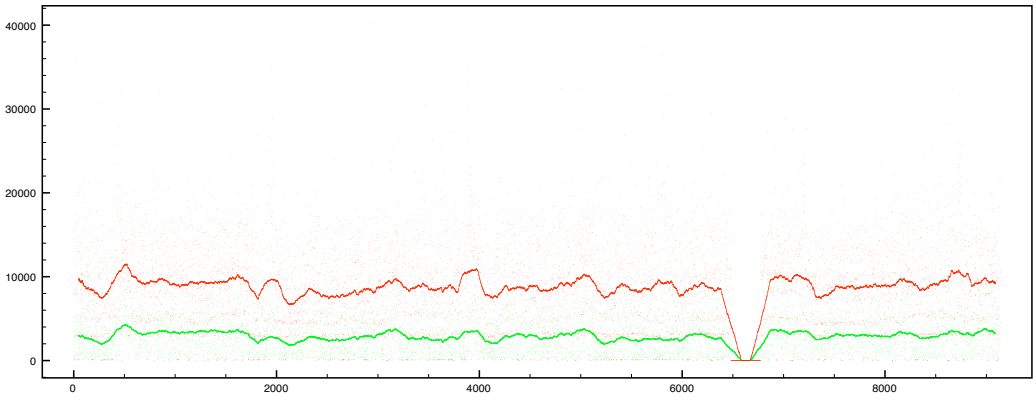
Ker funkcija vrača strukturo in ne primitivnega tipa, je generični sistem za beleženje ne zna zabeležiti. Zato je potrebno najprej pretvoriti bazno strukturo v javanski objekt, kar storimo v vrstici 27. Če ta ni NULL, ga zabeležimo v vrstici 31. Izpis v dnevniku je viden v vrstici 24.

Med tema dvema zapisoma so vidni še ostali zapisi v dnevniku. To so zapisi, ki so bili ustvarjeni znotraj shranjenih procedur. Iz baze jih je prebral $Epilogger.transferDatabaseLogs(conn)$ v vrstici 28 in jih poslal na standardni logger (datoteka, konzola, baza).

5.3.2 Metrike za perfomančne analize

Sistem z visoko odzivnostjo zahteva učinkovito diagnostiko za odpravljanje perfomančnih težav. V praksi se je izkazalo, da vse nepredvidljive perfomančne težave nastanejo v podatkovni bazi. Načeloma se za analizo perfomančnih težav uporabljajo orodja baze, a tam se vidijo le povprečja. Zato si Sokoban shranjuje čas izvajanja vsake iteracije in v dnevnik zapisuje za vsakih 10 iteracij, maksimalni čas izvajanja, povprečje in minumum. Statistike omogočajo grafični prikaz trendov (glej sliko 5.6).

Na izpisu 25 lahko vidimo statistike med hitrim delovanjem, na izpisu 26 pa med počasnim. Na tem izpisu se tudi dobro vidi, do kako izjemnih razlik lahko prihaja med hitrim in počasnim delovanjem. Natančen podatek kje in



Slika 5.6: Grafični prikaz izvajalnega časa zanke. Rdeča črta prikazuje povrečje maksimalnih časov znotraj 10 iteracij. Zelena črta, prikazuje povprečni izvajalni čas povprečij

Izpis 23 Funkcije kliče shranjeno proceduro ter dnevnik v izpisu 24

```
1  /**
2   * getMoveVia - returns next cont_move_order going through the location
3   *
4   * @param device for which device we are searching for the mission
5   * @param via Location which pass
6   * @param bCheckDest true if dest needs to be checked for empty place
7   * @return ContMoveOrder
8   * @throws SQLException in case of SQL Error
9   */
10 public static ContMoveOrder getMoveVia(Device device,
11                                         Location via,
12                                         boolean bCheckDest)
13 throws SQLException
14 {
15    try
16    {
17       csGetMoveVia.registerOutParameter(1, Types.STRUCT, T_CMO_PATH_RECORD);
18       csGetMoveVia.setInt(2, device.getId());
19       csGetMoveVia.setInt(3, via.getId());
20       csGetMoveVia.setString(4, bCheckDest  ? "Y":"N");
21       csGetMoveVia.execute();
22       Epilogger.debug(device.toShortString() +
23                       ", via=" + via +
24                       ", " + csGetMoveVia);
25
26       java.sql.Struct cmoPath = (java.sql.Struct) csGetMoveVia.getObject(1);
27       ContMoveOrder cmo = convertDbCmoToJavaCmo(cmoPath);
28       Epilogger.transferDatabaseLogs(conn);
29       if (cmo.getId() != 0)
30       {
31          Epilogger.debug("getMoveVia(" + device.toShortString() +
32                          ", via=" + via +
33                          "," + bCheckDest + "): " + cmo);
34       }
35       return cmo;
36    }
37    catch (SQLException sex)
38    {
39       Epilogger.severe(Resource.getEXC_SQLEXCEPTION(),
40                        "Src:" + via + ", " + csGetMoveVia);
41       throw sex;
42    }
43 }
```

