Univerza v Ljubljani

Fakulteta za elektrotehniko

Boštjan Murovec

Preprečevanje neizvedljivosti urnikov pri metahevurističnem razvrščanju proizvodnih procesov

Doktorska disertacija

Mentor: prof. dr. Peter Šuhel

Ljubljana, oktober 2002

Zahvala

Za vsestransko podporo ter prijateljsko vzpodbudo pri nastajanju doktorskega dela se iskreno zahvaljujem svojemu mentorju prof. dr. Petru Šuhlu. Na trnovi poti med znanim in neznanim bi se zanesljivo izgubil brez njegovih nasvetov ter zaupanja v moje delo.

Zahvaljujem se staršema, ki sta vedno verjela vame ter me vzpodbujala pri študiju in delu. Hvaležen sem jima za privzgojene delovne navade, natančnost ter vztrajnost, brez katerih to delo zanesljivo ne bi moglo nastati.

Sandri se zahvaljujem za podporo in potrpežljivost, s katero je prenašala moje prepogosto slabo razpoloženje in pomanjkanje časa. Hvaležen sem ji tudi za številne slovnične in pravopisne nasvete.

Povzetek

Razvrščanje proizvodnih procesov je zahtevna panoga kombinatoričnih optimizacijskih problemov, katere praktična vrednost se kaže neposredno pri večanju izkoristka proizvodnih resorjev ter pri s tem povezanem nižanju proizvodnih stroškov podjetij. Zahtevnost problematike je razvidna iz dejstva, da za večino optimizacijskih problemov s tega področja ne poznamo algoritmov reševanja, katerih časovna zahtevnost bi naraščala polinomsko v odvisnosti od velikosti problema. Posledica je, da v praksi izvajamo optimizacije aproksimativno, s čimer dobimo le blizu-optimalne rešitve v času, ki je sicer praktično sprejemljiv za planiranje in realizacijo proizvodnje.

Delo obravnava aproksimativno reševanje deterministočnega problema razvrščanja operacij, ki so podvržene tehnološkim omejitvam, po vnaprej znani množici proizvodnih resorjev z namenom doseči čim krajše trajanje izvajanja vseh opravil. Uporabili smo pristop lokalnega iskanja, ki temelji na iterativnem izboljševanju začetne rešitve s pomočjo spreminjanja zaporedja izvajanja operacij na posameznih strojih. Zaradi tehnoloških omejitev med operacijami je določena podmnožica teh sprememb neizvedljiva, kar predstavlja problem pri realizaciji lokalnega iskanja, saj se je potrebno takim spremembam izogniti. Dosedaj predlagane strategije preprečevanja neizvedljivosti so v splošnem nezadovoljive, saj se odpovedo preiskovanju določenih delov prostora rešitev, ali uvajajo drugačne vrste kompromisov.

Da bi težavo odpravili, smo razvili in teoretično utemeljili splošni postopek preprečevanja nastanka neizvedljivih rešitev, na katerem sloni naša izvedba lokalnega iskanja. Empirični rezultati spodbujajo uporabo predlaganega pristopa, saj smo s tremi testnimi problemi dobili rekordne rešitve.

Ključne besede: razvrščanje procesov, kombinatorična optimizacija, lokalno iskanje, tabu iskanje, genetski algoritmi.

Abstract

Production scheduling is a complex branch of combinatorial optimization problems. Its practical value can be directly measured by the ability of a cost reduction and performance enhancements of the production units. The fact that there are no known algorithms with polynomial time requirments for solving the majority of the algorithms in this area stresses the complexity of the topic. The aproximation algorithms which are able to deliver sub-optimal solutions in a manageable time span have been accepted as a necessity, for the practical purpose.

The emphasis of the thesis is the approximation algorithm for solving the deterministic job-shop combinatorial optimization problem with the makespan criterion. The underlying technique is the local search, which starts with an arbitrary feasible solution and iteratively seeks better ones by performing a small change of a processing order on some machine at each step. The existence of the technological constraints among technological operations results in the fact that some of the changes are infeasible, which possesses a problem for local search implementations. So far the proposed solutions to overcome the difficulty are unsatisfactory, since they give up on exploring some parts of a solution space or result in some other compromise.

As an answer to the problem, we developed and theoretically founded a generally utilizable algorithm for preventing infeasible solutions which deals with the infeasibility in a more satisfactory way. The empirical tests are relatively encouraging, since we acchieved new upper bounds on the optimal solution for the three open test instances.

Key words: scheduling, combinatorial optimization, local search, tabu search, genetic algorithms.

Vsebina

1. Uvod **1**

1.1 Definicija problema 3

1.2 Razširitve osnovnega modela in sorodni problemi 6

1.3 Ostali problemi razvrščanja opravil 7

1.4 Vzorčni primer Π_J instance 9

1.5 Gantt diagram 10

1.6 Regularne kriterijske funkcije 12

2. Prostor rešitev Π_J problema **13**

2.1 Pol-aktivni urniki 13

2.2 Aktivni urniki 17

2.3 Brez-čakalni urniki 19

2.4 Teorija kompleksnosti 19

3. Neusmerjeni vozliščno-uteženi graf **23**

3.1 Modeliranje tehnoloških omejitev med operacijami 23

3.2 Modeliranje zmogljivostnih omejitev strojev 25

3.3 Redukcija neusmerjenih povezav 27

3.4 Določitev urnika s pomočjo reduciranih povezav 30

3.5 Tranzitivnost predhodnosti in naslednosti 32

3.6 Glave, repi in kritična pot 33

3.7 Izvajanje premikov na urniku 36

3.7.1 Formalna definicija premikov 36

3.7.2 Teoretična izhodišča za izvajanje učinkovitih premikov . . 37

3.8 Topološko zaporedje 43
3.9 Delna izbira 46
3.10 Razširitve modeliranja z neusmerjenimi grafi 46

4. Pregled postopkov reševanja **47**
4.1 Delitev postopkov reševanja 47
4.2 Začetki razvoja Π_J problema 48
4.3 Drugi testni primeri instanc 49
4.4 Eksaktno reševanje problema 51
4.4.1 Osnovni postopek veji in omejuj 52
4.4.2 Izboljšane različice postopka veji in omejuj 53
4.5 Aproksimativno reševanje problema 58
4.5.1 Naključno generiranje aktivnih urnikov 58
4.5.2 Omejevanje preiskovalnega drevesa 59
4.5.3 Prioritetna pravila 59
4.5.4 Odpravljanje ozkega grla 60
4.6 Lokalno iskanje in meta-hevristika 60
4.6.1 Okolica za izvedbo lokalnega iskanja 62
4.6.2 Vrednotenje premikov v okolici 64
4.6.3 Postopno ohlajanje 67
4.6.4 Iskanje s tabu seznamom 71
4.6.5 Genetski algoritmi 74
4.6.6 Genetsko lokalno iskanje 79

5. Popravljalna tehnika **83**

5.1 Dosedanji pristopi k izogibanju neizvedljivosti 83

5.1.1 Slabosti okolice $\mathcal{H}_B$ 84

5.1.2 Prednosti in slabosti okolice $\mathcal{H}_D$ 87

5.2 Ideja popravljalne tehnike 90

6. Implementacija popravljalne tehnike **93**

6.1 Detekcija ciklov 93

6.1.1 Preureditev topološkega zaporedja 93

6.1.2 Detekcija ciklov na podlagi označenih operacij 96

6.2 Izbira popravljalnih premikov 100

6.3 Celotni postopek izvajanja premikov 105

7. Okolica urnika **109**

7.1 Formalna definicija predlagane okolice 109

7.2 Lastnosti predlagane okolice 110

8. Smernice za izvajanje empiričnih testov **117**

9. Iskanje s tabu seznamom **119**

10. Genetski algoritem **125**

11. Rezultati pri iskanju s tabu seznamom **129**

11.1 Rezultati tabu iskanja pri času izvajanja 10 ms 130

11.2 Rezultati tabu iskanja pri času izvajanja 50 ms 132

11.3 Rezultati tabu iskanja pri času izvajanja 100 ms 134

11.4 Rezultati tabu iskanja pri času izvajanja 500 ms 136

11.5 Rezultati tabu iskanja pri času izvajanja 1000 ms 138

11.6 Komentar rezultatov tabu iskanja 140

12.Rezultati uporabe genetskega algoritma 143

12.1 Primerjava naše okolice z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ 144

12.2 Primerjava naše okolice z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ 146

12.3 Komentar rezultatov genetskega algoritma 147

12.4 Dodatni testi z genetskim algoritmom 148

13.Diskusija 151

14.Predlogi za nadaljevanje raziskav 155

15.Izvirni prispevki 157

Literatura 159

Seznam tabel

1.1 Definicija vzorčne instance Π_J problema velikosti 4×3. 10

1.2 Primer urnika za izvedbo vzorčne Π_J instance. 10

2.1 Podajanje urnika z zaporedjem izvajanja operacij. 15

2.2 Zgornja meja števila pol-aktivnih urnikov. 15

2.3 Primer neizvedljivega urnika. 16

2.4 Odvisnost časa izvajanja algoritmov od njihove kompleksnosti. . . 21

6.1 Potek označevanja operacij z algoritmom 1. 99

6.2 Topološko zaporedje po izvedbi popravljalnega premika. 105

11.1 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 10 ms). 130

11.2 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 10 ms). 131

11.3 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 50 ms). 132

11.4 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 50 ms). 133

11.5 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 100 ms). 134

11.6 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 100 ms). 135

11.7 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 500 ms). 136

11.8 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 500 ms). 137

11.9 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 1000 ms). 138

11.10 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 1000 ms). 139

12.1 Izbira parametrov genetskega algoritma. 143

12.2 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji z genetskim algoritmom. 145

12.3 Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji z genetskim algoritmom. 146

12.4 Primerjava mutacijskih operatorjev. 148

Seznam slik

1.1 Prikaz urnika vzorčne Π_J instance s pomočjo Gantt diagrama. . . 11

2.1 Pol-aktivni urnik, ki pripada začetnemu urniku. 14

2.2 Primer neizvedljivega urnika. 16

2.3 Aktivni urnik, ki ga izvedemo iz pol-aktivnega na sliki 2.1. 17

2.4 Primer drugega aktivnega urnika. 18

3.1 Prikaz grafa $\mathcal{G}'_{\mathcal{A}}$, ki ustreza vzorčni instanci. 24

3.2 Prikaz grafa $\mathcal{G}'_{\mathcal{E}}$ za primer vzorčne instance. 25

3.3 Kompletna izbira $\mathcal{G}$. 26

3.4 Kompletna izbira, ki pripada neizvedljivemu urniku. 27

3.5 Reducirana kompletna izbira $\mathcal{G}_{\text{red}}$. 28

3.6 Reducirana kompletna izbira, ki pripada neizvedljivemu urniku. . 28

3.7 Hamiltonove poti podgrafov $\mathcal{G}_{\mathcal{E}1}$, $\mathcal{G}_{\mathcal{E}2}$ in $\mathcal{G}_{\mathcal{E}3}$. 30

3.8 Celotni podgrafi $\mathcal{G}_{\mathcal{E}1}$, $\mathcal{G}_{\mathcal{E}2}$ in $\mathcal{G}_{\mathcal{E}3}$. 31

3.9 Prikaz vseh predhodnic in naslednic operacije $2b$. 32

3.10 Ponoven prikaz urnika vzorčne Π_J instance. 33

3.11 Desno poravnani urnik vzorčne Π_J instance. 34

3.12 Urnik s slike 3.10 po izvedbi operacije $1c$ pred operacijo $3c$. 39

3.13 Urnik s slike 3.12 po izvedbi operacije $2a$ pred operacijo $4b$. 40

3.14 Urnik s slike 3.10 brez nekritičnih operacij. 40

3.15 Urnik s slike 3.14 po zamenjavi operacij $2a$ in $1b$. 41

4.1 Ponoven prikaz urnika vzorčne Π_J instance. 70

4.2 Urnik po zamenjavi vrstnega reda operacij $3c$ in $1c$. 70

5.1 Premik operacije $2a$ za operacijo $3b$ z uporabo okolice $\mathcal{H}_B$. 86

5.2 Premik operacije $2a$ za operacijo $3b$ z uporabo okolice $\mathcal{H}_D$. 88

5.3 Dvonivojska zasnova lokalnega iskanja. 90

5.4 Zasnova lokalnega iskanja z uporabo popravljalne tehnike. 91

6.1 Algoritem označevanja operacij, ki jih je potrebno v $\mathcal{T}(\mathcal{G}^{\curvearrowright})$ preurediti. 95

6.2 Algoritem za preureditev topološkega zaporedja. 97

6.3 Algoritem za izbiro popravljalnih premikov. 101

6.4 Algoritem za izvajanje varnih premikov. 106

6.5 Premik operacije $2a$ za operacijo $3b$ z uporabo okolice $\mathcal{H}_D$. 107

9.1 Algoritem za izvedbo iskanja s tabu seznamom. 120

Seznam uporabljenih simbolov

Simbol	Pomen
Π_{J}	obravnavani problem razvrščanja proizvodnih procesov
$\emptyset$	prazna množica
$\lvert\mathcal{Z}\rvert$	število elementov v množici $\mathcal{Z}$
$\max(a_1, \ldots, a_n)$	največja vrednosti izmed $a_1, \ldots, a_n$
$\min(a_1, \ldots, a_n)$	najmanjša vrednosti izmed $a_1, \ldots, a_n$
n	število opravil v instanci
m	število strojev v instanci
$\mathcal{J}$	množica vseh opravil v instanci
$\mathcal{J}_{\mathrm{i}}$	i-to opravilo v instanci
$\mathcal{M}$	množica vseh strojev v instanci
$\mathcal{M}_{\mathrm{j}}$	j-ti stroj v instanci
$n(i)$	število operacij v opravilu $\mathcal{J}_{\mathrm{i}}$
n_{tot}	skupno število operacij v vseh opravilih
$w_{i,j}$	j-ta operacija i-tega opravila
$p_{i,j}$	čas izvajanja operacije $w_{i,j}$
$t_{i,j}$	začetni čas izvajanja operacije $w_{i,j}$
$\tilde{t}_{i,j}$	začetni čas izvajanja $w_{i,j}$ v desno poravnanem urniku
$e_{i,j}$	končni čas izvajanja operacije $w_{i,j}$
$\tilde{e}_{i,j}$	končni čas izvajanja $w_{i,j}$ v desno poravnanem urniku
$\circledcirc$	fiktivna operacija, ki predstavlja začetek urnika
$\circledast$	fiktivna operacija, ki predstavlja konec urnika
w_i	operacija z indeksom i ($w_0 = \circledcirc$, $w_1, \ldots, w_{n(1)} \in \mathcal{J}_1, \ldots$)
$\mathrm{num}(w_k)$	indeks operacije w_k
$\mathcal{J}(w_k)$	opravilo, kateremu w_k pripada
$\mathcal{M}(w_k)$	stroj, na katerem se mora w_k izvršiti
$C_{\max}$	časovni interval izvedbe celotnega urnika
$C'_{\max}$	ocenjeni časovni interval izvedbe celotnega urnika
$C^*_{\max}$	najmanjši možni (optimalni) $C_{\max}$
$C^{\downarrow}_{\max}$	zgornja meja optimalnega $C_{\max}$
$w_i^{\diamond}$	i-ta podana operacija (en stroj)
$n^{\diamond}$	število vseh operacij pri razvrščanju (en stroj)
$p_i^{\diamond}$	čas izvajanja operacije $w_i^{\diamond}$ (en stroj)
$C_i^{\diamond}$	čas dokončanja operacije $w_i^{\diamond}$ (en stroj)
$L_i^{\diamond}$	zapoznelost operacije $w_i^{\diamond}$ (en stroj)
$T_i^{\diamond}$	počasnost operacije $w_i^{\diamond}$ (en stroj)

Seznam uporabljenih simbolov (nadaljevanje)

Simbol	Pomen
$E_i^\diamond$	zgodnost operacije $w_i^\diamond$ (en stroj)
$U_i^\diamond$	enota počasnosti operacije $w_i^\diamond$ (en stroj)
$d_i^\diamond$	predpisani končni čas operacije $w_i^\diamond$ (en stroj)
$\omega_i^\diamond$	teža pomembnosti operacije $w_i^\diamond$ (en stroj)
$R_i^\diamond$	čas pripravljenosti operacije $w_i^\diamond$ (en stroj)
$t_i^\diamond$	začetni čas izvajanja operacije $w_i^\diamond$ (en stroj)
$D_i^\diamond$	čas dokončanja operacije $w_i^\diamond$ (en stroj)
$C_{\max}^\diamond$	celotni izvršni čas (en stroj)
Φ	problem enega stroja ob podanih časih $R_i^\diamond$, $p_i^\diamond$ in $D_i^\diamond$
$\mathcal{G}'$	neusmerjeni graf, ki modelira Π_{J} instanco
$\mathcal{G}'_{\mathcal{A}}$	graf $\mathcal{G}'$ brez povezav v množici $\mathcal{E}$
$\mathcal{G}'_{\mathcal{E}}$	graf $\mathcal{G}'$ brez povezav v množici $\mathcal{A}$
$\mathcal{G}$	graf $\mathcal{G}'$ z usmerjenimi povezavami v množici $\mathcal{E}$
$\mathcal{G}_{\mathcal{E}}$	graf $\mathcal{G}$ brez povezav v množici $\mathcal{A}$
$\mathcal{G}_{\text{red}}$	graf $\mathcal{G}$, ki vsebuje množico $\mathcal{E}_{\text{red}}$ namesto množice $\mathcal{E}$
$\mathcal{G}_{\mathcal{E}\text{red}}$	graf $\mathcal{G}$, ki vsebuje samo povezave v množici $\mathcal{E}_{\text{red}}$
$\mathcal{G}_{\mathcal{E}i}$	graf z operacijami na stroju $\mathcal{M}_i$ in njihovimi povezavami
$\mathcal{G}^{\curvearrowright}$	graf $\mathcal{G}$ po izvedbi premika na urniku
$\mathcal{N}$	množica vozlišč grafa $\mathcal{G}'$
$\mathcal{N}_0$	množica vozlišč, ki vsebuje operaciji $\circledcirc$ in $\circledast$
$\mathcal{N}_i$	množica vozlišč, ki vsebuje operacije na stroju $\mathcal{M}_i$
$\mathcal{A}$	množica usmerjenih povezav grafa $\mathcal{G}'$
$\mathcal{E}$	množica neusmerjenih povezav grafa $\mathcal{G}'$
$\mathcal{E}_{\text{red}}$	reducirana množica $\mathcal{E}$
$\mathcal{E}_i$	množica povezav med operacijami na stroju $\mathcal{M}_i$
$\mathcal{E}_{\text{red}i}$	reducirana množica $\mathcal{E}_i$
$w_i \xrightarrow{\mathcal{A}} w_j$	usmerjena povezava v množici $\mathcal{A}$ od w_i do w_j
$w_i \xleftrightarrow{\mathcal{E}} w_j$	neusmerjena povezava v množici $\mathcal{E}$ med operacijama w_i in w_j
$w_i \xrightarrow{\mathcal{E}} w_j$	usmerjena povezava v množici $\mathcal{E}$ od w_i do w_j
$w_i \xrightarrow{\mathcal{E}_{\text{red}}} w_j$	usmerjena povezava v množici $\mathcal{E}_{\text{red}}$ od w_i do w_j
$P_{\text{J}}(w_i)$	neposredni tehnološki predhodnik operacije w_i
$\mathcal{P}_{\text{J}}(w_i)$	množica vseh tehnoloških predhodnikov operacije w_i
$S_{\text{J}}(w_i)$	neposredni tehnološki naslednik operacije w_i
$\mathcal{S}_{\text{J}}(w_i)$	množica vseh tehnoloških naslednikov operacije w_i

Seznam uporabljenih simbolov (nadaljevanje)

Simbol	Pomen
$P_{\mathrm{M}}(w_i)$	neposredni predhodnik operacije w_i na stroju
$\mathcal{P}_{\mathrm{M}}(w_i)$	množica vseh predhodnikov operacije w_i na stroju
$S_{\mathrm{M}}(w_i)$	neposredni naslednik operacije w_i na stroju
$\mathcal{S}_{\mathrm{M}}(w_i)$	množica vseh naslednikov operacije w_i na stroju
$\mathcal{P}(w_i)$	množica vseh predhodnikov operacije w_i
$\mathcal{S}(w_i)$	množica vseh naslednikov operacije w_i
$\mathcal{O}_i$	zaporedje izvajanja operacij na stroju $\mathcal{M}_{\mathrm{i}}$
$\mathcal{O}(w_a, w_b)$	del $\mathcal{O}_i$ od operacije w_a do operacije w_b (vključno)
$h(w_i)$	glava operacije w_i
$h'(w_i)$	ocena vrednosti glave operacije w_i
$q(w_i)$	rep operacije w_i
$q'(w_i)$	ocena vrednosti repa operacije w_i
$C(\mathcal{G})$	kritična pot grafa $\mathcal{G}$
r	število kritičnih blokov, ki sestavljajo $C(\mathcal{G})$
B_i	i-ti kritični blok na kritični poti
B_i^k	k-ta operacija i-tega kritičnega bloka
$\mathcal{T}(\mathcal{G})$	topološko zaporedje grafa $\mathcal{G}$
$\mathcal{T}(\mathcal{G}^{\curvearrowright})$	topološko zaporedje grafa $\mathcal{G}^{\curvearrowright}$
$\mathcal{T}(w_i, w_j)$	del $\mathcal{T}(\mathcal{G})$ med operacijama w_i in w_j (vključno)
$P_{\mathrm{T}}(w_i)$	neposredni topološki predhodnik operacije w_i
$S_{\mathrm{T}}(w_i)$	neposredni topološki naslednik operacije w_i
$\mathcal{L}$	množica označenih operacij za preureditev v $\mathcal{T}(\mathcal{G}^{\curvearrowright})$
$Q_{\mathrm{D}}(w_a, w_b)$	premik operacije w_a v desno za operacijo w_b
$Q_{\mathrm{L}}(w_a, w_b)$	premik operacije w_b v levo pred operacijo w_a
$Q(w_a, w_b)$	premik brez oznake smeri: $Q_{\mathrm{D}}(w_a, w_b)$ ali $Q_{\mathrm{L}}(w_a, w_b)$
$Q^*_{\mathrm{D}}(w_a, w_b)$	varen premik operacije w_a v desno za operacijo w_b
$Q^*_{\mathrm{L}}(w_a, w_b)$	varen premik operacije w_b v levo pred operacijo w_a
$Q^*(w_a, w_b)$	varen premik brez oznake smeri: $Q^*_{\mathrm{D}}(w_a, w_b)$ ali $Q^*_{\mathrm{L}}(w_a, w_b)$

1. Uvod

V serijski proizvodnji se nenehno srečujemo s problematiko *razvrščanja* (ang. *scheduling*), saj želimo čim uspešneje izvesti določen nabor operacij z uporabo po številu in kapaciteti omejenega nabora proizvodnih resorjev (Baker 1974, Koren 1985, Šuhel in sod. 1989, Kumar 1983).

Operacije niso vedno med seboj neodvisne, ampak so pogosto podvržene tehnološkim omejitvam (ang. *technological constraints*)[1]. To pomeni, da se posamezna operacija ne more izvršiti, dokler niso izvedene vse njene tehnološke predhodnice. Proizvodni resorji (v nadaljevanju: stroji) lahko v večini primerov naenkrat opravljajo samo po eno opravilo, poleg tega operacij med izvajanjem dostikrat ne smemo prekinjati.

Omenili smo samo nekaj težav in omejitev, na katere naletimo v praksi pri določanju *urnika* (ang. *schedule*), po katerem naj se operacije izvajajo. Slabo zastavljen urnik bo povzročil predolgo čakanje določenih operacij na svoje tehnološke predhodnice, stroji pa bodo stali neizkoriščeni. Pojav je posledica prekomernega kopičenja in zastojev operacij na drugih strojih, kjer le-te čakajo na izvršitev ter s tem blokirajo izvajanje svojih tehnoloških naslednic.

Teorija razvrščanja (ang. *scheduling theory*) je panoga, ki raziskuje postopke izdelave učinkovitih urnikov ob prisotnosti omejitev na katere naletimo v praksi pri izvajanju operacij. Teorija se intenzivno naslanja na matematična orodja in izsledke, za kar potrebuje ustrezne matematične modele, ki praktične probleme abstrahirajo v matematično kompaktno in precizno obliko. Težava, s katero se pri tem srečujemo, je, da preprosti modeli premalo natančno opisujejo realna dogajanja, zato je uporabnost z njimi pridobljenih rešitev omejena.

Po drugi strani velja, da je kompleksne in bolj precizne modele težje uporabljati ter da njihova kompleksnost pogosto prepreči pridobivanje uporabnih rešitev; zato je razvoj in definicija učinkovitih modelov proizvodnje odprta problematika teorije razvrščanja. Izkušnje kažejo, da se v konkretnih situacijah splošno

[1] Večkrat zasledimo v literaturi izraz prednostne omejitve (ang. *precedence constraints*).

zastavljeni modeli (za reševanje širše množice problemov) slabše obnašajo od namensko razvitih modelov za reševanje specifičnih in ozko usmerjenih problemov, kar je tudi za pričakovati.

Za teorijo razvrščanja so pomembni oboji, saj so splošneje zastavljeni modeli zanimivi za širši krog raziskovalcev in se zato hitreje razvijajo. Ob tem pridobljena spoznanja so velikokrat neposredno ali posredno uporabna tudi pri reševanju problemov z namensko razvitimi modeli.

V našem primeru se osredotočamo na obravnavo splošno namenskega *determinističnega modela razvrščanja operacij, ki so grupirane po opravilih ter znotraj posameznega opravila podvržene tehnološkim omejitvam* (ang. *deterministic job-shop scheduling problem*).

Eksaktno rešiti problem (precizneje: njegovo določeno instanco) pomeni poiskati urnik, katerega časovni interval, potreben za izvedbo vseh operacij (ang. *makespan*), je najkrajši. V splošnem je to nemogoče izvesti v doglednem času, saj problem kljub enostavnemu opisu (poglavje 1.1), ki še zdaleč ne daje slutiti njegove kompleksnosti, spada med težje rešljive matematične probleme nasploh (French 1982); čas, ki je potreben za eksaktno reševanje instanc z velikostnim redom 400 operacij, pogosto preseže življensko dobo naše galaksije. V praksi se zato poslužujemo aproksimativnega oziroma hevrističnega reševanja (Morton in Pentico 1993), kjer so dobljene rešitve slabše od optimalnih, časovna zahtevnost postopkov pa postane obvladljiva in sprejemljiva.

V pričujočem delu obravnavamo postopek hevrističnega reševanja zastavljenega problema s pomočjo tehnike, ki ji pravimo *lokalno iskanje* (ang. *local search*). Ideja temelji na iterativnem izboljševanju začetne rešitve, kjer se v vsakem koraku optimizacije na urniku izvede določena sprememba izmed vnaprej definirane množice sprememb, ki ji pravimo *okolica* urnika (ang. *neighborhood*). Ko z nobeno spremembo v okolici ne moremo doseči izboljšave urnika, smo dosegli lokalni minimum. Da bi bila nadaljnja optimizacija možna, se poslužujemo metahevrističnih pristopov, ki dovoljujejo spreminjanje urnikov tudi na način, ki njihovo kvaliteto poslabša. S tem je omogočen pobeg iz lokalnega minimuma in nadaljnje iskanje boljših rešitev.