Izpis 24 Programski vmesnik (API) od Sokobanovega vtičnika

```
1  08.01.2010-14:03:49.013 getMoveVia(): Debug [DEVICE[1,RBGA01], via=L[id=7,tag=
       RBGA010002], CALL: {call OUT := sokoban_pkg.get_move_via(1,7,'N')}
2   Result:{call 'Unkown SQL Type. Type=2002' := sokoban_pkg.get_move_via(IN,IN,
       IN)}
3   Query executed in 54 milliseconds.]
4  08.01.2010-14:03:48.000 Database Log: [get_move_via; start!]
5  08.01.2010-14:03:48.000 Database Log: [get_move_via; start candidate!]
6  08.01.2010-14:03:48.000 Database Log: [get_move_via; p_ctrl_dev_id=1,
       p_locat_id=7, l_cmo=cmo_id=44839, status_cmo=A, cont=528(202015),
       home_locat=(), cont_locat=11(QUFA010003), src=10(QUFA010002), dest=4(
       RACK01), via=4(RACK01), mfs_path_id=6, mfs_path_id_next=]
7  08.01.2010-14:03:48.000 Database Log: [get_move_via; l_move_allowed=1]
8  08.01.2010-14:03:48.000 Database Log: [get_next_avail_dest; p_cmo_dest_id=4,
       Possible dest=4]
9  08.01.2010-14:03:48.000 Database Log: [get_next_avail_dest; p_cmo_dest_id=4,
       possible move_id=3035]
10 08.01.2010-14:03:49.000 Database Log: [Feching next candidate. mfs_path_id
       =3035; queuing=N]
11 08.01.2010-14:03:49.000 Database Log: [get_next_avail_dest;
       check_for_empty_locat; p_ctrl_dev_id=1, l_cmo=cmo_id=44839, status_cmo=A,
       cont=528(202015), home_locat=(), cont_locat=11(QUFA010003), src=10(
       QUFA010002), dest=4(RACK01), via=4(RACK01), mfs_path_id=3035,
       mfs_path_id_next=]
12 08.01.2010-14:03:49.000 Database Log: [get_next_avail_dest; l_empty_ok=1,
       l_move_allowed=1]
13 08.01.2010-14:03:49.000 Database Log: [Found empty location]
14 08.01.2010-14:03:49.000 Database Log: [points_for_rack_direction; l_points=0,
       p_cmo.id=44839, p_cmo.dest_tag=RACK01, mfs_path_id=3035, p_device=QUFA01]
15 08.01.2010-14:03:49.000 Database Log: [Ima kaseta na lokaciji 4 cmo za v rack
       ?; l_cmo_id=]
16 08.01.2010-14:03:49.000 Database Log: [points_for_double_game; l_points=0,
       p_cmo.id=44839, p_cmo.dest_tag=RACK01, mfs_path_id=3035, p_device=QUFA01
       ]08.01.2010-14:03:49.000 Database Log: [get_next_avail_dest; l_cmo.points
       =0, l_points_best=-1000, l_cmo.id=44839, l_cmo.cont_tag=202015, l_cmo.
       dest_tag=RACK01]
17 08.01.2010-14:03:49.000 Database Log: [setiram l_cmo_best]
18 08.01.2010-14:03:49.000 Database Log: [get_move_via; l_cmo_drop; p_ctrl_dev_id
       =1, l_cmo_drop=cmo_id=44839, status_cmo=A, cont=528(202015), home_locat=()
       , cont_locat=11(QUFA010003), src=10(QUFA010002), dest=4(RACK01), via=4(
       RACK01), mfs_path_id=3035, mfs_path_id_next=]
19 08.01.2010-14:03:49.000 Database Log: [get_move_via; l_cmo_drop.cmo_id=44839,
       l_cmo_drop.cont_tag=202015, l_cmo_drop.dest=RACK01, l_cmo_drop.points=0]
20 08.01.2010-14:03:49.000 Database Log: [get_move_via; end candidate!]
21 08.01.2010-14:03:49.000 Database Log: [get_move_via; end!]
22 08.01.2010-14:03:49.014 getMoveVia(): [getMoveVia(DEVICE[1,RBGA01],L[id=7,tag=
       RBGA010002],,false): CMO[cmo_id=44839,status=A,cont=C[id=528,tag=202015,
       homeLocat=L[id=0,tag=]],contLocat=L[id=11,tag=QUFA010003],src=L[id=10,tag=
       QUFA010002],dest=L[id=4,tag=RACK01],via=L[id=4,tag=RACK01],path_id=6,true,
       p=0.0]]
```

Izpis 25 Statistike med hitrim delovanjem.

```
1  08.01.2010-14:10:21.865 calcStatistic(): Min/Avg/Max: (11ms/35ms/79ms)
2  08.01.2010-14:10:37.756 calcStatistic(): Min/Avg/Max: (9ms/42ms/178ms)
3  08.01.2010-14:11:38.311 calcStatistic(): Min/Avg/Max: (13ms/28ms/62ms)
4  08.01.2010-14:12:15.888 calcStatistic(): Min/Avg/Max: (10ms/39ms/134ms)
5  08.01.2010-14:13:02.077 calcStatistic(): Min/Avg/Max: (10ms/45ms/151ms)
6  08.01.2010-14:13:38.458 calcStatistic(): Min/Avg/Max: (11ms/27ms/62ms)
7  08.01.2010-14:14:19.310 calcStatistic(): Min/Avg/Max: (10ms/73ms/180ms)
8  08.01.2010-14:16:17.416 calcStatistic(): Min/Avg/Max: (10ms/87ms/201ms)
9  08.01.2010-14:17:05.155 calcStatistic(): Min/Avg/Max: (8ms/28ms/80ms)
10 08.01.2010-14:17:35.825 calcStatistic(): Min/Avg/Max: (8ms/35ms/136ms)
11 08.01.2010-14:18:16.417 calcStatistic(): Min/Avg/Max: (10ms/20ms/40ms)
12 08.01.2010-14:19:04.929 calcStatistic(): Min/Avg/Max: (8ms/49ms/212ms)
```