1.1 Definicija problema

Deterministični problem razvrščanja proizvodnih procesov (v nadaljevanju problem Π_J) je podan s končnim številom n opravil $\{\mathcal{J}_i\}_{i=1}^n$ iz množice $\mathcal{J}$, ki morajo biti izvedena s pomočjo končnega števila m strojev $\{\mathcal{M}_j\}_{j=1}^m$ iz množice $\mathcal{M}$. Vsako opravilo $\mathcal{J}_i$ je zaporedje $n(i)$ operacij $w_{i,1}, \ldots w_{i,n(i)}$, od katerih se mora vsaka izvesti na predpisanem stroju $\mathcal{M}_{i,k} \in \mathcal{M}$, za kar potrebuje predpisan čas izvajanja[2] $p_{i,k} > 0$. Skupno število operacij v vseh opravilih je $n_{\text{tot}} = \sum_{i=1}^n n(i)$. Operacije, ki pripadajo istemu opravilu, so podvržene tehnološkim omejitvam: začetni čas izvajanja $t_{i,k}$ operacije $w_{i,k}$, ki ni prva v opravilu ($k > 1$), ne sme biti manjši od končnega časa izvajanja ($e_{i,k-1} = t_{i,k-1} + p_{i,k-1}$) njene tehnološke predhodnice $w_{i,k-1}$; v primeru prve operacije mora veljati $t_{i,1} \geq 0$ za vsak $i = 1, \ldots, n$. Na vsakem stroju se lahko izvaja samo ena operacija naenkrat, kar predstavlja zmogljivostne omejitve (ang. *capacity constraints*). Poleg tega prekinitve izvajanja operacij (ang. *preempt*) niso dovoljene. Izvršni čas urnika $C_{\max}$ je dolžina časovnega intervala, v katerem se izvede vseh n_{tot} operacij. Zaradi tehnoloških omejitev velja $C_{\max} = \max_{i=1}^n (e_{i,n(i)})$. Rešiti problem pomeni poiskati tak urnik, ki bo imel najmanjši izvršni čas $C^*_{\max} = \min(C_{\max})$ izmed vseh izvedljivih urnikov. S tem ima izvršni čas urnika vlogo kriterijske funkcije. Ostali problemi razvrščanja imajo lahko drugačne kriterijske funkcije, kar je v nadaljevanju podrobneje obdelano.

V teoretičnih študijah se navadno obravnava restriktivnejši Π_J model, kjer se mora pri vsakem opravilu izvršiti natančno ena operacija na vsakem stroju. Število vseh operacij je tako $n_{\text{tot}} = n \cdot m$, zato govorimo o pravokotnem (ang. *rectangular*) Π_J problemu in navajamo dimenzionalnost instance kot $n \times m$. Omenimo še primer pravokotnega problema, kjer dodatno velja, da je število opravil enako številu strojev (velja relacija $n = m$), čemur pravimo instanca kvadratnih dimenzij (ang. *square in dimensionality*).

[2] Časi izvajanja so podani kot celoštevilčni mnogokratniki osnovne časovne enote. Proizvodni proces s časi izvajanja iz množice racionalnih števil modeliramo tako, da izberemo ustrezno majhno osnovno enoto ter čase izvajanja temu primerno skaliramo.

V tem delu bomo obravnavali samo instance, kjer se lahko izvede največ ena operacija vsakega opravila na posameznem stroju, kar je manj restriktiva zahteva od pravokotnosti.

Velikokrat smo v situaciji, ko obravnavamo določeno operacijo izmed n_{tot} operacij, ne želimo pa se poglabljati v podatek, kateremu opravilu pripada. Zato operacijam priredimo enotno številko v intervalu od 1 do n_{tot}, kar označimo z w_k (torej samo številka operacije k in ne par indeksov i, j), kjer $w_1, \ldots, w_{n(1)} \in \mathcal{J}_1$, $w_{n(1)+1}, \ldots, w_{n(1)+n(2)} \in \mathcal{J}_2$ in naprej do $w_{n_{\text{tot}}-n(n)+1} \ldots w_{n_{\text{tot}}} \in \mathcal{J}_{\text{n}}$[3]. Definirajmo tudi operator num(), ki vrne enotno številko operacije: $\text{num}(w_k) = k$. Opravilo, kateremu operacija w_k pripada, označimo kot $\mathcal{J}(w_k)$, stroj, na kateremu se le-ta izvršuje pa kot $\mathcal{M}(w_k)$.

Opisani model Π_{J} problema temelji na naslednjih predpostavkah, od katerih so nekatere izražene implicitno (French 1982).

1. Dve operaciji istega opravila se ne smeta izvajati istočasno.
2. Operacije med izvajanjem ne smemo prekiniti in je pozneje dokončati.
3. Pri nobenem opravilu se ne izvaja več kot ena operacija na posameznem stroju.
4. Vse operacije vseh opravil moramo izvesti v celoti.
5. Časi izvajanja operacij so neodvisni od urnika.
6. Operacija čaka sprostitev stroja, če je le-ta zaseden.
7. Za vsako operacijo obstaja samo en stroj, ki jo lahko izvrši.
8. Med izvajanjem urnika so lahko stroji prosti.
9. Noben stroj ne more izvajati več kot ene operacije naenkrat.
10. Stroj je vedno pripravljen na izvajanje.

[3] V literaturi se pogosto uporablja številčenje po drugačnem zaporedju: $w_1 = w_{1,1}$, $w_2 = w_{2,1}$ in tako naprej; torej najprej oštevilčimo vse prve operacije opravil, nato vse druge, ... Problem takega zaporedja je, da je primerno le za pravokotne instance, na katere se ne želimo omejiti.

11. Tehnološke omejitve so poznane vnaprej in so nespremenljive.

12. Vsi parametri (n, m, $p_{i,k}$) so deterministični in znani vnaprej.

Mattfeld (1996) dodatno poudarja še naslednje lastnosti zastavljenega modela.

13. Vsa opravila so pripravljena na izvajanje ob času nič.

14. Za nobeno opravilo ne zahtevamo, da se konča pred ostalimi.

15. Čas za pripravo stroja med izvajanjem operacij je zanemarljiv.

Nobena od navedenih predpostavk v praksi ne velja vedno, saj so situacije v proizvodnji preveč specifične, da bi lahko vse obravnavali s tako poenostavljenim modelom. Navedimo samo nekaj nasprotujočih primerov.

- Vse operacije niso nujno podvržene tehnološkim omejitvam, kot na primer pri sestavljanju končnega izdelka iz večjega števila polizdelkov, od katerih se vsak lahko proizvede neodvisno (lastnost 1).

- Stroje je potrebno med izvajanjem operacij čistiti, če so materiali obdelovancev različni (lastnost 5).

- Včasih operacija ne sme čakati sprostitve stroja, kot je to primer pri valjanju jekla, ki se ne sme ohladiti (lastnost 6).

- Na voljo imamo lahko več enakih strojev za izvajanje določene operacije (lastnost 7).

- Stroj se lahko pokvari (lastnosti 10, 12 in 15).

Vidimo, da je predpostavk, na katerih je problem zasnovan, veliko, zato je praktično nemogoče vpeljati splošen model, kjer bi vse opustili, saj bi bilo sestavljanje urnikov na tak način prezahtevno za praktično uporabo, ker je že osnovni Π_J problem izredno zahteven za reševanje.

1.2 Razširitve osnovnega modela in sorodni problemi

V odgovor na specifičnost in raznolikost zahtev proizvodnih situacij so se pojavili razširjeni modeli, ki osnovni Π_J problem posplošujejo, največkrat z odpravo samo ene od navedenih predpostavk, ki je v določeni situaciji najbolj moteča. Najpogosteje obravnavane razširitve so naslednje.

Generalizirani Π_J (ang. *generalized* Π_J) za izvrševanje vsake operacije predvideva večje število strojev. Časi izvajanja operacij so lahko za vsak stroj različni (Vaessens 1995, Mastrolilli in Gambardella 2000).

Π_J s pokvarljivimi stroji (ang. Π_J *with machine breakdowns*) predvideva, da stroji v določenih časovnih intervalih niso na voljo zaradi popravila ali zaradi kakršnegakoli drugega razloga (Holthaus 1999).

Večkratno izvajanje opravila na istem stroju (ang. *reentrant shop*) predvideva, da lahko posamezno opravilo vsebuje več operacij, ki se izvajajo na istem stroju (Morton in Pentico 1993).

Stohastični Π_J (ang. *stochastic* Π_J) ne predpostavlja deterministčnih, ampak stohastične parametre, ki opisujejo problem (Pinedo 1995).

Poleg naštetih razširitev obstaja tudi množica problemov, ki so problemu Π_J bolj ali manj sorodni, obravnavamo pa jih ločeno. Na tem mestu omenimo samo dva taka primera. Popolnejši pregled podajajo reference, kot so Morton in Pentico (1993) ter Pinedo (1995).

Π_J brez tehnoloških omejitev (ang. *open shop*) ne upošteva tehnoloških omejitev med operacijami. Situacije, ki jih lahko tako obravnavamo, so v praksi redke, vendar obstajajo. Tak primer je opremljanje izdelka z nalepko in tehtanje. Katerokoli od teh operacij lahko izvedemo najprej (ob predpostavki, da nalepka zanemarljivo vpliva na težo izdelka), ne moremo pa izvesti obeh naenkrat, ker bi lepljenje nalepke tehtanje motilo.

Π_J z uniformnimi opravili (ang. *flow shop*) predvideva, da se operacije vseh opravil izvajajo v istem zaporedju. Taka situacija obstaja na primer v valjarnah, kjer morajo različni izdelki skozi enake faze proizvodnje po enakem vrstnem redu.

Slednji problem bi lahko obravnavali kot navaden Π_J, kjer je s specifikacijo instance določeno, da so vsa opravila uniformna. Kljub temu problema obravnavamo ločeno, ker uniformnost odpira dodatne možnosti in poenostavitve, ki jih uspešno izkoriščamo pri snovanju algoritmov.

1.3 Ostali problemi razvrščanja opravil

V začetnem obdobju razvoja teorije razvrščanja procesov so bile raziskave intenzivno usmerjene v optimizacijo problemov z enim proizvodnim strojem (ang. *single machine problems*). Zanje razviti postopki optimizacije so večkrat uporabljeni kot gradniki postopkov za reševanje Π_J problema, zato jih na tem mestu na kratko omenjamo (kjer referenca ni podana, je povzetek narejen iz Morton in Pentico 1993).

Večina problemov v tej kategoriji predpostavlja, da moramo razvrstiti končno število $n^\diamond$ neodvisnih operacij $w_1^\diamond, \ldots, w_{n^\diamond}^\diamond$, ki se morajo vse izvršiti na enem stroju[4]. Za vsako od njih je predpisan izvršni čas $p_i^\diamond > 0$ $(1 \leq i \leq n^\diamond)$. Zaradi neodvisnosti operacij se celotni izvršni čas $C_{\max}^\diamond$ ne spreminja z zaporedjem, po katerem so le-te izvedene, saj je vedno $C_{\max}^\diamond = \sum_{i=1}^{n^\diamond} p_i^\diamond$; zato ta kriterijska funkcija ni zanimiva. Nadomešča jo cel spekter drugih kriterijskih funkcij, od katerih ima vsaka določeno praktično vrednost odvisno od proizvodne situacije. Skupna značilnost teh kriterijskih funkcij je, da so vse izražene kot funkcije končnih časov izvajanja posameznih operacij $C_i^\diamond$. Naštejmo nekaj najpomembnejših.

Uteženi izvršni čas (ang. *weighted completion time*) predpostavlja, da niso vse operacije enako pomembne. Namesto tega vsaki od njih pripišemo določeno

[4]Vse oznake, ki se nanašajo na probleme z enim strojem, imajo zaradi lažjega ločevanja dodan prepoznavni znak $^\diamond$.

težo $\omega_i^\diamond \geq 1$ (ali včasih $\omega_i^\diamond > 0$), katere vrednost izraža relativno pomembnost posamezne operacije. Uteženi izvršni čas, ki je kriterij za oceno urnika, izračunamo kot $\sum_{i=1}^{n^\diamond} \omega_i^\diamond C_i^\diamond$.

Največja zapoznelost (ang. *maximum lateness*) se uporablja, ko imamo za vsako operacijo $w_i^\diamond$ predpisan končni čas $d_i^\diamond$, do katerega se mora le-ta izvesti. Taka situacija nastopi, ko proizvajamo izdelke po naročilu s predpisanim dobavnim rokom. Vsaki operaciji predpišemo zapoznelost $L_i^\diamond = C_i^\diamond - d_i^\diamond$, ki je pozitivna, če operacija prekorači predpisani rok, in negativna v primeru, da je izvršena predčasno. Minimizirati želimo največjo kršitev predpisanega končnega časa $\max_{i=1}^{n^\diamond} L_i^\diamond$. Obstaja tudi razširjena verzija problema, kjer minimiziramo uteženo največjo kršitev predpisanega časa $\max_{i=1}^{n^\diamond} \omega_i^\diamond L_i^\diamond$.

Skupna počasnost (ang. *total tardiness*) želi minimizirati skupno zamudo vseh operacij, ne da bi nagradila prezgodnje izvrševanje. Vsaki operaciji $w_i^\diamond$ priredimo počasnost $T_i^\diamond = \max(L_i^\diamond, 0)$, ki je enaka 0, če je operacija izvršena v predpisanem roku, sicer je po vrednosti enaka njeni zapoznelosti. Minimizirati želimo $\sum_{i=1}^{n^\diamond} T_i^\diamond$ ali $\sum_{i=1}^{n^\diamond} \omega_i^\diamond T_i^\diamond$.

Število počasnih opravil (ang. *number of tardy jobs*) predpiše vsaki operaciji $w_i^\diamond$ enoto počasnosti $U_i^\diamond = \begin{cases} 1; & T_i^\diamond > 0 \\ 0; & sicer \end{cases}$. Minimizirati želimo število opravil, ki so počasna $\sum_{i=1}^{n^\diamond} U_i^\diamond$ ali $\sum_{i=1}^{n^\diamond} \omega_i^\diamond U_i^\diamond$.

Skupna zgodnost in počasnost (ang. *total earliness plus tardiness*) kaznuje tako prezgodnje kot tudi prepozno izvrševanje operacij. Problem temelji na predpostavki, da prezgodnje operacije povečujejo stroške skladiščenja. Vsaki operaciji $w_i^\diamond$ poleg počasnosti, ki smo jo že opisali, priredimo tudi zgodnost $E_i^\diamond = \max(0, -L_i^\diamond)$. Minimizirati želimo $\sum_{i=1}^{n^\diamond} (T_i^\diamond + E_i^\diamond)$ ali $\sum_{i=1}^{n^\diamond} (\omega_{\mathrm{T}i}^\diamond T_i^\diamond + \omega_{\mathrm{E}i}^\diamond E_i^\diamond)$.

V povezavi z reševanjem Π_J problema je še posebej pomemben naslednji problem razvrščanja operacij za izvajanje na enem stroju, ki ga označimo s Φ in je definiran takole (Carlier 1982, Grabowski in sod. 1986). Dana je končna množica $n^\diamond$ neodvisnih opravil $w_1^\diamond, \ldots, w_{n^\diamond}^\diamond$. Vsakemu opravilu $w_i^\diamond$ pripišemo čas, ob katerem

je le-to na voljo za izvajanje ($R_i^\diamond \geq 0$), čas izvajanja ($p_i^\diamond > 0$) ter čas, ki je potreben za dokončanje opravila (na drugem neodvisnem stroju) potem, ko se je na obravnavanem stroju že izvršilo ($D_i^\diamond \geq 0$). Minimizirati želimo izvršni čas vseh operacij $\max_i^{n^\diamond}(t_i^\diamond + p_i^\diamond + D_i^\diamond)$, kjer $t_i^\diamond$ pomeni začetni čas izvajanja operacije $w_i^\diamond$ ob upoštevanju pogoja $t_i^\diamond \geq R_i^\diamond$.

Obstaja tudi razširjena verzija gornjega problema, ki je bila namensko definirana kot gradnik za reševanje Π_J problema. Definicija je podobna kot v primeru problema Φ, le da model vsebuje dodatne prednostne omejitve[5] med določenimi pari operacij (Balas in sod. 1995).

Omenimo še *problem trgovskega potnika* (ang. *traveling salesman problem*), ki ne spada striktno med probleme razvrščanja proizvodnih procesov, v tem kontekstu pa se ga vseeno velikokrat omenja. Definiran je takole. Dano je končno število položajev (mest, postajališč, strank, ...) ter razdalje med njimi. Iščemo potovalno zaporedje, po katerem obhodimo vse položaje natančno enkrat in se nato vrnemo v izhodiščni položaj, pri tem pa prepotujemo najmanjšo možno razdaljo.

1.4 Vzorčni primer Π_J instance

Z namenom doseči nazornejšo razpravo v nadaljevanju, bomo na tem mestu definirali vzorčno instanco Π_J problema (tabela 1.1), ki nam bo omogočila vpogled v problematiko s pomočjo konkretnega primera. Instanca vsebuje $n_{\text{tot}} = 12$ operacij, od katerih vsaka pripada enemu od $n = 4$ opravil ter mora biti izvedena na enemu od $m = 3$ strojev. Za vsako operacijo je podan ustrezen stroj v stolpcu z oznako $\mathcal{M}$ ter čas izvajanja z oznako p. Tehnološke omejitve zahtevajo, da izvajanje operacij znotraj posameznega opravila sledi zaporedju $a \rightarrow b \rightarrow c$.

Potrebovali bomo tudi začetni urnik, po katerem naj se operacije izvedejo, zato si ga izberimo (tabela 1.2).

[5]Namerno nismo napisali, da gre za tehnološke omejitve, ker so to dejansko umetno tvorjene relacije, ki nastanejo pri postopku optimizacije Π_J problema s premikanjem ozkega grla (poglavje 4.5.4).

	operacije					
	a		b		c	
opravilo	$\mathcal{M}$	p	$\mathcal{M}$	p	$\mathcal{M}$	p
$\mathcal{J}_1$	2	4	1	7	3	3
$\mathcal{J}_2$	1	3	2	2	3	4
$\mathcal{J}_3$	2	2	1	4	3	3
$\mathcal{J}_4$	3	3	1	4	2	2

Tabela 1.1: Definicija vzorčne instance Π_{J} problema velikosti 4×3.

	zaporedje in začetni časi izvajanja							
stroj	o_1	t_{o_1}	o_2	t_{o_2}	o_3	t_{o_3}	o_4	t_{o_4}
$\mathcal{M}_1$	$4b$	3	$2a$	8	$1b$	12	$3b$	20
$\mathcal{M}_2$	$1a$	5	$2b$	11	$4c$	13	$3a$	17
$\mathcal{M}_3$	$4a$	0	$2c$	13	$3c$	24	$1c$	28

Tabela 1.2: Primer urnika za izvedbo vzorčne Π_{J} instance.

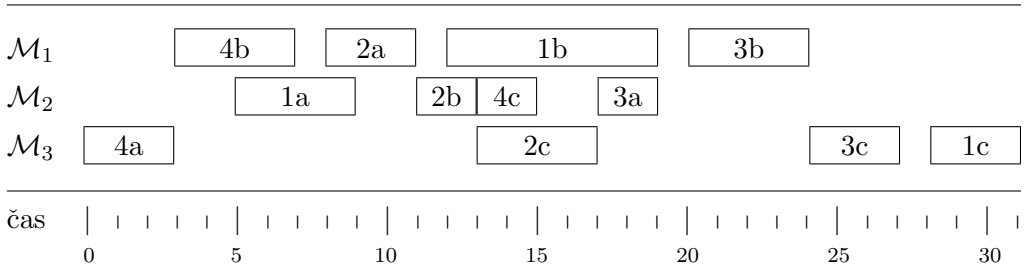
Gornja tabela predpisuje začetno zaporedje izvajanja operacij po strojih; poleg tega za vsako operacijo podaja začetni čas izvršitve, kar je nujno potrebno za nedvoumnost urnika. Kot primer si oglejmo dogajanje na stroju $\mathcal{M}_1$, kjer se najprej izvrši operacija $4b$: z izvrševanjem prične ob času 3, konča pa ob času 7, ker traja njeno izvajanje 4 časovne enote (tabela 1.1). Naslednja operacija je $2a$, ki se prične izvajati ob času 8, konča pa se ob času 11. Na enak način je predpisano tudi izvajanje ostalih operacij.

1.5 Gantt diagram

Gantt diagram (ang. *Gantt chart*) (Clark 1922) nudi nazoren način prikazovanja urnikov. Razvit je bil leta 1918 kot pomoč pri optimizaciji ameriške vojaške proizvodnje, vendar se je njegova uporaba hitro razširila na celotno področje razvrščanja tehnoloških procesov.

Ideja Gantt diagrama je preprosta. Abscisna os predstavlja časovni interval izvajanja urnika, na ordinatni osi pa so predstavljeni stroji (splošneje tehnološki resorji). Časovni intervali, v katerih je posamezni stroj zaseden, so prikazani z zapolnitvijo ustreznega območja na diagramu – najpogosteje z daljico ali s pravokotnikom. Označimo lahko, katera operacija se izvaja ter kateremu opravilu

le-ta pripada. Primer Gantt diagrama na sliki 1.1 prikazuje izvajanje urnika vzorčne instance, ki smo ga podali v tabeli 1.2.



Slika 1.1: Prikaz urnika vzorčne Π_{J} instance s pomočjo Gantt diagrama.

Z diagrama se jasno vidi, kdaj in s katero operacijo je posamezni stroj zaseden. Prikazani urnik je izvedljiv, saj se časovni intervali izvajanja opravil na posameznih strojih ne prekrivajo (upoštevane so zmogljivostne omejitve), poleg tega se operacije izvajajo v pravilnem tehnološkem zaporedju (upoštevane so tehnološke omejitve). Celotni čas, ki ga porabimo za izvedbo vseh opravil, če se držimo predlaganega urnika, je 31 časovnih enot; z drugačnim urnikom bi lahko potrebovali več ali manj časa.

Z diagrama na sliki 1.1 lahko hitro ugotovimo, kateri deli urnika so nespretno določeni; čas izvajanja bi se zanesljivo skrajšal, če bi operacijo $1c$ na stroju $\mathcal{M}_3$ izvedli pred operacijo $3c$. V splošnem postopek optimizacije ni tako preprost, saj so instance v praksi večje. Poleg tega je prikazani urnik namerno slabe kvalitete. Pri bližje-optimalnemu urniku bi potrebovali precej več napora (računalniške moči), da bi odkrili boljši razpored izvajanja operacij.

Do šestdesetih let dvajsetega stoletja je Gantt diagram predstavljal pomembno orodje za optimizacijo urnikov, danes pa je to nalogo prevzel neusmerjeni graf, ki je direktno integriran v področje diskretne matematike in je primernejši za opis računalniških algoritmov (Błażewicz in sod. 1996). Modeliranju Π_{J} instanc s pomočjo neusmerjenega grafa posvečamo celotno poglavje 3.

1.6 Regularne kriterijske funkcije

V podpoglavju 1.3 smo spoznali nekaj kriterijskih funkcij, ki se najpogosteje uporabljajo pri ocenjevanju urnikov. Nekatere izmed njih imajo posebno lastnost, ki ji pravimo *regularnost* (ang. *regularity*). Definiramo jo na naslednji način. Dan imamo urnik izvajanja operacij, kateremu pripada določena vrednost kriterijske funkcije K_1. Sedaj v tem urniku izvedemo spremembo, po kateri se katerakoli operacija ali podmnožica operacij (lahko tudi vse) prične izvajati ob zgodnejšem času kot v prvotnem urniku (pri tem nobene operacije ne zakasnimo). Spremenjenemu urniku pripada vrednost kriterijske funkcije K_2. Za regularne kriterijske funkcije velja, da je $K_2 \leq K_1$ pri kakršnikoli spremembi opisanega tipa.

Od opisanih kriterijskih funkcij samo "skupna zgodnost in počasnost" ne spada v to kategorijo, saj lahko s prezgodnjim izvajanjem določene operacije vrednost kriterijske funkcije poslabšamo. Za našo obravnavo je pomembno, da celotni izvršni čas $C_{\max}$ v primeru Π_{J} problema spada v obravnavano kategorijo.

2. Prostor rešitev Π_J problema

S pojmom prostor rešitev Π_J problema označujemo množico izvedljivih urnikov, ki pripadajo določeni Π_J instanci. Ker moramo v tej množici poiskati optimalni urnik (v primeru eksaktnega reševanja) ali čim bližjega optimalnemu (v primeru hevrističnega reševanja), je uspeh, s katerim bomo nalogo opravili, odvisen od karakteristik prostora rešitev. Najvažnejši podatek je zagotovo število urnikov, ki sestavljajo prostor rešitev. V primeru, da je le-teh sprejemljivo malo, lahko pregledamo in ovrednotimo vse vsebovane urnike ter izberemo najboljšega. Če pa je prostor rešitev prevelik in tako početje ni smotrno, obstaja velika verjetnost, da se bomo morali optimalnemu reševanju že v osnovi odpovedati.

Za Π_J problem s kriterijsko funkcijo celotni izvršni čas $C_{\max}$ lahko s preprostim sklepanjem ugotovimo, da je urnikov, ki pripadajo katerikoli instanci neskončno mnogo, ker ni nobene omejitve, s katero ne bi mogli izvajanja vsake operacije zakasniti poljubno dolgo. Situacija vseeno ni brezupna, saj smo predhodno ugotovili, da kriterijska funkcija $C_{\max}$ spada med regularne kriterijske funkcije. To pomeni, da je vsaka zakasnitev operacij nepotrebna, ker nam vrednost kriterijske funkcije lahko kvečjemu poslabša.

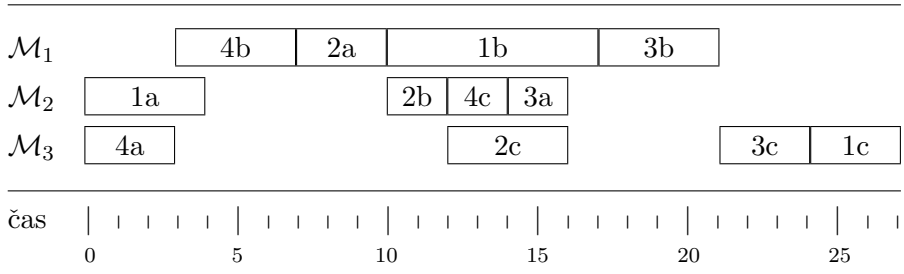
2.1 Pol-aktivni urniki

Glede na predhodno ugotovitev se pri iskanju optimalnih ali blizu-optimalnih rešitev omejimo na podmnožico *pol-aktivnih* (ang. *semi-active*) urnikov, ki jo definiramo na naslednji način (Baker 1974): določen urnik je pol-aktiven tedaj in samo tedaj, če nobene operacije ne moremo izvesti ob zgodnejšem času, ne da bi spremenili zaporedje izvajanja operacij ali kršili tehnoloških in/ali kapacitivnostnih omejitev. Vsak pol-aktivni urnik je povsem določen z zaporedji, po katerih se izvajajo operacije na strojih, zato pri specifikaciji urnika ni potrebno navajati začetnih časov izvajanja, ker so v podanih zaporedjih vsebovani implicitno.

Poljuben začetni urnik lahko spremenimo v le-temu pripadajočega pol-aktivnega na naslednji način: vsako operacijo, katere začetni čas izvajanja lahko zmanjšamo, ne da bi pri tem kakorkoli spremenili zaporedje izvajanja ostalih operacij in pri tem kršili tehnološke ter kapacitivnostne omejitve, pričnemo izvajati ob najzgodnejšem možnem času ter to ponavljamo, dokler take operacije po urniku obstajajo. Poljubnemu začetnemu urniku pripada točno določen pol-aktivni urnik neodvisno od zaporedja, po katerem smo začetne čase operacij spreminjali.

Kot primer si ponovno oglejmo urnik na sliki 1.1, za katerega se lahko hitro prepričamo, da ne spada v množico pol-aktivnih urnikov, saj v njem obstajajo operacije, ki jih lahko pričnemo izvajati ob zgodnejšem času, kot določa prvotni urnik. Operacija $1a$ na stroju $\mathcal{M}_2$ bi se lahko pričela izvajati ob času 0, ker nima (in ji ni potrebno čakati) niti tehnoloških predhodnic niti predhodnic na stroju. Podobno ugotovimo, da se operacija $2a$ na stroju $\mathcal{M}_1$ lahko prične izvajati eno časovno enoto prej, kot je določeno. S tem se odpre možnost zgodnejšega izvajanja operacije $2b$ na stroju $\mathcal{M}_2$, ki je tehnološki naslednik $2a$. Na podoben način lahko pohitrimo še operacije $2c$, $4c$, $1b$, $3a$, $3b$ (in posledično $3c$) ter $1c$.

Rezultirajoči pol-aktivni urnik je prikazan na sliki 2.1. Vidimo, da smo s pohitritvijo operacij pridobili 4 časovne enote, saj sedaj vsa opravila izvedemo v 27 časovnih enotah.



Slika 2.1: Pol-aktivni urnik, ki pripada začetnemu urniku.

Z omejitvijo optimizacijskih algoritmov na delovanje nad pol-aktivno množico urnikov smo problem iskanja optimalnega ali blizu-optimalnega urnika prevedli na iskanje ustreznega zaporedja izvajanja operacij, zato pravimo, da Π_J problem spada med sekvenčne (ang. *sequential*) optimizacijske probleme.

Urnikov, ki so rezultat optimizacije, torej ne podajamo tako, kot je prikazano v tabeli 1.2, ampak navajamo samo zaporedje operacij po strojih, kot prikazuje tabela 2.1.

stroj	zaporedje operacij			
$\mathcal{M}_1$	$4b$	$2a$	$1b$	$3b$
$\mathcal{M}_2$	$1a$	$2b$	$4c$	$3a$
$\mathcal{M}_3$	$4a$	$2c$	$3c$	$1c$

Tabela 2.1: Podajanje urnika z zaporedjem izvajanja operacij.

Ključno vprašanje je, koliko pol-aktivnih urnikov pripada določeni instanci. V primeru pravokotnega problema dimenzij $n \times m$ sklepamo takole. Na vsakem stroju se izvede natančno n operacij, ki jih lahko izvršimo v $n!$ različnih zaporedjih. Ker to velja za vsak stroj, je zgornja meja možnega števila pol-aktivnih urnikov podana kot $(n!)^m$. Izračun le-te je prikazan za nekaj velikosti instanc v tabeli 2.2.

problem $(n \times m)$	št. rešitev $(n!)^m$	problem $(n \times m)$	št. rešitev $(n!)^m$
1×1	1	10×10	$3,95 \cdot 10^{65}$
2×2	4	12×12	$1,46 \cdot 10^{104}$
3×3	216	15×10	$1,46 \cdot 10^{121}$
4×4	$3,32 \cdot 10^{5}$	15×15	$5,59 \cdot 10^{181}$
5×5	$2,48 \cdot 10^{10}$	20×10	$7,26 \cdot 10^{183}$
6×6	$1,39 \cdot 10^{17}$	20×15	$6,19 \cdot 10^{275}$

Tabela 2.2: Zgornja meja števila pol-aktivnih urnikov.

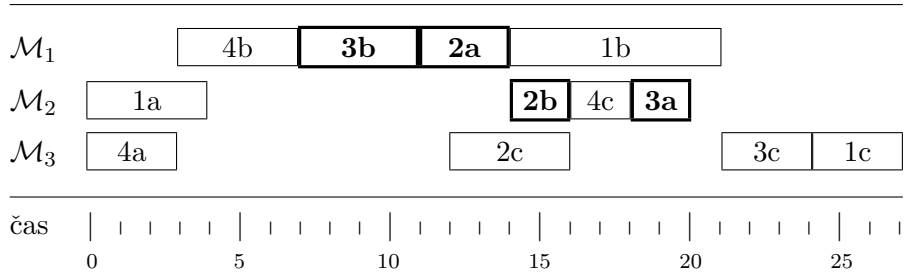
Opazimo, da je naraščanje števila možnih pol-aktivnih urnikov grozljivo hitro, kar je glavni razlog za izredno težavnost Π_{J} problema; zanj pravimo, da kombinatorično eksplodira.

Dodatno težavo predstavlja dejstvo, da poljubno zaporedje izvajanja operacij ne predstavlja *izvedljivega urnika* (ang. *feasible schedule*). Oglejmo si primer, ki je podan v tabeli 2.3.

stroj	zaporedje operacij			
$\mathcal{M}_1$	$4b$	$\mathbf{3b}$	$2a$	$1b$
$\mathcal{M}_2$	$1a$	$2b$	$4c$	$3a$
$\mathcal{M}_3$	$4a$	$2c$	$3c$	$1c$

Tabela 2.3: Primer neizvedljivega urnika.

Edina razlika v primerjavi z urnikom v tabeli 2.1 je, da smo operacijo $3b$, ki se je do sedaj izvajala zadnja na stroju $\mathcal{M}_1$, določili za izvajanje neposredno za operacijo $4b$, ki je na stroju prva. Slika 2.2 nas bo prepričala, da takega urnika ni mogoče izvršiti, ne da bi kršili tehnološke omejitve.



Slika 2.2: Primer neizvedljivega urnika.

Operacija $3b$ se časovno izvaja pred operacijo $3a$ na stroju $\mathcal{M}_2$, torej sklepamo, da jo moramo zakasniti, če želimo zadovoljiti tehnološke omejitve. Izkaže se, da s tem zakasnimo tudi operacijo $2a$, kar povzroči zakasnitev operacije $2b$ na stroju $\mathcal{M}_2$, s tem pa kasni tudi operacija $3a$; operacijo $3b$ lahko zakasnimo kolikor hočemo, operacija $3a$ se bo vedno izvedla pozneje, zato je urnik s takim zaporedjem izvajanja neizvedljiv.

Neizvedljivi urniki predstavljajo problem, ker je preverjanje izvedljivosti zamudno. Postopki lokalnega iskanja dosegajo blizu-optimalne rešitve tako, da izvajajo lokalne zamenjave zaporedja izvajanja operacij, na podoben način, kot smo iz urnika v tabeli 2.1 dobili urnik v tabeli 2.3. Dejstvo, da lahko z zamenjavo povzročimo neizvedljivost urnika, predstavlja resno oviro pri snovanju učinkovitih algoritmov lokalnega iskanja za reševanje Π_{J} problema in ostalih problemov, kjer so prisotne tehnološke omejitve (Koren 1985, Šuhel in sod. 1989, Kumar 1983).

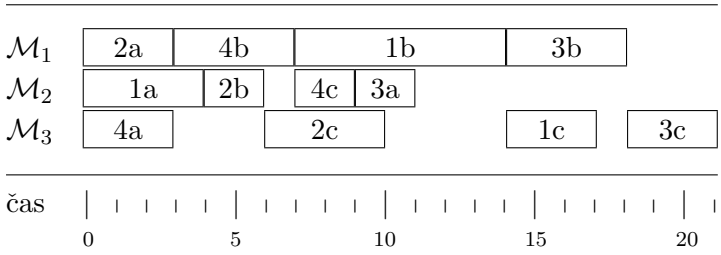
Do danes ni znan postopek, s katerim bi na enostaven način določili natančno število izvedljivih urnikov, saj le-to ni odvisno samo od dimenzionalnosti instance, temveč tudi od predpisa, po katerem se operacije posameznih opravil izvajajo na strojih. S pomočjo do sedaj izvedenih empiričnih poskusov na instancah 10×10 je ocenjeno, da je velikostni red neizvedljivih urnikov 15% (Mattfeld 1996).

2.2 Aktivni urniki

Pri iskanju optimalne rešitve se lahko še bolj omejimo, in sicer na množico *aktivnih* (ang. *active*) urnikov, ki so definirani na naslednji način (Giffler in Thompson 1960): urnik je aktiven tedaj in samo tedaj, ko nobene operacije ne moremo izvesti ob zgodnejšem času, ne da bi s tem zakasnili izvajanja katerekoli druge operacije, ali kršili tehnoloških in/ali kapacitivnostnih omejitev. Vsak aktivni urnik je hkrati tudi pol-aktiven, obratno pa ne drži.

Kot primer si zopet oglejmo sliko 2.1. Brez težav se lahko prepričamo, da prikazani urnik ni aktiven. Operacijo $1c$ na stroju $\mathcal{M}_3$ lahko izvršimo pred operacijo $3c$, ne da bi slednjo s tem zakasnili. Podobno ugotovimo za operacijo $2a$, $4c$ in $3a$.

Poskusimo torej spremeniti vrstni red izvajanja operacij. Najprej določimo, da se operacija $2a$ izvede pred operacijo $4b$; s tem se pohitri tudi izvajanje operacij $2b$, $4c$, $3a$ in $2c$. Nato premaknimo operacijo $1c$ pred $3c$. Rezultat prikazuje slika 2.3.

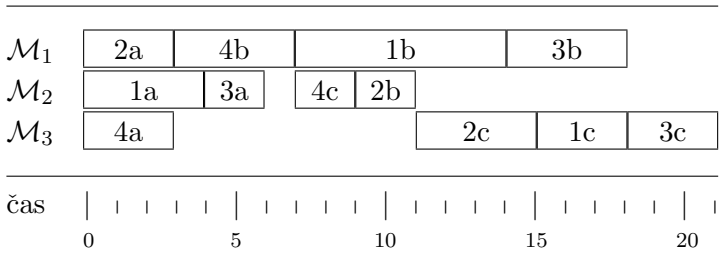


Slika 2.3: Aktivni urnik, ki ga izvedemo iz pol-aktivnega na sliki 2.1.

Giffler in Thompson (1960) sta dokazala, da se vsaj en optimalen urnik nahaja v množici aktivnih urnikov. Število aktivnih urnikov je težje določljivo kot v primeru pol-aktivnih urnikov, saj je še dodatno odvisno od predpisanih časov izvajanja operacij. Vemo le, da je aktivnih urnikov bistveno manj kot pol-aktivnih.

Lahko bi sklepali, da je primerneje izvajati optimizacijo v prostoru aktivnih urnikov, ki jih je manj kot pol-aktivnih, saj manjši prostor rešitev pomeni večjo verjetnost, da bomo našli optimalno rešitev. Vendar sklep ne drži, ker ima striktno vztrajanje pri aktivnih urnikih več slabosti kot prednosti.

Prva težava nastopi zaradi obstoja večjega števila aktivnih urnikov, ki pripadajo določenemu zaporedju izvajanja operacij oziroma pol-aktivnemu urniku. Če ponovno poskusimo spremeniti urnik na sliki 2.1 v aktivnega in tokrat najprej premaknemo operacijo $3a$ pred $2b$, nato $2a$ pred $4b$ in nazadnje $1c$ pred $3c$, dobimo rezultat na sliki 2.4.



Slika 2.4: Primer drugega aktivnega urnika.

V tem primeru imata oba aktivna urnika enak izvršni čas, ki znaša 21 enot, kar pa je zgolj naključje. Ugotovili smo torej, da v splošnem zaporedje operacij ne določa aktivnega urnika nedvoumno. Do danes ne poznamo postopka, s katerim bi lahko določili, kateri aktivni urnik je boljši od ostalih. Izvršni čas ne pove vsega, saj je velikokrat lažje pretvoriti urnik z večjo vrednostjo le-tega v optimalnega.

Druga težava pri uporabi aktivnih urnikov je, da majhna lokalna sprememba zaporedja operacij lahko povzroči večje število sprememb na različnih delih urnika, kar postopek optimizacije težko predvidi; lahko se zgodi, da te spremembe kratkovidno zmanjšajo izvršni čas urnika, s tem pa se ujamejo v lokalni minimum, iz katerega je težko pobegniti in nadaljevati optimizacijo k bližje-optimalnim rešitvam.

Tretja težava aktivnih urnikov je potratnost časa, ki ga moramo pri vsaki lokalni spremembi porabiti za doseganje aktivnosti. Aktivnost namreč ni potreben pogoj za optimalnost urnika, saj se s pomočjo slike 2.4 ni težko prepričati, da

lahko na primer operacijo $4c$ izvedemo za operacijo $2b$ in njeno izvajanje še dodatno zakasnimo za 8 časovnih enot, pa bo rezutirajoči urnik še vedno potreboval 21 časovnih enot za izvrševanje. Preverjati aktivnost te operacije je torej čista izguba časa. V optimalnem urniku je po aktivnem načelu razvrščena samo majhna množica tako imenovanih kritičnih operacij, ki jih bomo definirali v poglavju 3.6.

Zaradi opisanih slabosti vsi učinkoviti postopki lokalnega iskanja ostajajo v večjem prostoru pol-aktivnih urnikov in pri aktivnosti ne vztrajajo.

2.3 Brez-čakalni urniki

Obstaja še manjša množica tako imenovanih *brez-čakalnih* (ang. *non-delay*) urnikov, ki so definirani kot aktivni urniki, pri katerih prost stroj vedno prične izvajati operacijo takoj, ko je ena od njih na voljo (ko je dokončana njena tehnološka predhodnica).

Urnik na sliki 2.4 torej ni brez-čakalni, saj stroj $\mathcal{M}_2$ v času med časovnima enotama 6 in 7 čaka na izvrševanje operacije $4c$, medtem pa bi se $2b$ že lahko izvrševala.

Brez-čakalnih urnikov je dosti manj kot aktivnih, vendar je ta množica za izvajanje optimizacije manj privlačna. Dokazano je (Pinedo 1995), da se v njej optimalni urnik ne nahaja vedno. Poleg tega so zanjo značilne razne anomalije, kot so situacije, kjer večanje zmogljivosti strojev poveča izvršni čas urnika.

V nadaljevanju obravnavamo postopke optimizacije, ki delujejo nad množico pol-aktivnih urnikov. Ko ne bo eksplicitno določeno drugače, bomo za vsak urnik smatrali, da spada v to množico.

2.4 Teorija kompleksnosti

Teorija kompleksnosti (Garey in Johnson 1979) je veda, v okviru katere se razvijajo postopki za sistematično ugotavljanje kompleksnosti matematično ali računalniško opisanih problemov in algoritmov. Pojem kompleksnost algoritma označuje časovno in pomnilniško zahtevnost (kar označujemo s skupnim poj-

mom računalniška moč), ki ju moramo zagotoviti, da lahko algoritem izvedemo od začetka do konca. Kompleksnost določenega problema označuje potrebno računalniško moč za izvedbo najmanj kompleksnega algoritma, s katerim lahko problem rešimo.

Potreba po teoriji kompleksnosti se je pojavila, ko je postalo očitno, da so nekateri problemi bistveno težje rešljivi od drugih. Na primer za večino omenjenih problemov razvrščanja opravil na enem stroju (poglavje 1.3) poznamo algoritem, katerega potrebna računalniška moč ne narašča hitreje kot nek polinom nizke stopnje (ponavadi prve ali druge). Po drugi strani za problem Φ takega algoritma ne poznamo; pri vsakem znanem postopku reševanja narašča število potrebnih korakov, ki jih moramo izvršiti, hitreje od kateregakoli polinoma poljubne stopnje, ko število opravil raste preko vseh meja. Sklepamo, da je problem Φ težji od problemov, katerih potrebna računalniška moč narašča polinomsko v odvisnosti od velikosti instance.

Kompleksnost algoritmov označujemo s tako imenovano notacijo *veliki* O, s čimer zaobjamemo samo najhitreje rastoči del funkcije, ki opisuje kompleksnost algoritma. Na primer kompleksnost algoritma, katerega število korakov izvajanja narašča v odvisnosti od števila opravil v instanci n kot $17n^2 + 34n + 32$, ima kvadratno kompleksnost $O(n^2)$. Da je zanemaritev nižjih členov in vodilnega koeficienta upravičena, se lahko prepričamo s pomočjo naslednjega zgleda, ki ga podaja Schneier (1996).

Predpostavimo, da imamo na voljo računalnik, ki je zmožen izvršiti milijon računalniških operacij na sekundo. Z njegovo pomočjo moramo rešiti problem, katerega velikost instance n je milijon (na primer razporediti želimo milijon tehnoloških operacij, ki jih moramo izvršiti na enem stroju). V tabeli 2.4 so podana števila korakov in časi izvajanja nekaterih algoritmov z različnimi kompleksnostimi.

Vidimo, da algoritmi, ki nimajo polinomske odvisnosti števila operacij od velikosti instance, niso praktični za uporabo (razen nekaterih izjem; za problem Φ obstaja algoritem reševanja, ki je praktičen do okvirno 2000 operacij). Največkrat označujemo s pojmom učinkovit (ang. *tractable*) tisti algoritem, katerega komple-

kompleksnost		število operacij	čas izvajanja
konstantna	$O(1)$	1	10^{-6} s
linearna	$O(n)$	10^6	1 s
kvadratna	$O(n^2)$	10^{12}	11,6 dni
kubična	$O(n^3)$	10^{18}	32.000 let
eksponentna	$O(2^n)$	$10^{301.030}$	$10^{301.006}$ × starost vesolja

Tabela 2.4: Odvisnost časa izvajanja algoritmov od njihove kompleksnosti.

ksnost narašča polinomsko v odvisnosti od velikosti instance. To je bilo večkrat kritizirano, saj pri večjih stopnjah polinoma čas izvajanja ravno tako narašča nesprejemljivo hitro. Kljub temu bomo omenjeni pojem v tem delu uporabljali na ustaljeni način.

Če bi bila velikost prostora rešitev edino merilo za kompleksnost diskretnih kombinatoričnih optimizacijskih problemov, bi lahko reševanje s problemom Π_J že v osnovi opustili. Število pol-aktivnih urnikov, katerih kardinalnost prostora rešitev ima odvisnost $(n!)^m$, narašča celo hitreje od funkcije 2^n. V resnici kardinalnost prostora rešitev določa samo zgornjo mejo kompleksnosti problema, saj sama zase ne izključuje obstoja postopka, s katerim lahko sestavimo ali poiščemo optimalno rešitev v polinomskem času (mnogo problemov razvrščanja operacij na enem stroju ima kardinalnost prostora rešitev $n^\diamond!$, vendar zanje poznamo učinkovite postopke reševanja).

Teorija kompleksnosti klasificira algoritme v razrede glede na njihovo kompleksnost. Najpreprostejši so algoritmi, katerih časovna odvisnost od velikosti instance je polinomska; ti algoritmi spadajo v razred *polinomskih problemov*, P (ang. *polynomial*)[1]. Naslednji je razred *nedeterminističnih polinomskih problemov*, NP (ang. *non-deterministic polynomial*), ki vsebuje tudi razred P. V njega spadajo vsi problemi, katerih pravilnost dane rešitve lahko preverimo v polinomskem času[2]; to je, do rešitve se ne moremo nujno dokopati v polinomskem času, lahko le preverimo njeno pravilnost, če smo jo na primer generirali naključno (od tu beseda *nedeterministični* v imenu razreda). Do danes ni dokazano niti, da velja $P = NP$, niti, da je $P \subset NP$. Torej ni nemogoče, da obstajajo postopki, s pomočjo katerih lahko rešujemo NP probleme v polinomskem času.

[1]Natančnejša definicija zahteva vpeljavo pojma Turingov stroj, kar presega okvir tega dela.
[2]Natančnejša definicija zahteva vpeljavo nedeterminističnega Turingovega stroja.

Posebno skupino *NP* problemov tvorijo *NP-kompletni* (ang. *NP-complete*) problemi. Zanje je dokazano dvoje: (1) vsakega od njih je možno s polinomskim algoritmom reducirati na katerikoli drug problem v tej množici, (2) če obstaja polinomski algoritem vsaj za enega od njih, obstaja tudi za vse ostale *NP*-kompletne (in s tem vse *NP*) probleme.

Pomembnost teorije kompleksnosti za našo obravnavo je naslednja. Dokazano je, da spada Π_J problem v množico *NP*-kompletnih problemov (Lenstra in Rinnooy Kan 1979), ki jih je zelo veliko (samo v Garey in Johnson (1979) jih je navedenih več kot 300). Vsakega od njih je skušalo rešiti mnogo znanstvenikov, vendar nikomur ni uspelo odkriti polinomskega algoritma za njihovo reševanje. Če bi to uspelo samo enemu avtorju samo za en problem, bi to avtomatično pomenilo, da podobni algoritmi obstajajo za vse *NP*-kompletne probleme in s tem tudi za Π_J.

Zaključimo, da se moramo odpovedati reševanju Π_J problema s postopki, ki nam jamčijo optimalnost generiranih rešitev (Balas in sod. 1995). Namesto tega se je koristneje posvečati izboljšavam aproksimativnih postopkov, ki nam optimalnosti rešitev sicer ne jamčijo, zmožni pa so generirati blizu-optimalne rešitve v sprejemljivem času. To je za proizvodne obrate ključnega pomena, saj ne morejo čakati več desetletij ali življenskih dob galaksije na optimalen urnik, če lahko v nekaj urah dobijo blizu-optimalnega, ki je od optimalnega slabši samo za nekaj procentov.

Izsledki teorije kompleksnosti so tako radikalno spremenili pogled na računalništvo, da se obdobje pred njenim nastankom včasih označuje kot BC (*before complexity* – pred kompleksnostijo) (Parker 1995), poznejše obdobje pa kot AD (*advanced difficulty* – napredna ali ekstremna težavnost) (Jain in Meeran 1999).

3. Neusmerjeni vozliščno-uteženi graf

Za potrebe matematične analize ter za opisovanje algoritmov se Π_{J} instance najpogosteje prikazujejo s pomočjo *neusmerjenega vozliščno-uteženega grafa* (ang. *disjunctive node-weighted graph*) $\mathcal{G}' = \{\mathcal{N}, \mathcal{A} \cup \mathcal{E}\}$, ki sta ga predlagala Roy in Sussmann (1964). Oznake $\mathcal{N}$, $\mathcal{A}$ in $\mathcal{E}$ pomenijo množico *vozlišč* (ang. *nodes*), množico *usmerjenih povezav* (ang. *conjunctive arcs*) in množico *neusmerjenih povezav* (ang. *disjunctive arcs*). Vsaka operacija $w_{i,j}$ je predstavljena s svojim vozliščem, katerega utež[1] je enaka času izvajanja operacije $p_{i,j}$. Poleg vozlišč za vse operacije v instanci vsebuje množica $\mathcal{N}$ dve dodatni vozlišči za fiktivni operaciji *izvor* $\circledcirc$ in *ponor* $\circledast$, katerih utež je enaka nič. Operaciji predstavljata začetek in konec izvajanja urnika. Ker ju želimo obravnavati enako kot resnične operacije, jima priredimo indeksa 0 in $n_{\text{tot}} + 1$, torej $w_0 = \circledcirc$ in $w_{n_{\text{tot}}+1} = \circledast$ (resničnim operacijam smo priredili indekse v poglavju 1.1). Z množicama $\mathcal{A}$ in $\mathcal{E}$ modeliramo tehnološke omejitve med operacijami ter zmogljivostne omejitve strojev.

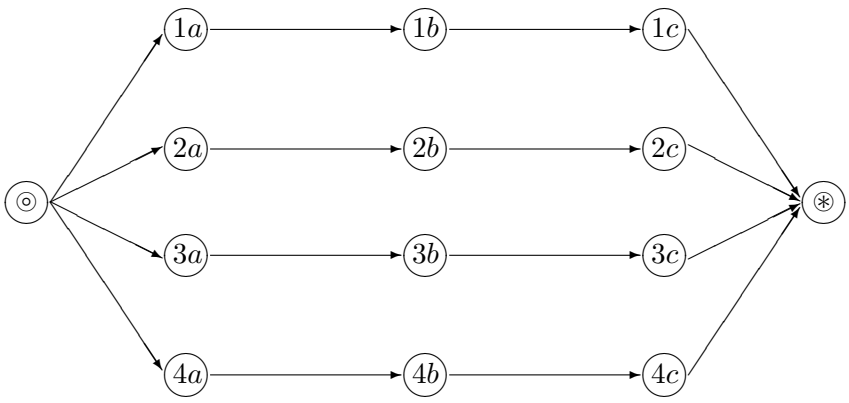
3.1 Modeliranje tehnoloških omejitev med operacijami

Neposredna tehnološka predhodnica (ang. *immediate technological predecessor*) operacije $w_{i,j}$ je $P_{\text{J}}(w_{i,j}) = w_{i,j-1}$, v primeru, da operacija ni prva v opravilu ($j > 1$); za prve operacije vseh opravil velja $P_{\text{J}}(w_{i,1}) = \circledcirc$; $i = 1, \ldots, n$. Podobno definiramo *neposredno tehnološko naslednico* (ang. *immediate technological successor*) operacije $w_{i,j}$ kot $S_{\text{J}}(w_{i,j}) = w_{i,j+1}$, v primeru, da operacija ni zadnja v opravilu $\big(j < n(i)\big)$; za zadnje operacije vseh opravil velja $S_{\text{J}}(w_{i,n(i)}) = \circledast$; $i = 1, \ldots, n$.

Za vsako resnično operacijo $w_{i,j}$ vsebuje množica $\mathcal{A}$ usmerjeno povezavo $P_{\text{J}}(w_{i,j}) \xrightarrow{\mathcal{A}} w_{i,j}$ (simbol $\xrightarrow{\mathcal{A}}$ označuje povezavo v množici $\mathcal{A}$). Nadalje se v tej množici nahajajo še povezave $w_{i,n(i)} \xrightarrow{\mathcal{A}} \circledast$ za vse $i = 1, \ldots, n$. Usmerjena povezava od poljubne operacije w_a do operacije w_b ($a \neq b; 1 \leq a, b \leq n_{\text{tot}}$) po-

[1] Večkrat zasledimo rahlo spremenjeno verzijo grafa, kjer so utežene povezave in ne vozlišča, kar pa semantike v ničemer ne spremeni.

meni, da se mora w_a v celoti izvesti, preden lahko pričnemo z izvajanjem w_b. Če sta w_a in w_b poljubni operaciji, med katerima obstaja *usmerjena pot* (ang. *directed path*), formirana iz povezav v množici $\mathcal{A}$, potem je w_b *tehnološka naslednica* (ang. *technological successor*) operacije w_a, kar je potreben in zadosten pogoj, da je w_a *tehnološka predhodnica* (ang. *technological predecessor*) operacije w_b. Večina operacij ima več tehnoloških predhodnic in naslednic, a vsaka samo po eno neposredno. Množico vseh tehnoloških predhodnic in množico vseh tehnoloških naslednic operacije w_a označimo s $\mathcal{P}_{\mathrm{J}}(w_a)$ in $\mathcal{S}_{\mathrm{J}}(w_a)$. Graf, ki je enak $\mathcal{G}'$, le da vsebuje samo povezave v množici $\mathcal{A}$, označimo z $\mathcal{G}'_{\mathcal{A}} = \{\mathcal{N}, \mathcal{A}\}$. Za primer vzorčne instance je le-ta prikazan na sliki 3.1.



Slika 3.1: Prikaz grafa $\mathcal{G}'_{\mathcal{A}}$, ki ustreza vzorčni instanci.