Izpis 26 Statistike med počasnim delovanjem

```
1  19.02.2011-12:03:20.184 calcStatistic(): Min/Avg/Max: (4ms/141ms/388ms)
2  19.02.2011-12:03:41.034 calcStatistic(): Min/Avg/Max: (27ms/1001ms/7767ms)
3  19.02.2011-12:04:05.819 calcStatistic(): Min/Avg/Max: (18ms/703ms/2497ms)
4  19.02.2011-12:04:24.391 calcStatistic(): Min/Avg/Max: (4ms/633ms/2290ms)
5  19.02.2011-12:05:10.562 calcStatistic(): Min/Avg/Max: (21ms/2716ms/13411ms)
6  19.02.2011-12:05:52.043 calcStatistic(): Min/Avg/Max: (66ms/1735ms/13241ms)
7  19.02.2011-12:06:18.967 calcStatistic(): Min/Avg/Max: (24ms/356ms/1377ms)
8  19.02.2011-12:06:46.788 calcStatistic(): Min/Avg/Max: (25ms/600ms/3199ms)
9  19.02.2011-12:07:30.640 calcStatistic(): Min/Avg/Max: (40ms/434ms/2158ms)
10 19.02.2011-12:08:21.236 calcStatistic(): Min/Avg/Max: (5ms/3163ms/17030ms)
11 19.02.2011-12:09:10.022 calcStatistic(): Min/Avg/Max: (17ms/635ms/2137ms)
12 19.02.2011-12:09:34.444 calcStatistic(): Min/Avg/Max: (34ms/564ms/2983ms)
13 19.02.2011-12:10:01.723 calcStatistic(): Min/Avg/Max: (5ms/2118ms/12204ms)
14 19.02.2011-12:10:33.997 calcStatistic(): Min/Avg/Max: (19ms/686ms/3085ms)
15 19.02.2011-12:11:42.708 calcStatistic(): Min/Avg/Max: (6ms/5533ms/15924ms)
16 19.02.2011-12:12:34.083 calcStatistic(): Min/Avg/Max: (5ms/1130ms/4586ms)
17 19.02.2011-12:13:39.280 calcStatistic(): Min/Avg/Max: (21ms/2522ms/14273ms)
```

s katerimi vhodnimi parametri je Sokoban porabil čas za obdelavo posamezne iteracije, je razviden v dnevniku.

V primeru, da se upočasnitev zgodi v bazni funkciji, katere klic ni zabeležen, obstaja varovalka. Tudi če klic funkcije v bazo ni zabeležen, bo *LoggableStatement* avtonomno zapisal v dnevnik klic funkcije z vhodnimi in izhodnimi vrednostmi. Funkcija na izpisu 27 nima nobenega klica za zapis v dnevnik, a če se klic procedure ne bo izvršil znotraj 100ms, bo *LoggableStatement* sam povzročil zapis v dnevniku, kot je vidno na izpisu 28.

Izpis 27 Primer funkcije brez eksplicitnega beleženja v dnevnik

```
1  public static ContMoveOrder getMoveFrom(Device device,
2                                          Location from,
3                                          boolean bCheckDest)
4  throws SQLException
5  {
6      csGetMoveFrom.registerOutParameter(1, Types.STRUCT, T_CMO_PATH_RECORD);
7      csGetMoveFrom.setInt(2, device.getId());
8      csGetMoveFrom.setInt(3, from.getId());
9      csGetMoveFrom.setString(4, bCheckDest  ? "Y":"N");
10     csGetMoveFrom.setWarningThreshold(500);
11     csGetMoveFrom.execute();
12
13     java.sql.Struct cmoPath = (java.sql.Struct) csGetMoveFrom.getObject(1);
14     ContMoveOrder cmo = convertDbCmoToJavaCmo(cmoPath);
15     return cmo;
16 }
```

Izpis 28 Funkcija se ni izvedla znotraj definiranega časa

```
1  10:53:49.925 WARNING: SQL statement took too long to execute. Execution time
       was 8807, threshold is 100; CALL: call OUT := sokoban_pkg.get_move_from
       (54,243,'Y')}
2  Result:{call OUT := sokoban_pkg.get_move_from(IN,IN,IN)}
3  Query executed in 8807 milliseconds.
```

Poglavje 6

Evolucija programa

Glavno gonilo evolucijskega procesa so zahteve po spremembah, ki jih narekujejo stranke, nove tehnologije, itd. Dodaten vir so ugotovljene napake in nove ideje za izboljšave in lažje vzdrževanje.

Zakone programske evolucije sta formulirala *Lehman* in *Belady* začenši z letom 1974 [13][6]. Zakoni opisujejo razmerje med gonilno silo napredka na eni strani, in silami, ki zaustavljajo razvoj na drugi strani. Lehman in Belady sta klasicifirala programe v tri skupine:

S-tip so tisti programi, ki se jih da formalno specificirati. Zadoščati morajo štirim pogojem:

1. problem je možno strogo formalno definirati,
2. problem mora biti rešljiv algoritmično,
3. mora biti dokazljivo, da je program pravilen in ustreza formalni specifikaciji,
4. specifikacija mora biti kompletna in končna glede na deležnikove (stakeholder) trenutne zahteve. Specifikacija mora eksplicitno definirati *vse* funkcijske in nefunkcijske zahteve deležnikov in še posebno strank in uporabnikov. Nefunkcijske zahteve vsebujejo obseg in natančnost spremenljivk, maksimalni pomnilniški prostor, omejitve izvajalnega časa, itd.

E-tip so programi, ki "delujejo v" ali "obravnavajo problem ali aktivnost realnega sveta". Ključna lastnost tega tipa je, da program postane sestavni del domene, znotraj katere deluje in ki jo obravnava. Program mora zato odražati vse lastnosti, ki imajo na kakršenkoli način vpliv

na rezultat. Da program ostane zadovoljiv, medtem ko se aplikacije, domene in njihove lastnosti spreminjanjo, je potebno programe tipa *E* nenehno spreminjati in posodabljati - morajo *E*volvirati. Programska evolucija je zato neposredna posledica in odsev sprememb v dinamičnem realnem svetu. Operacijski sistemi, podatkovne baze, transakcijski sistemi in nadzorni sistemi so primeri programov tipa *E*, ob tem pa lahko vsebujejo posamezne elemente, ki so tipa *S*.

P-tip programi so tisti, ki obravnavajo probleme, ki izgledajo določljivi, vendar je za uporabnika bolj pomembna pravilnost rezultata izvajanja v domeni, kjer bodo *uporabljeni*, kot pa skladnost s specifikacijo. Za dosego rezultata se običajno uporablja iterativen proces z uporabo prototipov in validacij. Programi tipa *P* vedno ustrezajo tudi tipu *S* ali *E*, zato nekateri menijo da je ta tip redundanten.