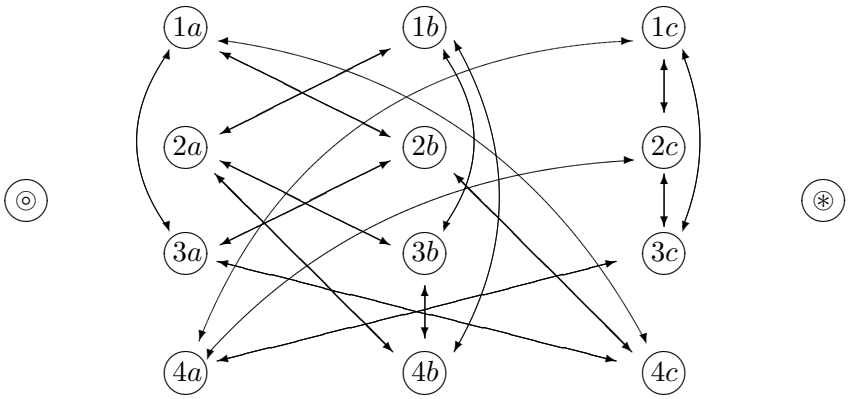
Kot primer gornjih definicij si oglejmo operacijo $2c$. Njena neposredna tehnološka predhodnica $P_{\mathrm{J}}(2c)$ je operacija $2b$, ostali tehnološki predhodnici sta še operaciji $2a$ in $\circledcirc$, zato je množica $\mathcal{P}_{\mathrm{J}}(2c) = \{\circledcirc, 2a, 2b\}$. Neposredna in tudi edina tehnološka naslednica $2c$ je fiktivna operacija $\circledast$, zato veljata zvezi $S_{\mathrm{J}}(2c) = \circledast$ in $\mathcal{S}_{\mathrm{J}}(2c) = \{\circledast\}$. Podobno ugotovimo, da sta neposredna tehnološka predhodnica in neposredna tehnološka naslednica operacije $4b$ operaciji $4a$ in $4c$. Velja tudi $\mathcal{P}_{\mathrm{J}}(4b) = \{\circledcirc, 4a\}$ in $\mathcal{S}_{\mathrm{J}}(4b) = \{4c, \circledast\}$.

Izvor $\circledcirc$ predstavlja začetek urnika, saj je le-to neposredni tehnološki predhodnih vseh prvih operacij v opravilih. Z njim je določena začetna točka časovnega intervala, v katerem se urnik izvaja. Povsem analogno je ponor $\circledast$ neposredni tehnološki naslednik vseh zadnjih operacij v opravilih, zato se lahko izvrši šele potem,

ko so vse ostale operacije opravljene. Na ta način je z njegovim začetnim časom določena končna točka časovnega intervala, v katerem je urnik v celoti opravljen. Z usmerjenimi povezavami smo na kompakten in nazoren način predstavili vse tehnološke omejitve, ki jih obravnavani Π_{J} model vsebuje.

3.2 Modeliranje zmogljivostnih omejitev strojev

Zmogljivostne omejitve strojev podajamo s pomočjo povezav v množici $\mathcal{E}$. Med operacijama w_a in w_b, ki se morata izvesti na istem stroju, vpeljemo neusmerjeno povezavo $w_a \overset{\mathcal{E}}{\longleftrightarrow} w_b$ (simbol $\overset{\mathcal{E}}{\longleftrightarrow}$ označuje neusmerjeno povezavo v množici $\mathcal{E}$). Rezultirajoči graf $\mathcal{G}'_{\mathcal{E}} = \{\mathcal{N}, \mathcal{E}\}$, ki je enak grafu $\mathcal{G}'$ brez povezav v množici $\mathcal{A}$, je prikazan na sliki 3.2.

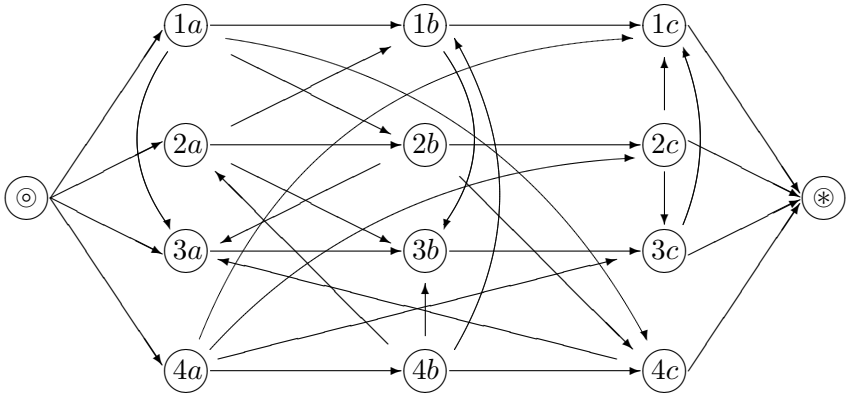


Slika 3.2: Prikaz grafa $\mathcal{G}'_{\mathcal{E}}$ za primer vzorčne instance.

Povezave v množici $\mathcal{E}$ niso usmerjene, ker ob specifikaciji instance ni določeno, v kakšnem zaporedju morajo stroji izvajati operacije. To je znano šele ob določitvi urnika; definirati urnik pomeni izbrati smer vsake od neusmerjenih povezav na tak način, da je rezultirajoči graf $\mathcal{G} = \{\mathcal{N}, \mathcal{A} \cup \mathcal{E}\}$ (ne samo $\mathcal{G}_{\mathcal{E}} = \{\mathcal{N}, \mathcal{E}\}$), dobljen iz grafa $\mathcal{G}'$, *acikličen* (ang. *acyclic*) (Mattfeld 1996). Acikličnost[2] pomeni, da od nobenega vozlišča ni mogoče priti nazaj do istega vozlišča po nobeni usmerjeni poti. Če je to izpolnjeno, pravimo grafu $\mathcal{G}$ *kompletna izbira* (ang. *complete selection*). Prisotnost usmerjene povezave $w_a \xrightarrow{\mathcal{E}} w_b$ (simbol $\xrightarrow{\mathcal{E}}$ označuje usmerjeno

[2]Terminologija v tuji literaturi ni enotna. V Christofides (1975) zasledimo za obravnavani pojem *cikel* izraz *circuit*, Mattfeld (1996) pa uporablja termin *cycle*.

povezavo v množici $\mathcal{E}$) določa, da se operacija w_a izvede (kadarkoli) pred operacijo w_b, zato je operacija w_b *naslednica operacije* w_a *na stroju* (ang. *machine successor*), kar implicira, da je w_a *predhodnica operacije* w_b *na stroju* (ang. *machine predecessor*). Za urnik, ki smo ga podali v tabeli 2.1, je graf $\mathcal{G}$ prikazan na sliki 3.3.



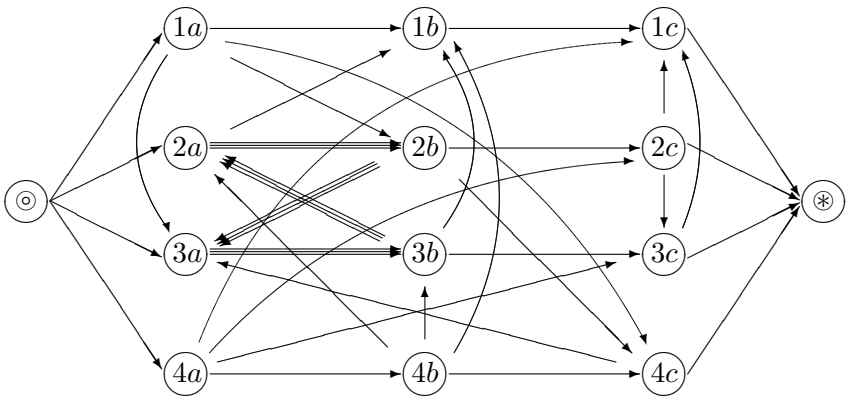
Slika 3.3: Kompletna izbira $\mathcal{G}$.

Za primer si poglejmo dogajanje na stroju $\mathcal{M}_2$, kjer je po urniku določeno naslednje zaporedje izvajanja operacij: $1a$, $2b$, $4c$ in nazadnje $3a$. S pomočjo slike 3.3 ugotovimo naslednje. Operacija $1a$ se mora izvesti prva zaradi prisotnosti povezav $1a \xrightarrow{\mathcal{E}} 2b$, $1a \xrightarrow{\mathcal{E}} 4c$ ter $1a \xrightarrow{\mathcal{E}} 3a$. Podobno se mora $2b$ izvesti pred $4c$ in $3a$ zaradi povezav $2b \xrightarrow{\mathcal{E}} 4c$ in $2b \xrightarrow{\mathcal{E}} 3a$. Operacija $4c$ je določena za izvajanje pred operacijo $3a$ zaradi $4c \xrightarrow{\mathcal{E}} 3a$. Zaporedje izvajanja je tako popolnoma določeno, saj kakršnakoli sprememba le-tega krši vsaj eno od omenjenih usmeritev. Podobno analizo lahko izvedemo na ostalih strojih.

Kompletna izbira grafa $\mathcal{G}$ ima naslednji pomembni lastnosti: (1) od kateregakoli vozlišča je po usmerjenih povezavah možno prispeti do končnega vozlišča $\circledast$, (2) v nobeno vozlišče se po nobeni poti ni možno vrniti, kar je izpolnjeno z že omenjeno zahtevo, da urnik ne sme vsebovati ciklov. Obe lastnosti skupaj zagotavljata izvedljivost urnika. Če prva lastnost ne bi bila izpolnjena, vozlišče $\circledast$ ne bi bilo tehnološki naslednik vseh operacij in bi se lahko pričelo izvajati preden bi bile do konca izvedene ostale operacije, s čimer bi dobili napačno oceno časovnega intervala, ki je potreben za izvedbo urnika. Neizpolnjevanje druge lastnosti bi po-

menilo, da je neko vozlišče predhodnik (in naslednik) samega sebe, zato se ne bi moglo izvesti, dokler ni samo že predhodno izvedeno, kar pomeni večno blokado izvajanja. Prva lastnost je avtomatično izpolnjena zaradi prisotnosti povezav v množici $\mathcal{A}$, zato ji ni potrebno posvečati posebne pozornosti. Nasprotno pa je drugo lastnost v splošnem težko vedno izpolniti, zato moramo biti pri snovanju algoritmov optimizacije previdni, da cikli v grafu $\mathcal{G}$ ne nastanejo.

Vrnimo se k neizvedljivemu urniku, ki smo ga podali v tabeli 2.3 in prikazali na sliki 2.2. Graf, ki temu urniku pripada, je prikazan na sliki 3.4.



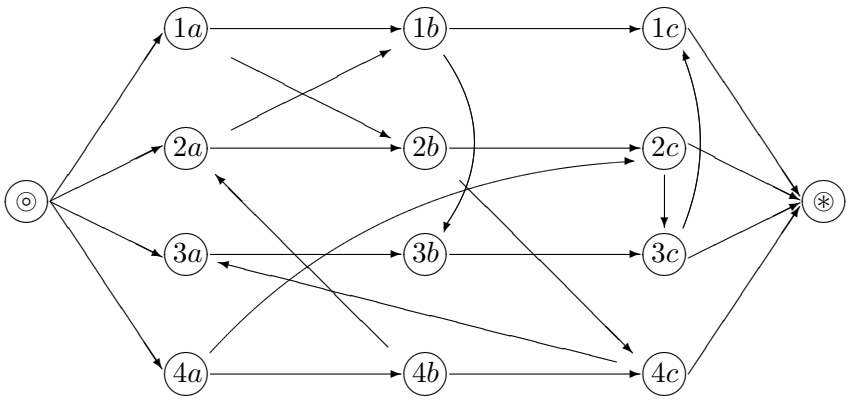
Slika 3.4: Kompletna izbira, ki pripada neizvedljivemu urniku.

S pozornim opazovanjem slike ugotovimo, da v grafu obstaja 4-vozliščni cikel $3b \xrightarrow{\mathcal{E}} 2a \xrightarrow{\mathcal{A}} 2b \xrightarrow{\mathcal{E}} 3a \xrightarrow{\mathcal{A}} 3b$. V splošnem velja, da so vsa vozlišča, ki sestavljajo katerikoli cikel v grafu $\mathcal{G}$, predhodniki in nasledniki samih sebe, zato je njihovo izvajanje večno blokirano. Poudarimo naj, da moramo pri analizi cikličnosti upoštevati celoten graf $\mathcal{G}$, saj sta v našem primeru grafa $\mathcal{G}'_{\mathcal{A}}$ in $\mathcal{G}_{\mathcal{E}}$ vsak zase aciklična.

3.3 Redukcija neusmerjenih povezav

Vrnimo se k izvedljivemu urniku. Ob pogledu na sliko 3.3 hitro opazimo, da je velik del povezav v množici $\mathcal{E}$ odvečnih. V primeru, da obstajata povezavi $w_a \xrightarrow{\mathcal{E}} w_b$ ter $w_b \xrightarrow{\mathcal{E}} w_c$, je povezava $w_a \xrightarrow{\mathcal{E}} w_c$, ki prav tako obstaja, nepotrebna, saj jo lahko izpeljemo. V računalniških algoritmih grafa $\mathcal{G}$ zato navadno ne implementiramo na opisani način, ampak uporabljamo množico $\mathcal{E}_{\text{red}}$, ki je maksimalna možna

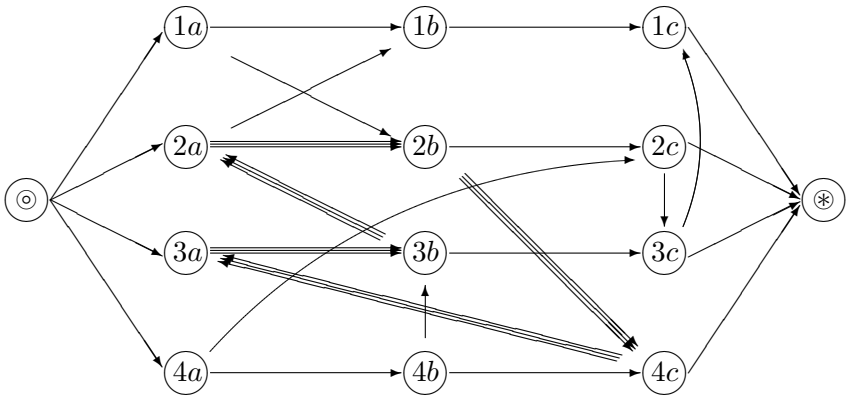
redukcija množice $\mathcal{E}$, kjer je še ohranjena lastnost, da je graf $\mathcal{G}_{\mathcal{E}} = \{\mathcal{N}, \mathcal{E}\}$ *tranzitivno zaprtje* (ang. *transitive closure*) grafa $\mathcal{G}_{\mathcal{E}\text{red}} = \{\mathcal{N}, \mathcal{E}_{\text{red}}\}$ (Christofides 1975). Graf $\mathcal{G}_{\text{red}} = \{\mathcal{N}, \mathcal{A} \cup \mathcal{E}_{\text{red}}\}$ je prikazan na sliki 3.5.



Slika 3.5: Reducirana kompletna izbira $\mathcal{G}_{\text{red}}$.

Graf $\mathcal{G}_{\text{red}}$ je (že na pogled) bolj pregleden od grafa $\mathcal{G}$, oba pa vsebujeta iste informacije, čeprav so sedaj nekatere relacije med operacijami zapisane posredno. Na primer da se operacija $2b$ izvede za operacijo $1a$ je neposredno razvidno iz obeh grafov. Zaporedje operacij $1a$ in $3a$ v primeru grafa $\mathcal{G}_{\text{red}}$ postane očitno šele, ko prepotujemo pot $1a \rightarrow 2b \rightarrow 4c \rightarrow 3a$, medtem ko v grafu $\mathcal{G}$ obstaja med operacijama direktna povezava.

Poglejmo si še sliko 3.6, kjer je prikazan graf $\mathcal{G}_{\text{red}}$, ki pripada neizvedljivemu urniku na sliki 3.4.



Slika 3.6: Reducirana kompletna izbira, ki pripada neizvedljivemu urniku.

V njem obstaja 5-vozliščni cikel $3b \xrightarrow{\mathcal{E}} 2a \xrightarrow{\mathcal{A}} 2b \xrightarrow{\mathcal{E}} 4c \xrightarrow{\mathcal{E}} 3a \xrightarrow{\mathcal{A}} 3b$. Prvotnega 4-vozliščnega cikla ni več, saj med operacijama $2b$ in $3a$ ni več direktne povezave. Ob ponovnem ogledu slike 3.4 ugotovimo, da isti 5-vozliščni cikel obstaja tudi v nereduciranem grafu. Zaključimo, da reducirana kompletna izbira $\mathcal{G}_{\text{red}}$ ohrani lastnost cikličnosti/acikličnosti kompletne izbire $\mathcal{G}$, ni pa nujno, da vsebuje vse cikle, ki so bili prvotno prisotni.

Zopet se vrnimo na obravnavo izvedljivih urnikov. Vidimo, da se kljub redukciji povezav predstavitev urnikov s pomočjo grafa $\mathcal{G}_{\text{red}}$ po dojemljivosti še zdaleč ne more primerjati z Gantt diagramom, zato bomo v nadaljevanju grafe uporabljali pri teoretičnih študijah in razlagah algoritmov, Gantt diagram pa nam bo še naprej služil za vizualno predstavitev urnikov.

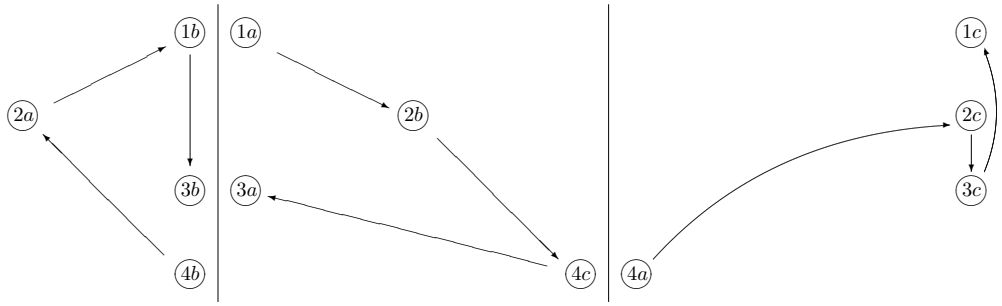
Določimo, koliko povezav vsebujeta obravnavani množici v primeru pravokotne instance dimenzij $n \times m$ (v splošnejšem primeru število povezav ni odvisno samo od dimenzionalnosti instance). Na vsakem stroju se izvaja n operacij, torej je med njimi $\big(n(n-1)\big)/2$ povezav. Vseh strojev je m, iz česar sledi $|\mathcal{E}| = \big(mn(n-1)\big)/2$. Za množico $\mathcal{E}_{\text{red}}$ izpeljemo formulo takole. Hamiltonove poti podgrafov $\mathcal{E}_{\text{red}i}$, ki imajo n vozlišč, vsebujejo $n-1$ povezav. Takih podgrafov je natanko m, zato je $|\mathcal{E}_{\text{red}}| = m(n-1)$.

Za primer instance 10×10 so velikosti naslednje: $|\mathcal{E}| = 450$ in $|\mathcal{E}_{\text{red}}| = 90$. Pri instanci 20×20 pa izračunamo $|\mathcal{E}| = 3800$ in $|\mathcal{E}_{\text{red}}| = 380$. Vidimo, da je prihranek z uporabo množice $\mathcal{E}_{\text{red}}$ namesto $\mathcal{E}$ lahko velik, tako pomnilniško kot tudi časovno, saj se med potekom optimizacije urnik spreminja, kar pomeni nenehno popravljanje orientacij povezav v eni od obeh množic. Kljub temu je uporaba množice $\mathcal{E}$ primernejša za teoretično obravnavo algoritmov, ker z njeno uporabo urnik popravljamo samo s spreminjanjem orientacij povezav. V primeru množice $\mathcal{E}_{\text{red}}$ je potrebno povezave tudi dodajati in odvzemati, kar dogajanje zamegli.

3.4 Določitev urnika s pomočjo reduciranih povezav

Definirajmo še, kaj pomeni določiti urnik s pomočjo množice $\mathcal{E}_{\text{red}} \subseteq \mathcal{E}$. V ta namen razdelimo množico vozlišč $\mathcal{N}$ na $m+1$ podmnožic $\mathcal{N}_0, \ldots, \mathcal{N}_m$, kjer posamezne podmnožice $\mathcal{N}_i$, $i = 1, \ldots, m$, vsebujejo operacije, ki se izvajajo na stroju i, množica $\mathcal{N}_0$ pa vsebuje vozlišči $\circledcirc$ in $\circledast$ (velja $\bigcup_0^m \mathcal{N}_i = \mathcal{N}$ in $\mathcal{N}_i \cap \mathcal{N}_j = \emptyset$; $i \neq j$; $0 \leq i, j \leq m$). Podobno storimo z množico $\mathcal{E}$, ki jo razdelimo na podmnožice $\mathcal{E}_1, \ldots, \mathcal{E}_m$, kjer posamezna podmnožica $\mathcal{E}_i$ vsebuje povezave med operacijami na stroju i (velja $\bigcup_1^m \mathcal{E}_i = \mathcal{E}$ in $\mathcal{E}_i \cap \mathcal{E}_j = \emptyset$; $i \neq j$; $1 \leq i, j \leq m$). Povsem analogno delitev izvedimo tudi nad množico $\mathcal{E}_{\text{red}}$, kjer dobimo množice $\mathcal{E}_{\text{red}1}, \ldots, \mathcal{E}_{\text{red}m}$ ($\bigcup_1^m \mathcal{E}_{\text{red}i} = \mathcal{E}_{\text{red}}$, $\mathcal{E}_{\text{red}i} \cap \mathcal{E}_{\text{red}j} = \emptyset$; $i \neq j$; $1 \leq i, j \leq m$).

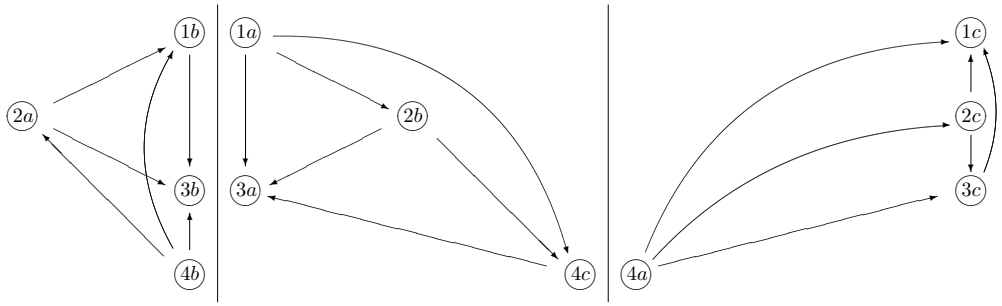
Da bi množica $\mathcal{E}_{\text{red}}$ definirala izvedljiv in nedvoumen urnik, rezultirajoči graf $\mathcal{G}_{\text{red}}$ ne sme vsebovati nobenega cikla, poleg tega mora vsaka podmnožica $\mathcal{E}_{\text{red}i}$ vsebovati povezave, ki tvorijo *Hamiltonovo pot* (ang. *Hamiltonian path*) grafa $\mathcal{G}_{\mathcal{E}i} = \{\mathcal{N}_i, \mathcal{E}_i\}$ (Mattfeld 1996, Christofides 1975). Opozarjamo na nepoenotenost literature pri definiciji pojma Hamiltonova pot. V Christofides (1975) je le-ta definirana kot pot, ki obhodi vsa vozlišča grafa natanko enkrat in se nato vrne v izhodiščno vozlišče. Mattfeldova različica pa povezave za vrnitev v izhodiščno vozlišče ne vsebuje. V našem primeru je uporabna slednja definicija, saj bi drugače vsak graf $\mathcal{G}_{\mathcal{E}i}$ vseboval cikel in bi bila vsa vozlišča nasledniki samih sebe. Hamiltonove izbire vseh podgrafov $\mathcal{G}_{\mathcal{E}i}$ so prikazane na sliki 3.7.



Slika 3.7: Prikaz Hamiltonovih poti podgrafov $\mathcal{G}_{\mathcal{E}1}$, $\mathcal{G}_{\mathcal{E}2}$ in $\mathcal{G}_{\mathcal{E}3}$ od leve proti desni.

Povezava med urnikom in Hamiltonovimi potmi je zelo enostavna. Pot poteka po operacijah v zaporedju, ki ga definira urnik. Kot primer si poglejmo izvajanje operacij na stroju $\mathcal{M}_3$, ki se morajo po urniku v tabeli 2.1 izvesti v zaporedju $4a$,

$2c$, $3c$ ter nazadnje $1c$. Na desnem grafu slike 3.7 opazimo natanko tak potek Hamiltonove poti. Za ilustracijo si na sliki 3.8 oglejmo še prikaz celotnih podgrafov $\mathcal{G}_{\mathcal{E}i}$.



Slika 3.8: Celotni podgrafi $\mathcal{G}_{\mathcal{E}1}$, $\mathcal{G}_{\mathcal{E}2}$ in $\mathcal{G}_{\mathcal{E}3}$ od leve proti desni.

S pomočjo reducirane množice neusmerjenih povezav vpeljimo naslednja pojma. Če obstaja povezava $w_a \xrightarrow{\mathcal{E}_{\text{red}}} w_b$, je operacija w_a *neposredna predhodnica operacija* w_b *na stroju* (ang. *immediate machine predecessor*), kar označimo $w_a = P_{\text{M}}(w_b)$. Relacija implicira, da je w_b *neposredna naslednica operacije* w_a *na stroju* (ang. *immediate machine successor*), kar označimo $w_b = S_{\text{M}}(w_a)$. Definirajmo še naslednji relaciji. Za vse prve operacije na strojih w_i ($|w_i| = m$ in $w_i \xrightarrow{\mathcal{E}} w_z \; \forall \; \mathcal{M}(w_z) = \mathcal{M}(w_i);\; z \neq i;\; z = 1, \ldots, n_{\text{tot}}$) velja $P_{\text{M}}(w_i) = \circledcirc$. Simetrično velja za vse zadnje operacije na strojih w_j ($|w_j| = m$ in $w_z \xrightarrow{\mathcal{E}} w_j \; \forall \; \mathcal{M}(w_z) = \mathcal{M}(w_j);\; z \neq j;\; z = 1, \ldots, n_{\text{tot}}$), da je $S_{\text{M}}(w_j) = \circledast$. S tem zagotovimo, da imajo vse nefiktivne operacije definirani obe neposredni predhodnici in naslednici, kar poenostavi diskusijo in zasnovo algoritmov. Podobno kot pri relacijah znotraj opravil velja, da imajo operacije v splošnem več predhodnic in naslednic na strojih, a samo eno neposredno. Množico vseh predhodnic in množico vseh naslednic operacije w_a na strojih označimo s $\mathcal{P}_{\text{M}}(w_a)$ in $\mathcal{S}_{\text{M}}(w_a)$.

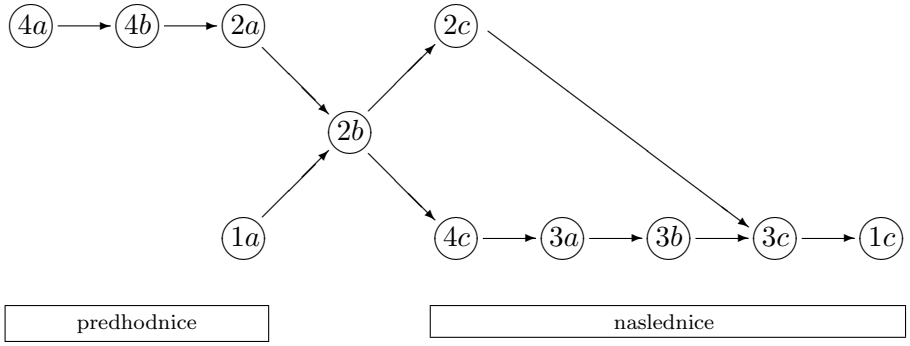
Za zgled pravkar vpeljanih definicij si poglejmo operacijo $2a$ na stroju $\mathcal{M}_1$ (sliki 3.7 in 3.8). Njena neposredna predhodnica na stroju $P_{\text{M}}(2a)$ je operacija $4b$, vse njene predhodnice na stroju pa so $\mathcal{P}_{\text{M}}(2a) = \{\circledcirc, 4b\}$. Neposredna naslednica $2a$ na stroju $S_{\text{M}}(2a)$ je $1b$. Množica vseh naslednic $2a$ na stroju je $\mathcal{S}_{\text{M}}(2a) = \{1b, 3b, \circledast\}$.

Operacija w_b je *naslednica* (ang. *successor*) operacije w_a, če obstaja usmerjena pot v grafu $\mathcal{G}$ ali $\mathcal{G}_{\text{red}}$ od w_a do w_b. V tem primeru je w_a *predhodnica* (ang. *predecessor*) operacije w_b. Množico vseh predhodnic in množico vseh naslednic operacije w_a označimo s $\mathcal{P}(w_a)$ in $\mathcal{S}(w_a)$.

S pomočjo neusmerjenega grafa (sliki 3.3 in 3.5) se lahko prepričamo v naslednje. Operacija $1c$ je naslednik operacije $4b$, ker v grafu $\mathcal{G}$ obstaja pot $4b \xrightarrow{\mathcal{E}} 2a \xrightarrow{\mathcal{A}} 2b \xrightarrow{\mathcal{A}} 2c \xrightarrow{\mathcal{E}} 3c \xrightarrow{\mathcal{E}} 1c$; s tem je $4b$ predhodnik operacije $1c$. Pri operacijah $1a$ in $4b$ nobena od teh relacij ne drži, ker ne obstaja pot niti od $1a$ do $4b$, niti od $4b$ do $1a$.

3.5 Tranzitivnost predhodnosti in naslednosti

Oglejmo si sliko 3.9, kjer so prikazane vse nefiktivne predhodnice in naslednice operacije $2b$.



Slika 3.9: Prikaz vseh predhodnic in naslednic operacije $2b$.

Z njeno pomočjo lahko opazujemo pomembno dejstvo: če je neka operacija w_b naslednica operacije w_a, so vse naslednice w_b tudi naslednice w_a. Operacija $4c$ je naslednica $2b$, ker obstaja med njima povezava v množici $\mathcal{E}$. Nadalje, operacija $3a$ je naslednica operacije $4c$ iz istega razloga. Posledica je, da obstaja v grafu pot med $2b$ in $3a$, kar po definiciji pomeni, da je $3a$ naslednica $2b$. Podoben sklep lahko izvedemo za vse naslednice $2b$.

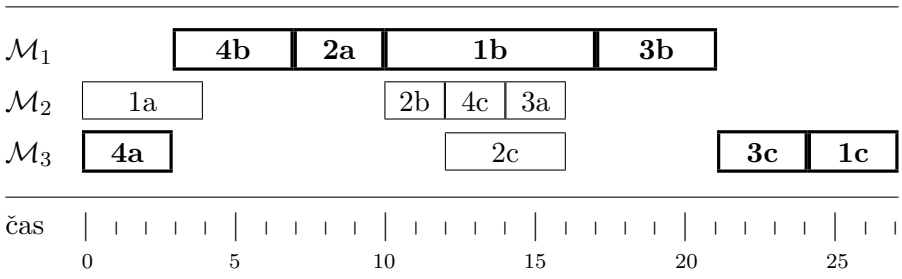
Obstaja tudi simetrična zveza: če je neka operacija w_a predhodnica operacije w_b, so vse predhodnice operacije w_a tudi predhodnice w_b. Podobno kot zgoraj

lahko ugotovimo, da je operacija $2a$ predhodnica $2b$, ker med njima obstaja povezava v množici $\mathcal{A}$. Nadalje je $4b$ predhodnica $2a$ zaradi povezave v množici $\mathcal{E}$. Posledica tega je, da med operacijama $4b$ in $2b$ obstaja usmerjena pot, zato je $4b$ predhodnica $2b$.

3.6 Glave, repi in kritična pot

S pomočjo množice $\mathcal{E}$ (ali $\mathcal{E}_{\text{red}}$) povsem definiramo urnik, po katerem se operacije izvršujejo. To pomeni, da je za vsako operacijo določen časovni interval, v katerem se le-ta izvaja. Ker nismo še ničesar povedali o povezavi med grafom $\mathcal{G}$ (ali $\mathcal{G}_{\text{red}}$) in časovnim prostorom urnika, bomo to storili sedaj.

Dolžina najdaljše utežene poti v $\mathcal{G}$ (ali $\mathcal{G}_{\text{red}}$) od $\circledcirc$ do katerekoli operacije w_i se imenuje *glava* (ang. *head*) operacije $h(w_i)$ in je enaka najmanjšemu času, ob katerem se w_i lahko prične izvajati. Simetrično definiramo *rep* (ang. *tail*) operacije $q(w_i)$ kot dolžino najdaljše utežene poti od w_i do $\circledast$; njegova vrednost predstavlja spodnjo mejo časovnega intervala, ki je potreben, da se urnik izvede v celoti, ko se je operacija w_i že izvršila. V pomoč k razlagi si oglejmo sliko 3.10, kjer je ponovno prikazan Gantt diagram vzorčnega urnika (pomen odebeljenih operacij bomo razložili kasneje).

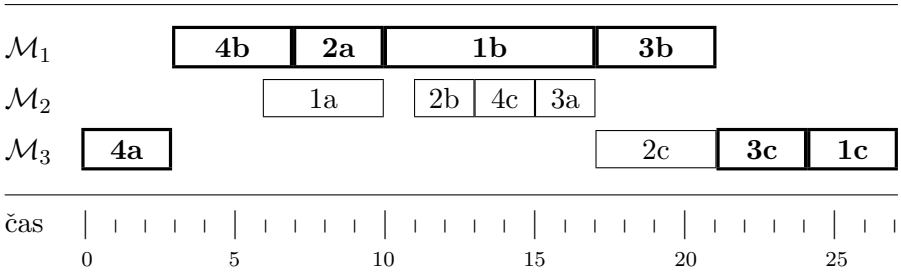


Slika 3.10: Ponoven prikaz urnika vzorčne Π_{J} instance.

S slik 3.10 in 3.5 je razvidno, da je glava operacije $1a$ enaka nič (edina vplivna pot v grafu je $\circledcirc \xrightarrow{\mathcal{A}} 1a$, katere dolžina je 0), zato se $1a$ lahko prične izvajati ob času 0. Podobno lahko ugotovimo za $4a$. Pri operaciji $4b$ ugotovitev ne velja, saj do nje pridemo po poti $\circledcirc \xrightarrow{\mathcal{A}} 4a \xrightarrow{\mathcal{A}} 4b$, ki ima dolžino 3

časovne enote, kolikor traja izvajanje operacije $4a$ (tabela 1.1). Najdaljša pot je torej dolga 3 enote, in kot vidimo na sliki 3.10 je to začetni čas izvajanja te operacije. Podobno lahko ugotovimo za operacijo $2b$, da je njena glava enaka $p_{4a} + p_{4b} + p_{2a} = 3 + 4 + 3 = 10 = h(2b)$, ker je $\circledcirc \xrightarrow{\mathcal{A}} 4a \xrightarrow{\mathcal{A}} 4b \xrightarrow{\mathcal{E}} 2a \xrightarrow{\mathcal{A}} 2b$ najdaljša pot med $\circledcirc$ in $2b$.

Pomen repov operacij si lahko ogledamo na sliki 3.11, kjer je prikazan tako imenovani desno poravnani urnik: izvajanje vsake operacije je v njem zakasnjeno kolikor je še mogoče, ne da bi se pri tem povečal celotni čas izvajanja.



Slika 3.11: Desno poravnani urnik vzorčne Π_{J} instance.

Rep operacije $1c$ na stroju $\mathcal{M}_3$ je enak 0, ker ima edina vplivna pot $1c \xrightarrow{\mathcal{A}} \circledast$ dolžino 0, zato se ta operacija konča v desno poravnanem urniku ob času $C_{\max} - q(1c) = 27 - 0 = 27$. Podobno ugotovimo za operacijo $1a$, da je njen rep enak $p_{1b} + p_{3b} + p_{3c} + p_{1c} = 7 + 4 + 3 + 3 = 17$, ker je najdaljša pot med $1a$ in $\circledast$ enaka $1a \xrightarrow{\mathcal{A}} 1b \xrightarrow{\mathcal{E}} 3b \xrightarrow{\mathcal{A}} 3c \xrightarrow{\mathcal{E}} 1c \xrightarrow{\mathcal{A}} \circledast$. Zato se $1a$ v desno poravnanem urniku konča ob času $C_{\max} - q(1a) = 27 - 17 = 10$. Končni in začetni čas izvajanja operacije w_i v desno poravnanem urniku bomo označili z $\tilde{e}_{w_i} = C_{\max} - q(w_i)$ in $\tilde{t}_{w_i} = \tilde{e}_{w_i} - p_{w_i}$. Tako lahko za operacijo $1a$ zapišemo $\tilde{t}_{1a} = 6$ in $\tilde{e}_{1a} = 10$.

Pomen repov je torej (med drugim) v tem, da z njihovo pomočjo ugotovimo, koliko lahko določeno operacijo zakasnimo. Podatek je bistven za uspešno izvedbo lokalnega iskanja, kjer gre za odločanje o tem, katero operacijo bomo zakasnili, s čimer dobimo možnost izvršiti drugo operacijo, ki je bolj ključna, ob zgodnejšem času.

Operacija w_b je *sosednja* (ang. *adjacent*) operaciji w_a, če se obe izvajata na istem stroju in velja relacija $e_a = t_b$. S slike 3.10 je razvidno, kateri pari operacij

so si sosednji. Na stroju $\mathcal{M}_1$ so to $(4b, 2a)$, $(2a, 1b)$ in $(1b, 3b)$, na stroju $\mathcal{M}_2$ $(2b, 4c)$ in $(4c, 3a)$ ter na stroju $\mathcal{M}_3$ $(3c, 1c)$. Operaciji $1a$ in $2b$ nista sosednji, čeprav se izvajata na istem stroju in si po zaporedju izvajanja sledita, saj se $2b$ ne prične z izvajanjem ob istem času, kot se $1a$ konča.

Ob pogledu na sliki 3.10 in 3.11 opazimo, da se operacije, ki so prikazane odebeljeno ($4a$, $4b$, $2a$, $1b$, $3b$, $3c$ in $1c$), v desno poravnanem urniku niso premaknile. Če za neko operacijo w_C ta ugotovitev drži, potem zanjo velja zveza $h(w_C) + p_{w_C} + q(w_C) = C_{\max}$. Tem operacijam pravimo *kritične operacije* (ang. *critical operations*). Najdaljša uteženа pot med izvorom in ponorom se imenuje *kritična pot* (ang. *critical path*) $C(\mathcal{G})$ grafa $\mathcal{G}$. Njena dolžina je enaka izvršnemu času urnika. Kritičnih poti je lahko v grafu več, za vsako od njih pa velja, da poteka samo skozi kritične operacije. Če želimo zmanjšati izvršni čas urnika, moramo zaporedje izvajanja operacij po strojih spremeniti na tak način, da se dolžina vseh kritičnih poti zmanjša.

Pri uporabi iterativnih optimizacijskih postopkov za reševanje Π_{J} problema se v primeru, da je kritičnih poti v grafu več, ponavadi osredotočimo samo na eno le-teh. Razlog je v tem, da urnik izboljšujemo po korakih. Tako najprej zmanjšamo prvo (v kakršnemkoli smislu) kritično pot, nato se lotimo naslednje in tako naprej, dokler nam to uspeva. Izbrano kritično pot razdelimo na zaporedje r *kritičnih blokov* (ang. *critical blocks*), $B_1, \ldots, B_r$, kjer vsak blok vsebuje maksimalno zaporedje sosednjih kritičnih operacij. Fiktivnih operacij $\circledcirc$ in $\circledast$ se ne uvršča v bloke in včasih tudi ne obravnava kot dela kritične poti. Oznaka B_i^k pomeni k-to kritično operacijo i-tega bloka ($1 \leq i \leq r$; $1 \leq k \leq |B_i|$). Za kritično pot in kritične bloke veljajo naslednje zveze: $B_i \cap B_j = \emptyset$ $(i \neq j; 1 \leq i, j \leq r)$, $\bigcup_{i=1}^{r} B_i = C(\mathcal{G})$ in $\sum_{i=1}^{r} |B_i| = |C(\mathcal{G})|$.

Po pravkar vpeljanih definicijah vsebuje graf urnika s slike 3.10 kritično pot $(\circledcirc \xrightarrow{\mathcal{A}}) \; 4a \xrightarrow{\mathcal{A}} 4b \xrightarrow{\mathcal{E}} 2a \xrightarrow{\mathcal{E}} 1b \xrightarrow{\mathcal{E}} 3b \xrightarrow{\mathcal{A}} 3c \xrightarrow{\mathcal{E}} 1c \; (\xrightarrow{\mathcal{A}} \circledast)$, ki jo razdelimo na $r = 3$ kritičnih blokov: $B_1 = \{4a\}$, $B_2 = \{4b, 2a, 1b, 3b\}$ in $B_3 = \{3c, 1c\}$. Za zgled zapišimo tudi zvezi $B_2^3 = 1b$ in $B_3^1 = 3c$.

Če si ponovno pogledamo primer urnika na sliki 3.10, opazimo, da zadnja operacija bloka vedno pripada istemu opravilu kot prva operacija naslednjega

bloka (izjema je seveda zadnja operacija zadnjega bloka). Formalneje zapišemo ugotovitev takole: $B_i^{|B_i|} \xrightarrow{\mathcal{A}} B_{i+1}^1$ ali tudi $S_{\text{J}}(B_i^{|B_i|}) = B_{i+1}^1$ $\big(1 \leq i \leq (r-1)\big)$. Kot bomo videli v nadaljevanju, je ta lastnost izredno pomembna, saj se zaradi nje operacije posameznih blokov medsebojno ne prehitevajo: če želimo skrajšati izvršni čas urnika, moramo spremeniti vsaj en kritični blok na vsaki kritični poti, prisotni v grafu $\mathcal{G}$.

3.7 Izvajanje premikov na urniku

Različni urniki, ki pripadajo isti instanci problema, se razlikujejo samo po usmeritvah povezav v množici $\mathcal{E}$, katerim pripadajo različna zaporedja izvajanja operacij po strojih. Cilj optimizacije je doseči tako kombinacijo usmeritev, da bo rezultirajoči urnik imel čim krajšo kritično pot in s tem ustrezno majhen izvršni čas $C_{\max}$. Ko izvajamo optimizacijo iterativno (kot pri postopku, ki ga v tem delu obravnavamo), moramo imeti na voljo mehanizem, s katerim na urniku izvajamo potrebne spremembe oziroma premike operacij po strojih. Ta mehanizem sedaj natančno definirajmo.

3.7.1 Formalna definicija premikov

V našem primeru ločujemo dva različna tipa premikov na urniku, to sta premik v desno in premik v levo. Za njuni formalni definiciji moramo vpeljati naslednje pojme.

Naj $\mathcal{O}_i$ označuje zaporedje izvajanja operacij na stroju $\mathcal{M}_{\text{i}}$, $1 \leq i \leq m$. Nadalje, u in v sta dve poljubni operaciji, ki se obe izvajata na stroju $\mathcal{M}_{\text{i}}$, torej velja $u \in \mathcal{O}_i$ in $v \in \mathcal{O}_i$; zaradi enostavnejše definicije zahtevamo, da je operacija u predhodnica operacije v na stroju, iz česar sledi, da se operacija u nahaja v zaporedju $\mathcal{O}_i$ pred operacijo v. Notacija $\mathcal{O}(u, v)$ označuje del zaporedja $\mathcal{O}_i$ od operacije u do operacije v. Opravili, katerima operaciji pripadata, označimo s $\mathcal{J}_{\text{u}}$ in $\mathcal{J}_{\text{v}}$; ker smo se pri definiciji problema omejili na instance, pri katerih se izvaja največ ena operacija vsakega opravila na posameznem stroju, velja zveza $\mathcal{J}_{\text{u}} \neq \mathcal{J}_{\text{v}}$.

Premik v desno, ki ga označimo s $Q_{\mathrm{D}}(u, v)$, spremeni vrstni red izvajanja operacij na stroju $\mathcal{M}_{\mathrm{i}}$ tako, da se po izvedbi premika operacija u nahaja neposredno za operacijo v v zaporedju izvajanja operacij $\mathcal{O}_i$. S tem postane veljavna zveza $S_{\mathrm{M}}(v) = u$. V množici $\mathcal{E}$ se premik $Q_{\mathrm{D}}(u, v)$ odraža tako, da spremeni usmeritev povezav, ki potekajo od operacije u do vseh operacij v množici $\mathcal{O}(u, v)\backslash\{u\}$. Pri premiku v desno pravimo, da je u premikajoča operacija, v pa ciljna operacija.

Premik v levo, ki ga označimo s $Q_{\mathrm{L}}(u, v)$, spremeni vrstni red izvajanja operacij na stroju $\mathcal{M}_{\mathrm{i}}$ tako, da se po izvedbi premika nahaja operacija v neposredno pred operacijo u v zaporedju izvajanja operacij $\mathcal{O}_i$. S tem postane veljavna zveza $P_{\mathrm{M}}(u) = v$. V množici $\mathcal{E}$ se premik $Q_{\mathrm{L}}(u, v)$ odraža tako, da spremeni usmeritev vseh povezav, ki potekajo od operacij v množici $\mathcal{O}(u, v)\backslash\{v\}$ do operacije v. Pri premiku v levo pravimo, da je v premikajoča operacija, u pa ciljna operacija.

Kadar se operacija u nahaja v zaporedju $\mathcal{O}_i$ neposredno pred operacijo v, učinkuje premik $Q_{\mathrm{D}}(u, v)$ enako kot premik $Q_{\mathrm{L}}(u, v)$. V takih primerih je smer premika nepomembna, zato takemu premiku pravimo *zamenjava* in ga označimo s $Q(u, v)$. Uporabljanje oznake brez smeri ni striktno ali obvezujoče.

3.7.2 Teoretična izhodišča za izvajanje učinkovitih premikov

V poglavju 2.1 smo ugotovili, da je različnih urnikov neobvladljivo veliko, zaradi česar je skrajno nesmotrno iskati rešitev problema z izčrpnim pregledovanjem celotnega prostora rešitev. Tudi naključno generiranje urnikov je obsojeno na neuspeh, ker je verjetnost, da bomo naleteli na dobro rešitev, premajhna. Boljše možnosti za uspeh imamo, če nad množico $\mathcal{E}$ izvajamo take spremembe, ki z veliko verjetnostjo vodijo k manjšanju izvršnega časa. Na tem mestu si bomo pogledali nekatera teoretična izhodišča za izvajanje smotrnih premikov na urniku, brez katerih je praktično nemogoče zasnovati uspešen postopek optimizacije Π_{J} problema.

Prvi pomembni ugotovitvi, ki ju bomo opisali, je pred več kot tridesetimi leti odkril Balas (1969). Vsaj prva od njiju je od takrat naprej vgrajena v vse uspešne postopke Π_{J} optimizacije. Ugotovitvi sta naslednji.

Teorem 1. *Imamo izvedljiv urnik $\mathcal{G}(1)$ z izvršnim časom $C_{\max}(1)$. V primeru da v pripadajoči množici $\mathcal{E}$ spremenimo usmeritve poljubni podmnožici povezav, od katerih nobena ne pripada nobeni kritični poti v $\mathcal{G}(1)$, bo rezultirajoči urnik $\mathcal{G}(2)$ (če bo še vedno izvedljiv) imel izvršni čas $C_{\max}(2)$ večji ali kvečjemu enak od prvotnega izvršnega časa: $C_{\max}(2) \geq C_{\max}(1)$.*

Ugotovitve ni težko dokazati. Od kritičnih poti v grafu je neposredno odvisen izvršni čas urnika. Kadar graf spreminjamo na opisani način, ne spremenimo nobene od kritičnih poti. Lahko se zgodi, da postane neka druga pot v grafu daljša, od dosedanjih kritičnih poti, s čimer sama postane kritična. Prvotna kritična pot je v novem grafu zanesljivo prisotna še naprej, neodvisno od tega, ali je v njem še vedno kritična ali ne. □

Teorem 2. *Imamo izvedljiv urnik $\mathcal{G}(1)$. Ko v množici $\mathcal{E}$ spremenimo usmeritev povezave med sosednjima operacijama B_x^y in B_x^{y+1} v kateremkoli kritičnem bloku, je tudi rezultirajoči urnik $\mathcal{G}(2)$ zanesljivo izvedljiv.*

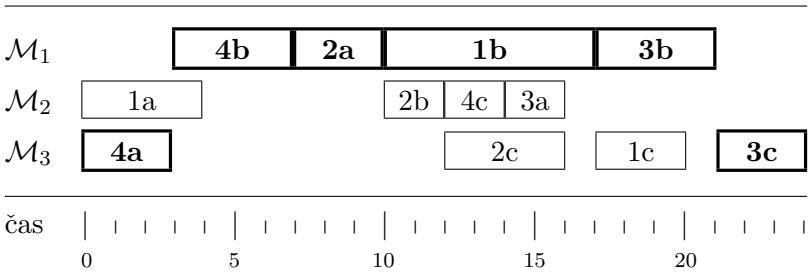
Trditev velja samo, če so izvršni časi vseh operacij večji od 0. To pri realnih problemih vedno velja, saj nobenega opravila ne moremo izvršiti neskončno hitro. Za dokaz predpostavimo, da je zamenjava vrstnega reda operacij B_x^y in B_x^{y+1} povzročila neizvedljiv urnik in s tem cikel v grafu $\mathcal{G}(2)$. Pred zamenjavo vrstnega reda je v grafu $\mathcal{G}(1)$ obstajala povezava $B_x^y \xrightarrow{\mathcal{E}} B_x^{y+1}$, po zamenjavi pa graf $\mathcal{G}(2)$ vsebuje povezavo $B_x^{y+1} \xrightarrow{\mathcal{E}} B_x^y$. Ker predpostavljamo, da je graf $\mathcal{G}(2)$ cikličen, je morala v $\mathcal{G}(1)$ obstajati še ena povezava od operacije B_x^y do B_x^{y+1}, kar je možno edino preko tehnološke naslednice prve operacije $S_{\mathrm{J}}(B_x^y)$ in tehnološke predhodnice druge operacije $P_{\mathrm{J}}(B_x^{y+1})$, saj od vsake operacije vodita največ dve poti: preko tehnološke naslednice ali preko naslednice na stroju. Nadalje, če obstaja v grafu $\mathcal{G}(1)$ pot od $S_{\mathrm{J}}(B_x^y)$ do $P_{\mathrm{J}}(B_x^{y+1})$, bi kritična pot zanesljivo potekala tudi preko njiju, saj je odsek poti $B_x^y \xrightarrow{\mathcal{E}} B_x^{y+1}$ zanesljivo krajši od $B_x^y \xrightarrow{\mathcal{A}} S_{\mathrm{J}}(B_x^y) \xrightarrow{?} \cdots \xrightarrow{?} P_{\mathrm{J}}(B_x^{y+1}) \xrightarrow{\mathcal{A}} B_x^{y+1}$. Zaradi tega je nemogoče, da bi operaciji B_x^y in B_x^{y+1} pripadali istemu kritičnemu bloku, oziroma da bi bili na kritični poti sosednji. To pomeni, da smo zašli v protislovje, s čimer je trditev dokazana. □

Teorem 1 govori o tem, na kakšen način izvršnega časa zanesljivo ni možno zmanjšati; to je s spreminjanjem delov urnika, ki ne pripadajo kritičnim potem. Torej se moramo osredotočiti na spremembe zaporedij operacij, ki so kritične. S tem se pojavi vprašanje, katere kritične zamenjave lahko izvajamo brez bojazni, da bomo zašli v neizvedljivost, o čemer govori teorem 2.

Naslednja trditev, ki so jo prispevali Matsuo in sod. (1988), vpelje dodaten pogoj za zmanjšanje izvršnega časa.

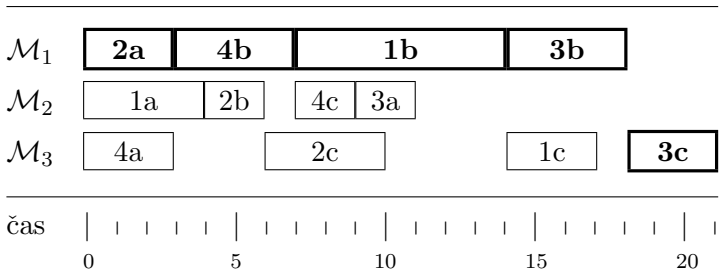
Teorem 3. *Zamenjava zaporedja izvajanja kritičnih operacij B_x^y in B_x^{y+1} lahko zmanjša izvršni čas urnika samo, če vsaj ena od operacij $P_{\mathrm{M}}(B_x^y)$ ali $S_{\mathrm{M}}(B_x^{y+1})$ ne pripada nobeni kritični poti.*

Formalni dokaz trditve si lahko ogledamo v originalnem prispevku. Na tem mestu bomo skušali ozadje teorema predstaviti zgolj empirično. V ta namen se vrnimo k urniku s slike 3.10 in si oglejmo možnosti, ki jih imamo na razpolago, za zmanjšanje izvršnega časa. Operacija $1c$ bi se lahko izvedla ob zgodnejšem času, če ne bi čakala operacije $3c$. Če njun vrstni red zamenjamo, se izvršni čas urnika res zmanjša (slika 3.12), saj operacija $P_{\mathrm{M}}(3c) = 2c$ ni kritična; v nasprotnem primeru bi operacija $2c$ prenehala z izvajanjem tik preden bi se v urniku na sliki 3.10 pričela izvajati operacija $3c$. Med operacijama $2c$ in $3c$ stroj ne bi bil prost, zato bi se po opisani zamenjavi pričela operacija $1c$ izvajati ob istem času, kot se je v prejšnjem urniku pričela izvajati operacija $3c$, s tem pa bi se $3c$ zakasnila prav toliko, kolikor bi se $1c$ pohitrila in izvršni čas bi ostal enak.



Slika 3.12: Urnik s slike 3.10 po izvedbi operacije $1c$ pred operacijo $3c$.

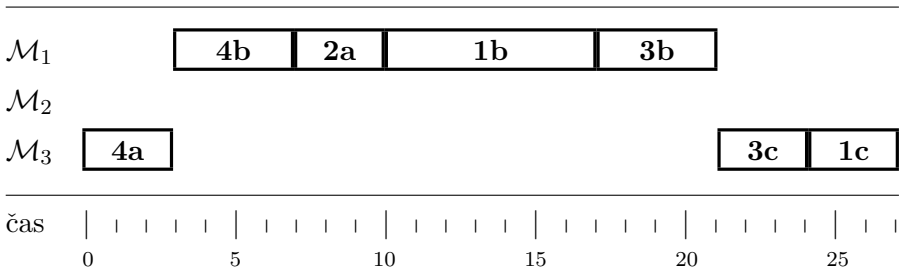
V novem urniku je kritični blok $B_2 = \{4b, 2a, 1b, 3b\}$ še vedno prisoten, zato poskusimo premakniti operacijo $2a$ pred operacijo $4b$. Rezultat prikazuje slika 3.13, kjer ugotovimo, da se je izvršni čas zopet zmanjšal.