Sokoban je program tipa *E*, ki evolvira. Zakoni te evolucije izhajajo iz neposrednih opažanj in meritev raznovrstnih sistemov in veljajo za vse sisteme tipa *E*, ne glede na specifično prakso programiranja ali upravljanja. Ti zakoni so:

1. **Nenehne spremembe** - sistem tipa *E* se mora nenehno prilagajati, drugače postane postopoma manj sprejemljiv za uporabo.

2. **Naraščajoča kompleksnost** - med spreminjanjem sistema tipa *E* se njegova kompleksnost povečuje in postaja vedno težji za postopni razvoj, razen če vzporedno vlagamo napore v zmanjševanje kompleksnosti oziroma uvajamo spremembe za lažje vzdrževanje.

3. **Samoregulacija** - evolucijo sistema tipa *E* krmilijo povratne informacije (feedback).

4. **Ohranjanje organizacijske stabilnosti** - popvrečni obseg aktivnosti pri spreminjanju sistema tipa *E* je skozi življensko dobo produkta konstanta.

5. **Ohranjanje poznavanja** - postopna rast sistemov tipa *E* je v splošnem omejena s potrebo po poznavanju sistema. Vsi, ki so s sistemom povezani (razvijalci, prodajalci, uporabniki, podpora, idr.), morajo z vsako spremembo še vedno obvladati vsebino in obnašanje sistema, da je evolucija uspešna. Pretirana rast to preprečuje, zato ostaja povprečna postopna rast nespremenjena.

6. **Nenehna rast** - funkcionalnost sistema tipa E je treba nenehno izboljševati, da se ohrani zadovoljstvo uporabnikov.

7. **Upadajoča kakovost** - zdi se, da bo kvaliteta sistema tipa E upadala, če sistema ne dosledno prilagajamo in izboljšujemo glede na spremembe delovnega okolja. Primer: aplikacija napisana za okolje Windows NT4 v okolju Windows 7 ne naredi dobrega vtisa.

8. **Povratne informacije** - s stališča povratnih informacij so evolucijski procesi tipa E kompleksni: so večnivojski, imajo več povratnih zank in več izvorov povratnih informacij (uporabniki, komerciala, podpora, uprava, ...). Do tega pride zato, ker so močno integrirani v svoje okolje.

Kot bomo videli v nadaljevanju, navedeni zakoni veljajo tudi za Sokoban. Nova sladišča in nove naprave, nove zahteve strank zahtevajo nenehne spremembe (zakon št. 1), kar povečuje kompleksnost. Zato kodo v Sokobanu preuredim (refactoring), če pridem do spoznanja, da bo sprememba uvedla preveč kompleksnosti (zakon št. 2). Tudi zakon št. 7 vpliva na Sokoban, čeprav je neinteraktiven program. Upravitelj sistema in podatkovne baze zagotovo ne bi dobil dobrega vtisa, če bi Sokoban leta 2011 zahteval Oracle verzije 8.1.7 in Javo 1.4, kar je zahteval Sokoban 1.x. Verzija 4.x podpira podatkovno bazo Oracle verzij 10.2 do Oracle 11.2 in Javo 1.4 do 1.6. Ohranjanje poznavanja (zakon št. 5) je ključno za oddelek podpore strankam. Velike spremembe je zato težje uvesti, saj jih je potrebno podpreti 24/7 že s prvim dnem. Postopne spremembe je bistveno lažje razumeti, kot spremembo celotnega sistema. In končno zakon 8 - mnoge izboljšave so bile narejene na osnovi povratnih informacij (npr. za lažjo analizo, večjo robustnost).

Sledi podrobnejši opis posameznih verzij in razlaga, zakaj evolucijski pristop ni bil primeren za verzijo 4.x. Interno vodimo verzije trinivojsko v obliki *"major.minor.patch"*. *Major* verzija prinaša večje izboljšave, ki po naravi niso v celoti evolucijske narave. Prehod na višjo *major* verzijo zahteva nadgradnjo vseh podpornih sistemov in večje spremembe tudi pri podatkovni bazi in konfiguraciji. Take prehode izvajamo le v primeru večjih nadgradenj, ki praviloma zahtevajo nekaj dni testiranja in odpravljanja problemov v realnem sistemu, kar v večini onemogoča produkcijo.

Minor verzije vsebujejo evolucijske izboljšave. Z njimi zagotavljamo nenehno rast (zakon št. 6). Nadgradnje na višje minor verzije zahtevajo minimalne posege v konfiguracijo ali podatkovno bazo in jih je praviloma moč izvesti v eni uri.

Patch verzije vsebujejo popravke ali pa tudi nove manjše funkcionalnosti, ki ne zahtevajo popravkov konfiguracije ali podatkovne baze. Nadgradnjo je možno izvesti v 15min in mora biti 100% kompatibilna za nazaj.

6.1 Sokoban 1.x (2003)

Prva verzija Sokobana je podpirala le zaprti sistem skladišča z enim regalnim dvigalom. Vse periferne transportne naprave so se morale neposredno stikati s skladiščem. Na sliki 6.1 je prikazan primer skladišča, ki bi ga Sokoban 1.x lahko podpiral z izjemo ULB naprave, ki se nahaja na dnu.