Slika 3.13: Urnik s slike 3.12 po izvedbi operacije $2a$ pred operacijo $4b$.

S pomočjo tega primera lahko poudarimo, da teorem 3 predpisuje samo *potrebni* pogoj za zmanjšanje izvršnega časa, ne pa *zadostnega*. Urnik na sliki 3.13 vsebuje kritični blok $B_1 = \{2a, 4b, 1b, 3b\}$. V primeru, da bi sedaj izvedli premik operacije $4b$ pred $2a$, bi zopet dobili urnik na sliki 3.12, s čimer bi se izvršni čas povečal.

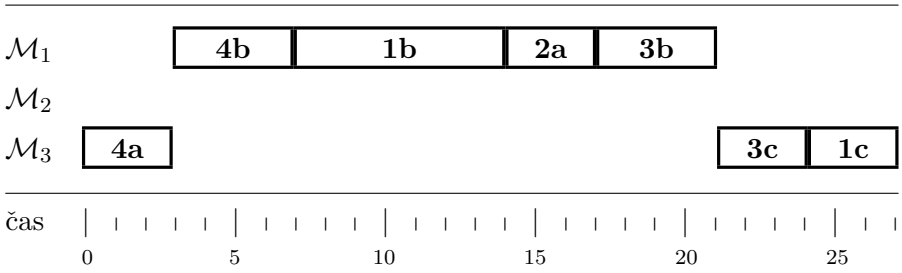
Sedaj si oglejmo primer, ko zamenjamo vrstni red zaporednih kritičnih operacij, ki ne izpolnjujejo zahtev teorema 3. Dogajanje bomo prikazali s pomočjo urnika na sliki 3.10, ki mu bomo zaradi nazornosti odstranili vse nekritične operacije, kakor je prikazano na sliki 3.14.



Slika 3.14: Urnik s slike 3.10 brez nekritičnih operacij.

Zamenjajmo vrstni red operacijama $2a$ in $1b$, kakor je prikazano na sliki 3.15. Ker v urniku na sliki 3.14 velja, da sta obe operaciji $P_{\mathrm{M}}(2a) = 4b$ in $S_{\mathrm{M}}(1b) = 3b$ kritični, se izvršni čas ni mogel zmanjšati. Opisana zamenjava je sicer pohitrila

izvajanje operacije $1b$, vendar je prav toliko zakasnila izvajanje operacije $2a$. Obe zamenjani operaciji čakata na konec izvajanja kritične operacije $4b$, operacija $3b$ pa čaka konec izvajanja vseh treh.



Slika 3.15: Urnik s slike 3.14 po zamenjavi operacij $2a$ in $1b$.

Teorem 3 lahko preprosteje navedemo na naslednji način. Kadar optimiramo urnik z zamenjavami sosednjih kritičnih operacij, so spremembe na robovih kritičnih blokov edine, ki so zmožne same zase zmanjšati izvršni čas urnika. □

Naslednji teorem, ki so ga predstavili ter dokazali Brucker in sod. (1994a) je generalizacija teorema 3 na zamenjave zaporednih kot tudi nezaporednih operacij v kritičnem bloku.

Teorem 4. *Obstaja izvedljiv urnik $\mathcal{G}(1)$ z izvršnim časom $C_{\max}(1)$. Če obstaja urnik $\mathcal{G}(2)$ z manjšim izvršnim časom $C_{\max}(2) < C_{\max}(1)$, se mora v njem obvezno vsaj ena ne-prva (ne-zadnja) operacija vsaj enega kritičnega bloka urnika $\mathcal{G}(1)$ izvesti v $\mathcal{G}(2)$ pred (za) vsemi ostalimi operacijami v kritičnem bloku.*

Formalni dokaz si lahko ogledamo v originalnem prispevku in je dejansko razširitev dokaza o veljavnosti teorema 3. Prva operacija B_x^1 kritičnega bloka B_x vedno čaka tehnološko predhodnico, ki je zadnja operacija predhodnega bloka $B_{x-1}^{|B_{x-1}|}$. Vse ostale operacije v kritičnem bloku B_x se ne morejo izvršiti, dokler ni končano izvajanje operacije B_x^1. S tem, ko premaknemo ne-prvo operacijo kritičnega bloka pred njega, tej operaciji ni več potrebno čakati začetnice bloka, s čimer je odprta možnost hitrejšega izvajanja celotnega urnika.

Analogno velja za premik ne-zadnjih operacij na konec bloka B_x. Prva operacija naslednjega bloka B_{x+1}^1 je tehnološka naslednica zadnje operacije bloka B_x,

zato opercija B^1_{x+1} mora čakati operacijo $B^{|B_x|}_x$, s tem pa v bistvu čaka celotni blok B_x. S tem, ko premaknemo ne-zadnjo operacijo za blok, se odpira možnost hitrejšega izvajanja zadnje operacije v bloku in s tem vseh kritičnih blokov, ki spremenjenemu bloku sledijo. □

Teorem 4 vpelje večjo množico premikov, s katerimi se izvršni čas lahko zmanjša. Neugodna lastnost razširjene množice premikov je, da zanjo ne velja trditev o ohranjanju izvedljivosti: zamenjava vrstnega reda katerihkoli dveh operacij, ki nista sosednji na kritični poti, lahko vodi v neizvedljiv urnik. Nowicki in Smutnicki (1996) s svojim prispevkom gornji teorem dodatno dopolnjujeta. Njune ugotovitve lahko strnemo v naslednjo trditev.

Teorem 5. *Kadar premaknemo ne-prvo operacijo prvega kritičnega bloka na njegov začetek in se pri tem ne spremeni konec kritičnega bloka (to je v primeru, ko ne premaknemo zadnje operacije v bloku), se izvršni čas urnika zanesljivo ne more zmanjšati. Velja tudi analogna trditev: kadar premaknemo ne-zadnjo operacijo zadnjega kritičnega bloka na njegov konec in se pri tem ne spremeni začetek kritičnega bloka (to je v primeru, ko ne premaknemo prve operacije v bloku), se izvršni čas urnika zanesljivo ne more zmanjšati.*

Zopet ne bomo navedli formalnega dokaza, ki se nahaja v originalnem prispevku, pač pa bomo trditev empirično osvetlili. Prvi kritični blok se prične izvajati ob času 0, kar pomeni, da njegova prva operacija ne čaka svoje tehnološke predhodnice. Izvajanja bloka z zamenjavo začetnega dela tako ni možno pospešiti, saj se zaradi tega njegova zadnja operacija ne bo izvedla nič hitreje, s tem pa tudi nadaljnji kritični bloki ne. Izjema je zadnja operacija prvega bloka, katero čaka naslednji blok. Premik te operacije proti začetku bloka lahko pospeši izvajanje naslednjega bloka, vendar to lahko obravnavamo kot premik ne-zadnje operacije na konec bloka, kar pokriva že teorem 4.

Za zadnji kritični blok velja analogna razlaga. Urnik je izveden po koncu izvajanja zadnjega kritičnega bloka (predpostavimo eno kritično pot, s čimer ne izgubimo na splošnosti), kar pomeni, da za njim ni naslednjega bloka, ki bi le-tega čakal, zato sprememba na koncu bloka ne more pospešiti nadaljnjih blokov in s tem zmanjšati izvršnega časa. □

Množica premikov, ki jo favorizirata teorema 4 in 5, je dovolj privlačna, da smo se pripravljeni odpovedati izvedljivosti, ki jo ponuja vztrajanje pri zamenjavah zaporedja sosednjih kritičnih operacij. Zato nas zanima, katere premike lahko izvedemo v danem trenutku, ne da bi urnik postal neizvedljiv. Odgovor nam ponuja naslednja trditev, ki sta jo prispevala Dell'Amico in Trubian (1993).

Teorem 6. *Imamo izvedljiv urnik $\mathcal{G}(1)$, v katerem obstaja kritični blok B_x, ki je zaporedje naslednjih operacij: $B_x^1, \ldots, B_x^{a-1}, B_x^a, B_x^{a+1}, \ldots, B_x^{|B_x|}$. Denimo, da smo premaknili operacijo B_x^a na začetek bloka, s čimer smo dobili zaporedje $B_x^a, B_x^1, \ldots, B_x^{a-1}, B_x^{a+1}, \ldots, B_x^{|B_x|}$. Rezultirajoči urnik bo neizvedljiv takrat in samo takrat, ko obstaja v grafu $\mathcal{G}(1)$ pot vsaj od ene operacije $S_{\mathrm{J}}(B_x^1), \ldots, B_x^{a-1}$ do $P_{\mathrm{J}}(B_x^a)$.*

Pri premiku operacije B_x^a na konec kritičnega bloka, dobimo zaporedje izvajanja $B_x^1, \ldots, B_x^{a-1}, B_x^{a+1}, \ldots, B_x^{|B_x|}, B_x^a$. V tem primeru bo rezultirajoči urnik neizvedljiv takrat in samo takrat, ko obstaja v grafu $\mathcal{G}(1)$ pot od $S_{\mathrm{J}}(B_x^a)$ do katerekoli operacije $P_{\mathrm{J}}(B_x^{a+1}), \ldots, B_x^{|B_x|}$.

Trditev bomo dokazali samo za premik operacije B_x^a na začetek bloka (pri premiku na konec bloka je dokaz povsem analogen). Ker je v $\mathcal{G}(1)$ operacija B_x^a naslednica vseh operacij $B_x^1, \ldots, B_x^{a-1}$ na stroju, obstajajo v grafu $\mathcal{G}(1)$ naslednje usmerjene povezave $B_x^1 \xrightarrow{\mathcal{E}} B_x^a, \ldots, B_x^{a-1} \xrightarrow{\mathcal{E}} B_x^a$; pri izvedbi obravnavanega premika vse te povezave spremenijo smer. Denimo, da obstaja v grafu $\mathcal{G}(1)$ povezava med $S_{\mathrm{J}}(B_x^1)$ in $P_{\mathrm{J}}(B_x^a)$. V tem primeru bo v rezultirajočem grafu obstajal cikel $B_x^a \xrightarrow{\mathcal{E}} B_x^1 \xrightarrow{\mathcal{A}} S_{\mathrm{J}}(B_x^1) \xrightarrow{?} \cdots \xrightarrow{?} P_{\mathrm{J}}(B_x^a) \xrightarrow{\mathcal{A}} B_x^a$. Analogna ugotovitev velja za operacije $B_x^2, \ldots, B_x^{a-1}$. V primeru da ne obstaja nobena pot med $S_{\mathrm{J}}(B_x^1), \ldots, S_{\mathrm{J}}(B_x^{a-1})$ in $P_{\mathrm{J}}(B_x^a)$, tudi obstoj cikla po premiku ni možen, saj se po nobeni poti ne moremo vrniti od obravnavanih operacij do premaknjene operacije B_x^a. □

3.8 Topološko zaporedje

Pri izvajanju modifikacij na urniku prihaja do sprememb zaporedij, po katerih se operacije na posameznih strojih izvajajo (menjajo se usmeritve povezav v množici $\mathcal{E}$), zaradi česar se spreminjajo glave in repi operacij.

Zanima nas, kako na učinkovit način izračunati spremenjene vrednosti le-teh. Lahko bi za vsako operacijo ponovno poiskali najdaljšo pot od ⊙ do nje in nato še od nje do ⊛, kar je zamudno in nepraktično.

Boljšo rešitev ponujata rekurzivni definiciji za glavo in rep. Naj bo w_i operacija, kateri želimo izračunati vrednost glave. Zaradi preglednosti vpeljimo oznaki $pm = P_{\mathrm{M}}(w_i)$ in $pj = P_{\mathrm{J}}(w_i)$. Najdaljša pot od ⊙ do w_i zanesljivo vodi skozi eno od operacij pm ali pj, zato lahko glavo w_i izračunamo s pomočjo formule $h(w_i) = \max\big(h(pm) + p_{pm}, h(pj) + p_{pj}\big) = \max\big(e_{pm}, e_{pj}\big)$. Ideja je preprosta: potrebujemo dolžini najdaljših poti do operacij pm in pj, ki sta enaki njunima glavama. Ti vrednosti povečamo za prispevek pm ali pj k dolžini poti do w_i, ki je enak njunemu času izvajanja, in že imamo vrednost dveh najdaljših poti od ⊙ do w_i, med katerima izberemo daljšo. Edina težava je, da moramo vrednosti $h(pm)$ in $h(pj)$ že poznati, kar lahko rešimo s tem, da preračunavamo glave operacij v pravilnem zaporedju.

Tu nam priskoči na pomoč *topološko zaporedje* (ang. *topological sort*) $\mathcal{T}(\mathcal{G})$ grafa $\mathcal{G}$ (Cormen in sod. 1997). To je urejen seznam vozlišč, v katerem se vsako vozlišče nahaja kjerkoli za vsemi svojimi predhodniki in pred vsemi svojimi nasledniki. Če glave operacij preračunavamo po zaporedju, ki je dano s $\mathcal{T}(\mathcal{G})$, bo naš izračun vedno pravilen. Pomemben podatek je tudi glava vozlišča ⊛, saj predstavlja izvršni čas urnika. Izračunamo jo nazadnje kot $h(\circledast) = \max_i^n \left(e_{w_{i,n(i)}}\right)$.

Za zgled si poglejmo enega od možnih topoloških zaporedij urnika na sliki 3.10:

$$\mathcal{T}(\mathcal{G}) = \{\odot, 1a, 4a, 4b, 2a, 1b, 2b, 4c, 2c, 3a, 3b, 3c, 1c, \circledast\}. \tag{3.1}$$

Analizirajmo operacijo $2b$, ki se v topološkem zaporedju nahaja na sedmem mestu. Operaciji $P_{\mathrm{J}}(2b) = 2a$ in $P_{\mathrm{M}}(2b) = 1a$ se nahajata pred njo: na petem in drugem mestu. Operaciji $S_{\mathrm{J}}(2b) = 2c$ in $S_{\mathrm{M}}(2b) = 3a$ se nahajata za $2b$: na devetem in desetem mestu. Prepričamo se lahko, da veljajo podobne ugotovitve za vse nefiktivne operacije, zato je podano topološko zaporedje pravilno.

Glave operacij preračunamo takole. Prva operacija v topološkem zaporedju je vedno ⊙, za katero po definiciji velja: $h(\odot) = 0$ in s tem $e_{\odot} = 0$. Pravi izračun pričnemo pri operaciji $1a$: $h(1a) = \max(e_{\odot}, e_{\odot}) = \max(0, 0) = 0$. Enak rezultat

dobimo pri $4a$. Sledi $4b$: $h(4b) = \max(e_{\odot}, e_{4a}) = \max(0, 3) = 3$. Že smo pri operaciji $2a$, za katero ugotovimo naslednje: $h(2a) = \max(e_{4b}, e_{\odot}) = \max(7, 0) = 7$. Opisani postopek nadaljujemo, dokler ne dosežemo vozlišče $\circledast$, katerega glava je $h(\circledast) = \max(e_{1c}, e_{2c}, e_{3c}) = \max(27, 16, 24) = 27 = C_{\max}$.

Z analognim postopkom lahko izračunamo tudi repe operacij. Naj bo w_i operacija, kateri želimo izračunati vrednost repa. Podobno, kot smo to storili zgoraj, vpeljemo oznaki $sm = S_{\mathrm{M}}(w_i)$ in $sj = S_{\mathrm{J}}(w_i)$. Najdaljša pot od w_i do $\circledast$ zanesljivo vodi skozi eno od operacij sm ali sj, zato lahko rep w_i izračunamo s formulo $q(w_i) = \max\big(q(sm) + p_{sm}, q(sj) + p_{sj}\big)$. Po definiciji je $q(\circledast) = 0$. Repe računamo po topološkem zaporedju od zadaj naprej, saj pri izračunu repa w_i potrebujemo vrednosti repov obeh njenih neposrednih naslednic.

Izračun pokažimo na primeru. Zadnja nefiktivna operacija v topološkem zaporedju je $1c$: $q(1c) = \max\big(q(\circledast) + p_{\circledast}, q(\circledast) + p_{\circledast}\big) = 0$. Sledi operacija $3c$, za katero velja $q(3c) = \max\big(q(1c) + p_{1c}, q(\circledast) + p_{\circledast}\big) = \max(3, 0) = 3$. Postopek nadaljujemo, dokler ne pridemo do vozlišča $\odot$, katerega rep izračunamo s formulo $q(\odot) = \max_{i=1}^{n}\big(q(w_{i,1}) + p_{w_{i,1}}\big)$. S to vrednostjo lahko vsaj delno preverimo pravilnost izračunov, saj mora veljati $q(\odot) = h(\circledast) = C_{\max}$, sicer je med izračunom prišlo do napake.

V nadaljevanju potrebujemo še naslednje pojme. Neposredna topološka predhodnica $P_{\mathrm{T}}(w_i)$ je operacija, ki se nahaja v $\mathcal{T}(\mathcal{G})$ neposredno pred w_i. Simetrično velja, da je neposredna topološka naslednica $S_{\mathrm{T}}(w_i)$ operacija, ki se v $\mathcal{T}(\mathcal{G})$ nahaja neposredno za w_i. Relaciji $P_{\mathrm{T}}(\odot)$ in $S_{\mathrm{T}}(\circledast)$ nista definirani. Notacija $\mathcal{T}(w_i, w_j)$ označuje del topološkega zaporedja med operacijama w_i in w_j (vključno z njima). Kot zgled zapišimo za topološko zaporedje 3.1 naslednje zveze: $P_{\mathrm{T}}(1b) = 2a$, $S_{\mathrm{T}}(1b) = 2b$ in $\mathcal{T}(4b, 2b) = \{4b, 2a, 1b, 2b\}$.

Ko urnik spreminjamo s tem, da menjamo zaporedje izvajanja operacij na strojih, se določenim operacijam spremenita množici predhodnikov in naslednikov, zato je potrebno topološko zaporedje popraviti, sicer ne bi odražalo novo nastale situacije.

3.9 Delna izbira

Ko vsaj ena povezava v množici $\mathcal{E}$ ni orientirana, pravimo grafu $\mathcal{G}$ *delna izbira* (ang. *partial selection*), ki urnika ne definira v celoti. Namesto tega pripada delni izbiri množica vseh urnikov, ki jih lahko tvorimo z vsemi možnimi usmeritvami neusmerjenih povezav, ki zagotavljajo acikličnost popolne izbire $\mathcal{G}$.

3.10 Razširitve modeliranja z neusmerjenimi grafi

V nadaljnjih poglavjih tega dela uporabljamo neusmerjene grafe in pripadajoče kompletne izbire natančno tako, kot je bilo opisano do sedaj. Kljub temu nakažimo nekatere razširitve osnovnega koncepta, s katerimi je možno modelirati širši spekter problemov (White in Rogers 1990).

Operacijam v instanci lahko priredimo več tehnoloških predhodnic ali naslednic, če dovolimo vstopanje ali izstopanje večjega števila usmerjenih povezav v posamezna vozlišča.

Zamenjava izvršnega časa s kriterijsko funkcijo maksimalna zapoznelost (poglavje 1.3) je na prvi pogled težavna, ker ne moremo več zasnovati optimizacije na podlagi analize kritične poti (poglavje 3.7.2). Izkaže se, da temu ni tako, saj lahko zapoznelost opravil ponazorimo z dodatnimi fiktivnimi zadnjimi operacijami v opravilu, ki so utežene z negativnimi vrednostmi dolžine predpisanega časovnega intervala za vsako opravilo. S tem postane dolžina kritične poti skozi te operacije pozitivna, če ustrezno opravilo kasni; v nasprotnem primeru je njena dolžina negativna. Na ta način se ohrani osnovni princip optimizacije: manjšanje kritične poti v grafu.

Na podoben način je možno modelirati nepripravljenost vseh opravil za izvajanje ob času nič, čase izvajanja, ki so odvisni od zaporedja izvajanja na stroju, ter transportne zakasnitve.

4. Pregled postopkov reševanja

V tem poglavju podajamo pregled postopkov reševanja Π_J problema. Opisujemo le glavne dosežke na tem področju, saj bi bila izčrpna predstavitev preobsežna. Za Π_J problem je namreč značilen izredno pester in bogat razvoj. Verjetno obstaja malo problemov, ki so pritegnili tako veliko množico raziskovalcev, kakor prav ta. Izčrpnejše preglede metod reševanja najdemo v referencah Jain in Meeran (1999), Jain (1998), Vaessens in sod. (1996) ter Błażewicz in sod. (1996).

4.1 Delitev postopkov reševanja

Glede na način pregledovanja prostora rešitev se postopki reševanja Π_J problema delijo na *eksaktne* (ang. *exact*) in *aproksimativne* (ang. *approximative*) (podobno lahko trdimo za večino kombinatoričnih optimizacijskih problemov). Eksaktni postopki jamčijo optimalnost dobljene rešitve, medtem ko pri aproksimativnih take garancije nimamo. Aproksimativni postopki obstajajo zgolj zato, ker do danes ne poznamo eksaktnega načina reševanja, katerega časovna in prostorska kompleksnost bi bila praktično sprejemljiva (poglavje 2.4). Aproksimativni postopki so pogosto edina možnost, ki jo imamo na razpolago; s tem, ko se odpovedo optimalnosti postane njihova kompleksnost zadosti majhna, da so praktični za uporabo.

Druga delitev se nanaša na način tvorjenja rešitve, kjer ločujemo dve skupini. V prvi so postopki, ki urnik gradijo od začetka, zato jim pravimo *konstruktivni postopki* (ang. *constructive algorithms, constructive methods*). Za drugo skupino pa je značilno, da (največkrat zelo slabo) naključno ali kako drugače generirano začetno rešitev iterativno popravljajo v vedno boljšo; pravimo jim *iterativni postopki* (ang. *iterative algorithms, iterative methods*).

V nadaljevanju si bomo ogledali najpomembnejše predstavnike omenjenih skupin postopkov ter tako dobili pregled nad dobrimi in slabimi lastnostmi različnih pristopov, ki so nam na voljo za reševanje.

4.2 Začetki razvoja Π_J problema

Zanimivo je, da ni enotnega mnenja o tem, kdaj se je Π_J problem pojavil in kdo ga je prvi vpeljal. Reference raziskav, ki obravnavajo razvrščanje opravil s pomočjo proizvodnih strojev, segajo v petdeseta leta dvajsetega stoletja. Med njimi najdemo že omenjeni problem Π_J z uniformnimi opravili (Johnson 1954) ter njegovo razširitev v klasični Π_J problem (Jackson 1956).

Kljub temu, da omenjeni deli citirata še zgodnejše reference s področja optimizacije tehnoloških procesov, se nekako smatra, da pripada začetek razvoja Π_J problema prav prispevku Johnson (1954). Pripisuje se mu vpeljava kriterijske funkcije $C_{\max}$; poleg tega je v delu podan učinkovit algoritem za določanje optimalnih urnikov v primeru uniformnega Π_J z dvema strojema, ki se ga da razširiti na navadni Π_J z dvema strojema.

Poleg omenjenih del je za obdobje petdesetih let značilen pojav velikega števila kombinatoričnih optimizacijskih problemov, ki so eksaktno rešljivi z učinkovitimi algoritmi. Za optimizacijo industrijske proizvodnje so zlasti pomembni problemi razvrščanja opravil na enem stroju glede na različne kriterijske funkcije[1].

Raziskave, ki so bile specifično usmerjene v Π_J problem, so doživele razcvet s pojavom knjige *Industrial Scheduling*, ki sta jo uredila Muth in Thompson (1963). V njej so sistematično zbrani vsi do takrat znani rezultati na tem področju. Knjiga vsebuje tudi prve tri testne primere instanc (ang. *benchmark problems*), s pomočjo katerih so lahko različni avtorji med seboj primerjali uspešnost svojih algoritmov. Objavljene instance so z leti pridobile zgodovinsko vrednost in se jih kljub poplavi sodobnejših testnih primerov še vedno uporablja za testiranje algoritmov.

Testni primeri so bili objavljeni v poglavju, ki sta ga prispevala avtorja Fisher in Thompson (1963), zato so dobili predpono **ft**, čeprav zasledimo tudi oznako **mt** po inicialkah urednikov knjige. Celotne oznake instanc so **ft06**, **ft10** in **ft20**, kjer številka za predpono podaja število opravil. Instance so pravokotne z dimenzijami

[1] S tem ni mišljeno, da so vsi problemi razvrščanja opravil na enem stroju rešljivi učinkovito. Za problem Φ, ki smo ga opisali v poglavju 1.3, ne poznamo učinkovitega algoritma.

6×6, 10×10 ter 20×5. Avtorja sta si prizadevala, da bi primeri kar najbolje ponazarjali realne proizvodne situacije, zato sta strojem z nižjimi indeksi dodelila več začetnih operacij znotraj opravil, poznejše operacije pa se pogosteje izvajajo na strojih z večjimi indeksi. Danes so vsi trije testni primeri rešeni. Optimalni urniki, ki instancam pripadajo, imajo izvršne čase 55, 930 in 1165.

Problem **ft06** je bil rešen s pomočjo računalnika leta 1971 (Florian in sod. 1971), skoraj 8 let po objavi. Zanimivo je, da je optimalni urnik najprej zgradila skupina študentov brez pomoči računalnika in optimalnost rešitve tudi dokazala s skrbnim opazovanjem odnosov med operacijami.

Za rešitev problema **ft20** je bilo potrebnih 12 let. Prva avtorja sta bila McMahon in Florian (1975).

Problem **ft10** se je izkazal za daleč najtežjega od vseh treh testnih primerov. Na optimalni urnik je bilo potrebno čakati več kot 20 let (Lageweg 1984), optimalnosti dobljene rešitve pa ni bilo možno dokazati nadaljnjih 5 let, dokler to ni uspelo avtorjema Carlier in Pinson (1989). Težavnost problema **ft10** je postala pregovorna in po splošnem prepričanju je to najbolj intenzivno raziskovan in najpogosteje citiran testni primer v vsej zgodovini Π_J problema.

4.3 Drugi testni primeri instanc

Za primerjavo različnih algoritmov reševanja, potrebujemo ustrezne testne primere instanc. Tri predhodno omenjene instance še zdaleč ne zadoščajo testnim potrebam, saj so relativno majhne. Poleg tega sama dimenzionalnost instance ni edini podatek, s katerim bi testne probleme klasificirali med lažje ali težje[2], saj na težavnost instance vpliva mnogo dejavnikov, ki so se pokazali kot ključni v teku razvoja in testiranja različnih algoritmov reševanja; zato je potreba po večjem številu testnih primerov očitna.

Ker bomo algoritem, ki ga predlagamo v nadaljevanju, preizkusili na večjem številu problemov, je prav, da tej problematiki posvetimo nekoliko več pozornosti.

[2]Veliko zlasti aproksimativnih algoritmov lažje najde optimalni urnik pri reševanju problema **ft20** kot v primeru instance **ft10**.

Skupini **ft** so sledili testni primeri, ki jih je objavili Lawrence (1984). V njej se nahaja štirideset testnih primerov, ki so po velikosti razdeljeni v osem različnih skupin: 10×5, 15×5, 20×5, 10×10, 15×10, 20×10, 30×10 in 15×15. Problemi so imenovani od **la1** do **la40**. Danes je vseh štirideset problemov rešenih, kljub temu, da imata zadnji dve skupini relativno velike dimenzije. Omenjeni testi so dober primer, kako na težavnost instance poleg dimenzionalnosti vpliva več dejavnikov. Za celotno skupino problemov 30×10 so bili optimalni urniki generirani samo štiri leta pozneje (Adams in sod. 1988), medtem ko je bilo potrebno za tri probleme iz skupine 15×10 čakati kar tri leta dlje (Applegate in Cook 1991).

Adams in sod. (1988) so objavili dva problema velikosti 10×10 (**abz5** in **abz6**) in tri probleme velikosti 20×15 (od **abz7** do **abz9**). Po več kot desetih letih sta dva primera iz zadnje skupine še vedno odprta.

Applegate in Cook (1991) sta objavila deset testnih primerov velikosti 10×10, ki se imenujejo od **orb1** do **orb10**. Generirala jih je skupina ljudi v Bonnu leta 1986, ki je bila izzvana, naj naredi probleme težke, kolikor je mogoče. Nekatere od teh instanc so bile s strani avtorjev označene kot pogubljene (ang. *doomed*) ali smrtne (ang. *deadlier*), vendar sta jih Applegate in Cook (1991) uspela rešiti.

Storer in sod. (1992) so objavili dvajset problemov (od **swv1** do **swv20**), od tega jih ima pet velikost 20×10 ter 20×15 in deset 50×10. Zanimivo je, da jih samo polovica v zadnji skupini spada med težke probleme, od katerih danes niso rešeni samo trije. V predzadnji skupini so vse instance še vedno odprte. To velja tudi za dve instanci iz prve skupine.

Yamada in Nakano (1992) sta prispevala štiri primere velikosti 20×20 (imenujejo se od **yn1** do **yn4**); vsi se smatrajo za izredno težavne in so še vedno odprti.

Taillard (1993) je objavil osemdeset testnih primerov (od **td1** do **td80**). Njihove velikosti so dokaj različne: 15×15, 20×15, 20×20, 30×15, 30×20, 50×15, 50×20 in 100×20. Skupina problemov 100×20 in 50×15 je bila rešena relativno hitro (Taillard 1994, Nowicki in Smutnicki 1996), medtem ko sta skupini 20×20 in 30×20 v celoti odprti. Zanimivo je, da je odprta tudi večina problemov 15×15,

medtem ko ista ugotovitev za skupini 20×15 in 30×15 ne preseneča. Rezultati teh testnih primerov se v literaturi ne navajajo pogosto. Razlog je verjetno v nemoči postopkov optimizacije pri njihovem reševanju.

4.4 Eksaktno reševanje problema

Uspeh pri odkrivanju učinkovitih algoritmov za omejeno množico problemov, kot so že omenjena razvrščanja opravil na enem stroju[3], je spodbudil nemalo raziskovalcev, da so vložili veliko truda v razvoj podobnih algoritmov za reševanje problema Π_{J} ter njegovih izvedenk. Verjetno je bila večina raziskav v šestdesetih letih prejšnjega stoletja usmerjena prav v razvoj eksaktnih metod.

Rezultat prizadevanj je bil ničen. Vedno znova se je izkazalo, da praktično vse razširitve (npr. Π_{J}, kjer dopuščamo več kot dva stroja in dve opravili) učinkovito rešenih problemov postanejo časovno in prostorsko neobvladljive (teorija kompleksnosti v tem obdobju še ni obstajala); čas, potreben za rešitev problema, in poraba pomnilnika naraščata prehitro v odvisnost od velikosti instance, da bi bil katerikoli predlagani algoritem reševanja praktičen za uporabo (razen v primerih relativno majhnih instanc).

Eksaktno reševanje temelji na pregledovanju celotnega prostora rešitev; v primeru Π_{J} problema algoritem generira vse izvedljive pol-aktivne ali aktivne urnike ter si zapomni tistega, ki ima najboljšo vrednost kriterijske funkcije. Postopku pravimo *eksplicitna enumeracija* (ang. *explicit enumeration*) ali samo enumeracija (ang. *enumeration*). V osnovi je ideja preprosta in tudi enostavna za realizacijo s pomočjo večine programskih jezikov. Velika težava pristopa je v tem, da moramo generirati in ovrednotiti vsaj vse aktivne urnike, če ne želimo izpustiti optimalne rešitve.

Težavo delno omilimo z razširitvijo osnovne ideje, kjer urnike generiramo sistematično na tak način, da nam ni potrebno pregledati vseh; določene dele prostora rešitev lahko izpustimo iz preiskovanja, ker zanje lahko vnaprej ugotovimo, da ne vsebujejo optimalne rešitve. Takemu pristopu, ki je dosti kompleksnejši za reali-

[3]Podoben uspeh je bil dosežen tudi pri nekaterih drugih kombinatoričnih optimizacijskih problemih, od katerih večina ni neposredno povezana z optimizacijo tehnoloških procesov.

zacijo, pravimo *implicitna enumeracija* (ang. *implicit enumeration*). To je edini znani pristop, ki se ga splača uporabljati, v primeru da se eksaktnim rešitvam ne želimo odpovedati; uporabljamo ga seveda samo v primeru majhnih instanc, ki jih na ta način še lahko obvladamo. V tem trenutku velja, da so instance dimenzij približno 20×10 največ, kar je možno rešiti eksaktno (Jain in Meeran 1999).

4.4.1 Osnovni postopek veji in omejuj

Princip implicitne enumeracije najuspešneje utelešajo postopki *veji in omejuj* (ang. *branch & bound*). Osnovna ideja pristopa je izgradnja preiskovalnega drevesa, katerega izhodiščno vozlišče ponazarja graf $\mathcal{G}'$ (poglavje 3.2); torej graf, v katerem so vse povezave v množici $\mathcal{E}$ neusmerjene. Vsako vozlišče preiskovalnega drevesa predstavlja množico izvedljivih urnikov, ki jih lahko dobimo z vsemi možnimi kombinacijami še neusmerjenih povezav. V naslednjem koraku izberemo eno od še neusmerjenih povezav (npr. $a \overset{\mathcal{E}}{\longleftrightarrow} b$) in tvorimo dve novi vozlišči preiskovalnega drevesa: v prvem vozlišču se nahaja graf $\mathcal{G}'(1)$, ki vsebuje povezavo $a \overset{\mathcal{E}}{\longrightarrow} b$, v drugem pa graf $\mathcal{G}'(2)$, ki vsebuje povezavo $b \overset{\mathcal{E}}{\longrightarrow} a$. Oba nova grafa vsebujeta tudi vse usmerjene povezave, ki jih ima graf v vozlišču, iz katerega izhajata. Če bi postopek nadaljevali toliko časa, da bi v vseh vejah drevesa usmerili vse neusmerjene povezave, bi izvedli eksplicitno enumeracijo celotnega semi-aktivnega prostora rešitev. Vozlišča preiskovalnega drevesa, v katerih se nahajajo ciklični grafi, lahko izpustimo iz nadaljnje optimizacije. Kar zadeva korektnost postopka je vseeno, ali cikličnost preverjamo takoj ali po končani optimizaciji; kljub temu stremimo za tem, da cikle detektiramo čim hitreje, saj s tem manjšamo časovno in prostorsko kompleksnost optimizacije, ker cikličnih vej ne razširjamo naprej (Bakshi in Arora 1969).

Implicitno enumeracijo dosežemo z naslednjo razširitvijo gornjega postopka. Dolžina kritične poti v popolnoma usmerjenem grafu $\mathcal{G}$ je enaka izvršnemu času urnika $C_{\max}$. Tako je dolžina kritične poti v delno usmerjenem grafu $\mathcal{G}'(\mathrm{x})$ lahko merilo za *spodnjo mejo* (ang. *lower bound*) izvršnega časa kateregakoli urnika, ki ga dobimo iz $\mathcal{G}'(\mathrm{x})$ s poljubno usmeritvijo še neusmerjenih povezav, saj dodajanje novih povezav obstoječe kritične poti ne more zmanjšati, lahko jo le poveča.

Optimizacijo torej izvajamo na naslednji način. Vsake toliko časa eno od preostalih nedokončno usmerjenih vozlišč razvijemo do popolnega urnika (usmerimo vse povezave v množici $\mathcal{E}$). Dobljeni urnik ima določen izvršni čas $C_{\max}(\mathrm{x})$. Kadar je to urnik z najkrajšim znanim izvršnim časom do tega trenutka, si ga zapomnimo, $C_{\max}(\mathrm{x})$ pa postane trenutna *zgornja meja* (ang. *upper bound*) optimalnega izvršnega časa $C^*_{\max}$, saj je optimalni izvršni čas lahko le manjši ali kvečjemu enak izvršnemu času katerekoli izvedljive rešitve. Ko imamo novo zgornjo mejo izvršnega časa, pregledamo obstoječe veje preiskovalnega drevesa in preprečimo preiskovanje v tistih vozliščih, katerih spodnja meja je večja ali enaka trenutno znani zgornji meji, saj iz teh vej boljših rešitev od trenutno znane ne moremo dobiti.

Optimizacija je končana, ko imajo vse še ne preiskane veje večjo ali enako spodnjo mejo izvršnega časa od trenutno znane zgornje meje le-tega. Takrat nismo samo našli urnika z optimalnim izvršnim časom, ampak smo njegovo optimalnost tudi dokazali. To je bistvena razlika v primerjavi z aproksimativnimi postopki, ki optimalnosti ne morejo dokazati in se v primeru, ko dobijo optimalen urnik, ne znajo ustaviti, ampak preiskovanje nadaljujejo, dokler ni izpolnjen pogoj za prekinitev, ki je ponavadi dovoljen časovni interval ali število iteracij brez izboljšave trenutne rešitve.

Opisani postopek je za realizacijo dokaj preprost ter za razlago principa veji in omejuj pedagoško primeren, vendar je praktično neuporaben[4]. Preiskovalno drevo je tako veliko, da bi danes z močnejšim osebnim računalnikom komaj reševali probleme velikosti 6×6, nikakor pa ne 10×10. Od šestdesetih let do tega trenutka so se pojavile številne boljše različice postopka veji in omejuj, ki si jih bomo ogledali v nadaljevanju.

4.4.2 Izboljšane različice postopka veji in omejuj

Predhodno opisani postopek deluje nad množico pol-aktivnih urnikov. Preiskovalno drevo bi se zanesljivo zmanjšalo, če bi uspeli omejiti področje delovanja

[4]To tudi ni bil prvi predlagani postopek veji in omejuj, ker je zasnovan na neusmerjenem grafu, ki se je pojavil šele leta 1964.

zgolj na aktivne urnike (poglavje 2.2), za katere sta Giffler in Thompson (1960) dokazala, da vsebujejo vsaj eno optimalno rešitev. Avtorja predlagata aproksimativni postopek reševanja z uporabo aktivnih urnikov, ki pa ga ni težko razširiti v eksaktni postopek veji in omejuj.

Zopet gre za eksaktno konstruktivno metodo, ki poteka na naslednji način. Pričnemo s praznim urnikom. Določimo množico operacij, ki se lahko pričnejo izvajati takoj; to so operacije, ki imajo svoje tehnološke predhodnice že izvršene (na samem začetku so to prve operacije vseh opravil v instanci). Določimo časovne konflikte med operacijami, ki se izvajajo na istem stroju. Konflikt nastane samo, če se možna časovna intervala izvajanja prekrivata vsaj za eno časovno enoto (npr. taki operaciji sta a in b). Za vsak konflikt vzpostavimo dve veji preiskovalnega drevesa: v eni se izvrši operacija a pred operacijo b, v drugi pa je vrstni red zamenjan. Postopek nadaljujemo, dokler ne zgradimo celotnega urnika v vseh vejah. Spodnjo mejo preiskovalnih vozlišč lahko predstavlja izvršni čas delnega urnika; boljše rezultate pa dosežemo, če v oceni upoštevamo tudi tehnološke naslednice že fiksiranih operacij.

To je eden redkih postopkov, kjer aktivni urniki dejansko prispevajo k manjši kompleksnosti preiskovalnega drevesa in kjer slabosti, ki smo jih navedli v poglavju 2.2, ne pridejo do izraza. Kljub temu se ta postopek danes ne uporablja, saj je njegovo preiskovalno drevo še vedno preveliko.

Pomembno pridobitev za Π_J problem je predstavljala vpeljava neusmerjenega grafa, s čimer se je odprla možnost ugotavljanja šibkih točk urnika s pomočjo analize kritične poti in kritičnih operacij. S tem, ko so bile nekatere operacije identificirane kot kritične, so se optimizacijski postopki lahko bolje osredotočili na potencialno obetajoče dele prostora rešitev, s čimer je postalo iskanje optimalnih (in v primeru aproksimativnega reševanja blizu-optimalnih) urnikov precej uspešnejše. Prvi večji uspeh uporabe kritične poti je dosegel Balas (1969), ki je ugotovil, da je za zmanjšanje izvršnega časa nujno potrebno spremeniti kritično pot v grafu. Njegov postopek veji in omejuj je vozlišča preiskovalnega drevesa gradil tako, da je v vsakem koraku spremenil usmeritev natančno ene povezave v množici $\mathcal{E}$, ki se je nahajala na kritični poti. Dokazal je, da je na ta način možno doseči optimalno rešitev iz poljubno izbranega začetnega urnika, kar je zadosten

pogoj, da se njegov postopek veji in omejuj uvršča med eksaktne metode. Z lastno implementacijo tega postopka so Florian in sod. (1971) uspeli rešiti že omenjeni testni problem **ft06**.

Izkušnje s predlaganimi postopki reševanja so pokazale veliko problemov, ki jih je potrebno premostiti, da bi se uporabnost pristopa veji in omejuj lahko razširila na instance večjih dimenzij. Glavno oviro so predstavljale šibke spodnje meje izvršnega časa, ki se jih je lahko pripisalo preiskovalnim vozliščem. Ker so le-te predstavljale realne ocene šele v poznejših fazah optimizacije, je bilo nemogoče dovolj zgodaj omejiti razvejevanje preiskovalnega drevesa na sprejemljivo razsežnost.

To spoznanje je motiviralo veliko število raziskovalcev, ki so vložili ogromno truda v iskanje učinkovitega postopka za računanje boljših spodnjih mej. V prispevku Lageweg in sod. (1977) je zbran povzetek metod, ki so bile rezultat teh prizadevanj; klasificirane so v pet skupin, od katerih prve štiri obsegajo spodnje meje, ki so izračunljive v polinomskem času, v zadnjo skupino pa spadajo NP-kompletne metode. Izkazalo se je, da nobena metoda ni izpolnila pričakovanj; preiskovalno drevo je ostalo neobvladljivo veliko. Vse metode so bile zasnovane, ali pa se izkazale za slabše od ocene, ki je temeljila na reševanju problema Φ na kritičnem stroju. Ta problem je sam zase NP-kompleten, vendar dosti lažje rešljiv od Π_{J} problema. Poleg tega vanj vstopajo samo operacije na enem stroju, s čimer je lahko velikost Φ instanc za več velikostnih redov manjša od instance Π_{J}. Ker vsaka izboljšava spodnje meje pomeni ogromen prihranek zaradi dodatnega rezanja preiskovalnega drevesa, je bilo še kako upravičeno reševati veliko množico lažjih NP-kompletnih problemov s ciljem zmanjšati grozljivo kompleksnost originalnega problema.

Barker in McMahon (1985) sta implementirala postopek veji in omejuj, kjer so vsakemu vozlišču preiskovalnega drevesa prirejeni kompletna izbira, kritična operacija in njej pripadajoči kritični blok (ki nista definirana enako kot v poglavju 3.6) ter spodnja meja. Vejanje poteka na podlagi identifikacije kritične operacije, to je najzgodnejše operacije w_x, za katero velja $t_x + q(x) = C^{\downarrow}_{\max}$, kjer je $C^{\downarrow}_{\max}$ trenutno znana zgornja meja izvršnega časa urnika. Operaciji w_x pripada

ustrezen kritični blok, ki predstavlja zaporedje sosednjih operacij na stroju, ki se konča z operacijo w_x. Ob vejanju se v novih vozliščih spremenijo in fiksirajo relacije med operacijo w_x in ostalimi operacijami kritičnega bloka. Avtorja sta s tem postopkom uspela dobiti urnik z izvršnim časom 960 za testni problem **ft10**.

Naslednji pomemben korak sta prispevala avtorja Carlier in Pinson (1989). Njun postopek veji in omejuj je uporabljal izboljšano metodo za reševanje problema Φ, ki je bila namensko razvita za reševanje Π_{J} problema (Carlier 1982). Pomembnejša od tega pa je uvedba nove tehnike v izgradnjo preiskovalnega drevesa; tehnika se imenuje *takojšnja izbira* (ang. *immediate selection*). Avtorja sta sestavila množico enačb, s katerima sta v posameznih korakih optimizacije uspela dokazati, da se mora določena operacija izvršiti pred ali za neko množico operacij na istem stroju, saj z drugačno razporeditvijo izvršnega časa ni možno zmanjšati. V takem primeru je preiskovalno drevo naraščalo bistveno počasneje, saj ni bilo potrebno tvoriti novih vej za vse možne razporede operacij na kritični poti. S tem postopkom sta avtorja uspela dokazati optimalnost rešitve 930 za notorični problem **ft10**. Še vedno se ni smatralo, da je ta instanca obvladljiva, saj sta avtorja pri reševanju uporabila bližnjico: rezultat 930, ki ga je odkril Lageweg (1984), je bil vhodni podatek v njun postopek. S tem sta si že v izhodišču zagotovila dobro zgornjo mejo, kar jima je omogočalo rezanje velikega števila vej dosti hitreje, kot bi sicer bilo mogoče. Kljub temu je dosežek požel veliko slavo in je bil intenzivno citiran.

Avtorja Applegate in Cook (1991) sta razvila nove metode za določanje bistveno boljših spodnjih mej, s čimer se je tudi občutno povečal računalniški čas za njihovo določanje. Očitno je postajalo, da močnejših spodnjih mej ni mogoče dobiti brez velikega naraščanja potrebne računalniške moči za njihov izračun.

Grabovski in sod. (1986) so postopek za reševanje Φ problema, ki ga je predlagal Carlier (1982), dodatno izboljšali (zopet z namenom prispevati k Π_{J}); empirično so pokazali, da njihov pristop potrebuje manjše število korakov kot predhodna metoda. Pomembnejša od tega pa je uvedba pojma kritični blok operacij (na enem stroju). Avtorji so pokazali, da je za zmanjšanje izvršnega časa Φ urnika potrebno vsaj eno ne-prvo (ne-zadnjo) operacijo kritičnega bloka prestaviti

na začetek (konec) le-tega. To spoznanje, ki so ga Brucker in sod. (1994a) razširili na Π_J problem, je bilo izhodišče za delitev kritične poti grafa, ki predstavlja Π_J urnik, na kritične bloke (poglavje 3.6). S pomočjo teorema 4 se je dalo zasnovati bistveno učinkovitejše vejanje preiskovalnega drevesa v primerjavi s predlogom, ki ga je podal Balas (1969). Dodatno pohitritev reševanja je prispevala močno izboljšana tehnika takojšnje izbire (Brucker in sod. 1994b). S tem se je smatralo, da je problem **ft10** povsem obvladljiv; čas za njegovo izvrševanje se je spustil na velikostni red nekaj minut.

V zvezi z eksaktnim reševanjem problema omenimo še dosežek avtorjev Martin in Shmoys (1996), ki sta postopek veji in omejuj zasnovala v časovnem prostoru s pomočjo časovnih oken, v katerih se operacije lahko izvajajo, dokler še dosegamo boljše rezultate od znane zgornje meje. Vpeljala sta tako imenovano tehniko *britja* (ang. *shaving*), kjer s pomočjo računalniško izredno intenzivnega postopka krčita (brijeta) možne začetke in/ali konce časovnih oken operacij. Empirični rezultati kažejo, da je njihova tehnika močnejša od postopkov Brucker in sod. (1994a,b).

V zadnjem času praktično ni primera eksaktnega reševanja, ki ga ne bi kombinirali z aproksimativnimi postopki, o katerih bo govora v nadaljevanju. Na ta način si zagotovimo dobro izhodiščno zgornjo mejo (podobno, kot sta to storila Carlier in Pinson (1989) za reševanje problema **ft10**). Poleg tega imamo na voljo vsaj aproksimativno rešitev, kadar eksaktni del postopka odpove. Na ta način skušamo izkoristiti prednosti obeh pristopov s poenotenimi postopki reševanja.

Na tem mestu omenimo še, da velja Π_J problem za enega najbolj trmastih in težko rešljivih problemov med diskretnimi kombinatoričnimi optimizacijskimi problemi (French 1982). Videli smo, da po petdesetih letih raziskav in razvoja komaj rešujemo instance z 200 operacijami, kar je nenavadno malo. Za primerjavo omenimo, da je uspel Carlier (1982) rešiti več instanc problema Φ z 10,000 operacijami. V primeru problema trgovskega potnika (poglavje 1.3) algoritmi uspešno rešujejo instance z do 4000 mesti (Jain in Meeran 1999). Neobvladljivost problema Π_J je po vsem raziskovalnem naporu, ki je bil vanj vložen, s tako skromnim rezultatom postala naravnost pregovorna.

4.5 Aproksimativno reševanje problema

Skromni dosežki pri razvoju eksaktnih postopkov reševanja in nespodbudne ugotovitve teorije kompleksnosti so usmerili dosti več raziskav na področje aproksimativnih postopkov reševanja. Ti postopki so primerni za praktično uporabo, kjer se ne moremo zadovoljiti z ugotovitvijo, da imamo na voljo premalo časa in/ali pomnilnika, da bi se lahko dokopali do vsaj blizu-optimalnega urnika. S tem, ko ne vztrajamo na preiskavi celotnega prostora rešitev, oba računalniška resorja kontroliramo mi in ne instanca, ki jo rešujemo. Na primer: določimo lahko, da se postopek prekine po eni uri izvajanja, ali ko pomnilniška poraba preseže deset mega zlogov zasedenega prostora. Kot rezultat bomo dobili najboljši urnik, ki se je nahajal v tistem delu prostora rešitev, ki nam ga je v tem času uspelo preiskati. Vedno pa bomo dobili vsaj slabo rešitev[5], kar je bolje kot nobena rešitev; zasnovi čim optimalnejše proizvodnje, kar je naš cilj, se ne bomo odpovedali zaradi nemočnih eksaktnih algoritmov.

Aproksimativnim postopkom pravimo tudi *hevristični* (ang. *heuristic*) ali izkustveni postopki. Izraz verjetno izvira iz njihove podobnosti v primerjavi s človekovim (izkustvenim) pristopom k reševanju težkih problemov.

4.5.1 Naključno generiranje aktivnih urnikov

Omenili smo že prispevek avtorjev Giffler in Thompson (1960), ki govori o aktivnih urnikih. Opisali smo tudi postopek veji in omejuj, ki je zasnovan na njunih izsledkih. Postopek v omenjenem prispevku ne gradi preiskovalnega drevesa, ampak se v vsakem koraku izgradnje urnika naključno odloči, katero izmed konfliktnih operacij bo izvršil najprej. Vsakič, ko postopek ponovimo, bomo zaradi drugačnih izbir generatorja naključnih števil dobili različen aktivni urnik. Dlje, ko izgradnjo urnikov ponavljamo, večjo verjetnost imamo, da bomo naleteli na optimalno rešitev. Pomnilniška poraba je praktično neodvisna od števila ponovitev, časovna zahtevnost pa narašča linearno. Danes se ta postopek ne uporablja več, ker obstaja mnogo boljših.

[5]Predpostavljamo, da časovne in pomnilniške omejitve ne bodo neracionalno skromne. V času ene pikosekunde z maksimalno porabo treh zlogov pomnilnika verjetno noben aproksimativni postopek ne bi mogel sestaviti niti slabega urnika.

4.5.2 Omejevanje preiskovalnega drevesa

Z majhno modifikacijo lahko vsak postopek veji in omejuj spremenimo v aproksimativnega. V vsakem vozlišču preiskovalnega drevesa ne obdržimo vseh možnih vej, ampak samo b najboljših (na primer v smislu ocenjene spodnje meje). Parameter b je vnaprej določen in se lahko spreminja z globino drevesa ter ostalimi spremenljivkami. Pristop poznamo pod imenom *širinsko preiskovanje* (ang. *beam search*) (Morton in Pentico 1993). Njegova slabost je, da je preiskovalno drevo še vedno prisotno, zato so pomnilniške zahteve velike. Tudi rezultati so navadno slabši od tistih, ki jih dobimo s pozneje razvitimi postopki, zato se ta pristop danes praktično ne uporablja, vsaj ne sam zase.

4.5.3 Prioritetna pravila

Skozi zgodovino Π_J problema je bila velika pozornost raziskovalcev usmerjena v *prioritetna pravila* (ang. *priority dispatch rules*). Ideja je v tem, da pričnemo reševati instanco s praznim urnikom, v katerega dodajamo operacije eno po eno. V vsakem koraku izberemo tisto operacijo, ki je favorizirana s strani izbranega prioritetnega pravila. V prispevku Panwalkar in Iskander (1977) je zbranih 113 prioritetnih pravil, kot so najkrajši izvršni čas, največje število še ne izvršenih opravil, največji skupni čas še ne izvršenih opravil, najmanjše razmerje izvršnega časa operacije proti celotnemu času opravila in ostale. Empirični rezultati kažejo, da nobeno pravilo ne daje izrazito boljših rezultatov od ostalih. Obstaja sicer majhna množica (velikostni red 10-15) opravil, ki v povprečju dajajo boljše rezultate, vendar se uspeh močno razlikuje od instance do instance (Chang in sod. 1996).

Prioritetna pravila so privlačna zaradi izredno majhne računske zahtevnosti (urnik se zgradi v enem prehodu, kjer v vsakem koraku izberemo naslednjo operacijo s skoraj trivialnim postopkom), vendar so rezultati temu primerno slabi. V zadnjem času se je pojavilo nekaj pristopov, kjer so rezultati precej spodbudnejši (na primer Werner in Winkler 1995), vendar gre v takih primerih za kombinacijo prioritetnih pravil z drugimi metodami (na primer omejevano preiskovalno drevo ali lokalno iskanje, o katerem bomo govorili v nadaljevanju), s čimer se tudi računska kompleksnost postopkov močno poveča.

4.5.4 Odpravljanje ozkega grla

Adams in sod. (1988) so predlagali zanimiv postopek reševanja, ki so ga poimenovali *premikanje ozkega grla* (ang. *shifting bottleneck procedure*). Gre za razbijanje celotnega Π_J problema na m Φ problemov, ki se jih reši s pomočjo postopka Carlier (1982). Dobljene rezultate se primerja med seboj in identificira stroj, ki predstavlja ozko grlo; zaporedje izvajanja na tem stroju se fiksira. Ostale stroje se zopet reši kot $(m-1)$ Φ problemov, vendar tokrat z upoštevanjem fiksiranega stroja. Postopek se nadaljuje, dokler niso vsi stroji fiksirani. Celotna ideja vsebuje tudi lokalno reoptimizacijo že fiksiranih strojev ter uporabo parcialne enumeracije, kjer so v posamezni fazi postopka različni stroji identificirani kot ozko grlo. To je bil prvi aproksimativni pristop, ki je uspel poiskati[6] urnik z izvršnim časom 930 za problem **ft10**. S tem je privabil veliko pozornost raziskovalcev, ki so v njem odkrili nekatere šibke točke in predlagali ustrezne izboljšave (na primer Dauzère-Pérès in Lasserre 1993 ter Balas in sod. 1995).