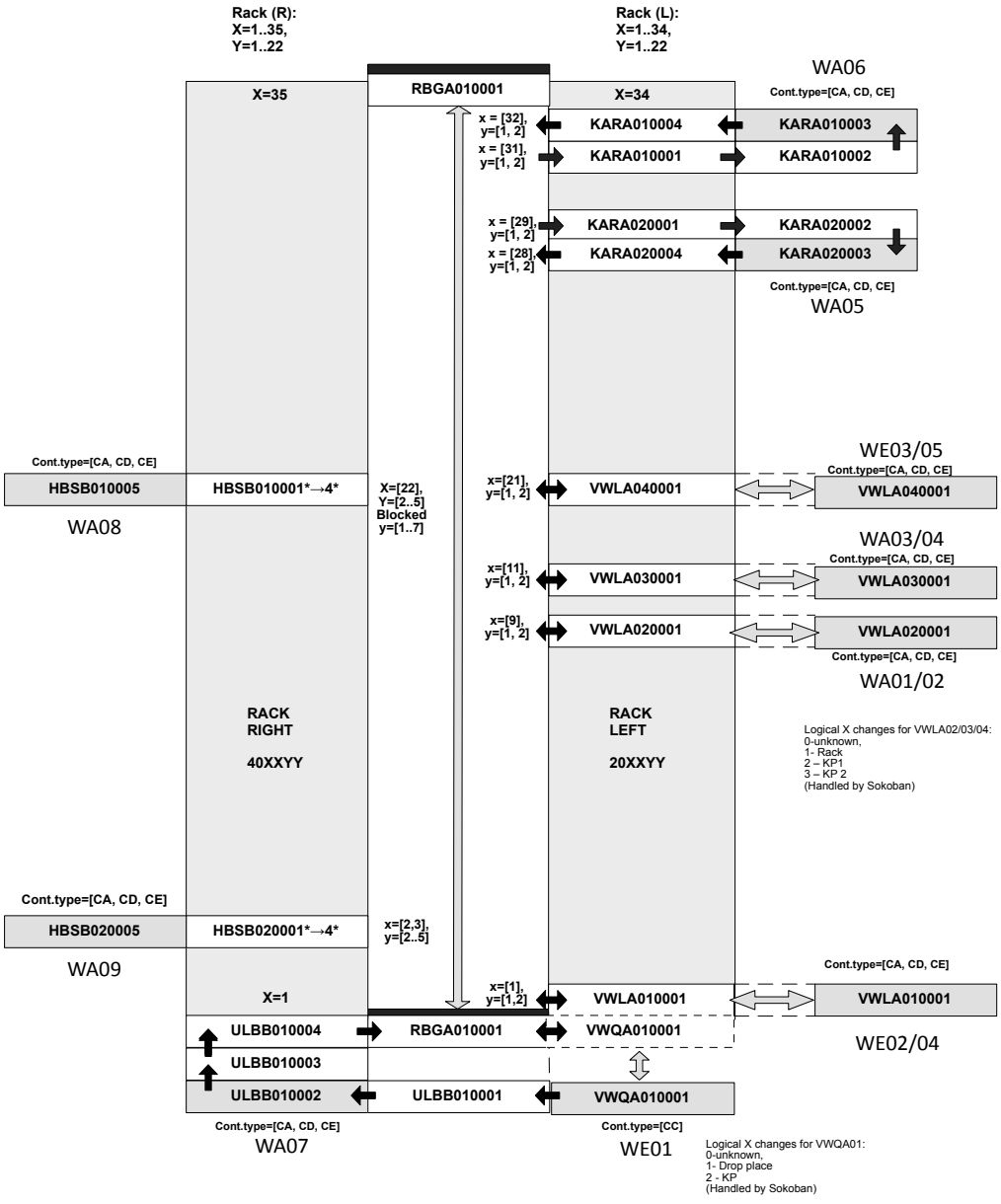
Skladišče ni bilo predstavljeno z grafom. V programski kodi je bilo privzeto, da regalno dvigalo dostavlja transportne enote na periferne transportne naprave z delovnimi mesti. Grob opis delovanja je prikazan na izpisu 29.

Ob inicializaciji Sokoban 1.x naloži lokacijo regalnega dvigala v spremenljivko r in vsa delovna mesta v množico W, kar je vidno v vrsticah 16 in 17. Nato vstopi v neskončno zanko, znotraj katere se izvaja procesiranje.

Zanka se začne v vrstici 19 z ugotavljanjem ali se transportna enota že nahaja na dvigalu in kam je namenjena. Če ima dvigalo že naloženo transportno enoto, se pokliče funkcija **Določi-Odlagalno-Lokacijo**. Ta določi odlagalno mesto na napravi, če je cilj delovno mesto, oziroma skladiščno lokacijo, če je cilj skladišče. Za tem se ustvari premik za odlaganje v vrstici 22.

Če je dvigalo prazno, se izvede **Poišči-Premik-Dvigalo**. Ta poišče optimalen izvedljiv premik za dvigalo. Pri tem uporablja podfunckijo **Premik-Dvigalo**, ki poišče mesto TE za ***naslednji izvedljiv*** premik. Išče tako premike iz skladišča do delovnih mest, kot premike iz delovnih mest proti skladišču. Če tak premik najde, ustvari nalog za premik za nalaganje transportne enote na dvigalo (v vrstici 26).

V vrstici 29 se izvede procedura **Opravi-Premike-Perifernih-Naprav**, ki pregleda vse periferne naprave in ustvari naloge za premik po napravi, npr. *HBSB010005* → *HBSB010001*, *KARA010001* → *KARA010003* ali *KARA010003* → *KARA010004*. Premiki med napravami, razen med osrednjim regalnim dvigalom, v Sokobanu 1.x niso bili podprti. Na koncu se je izvedel še čakalni interval pred naslednjo iteracijo. Sokoban 1.0 je upravljal le eno samo skladišče.



Slika 6.1: Primer enostavnega zaprtega sistema, kjer se vse naprave stikajo s skladiščem.

Izpis 29 Sokoban 1.x

```
1: function Poišči-Premik-Dvigalo(W, r)
2:     p_h ← 0                                   ▷ Najvišja prioriteta do sedaj
3:     s_b ← NIL                                 ▷ Najboljši izvor do sedaj
4:     for all w ∈ W do
5:         s_t ←Premik-Dvigalo(r, w)             ▷ Poišči izvedljiv premik
6:         p_t ←Določi-Prioriteto(s_t)
7:         if p_t > p_h then                     ▷ Višja od trenutno najvišje?
8:             p_h = p_t                         ▷ Da. Shrani trenutno stanje
9:             s_b = s_t
10:        end if
11:    end for
12:    return s_b          ▷ Vrni lokacijo TE za najoptimalnejši premik
13: end function
14:
15: procedure Sokoban-1
16:    r ← RBGA010001                            ▷ Lokacija dvigala
17:    W ← {KARA010003, KARA020003,..., HBSB020005}   ▷
       Seznam delovnih mest
18:    loop
19:        d_t ←Preberi-Cilj-TE(r)               ▷ Preberi cilj za TE
20:        if d_t ≠ NIL then
21:            d_n ← Določi-Odlagalno-Lokacijo(d_t)
22:            Ustvari-Premik(r, d_n)            ▷ Odloži TE
23:        else                                  ▷ TE ni na dvigalu?
24:            s ←Poišči-Premik-Dvigalo(W, r)
25:            if s ≠ NIL then
26:                Ustvari-Premik(s, r)          ▷ Naloži TE na dvigalo
27:            end if
28:        end if
29:        Opravi-Premike-Perifernih-Naprav
30:        Čakaj(1000)   ▷ Čakaj 1s pred izvajanjem naslednje iteracije
31:    end loop
32: end procedure
```

6.2 Sokoban 2.x (2003)

Na omejitve Sokobana 1.x smo naleteli že pri drugem skladišču. Skladišče je vsebovalo napravo, ki je imelo dve delovni mesti. Za primer vzemimo *KARA010002* in *KARA010003* na sliki 6.1. Funkcija PREMIK-DVIGALO je v verziji 1.x vračala naslednji premik za delovno mesto v zaporedju, ki ga je določil ERP. V tem primeru se je zgodilo, da je funkcija vrnila en premik za transpotno enoto v *KARA010002* in enega za *KARA010003*. Ker se s funkcijo DOLOČI-PRIORITETO določi prioriteto glede na zasedenost naprav in performančnih kriterijev, ne pa tudi vrstnega reda, kot zahteva ERP, se bi lahko zgodilo da bi transportne enote prišle na delovno mesto v napačnem vrstnem redu.

Problem sem rešil tako, da funkcija PREMIK-DVIGALO kot parameter sprejme napravo. Če ima naprava več delovnih mest, potem funkcija vrne le en možen premik, ki ustreza vrstnemu redu zahtevenem od ERP. Zato v inicializaciji Sokoban 2.x naloži seznam perifernih naprav v množico D, kar je vidno v vrstici 17 na izpisu 30.

V podverzijah od 2.1 do 2.7 so bile implementirane še sledeče izboljšave:

Podpora stražarju - v vsaki iteraciji se Sokobanu javi stražarju in mu s tem sporoči, da nemoteno obratuje (vrstica 32)

Čakanje na obvestila - Sokoban ne sprašuje več baze vsako sekundo ampak čaka na obvestilo (vrstica 33). Sprožilci v bazi pošljejo obvestilo, ko se spremenijo podatki, ki vplivajo na Sokoban. Če obvestila 60s ni, se časovna kontrola izteče in se naslednja iteracije vseeno izvede, kar omogoči redno javljanje stražarju.

Podpora dveh regalnih dvigal na skupni osi - podpora večjim skladiščem, katerih pretočnost in zanesljivost zahteva dve regalni dvigali na skupni osi. Regalno dvigalo tako ni več eno, zato je bilo potrebno razširiti spremenljivko z regalnim dvigalom v množico in razširiti iteracijo še za posamezno regalno dvigalo (vrstica 19).

Podpora več skladiščem - podpiral je več skladišč z zaprtim sistemom, prileţnimi perifernimi napravami in med sabo nepovezanih.

Instrumentacija - merjenje in protokoliranje podatkov za analizo performančnih problemov (opisano v sekciji 5.3.2 na strani 76).

Dnevniki - izboljšani dnevniki z beleženjem klicov shranjenih baznih procedur (opisano v sekciji 5.3.1 na strani 75).

Sokoban 2.x je upravljal približno 20 različnih skladišč.

Izpis 30 Sokoban 2.x

```
1: function Poišči-Premik-Dvigalo(N, r)
2:     p_h ← 0                                   ▷ Najvišja prioriteta do sedaj
3:     s_b ← nil                                 ▷ Najboljši izvor do sedaj
4:     for all n ∈ D do
5:         s_t ←Premik-Dvigalo(r, n)             ▷ Poišči izvedljiv premik
6:         p_t ←Določi-Prioriteto(s_t)
7:         if p_t > p_h then                     ▷ Višja od trenutno najvišje?
8:             p_h = p_t                         ▷ Da. Shrani trenutno stanje
9:             s_b = s_t
10:        end if
11:    end for
12:    return s_b          ▷ Vrni lokacijo TE za najoptimalnejši premik
13: end function
14:
15: procedure Sokoban-2
16:    R ← {RBGA010001}                          ▷ Lokacija dvigala
17:    N ← {KARA01, VWLA01,..., HBSB02}          ▷ Seznam prileţnih
    naprav
18:    loop
19:        for all r ∈ R do
20:            d_t ←Preberi-Cilj-TE(r)           ▷ Preberi cilj za TE
21:            if d_t ≠ NIL then
22:                d_n ← Določi-Odlagalno-Lokacijo(d_t)
23:                Ustvari-Premik(r, d_n)        ▷ Odloži TE
24:            else                              ▷ TE ni na dvigalu?
25:                s ←Poišči-Premik-Dvigalo(N, r)
26:                if s ≠ NIL then
27:                    Ustvari-Premik(s, r)      ▷ Naloži TE na dvigalo
28:                end if
29:            end if
30:        end for
31:        Opravi-Premike-Perifernih-Naprav
32:        Počohaj-Se                            ▷ Javi se straţarju (watchdog)
33:        Čakaj-Obestilo(60)                    ▷ Časovna kontrola se izteče v 60s
34:    end loop
35: end procedure
```

6.3 Sokoban 3.x

Z verzijo 3.x sem načrtoval podpreti dve povezani skladišči z zaprtim sistemom, vendar je bila verzija 4, ki podpira tudi takšna skladišča pravočasno nared, tako da do izdaje verzije 3.x ni nikoli prišlo.

6.4 Sokoban 4.x (2006)

V začetku leta 2006 je bilo potrebno logistično podpreti skladišča dveh strank, ki so presegla zmogljivosti Sokobana 2.x. Ena stranka je imela skladišče z odprtim sistemom palet, prikazanim na sliki 4.1 na strani 28. Skladišče ima štiri regalna skladišča in velik sistem perifernih enot. Druga stranka je imela povezani dve skladišči z zaprtim sistemom. Torej so lahko transportne enote iz enega prehajale v drugega.

Kljub temu, da je bilo z Sokoban 2.x podprtih približno 20 skladišč, sta bila nova sistema preobsežna, da bi bilo smiselno jih podpirati z verzijo 2.x. Arhitekturne omejitve so bile prevelike.

Potrebno je bilo zasnovati in na novo implementirati program. Nastal je Sokoban 4.x, ki je širše opisan v tej nalogi. Verzija 4.0 je bila prva, ki je modelirala skladišče s pomočjo grafa. Implementacija je bila popolnoma nova in nima skupne kode z verzijo 2.x. Ohranil sem vse funkcionalnosti verzije 2.x in dodal veliko novih funkcionalnosti in izboljšav (vtičniki, sistem za obveščanje o spremebah itd.), ki so se izkazale za koristne med postopnim razvojem in vzdrževanjem verzije 2.x.

Sokoban verzije 4.x upravlja cca. 40 skladišč. Na verzijo 4.x smo ob primernih priložnostih nadgradili številna skladišča verzije 2.x.

6.5 Sokoban 5.x (2010)

V avtomatiziranih skladiščih ni vedno dovolj, da se premikajo le transportne enote. Avtomatizacija se stalno izpopolnjuje in razširja. Poleg transportnih enot moramo skrbeti tudi za premike in druge obravnave materiala, ki imajo svoje zakonitosti. Primer takega procesa je komisioniranje z avtomatskimi napravami - sistem mora sam pripraviti naročeno količino materiala.

V verziji 5.0 lahko Sokoban poleg transporta obdela tudi komisionirni nalog. V komisionirnem nalogu so navedeni materijali in količine, ki jih je stranka naročila. Postopek je naslednji: najprej pripelje Sokoban transportne enote z ustreznim materialom do avtomatske komisionirne naprave. Komisioniranje

se nato začne s pripravo novega praznega nosilca/palete. Zatem Sokoban krmili komisionirno napravo, ki prenaša material iz transportne enote na paleto, dokler ni količina ustrezna. Prenešen materiali nato stehta s pomočjo avtomatske tehtnice. Če teža ustreza prenešeni količini, Sokoban odpelje transportno enoto z materialom nazaj v skladišče in h komisionirni napravi pripelje nov material. Ko je paleta polna, oziroma je zbran ves material za naročilo, pošlje paleto skozi pakirno napravo, nalepi nalepke z oznako in vsebino in jo odpelje po avtomatskem transportu proti izhodnemu mestu, kjer jo prevzame viličar in naloži na kamion.

Poglavje 7

Zaključek

V diplomski nalogi sem prikazal kako je zasnovam program, s katerim sem rešil ključni element sistema za nadzor materialnega toka v avtomatskem transportnem sistemu. Predstavil sem modeliranje skladišča z grafom, kako je razstavljeno na gradnike in opisal elementarne funkcije, s katerimi iščemo premike po skladišču z uporabo algoritma za iskanje najkraše poti. Pojasnil sem zakaj je program neinteraktiven, kako reagira na sporočila iz baze, kako so implementirani vtičniki, dnevniki, konfiguracija itd. in kako se je program razvijal skozi čas, da je podpiral nove "nivoje" skladišč.

Na omejenem prostoru ni bilo moč opisati vseh težav, ki jih je bilo treba rešiti, pa tudi celotna problematika realnih programov v realni uporabi ni tema te naloge. Glavne težave med implementacijo so tiste, ki izhajajo iz neidealnega okolja, predvsem podatkovne baze, na kateri sloni večina implementacije.

Naj naštejem nekaj glavnih:

- iskalni algoritem je spisan v programskem jeziku Java in uvožen v podatkovno bazo kot javanska shranjena procedura (Java stored procedure). S tem se je pojavila znatna razlika v času iskanja poti. Ob hladnem-startu (brez JIT - Just In Time compiler) na Sun Java 1.4 je iskanje poti trajalo 50ms. Znotraj podatkovne baze Oracle 9.2 je enako iskanje trajalo kar 9s! Zato sem uvedel predpomnjenje izračunov poti opisanih v sekciji 4.2.1. V podatkovni bazi Oracle 10.2 je bilo iskanje približno 10x hitrejše kot v verziji 9.2. Če sem procedure prevedel z JAccelerator (prevajalnik), sem s tem pospešil izvajanje za faktor 3 - kar pa je še vedno precej počasnejše od samostojnega JVM. Od verzije 11.1 naprej ima tudi Oracle JVM svoj JIT, tako da je iskanje poti znatno hitrejše brez dodatnih zapletov.

- SQL poizvedbe, ki iščejo premik v in skozi vozlišče so relativno kom-

pleksne. To povroča nestabilnost izvajalnega časa v Oracle verziji 9.2 in 10.2 - dogaja se, da postane SQL poizvedba nenadoma za nekaj časa 100x počasnejša. Vzroka nismo uspeli ugotoviti, kljub večkratnim poskusom in Oraclovi podpori. Verjetno gre za težave v dinamičnem Cost Based Optimizerju.

- Precej zapletena je bila tudi uvedba dveh naprav na skupni osi. Napravi, ki pokrivata isto križišče, ne smeta obe hkrati dostavljati transportne enote na isto vozlišče, saj je tam prostora le za eno. S tem nastane težava, saj postane vozlišče zasedeno že korak prej: s premikom transportne enote v sosednje križišče z namenom premika na ciljno vozlišče.

  Analogna situacija v igrici Sokoban bi bili skladiščniki, ki premikajo zaboje na skupnem področju. Že prej je lahko več skladiščnikov premikalo zaboje sočasno, vendar vsak na svojem območju, za prehod med območji pa je bil določen en sam skladiščnik. Po tej spremembi nastane območje z več skladiščniki, ki si lahko "hodijo v zelje". Vodenje soodvisnih skladiščnikov je bistveno bolj zahtevno, saj je treba upoštevati več kot en korak vnaprej.

- Zahtevno je tudi testiranje. Če pride do napake, je potrebno situacijo ponoviti in transportno enoto postaviti nazaj. To je fizično povsem nepraktično, zato smo napisali simulatorje, ki pa niso 100% enaki napravam. Časa za testiranje s pravimi napravami je na voljo malo, ker so projektni termini običajno kratki, prekinitve produkcije pa stranka, razen izjemoma, ne dovoli, saj vsaka ura, ko skladišče ne obratuje, pomeni visoke izgube.

V celoti se je opisana rešitev izkazala za zelo dobro. Danes Sokoban uspešno upravlja preko 50 avtomatiziranih skladišč po Evropi. Razvoj poteka od leta 2003 naprej in koncept še vedno uspešno deluje tudi v novih sistemih.

7.1 Ideje za naprej

V večjih skladiščih z odprtim sistemom predstavljenim v razdelku 4.1 vidim možnost uporabe algoritmov za doseganje maksimalnega pretoka. Blago prihaja iz proizvodnje na več vhodih v skladišče, se premika k skladiščnim lokacijam ali direktno na izhodne lokacije. V trenutni verziji pošiljamo blago vedno po najoptimalnejših poteh, ki pa so fiksne. Za uporabo alternativnih poti v primeru ozkega grla, je to potrebno implementirati vtičnik, kar pa ni najbolj elegantna in optimalna rešitev.

Problem bi lahko rešili z modeliranjem skladišča na pretočni mreži. Pretočna mreža je sestavljena iz usmerjenega grafa $G = (V, E)$, kjer ima vsaka povezava $(u, v) \in E$ definirano pozitivno kapaciteto pretoka.

Usmerjene povezave v pretočni mreži si lahko predstavljamo kot "kanal" za pretok blaga. Kanal ima definirano kapaciteto, podano kot maksimalno hitrost, s katero se lahko blago premika po kanalu, npr. 100 transportnih enot na uro, 200 litrov na minuto ali 20 kosov/s. Vozlišča v pretočni mreži so križišča kanalov, preko katerih blago potuje brez zadrževanja. Torej mora blago z enako hitrostjo kot vstopa v vozlišče tudi izstopiti iz njega. Posebna vozlišča so izvorna vozlišča, to so vhodi iz produkcije, ter ponorna vozlišča, kjer blago ponikne (izhodi).

Maksimalni pretok učinkovito izračunamo npr. s Ford Fulkersonovo metodo. Ta metoda vključuje več različnih algoritmov z različnimi izvajalnimi časi.

Sokoban že v obstoječi izvedbi išče pot sproti, t.j. za vsak nov premik na poti znova izračuna optimalno pot do cilja z uporabo iskalnega algoritma. Iskalni algoritem je implementiran po načrtovalskem vzorcu Strategija (glej sekcijo), tako da menjava algoritma ni težavna in omogoča uporabo najustreznejšega algoritma za vsako skladišče posebej.

Povezave so predstavljene objektno, tako, da dodajanje kapacitet ne predstavlja težave. Največja težava za uvedbo maksimalnega pretoka je ažurnost trenutne obremenjenosti kanala. Sokoban v trenutni izvedbi v pomnilniku ne hrani nobenega konteksta. Ob vsaki iteraciji si stanje osveži iz baze. V primeru pretočne mreže bi moral osvežiti celotno mrežo, kar bi bilo prepočasno. Sprememba arhitekture, ki bi omogočala, da bi se vsi premiki v celotnem skladišču preslikali tudi v delovni pomnilnik Sokobana, pa zahteva nekaj strukturnih sprememb.

Slike

Seznam izpisov

Literatura

[1] MaFiRa-Wiki, Graf. Dostopno na:
http://wiki.fmf.uni-lj.si/wiki/Graf

[2] MaFiRa-Wiki, Dijkstrov algoritem. Dostopno na:
http://wiki.fmf.uni-lj.si/wiki/Dijkstrov_algoritem

[3] Wikipedia, Graph (mathematics). Dostopno na:
http://en.wikipedia.org/wiki/Graph_(mathematics)

[4] Wikipedia, Fibonacci heap. Dostopno na:
http://en.wikipedia.org/wiki/Fibonacci_heap

[5] Wikipedia, Graph theory. Dostopno na:
http://en.wikipedia.org/wiki/Graph_theory

[6] Wikipedia, Lehman's laws of software evolution. Dostopno na:
http://en.wikipedia.org/wiki/Lehman's_laws_of_software_evolution

[7] David Eppstein, *ICS 161: Design and Analysis of Algorithms*, Dostopno na: http://www.ics.uci.edu/~eppstein/161/960201.html

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, 1992

[9] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, Wiley, New York, NY, 1998

[10] Igor Kononenko, Marko Robnik Šikonja, *Algoritmi in podatkovne strukture I*, Založba FE in FRI, 2003

[11] Igor Kononenko, Marko Robnik Šikonja, *Osnove algoritmov in podatkovnih strukture II*, Založba FE in FRI, 2004

[12] Igor Kononenko, Marko Robnik Šikonja in Zoran Bosnić, *Programiranje in algoritmi*, Založba FE in FRI, 2008

[13] Nazim H. Madhavji; Juan Fernandez-Ramil and Dewayne Perry, *Software evolution and feedback - theory and practice*, Wiley, 2006.