4.6 Lokalno iskanje in meta-hevristika

Ena najuspešnejših hevrističnih tehnik optimizacije je lokalno iskanje. Gre za iterativno metodo, kjer začetni izvedljivi urnik spreminjamo po korakih z izvajanjem majhnih lokalnih premikov; to so na primer zamenjave vrstnega reda izvajanja dveh operacij na stroju ali druga množica izbranih sprememb (eno od možnosti, ki jo uporabljamo v nadaljevanju, smo podrobno opisali v poglavju 3.7). Vseh možnih premikov na urniku je v splošnem zelo veliko, zaradi česar bi postopek lokalnega iskanja potreboval nesprejemljivo veliko časa za njihovo ovrednotenje, kar je predpogoj za smotrno odločitev, kateri premik naj se izvede v naslednjem koraku. Težavo rešujemo z uvedbo okolice urnika, ki vsebuje samo tiste premike, za katere smo se na tak ali drugačen način odločili, da so za optimizacijo koristne. Princip lokalnega iskanja je torej naslednji: trenutnemu urniku določimo okolico in ovrednotimo vse njene premike. Če kateri od premikov uspe zmanjšati izvršni čas, ga izvedemo in vse skupaj ponovimo.

[6]Namerno nismo napisali *problem je uspel rešiti*, ker kot pri večini aproksimativnih pristopov optimalnosti urnika ni bilo mogoče dokazati.

Prednost lokalnega iskanja je v tem, da je vsaj v principu sposobno delovati prav na mestu, kjer je to najbolj potrebno, in to na ustrezen način. Ostale tehnike aproksimativnega iskanja v tem pogledu zaostajajo. Pri uporabi prioritetnih pravil ne moremo izvesti poljubne spremembe na urniku oziroma sploh ne moremo zgraditi poljubnega urnika, ker nobeno prioritetno pravilo ne pokriva vseh kombinacij izvajanja operacij po strojih. Pri postopku premikanja ozkega grla je problem podoben, saj v vsakem koraku reoptimiramo celoten stroj, ne moremo pa spremeniti zaporedja izvajanja samo dvema operacijama. Tako ne preseneča dejstvo, da so tehnike lokalnega iskanja postale zelo popularne.

Naslednja prednost takega pristopa je enostavna možnost razširitve na meta-hevristične postopke. Predhodno opisani princip lokalnega iskanja izvaja bolj ali manj kratkovidne (ang. *myopic*) premike na urniku, zato se prej ali pozneje ujame v lokalni minimum; to je stanje, ko noben premik v okolici ne more sam zase izboljšati kriterijske funkcije, trenutna rešitev pa je slabša od optimalne. Lokalni mimimumi so v primeru Π_J problema relativno gosto posejani preko celotnega prostora rešitev in predstavljajo za postopke lokalnega iskanja velik problem (Mattfeld in sod. 1999). Da bi bila nadaljnja optimizacija možna tudi po dosegu lokalnega minimuma, se poslužujemo meta-hevrističnih postopkov, kjer dovoljujemo tudi izvajanje premikov, ki slabšajo kriterijsko funkcijo. Na ta način imamo možnost pobega iz lokalnega minimuma in preiskovanje ostalih delov prostora rešitev, kjer se mogoče nahajajo optimalne[7] ali blizu-optimalne rešitve.

Meta-hevristični pristopi veljajo danes za najuspešnejšo tehniko optimiranja tehnoloških procesov. Slaba stran njihovega izrazito lokalnega delovanja pa je, da ne preiskujejo celotnega prostora rešitev, ampak se zadržujejo v bližini začetne rešitve (Jain in sod. 2000). Da bi slabost odpravili, se lokalno iskanje največkrat uporablja v kombinaciji z genetskimi algoritmi (o katerih bomo govorili v nadaljevanju) ali kakim drugim mehanizmom globalnega in s tem manj preciznega preiskovanja. Na ta način združimo pozitivne lastnosti obeh pristopov. Dober pregled postopkov lokalnega iskanja za reševanje Π_J problema se nahaja v Vaessens in sod. (1996).

[7]Primeri instanc, za katere poznamo več različnih urnikov z optimalnim izvršnim časom, niso redki.

4.6.1 Okolica za izvedbo lokalnega iskanja

Pri izvedbi lokalnega iskanja je ključnega pomena ustrezno definirana okolica, od katere je odvisen uspeh celotnega postopka optimizacije (Mattfeld 1996, Jain in sod. 2000, Jain 1998). Idealna okolica, kateri se lahko samo bolj ali manj približamo, ima naslednje med seboj protislovne lastnosti (Mattfeld 1996).

Koreliranost: urnik po izvedbi premika se ne sme bistveno razlikovati od izhodiščnega urnika; v nasprotnem primeru prostora rešitev ne moremo natančno preiskovati.

Izvedljivost: če je pred izvedbo premika urnik izvedljiv (ustrezen graf ne vsebuje ciklov), naj se izvedljivost ohrani tudi po izvedbi kateregakoli premika v okolici.

Napredovanje: okolica naj vsebuje samo tiste premike, ki z veliko verjetnostjo vodijo k izboljšanju trenutnega urnika.

Velikost: okolica naj bo čim manjša, da pregledovanje in ocenjevanje primernosti njenih premikov ne porabi veliko časa.

Povezljivost: okolica mora vsebovati take premike, da je z njimi možno v končnem številu korakov priti od kateregakoli začetnega urnika do optimalnega.

Pri definiciji okolice moramo narediti več kompromisov. Primer: okolice z relativno velikim številom premikov niso praktične, ker postopek lokalnega iskanja izgubi pregled nad dogajanjem, poleg tega je vrednotenje vseh premikov časovno zamudno opravilo; okolice z relativno majhnim številom premikov pa po drugi strani večajo možnost, da se potencialno dobri premiki v njih ne nahajajo. Problem ustrezne okolice ostaja odprto vprašanje na področju raziskav Π_J problema. Najbolj znane in teoretično pomembne so naslednje okolice, ki so se formirale skladno s spoznanji in razvojem teorije Π_J problema.

Balas (1969) okolica $\mathcal{H}_B$ vsebuje vse zamenjave zaporedja dveh sosednjih operacij na stroju, če se obe nahajata na kritični poti. Smisel take definicije je

v tem, da zamenjava dveh nekritičnih operacij ne more zmanjšati izvršnega časa urnika, saj dobljeni graf še vedno vsebuje prvotno kritično pot (teorem 1, poglavje 3.7). Poleg tega je rezultat vseh njenih premikov izvedljiv urnik (teorem 2). Dokazano je, da ima ta okolica lastnost povezljivosti. Njena slaba stran pa je, da vsebuje veliko premikov, ki sami zase ne vodijo k izboljšanju izvršnega časa urnika (teorema 3 in 4).

Matsuo in sod. (1988) okolica $\mathcal{H}_M$ vsebuje vse zamenjave dveh sosednjih operacij na stroju, ki se nahajata na kritični poti, s tem da se vsaj ena od njih nahaja na robu bloka kritične poti (izbira sledi iz teorema 3). Ta okolica je občutno manjša od prejšnje (velja relacija $\mathcal{H}_M \subseteq \mathcal{H}_B$), je pa zato izgubila lastnost povezljivosti.

Nowicki in Smutnicki (1996) nadalje reducirata okolico $\mathcal{H}_M$ s pomočjo njunega dokaza, da sprememba začetne operacije začetnega bloka in simetrično končne operacije končnega bloka kritične poti ne more reducirati kriterijske funkcije (če se ob tem ne spremeni konec začetnega bloka oziroma začetek končnega bloka; teorem 5). Okolica $\mathcal{H}_N$ torej iz okolice $\mathcal{H}_M$ odstrani dve zamenjavi, ki teorema ne upoštevata. Velika prednost te okolice je, da je izredno majhna in zato učinkovita, saj njena nadaljnja krčitev ni možna brez izločanja premikov, ki so za optimizacijo obetavni (Jain 1998). Njena slabost pa je nepovezljivost, saj velja $\mathcal{H}_N \subseteq \mathcal{H}_M$.

Dell'Amico in Trubian (1993) definirata več okolic. Na tem mestu opišimo samo tisto (označimo jo s $\mathcal{H}_D$), ki je zanimiva za nadaljnjo obravnavo postopka, o katerem govori to delo. V okolici $\mathcal{H}_D$ se nahajajo vsi premiki, ki katerokoli ne-prvo (ne-zadnjo) operacijo kateregakoli bloka na kritični poti premaknejo na začetek (konec) bloka, kar se v celoti podreja teoremu 4. Neugodna lastnost take izbire so potencialno neizvedljivi premiki (teorem 6). V primeru, da bi bil rezultirajoči urnik neizvedljiv, se premik izvrši proti začetku ali koncu bloka do tistega mesta, kjer je dobljeni urnik še vedno izvedljiv (vedno je možno izvesti premik vsaj za en položaj zaradi veljavnosti teorema 2). Smisel premikanja notranjih operacij izven blokov je torej

v tem, da je to edini možni način zmanjševanja izvršnega časa urnika. Pomen premikanja operacij, ki vodijo v neizvršljive rešitve na pozicijo, kjer se izvedljivost še ohrani, je v tem, da novodobljena kritična pot poteka skozi neposredno tehnološko predhodnico ali naslednico premaknjene operacije. S tem se odpira možnost odprave ciklov v grafu s premikom tehnološke predhodnice ali naslednice izven svojega bloka. Na ta način je dosežena lastnost povezljivosti, hkrati pa se ohrani občutno večji odstotek premikov, ki so potencialno napredujoči (v primerjavi z okolico $\mathcal{H}_B$, ki ima tudi lastnost povezljivosti). V poglavju 5.1.2, kjer je pomen te okolice osvetljen na konkretnem primeru, je ilustrirano dogajanje na urniku med izvajanjem neizvedljivega premika.

Yamada in sod. (1994) ter Balas in Vazacopoulos (1998) okolica $\mathcal{H}_V$ je konceptualno podobna okolici $\mathcal{H}_D$. Od nje se razlikuje le po tem, da so premiki, ki (dejansko ali ocenjeno) vodijo v neizvedljivost urnika, enostavno zavrženi. Velja torej $\mathcal{H}_V \subseteq \mathcal{H}_D$, s čimer je izgubljena povezljivost.

Preden predstavimo postopke lokalnega iskanja, ki omenjene okolice uporabljajo, si oglejmo še metode vrednotenja premikov v okolici.

4.6.2 Vrednotenje premikov v okolici

Eno od ključnih vprašanj lokalnega iskanja je izbira premika v okolici urnika, ki naj se izvede v trenutnem koraku. Premiki morajo biti na določen način ovrednoteni, da lahko algoritem izbere najboljšega[8]. Kot kriterij se praktično vedno uporablja rezultirajoči izvršni čas urnika po izvedbi premika[9]. V prvih izvedbah lokalnega iskanja je bilo vrednotenje premikov implementirano z dejansko izvedbo vsakega premika na trenutnem urniku ter pomnenju rezultirajočega izvršnega časa. Tak pristop je računalniško potraten, saj je potrebno v vsaki iteraciji za vsak premik v okolici preračunati v povprečju polovico glav operacij[10], da lahko določimo izvršni čas urnika.

[8] *Najboljši* je mišljen zgolj v smislu izbranega kriterija, saj gre pri lokalnem iskanju vedno za kratkovidne premike, kar smo predhodno že poudarili.

[9] Druga možnost je razlika novega in trenutnega izvršnega časa, kar je pomembno v nekaterih primerih, kot so algoritmi postopnega ohlajanja, o čemer govorimo v nadaljevanju.

[10] Pri začetnih izvedbah lokalnega iskanja se tipično preračunajo glave vseh operacij, saj so šele Ten Eikelder in sod. (1997) predstavili algoritem, kjer lahko del oparacij izpustimo iz izračuna.

Boljšo rešitev ponujajo postopki aproksimativnega ugotavljanja izvršnega časa na podlagi hitrih ocen. Zanje je najpogosteje značilno, da v primeru povečanja izvršnega časa vrnejo natančno vrednost le-tega; v primeru zmanjšanja pa je njihov rezultat spodnja meja resničnega izvršnega časa, ki je tako lahko večji.

Oglejmo si najprej postopek ocenjevanja, ki je primeren pri uporabi okolice $\mathcal{H}_B$ (Taillard 1994). Premiki v okolici $\mathcal{H}_B$ vedno spremenijo vrstni red dvema kritičnima operacijama u in v, ki sta na kritični poti sosednji. V primeru, da je operacija w_x kritična, zanjo velja zveza:

$$h(w_x) + p_x + q(w_x) = C_{\max},$$

zato je spodnja meja izvršnega časa urnika po izvedbi njenega premika enaka trenutnemu izvršnemu času povečanemu (zmanjšanemu) za toliko, kolikor smo le-to premaknili v desno (levo). Trditev je utemeljena na naslednji način.

Izvršni čas operacije se ne spremeni, ker je podan ob specifikaciji instance. Poleg tega predpostavimo, da se ob premiku ne spremeni niti rep operacije $q(w_x)$, kar ni vedno res. Če smo torej kritično operacijo w_x premaknili v desno (levo), smo s tem povečali (zmanjšali) njeno glavo $h(w_x)$, s čimer je skupna vsota vseh treh členov v zadnji enačbi ustrezno večja (manjša). Ob premiku $Q(u, v)$ se velikokrat operacija v pohitri, operacija u pa zakasni, zato moramo pri oceni upoštevati obe. V poglavju 3.8 smo podali rekurzivni formuli za izračun glav in repov, ki ju lahko direktno uporabimo pri izračunu ocenjenih vrednosti le-teh. Na podlagi gornjih ugotovitev in ob upoštevanju stanja po premiku $Q(u, v)$, kjer velja $v = P_{\mathrm{M}}(u)$ oziroma $u = S_{\mathrm{M}}(v)$, se ni težko prepričati, da lahko izvršni čas urnika po izvedbi obravnavanega premika ocenimo na naslednji način (oznake z apostrofom pomenijo ocenjene vrednosti):

$$
\begin{aligned}
h'(w_v) &= \max\bigl(h(P_{\mathrm{M}}(u)) + p_{P_{\mathrm{M}}(u)}, h(P_{\mathrm{J}}(v)) + p_{P_{\mathrm{J}}(v)}\bigr),\\
h'(w_u) &= \max\bigl(h'(w_v) + p_v, h(P_{\mathrm{J}}(u)) + p_{P_{\mathrm{J}}(u)}\bigr),\\
q'(w_u) &= \max\bigl(q(S_{\mathrm{M}}(v)) + p_{S_{\mathrm{M}}(v)}, q(S_{\mathrm{J}}(u)) + p_{S_{\mathrm{J}}(u)}\bigr),\\
q'(w_v) &= \max\bigl(q'(w_u) + p_u, q(S_{\mathrm{J}}(v)) + p_{S_{\mathrm{J}}(u)}\bigr),\\
C'_{\max} &= \max\bigl(h'(w_v) + p_v + q'(w_v), h'(w_u) + p_u + q'(w_u)\bigr).
\end{aligned}
$$

Vidimo, da je določanje izvršnega časa na ta način mnogo hitrejše kot v primeru natančnega izračuna, saj se je kompleksnost metode zmanjšala z $O(n_{\mathrm{tot}})$ na $O(1)$. Dobljeni izvršni čas $C'_{\max}$ je točen v primeru, da je vsaj ena od operacij u ali v po premiku še naprej kritična. V nasprotnem primeru velja, da je dobljena ocena enaka spodnji meji dejanskega izvršnega časa.

V primeru uporabe splošnejše okolice, ki premika operacije preko celotnega kritičnega bloka (na primer $\mathcal{H}_D$), lahko gornji postopek posplošimo na način, ki sta ga predlagala Dell'Amico in Trubian (1993).

Denimo, da smo spremenili zaporedje izvajanja kritičnega bloka B_i in tako v novem urniku dobili zaporedje izvajanja operacij $w_{\mathrm{c}1}, \ldots, w_{\mathrm{c}|B_i|}$, kjer velja $w_{\mathrm{c}x} \in B_i$ za vsak $1 \leq x \leq |B_i|$ in $w_{\mathrm{c}a} \neq w_{\mathrm{c}b}$ za vsak $a \neq b$, kjer je $1 \leq a, b \leq |B_i|$. Izvršni čas urnika z zaporedjem izvajanja operacij $w_{\mathrm{c}x}$ ocenimo na naslednji način.

$$
\begin{aligned}
h'(w_{\mathrm{c}1}) &= \max\bigl(h(P_{\mathrm{M}}(B_i^1)) + p_{P_{\mathrm{M}}(B_i^1)}, h(P_{\mathrm{J}}(w_{\mathrm{c}1})) + p_{P_{\mathrm{J}}(w_{\mathrm{c}1})}\bigr),\\
h'(w_{\mathrm{c}2}) &= \max\bigl(h'(w_{\mathrm{c}1}) + p_{w_{\mathrm{c}1}}, h(P_{\mathrm{J}}(w_{\mathrm{c}2})) + p_{P_{\mathrm{J}}(w_{\mathrm{c}2})}\bigr),\\
&\vdots\\
h'(w_{\mathrm{c}|B_i|}) &= \max\bigl(h'(w_{\mathrm{c}(|B_i|-1)}) + p_{w_{\mathrm{c}(|B_i|-1)}}, h(P_{\mathrm{J}}(w_{\mathrm{c}|B_i|})) + p_{P_{\mathrm{J}}(w_{\mathrm{c}|B_i|})}\bigr),
\end{aligned}
$$

$$
\begin{aligned}
q'(w_{\mathrm{c}|B_i|}) &= \max\bigl(q(S_{\mathrm{M}}(B_i^{|B_i|})) + p_{S_{\mathrm{M}}(B_i^{|B_i|})}, q(S_{\mathrm{J}}(w_{\mathrm{c}|B_i|})) + p_{S_{\mathrm{J}}(w_{\mathrm{c}|B_i|})}\bigr),\\
q'(w_{\mathrm{c}(|B_i|-1)}) &= \max\bigl(q'(w_{\mathrm{c}|B_i|}) + p_{w_{\mathrm{c}|B_i|}}, q(S_{\mathrm{J}}(w_{\mathrm{c}(|B_i|-1)})) + p_{S_{\mathrm{J}}(w_{\mathrm{c}(|B_i|-1)})}\bigr),\\
&\vdots\\
q'(w_{\mathrm{c}1}) &= \max\bigl(q'(w_{\mathrm{c}2}) + p_{w_{\mathrm{c}2}}, q(S_{\mathrm{J}}(w_{\mathrm{c}1})) + p_{S_{\mathrm{J}}(w_{\mathrm{c}1})}\bigr),
\end{aligned}
$$

$$
C'_{\max} = \max_{x=1}^{|B_i|} \bigl(h'(w_{\mathrm{c}x}) + p_{w_{\mathrm{c}x}} + q'(w_{\mathrm{c}x})\bigr).
$$

Prikazani postopek ocenjevanja izvršnega časa ima računsko kompleksnost $O(|B_i|)$, kar je znaten prihranek v primerjavi s kompleksnostjo $O(n_{\mathrm{tot}})$, ki je potrebna za natančen izračun. Tak način ocenjevanja premikov uporablja tudi postopek, o katerem govori to delo.

V nadaljevanju tega poglavja so predstavljeni najpomembnejši postopki lokalnega iskanja.

4.6.3 Postopno ohlajanje

Postopno ohlajanje (ang. *simulated annealing*) kot meta-hevristični postopek lokalnega iskanja predstavlja analogijo postopnega ohlajanja trdnih snovi v fiziki. Kot orodje za reševanje kombinatoričnih optimizacijskih problemov (konkretno za problem trgovskega potnika) so ga neodvisno uvedli Kirkpatrick in sod. (1983)[11] ter Černy (1985). V zasnovi gre za klasično lokalno iskanje, kjer vsaki rešitvi pripada določena okolica premikov skupaj z njihovimi ocenami primernosti. Specifičnost postopnega ohlajanja je v načinu sprejemanja premikov, ki je stohastičen na podlagi Metropolis kriterija (Metropolis in sod. 1953).

[11] Kirkpatrick in sod. (1983) so v citiranem prispevku demonstrirali uporabnost metode tudi na nekaterih drugih optimizacijskih problemih, ki ne spadajo v skupino optimiranja tehnoloških procesov.

Postopek je naslednji. Naključno izberemo premik (označimo ga z x) v izbrani okolici in določimo izvršni čas urnika po njegovi izvršitvi. Premik na urniku izvedemo z verjetnostjo, ki jo podaja spodnja enačba:

$$P(x) = \min\left\{1, \exp\left(-\frac{C_{\max}(x) - C_{\max}(x_{-1})}{T}\right)\right\}.$$

Oznake imajo naslednji pomen: $P(x)$ verjetnost izvedbe premika x, $C_{\max}(x_{-1})$ izvršni čas urnika pred izvedbo premika, $C_{\max}(x)$ izvršni čas urnika po izvedbi premika, T trenutna absolutna temperatura, ki mora biti pozitivna.

Opazimo, da premike, ki izvršni čas zmanjšajo, izvedemo z verjetnostjo ena. V nasprotnem primeru pa je verjetnost izvedbe odvisna od s temperaturo T normiranega poslabšanja izvršnega časa. Ob začetku optimizacije je temperatura T nastavljena na relativno veliko vrednost, zato je v teku optimizacije sprejeto veliko število nazadujočih premikov, s čimer postopek preiskuje prostor rešitev globalno in se težko ujame v lokalne minimume. Med izvajanjem optimizacije temperaturo postopno nižamo, zato dopuščamo vedno manj nazadujočih premikov. Ko je temperatura dovolj nizka, da praktično ni možno sprejeti nobenega nazadujočega premika več, postopek obtiči v enem od lokalnih minimumov. Ideja postopnega ohlajanja je torej v začetnem grobem in s tem globalnem preiskovanju prostora rešitev, ki se zvezno prelevi v precizno in lokalno omejeno iskanje, ki deluje samo še v okolici najboljše najdene rešitve.

Prvo implementacijo postopnega ohlajanja za reševanje Π_{J} problema so razvili Van Laarhoven in sod. (1992). Uporabili so okolico $\mathcal{H}_B$, ki je enostavna za implementacijo; njene slabosti, ki smo jih predhodno omenili, pa so povzročile, da so se kmalu pojavili učinkovitejši postopki. Avtorji so tudi dokazali (zmotno, kot je opisano v nadaljevanju) asimptotično konvergenco postopka k optimalni rešitvi v primeru neskončnega časa izvajanja; končnim časom izvajanja naj bi seveda ustrezale boljše ali slabše blizu-optimalne rešitve.

Matsuo in sod. (1988) uporabljajo bistveno učinkovitejšo okolico $\mathcal{H}_M$. Poleg boljše okolice je pristop prinesel tako imenovano *strategijo gledanja nazaj in gledanja naprej* (ang. *look back and look ahead strategy*), ki je detektirala možnosti dodatnega izboljševanja urnikov s pregledovanjem medsebojnih odnosov tehnoloških

predhodnic in naslednic operacij na robovih kritičnih blokov ter s tem odkrivala dodatne premike za zmanjševanje izvršnega časa, ki se v okolici neposredno ne nahajajo.

Yamada in sod. (1994) so razvili metodo, ki so jo poimenovali *postopno ohlajanje s kritičnimi bloki* (ang. *critical block simulated annealing*), kjer kritične bloke tretirajo podobno kot Brucker in sod. (1994). To je bila novost pri uporabi postopnega ohlajanja, saj so ostali postopki ostajali pri varni zamenjavi sosednjih kritičnih operacij na kritični poti.

Yamada in Nakano (1995a) sta izvedla postopno ohlajanje na osnovi generiranja aktivnih urnikov (Giffler in Thompson 1960). Nadalje sta ista avtorja v postopno ohlajanje integrirala reoptimizacijo urnikov na podlagi premikanja ozkega grla (Yamada in Nakano 1996a).

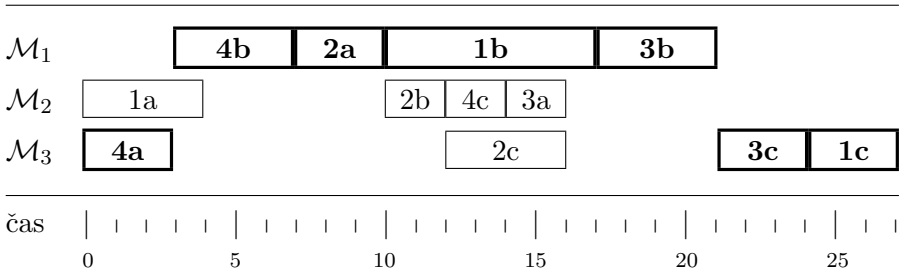
Omenjene izboljšave so dajale boljše in boljše rezultate, vendar noben postopek še zdaleč ni kazal znakov konvergence k optimalnim rešitvam, kot je to napovedoval zgrešeni dokaz Van Laarhoven in sod. (1992). Lastnost nekonvergence je bila tako očitna (Sadeh 1996), da je bila razvita metoda napovedovanja slabih rezultatov na podlagi empiričnega opazovanja prvih nekaj iteracij optimizacije (Sadeh in Nakakuki 1997). Pojav je bil teoretično pojasnjen šele v zadnjem času, ko je Kolonko (1998) odkril, da je prišlo do napake pri preureditvi dokaza, ki velja za problem trgovskega potnika. Dokaz je veljaven le, če ima uporabljena okolica lastnost simetričnosti, ki je definirana na naslednji način.

Obstaja trenutna rešitev problema[12] $\mathcal{R}(1)$, kateri pripada okolica $\mathcal{H}(1)$, ki vsebuje premik x_1. Če ta premik izvedemo, dobimo novo rešitev $\mathcal{R}(2)$, kateri pripada okolica $\mathcal{H}(2)$. V primeru da uporabljena okolica izkazuje lastnosti simetričnosti, mora $\mathcal{H}(2)$ vsebovati premik x_2, s katerim se lahko vrnemo na prvotno rešitev $\mathcal{R}(1)$.

To v primeru trgovskega potnika vedno velja, pri problemu Π_{J} pa praktično nikoli; vse okolice, ki so zasnovane na podlagi kritične poti, te lastnosti ne morejo imeti, saj lahko po izvedbi kateregakoli premika poteka kritična pot po povsem

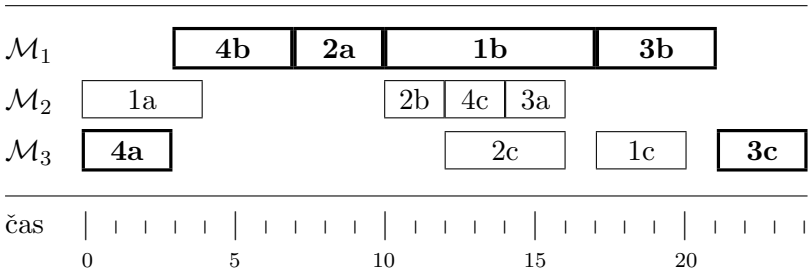
[12]Definicije ne želimo omejiti na probleme razvrščanja tehnoloških procesov, zato nismo napisali, da gre za urnik.

drugem delu urnika in premaknjenih operacij sploh ne vsebuje. Za ilustracijo si oglejmo urnik na sliki 4.1, ki je ponovitev slike 3.10.



Slika 4.1: Ponoven prikaz urnika vzorčne Π_{J} instance.

Denimo, da uporabljamo okolico $\mathcal{H}_B$, v kateri se nahajajo premiki, ki zamenjajo vrstni red naslednjih parov operacij: $(4b, 2a)$, $(2a, 1b)$, $(1b, 3b)$ in $(3c, 1c)$. Če sedaj izvedemo zamenjavo vrstnega reda operacij $3c$ in $1c$, dobimo urnik na sliki 4.2.



Slika 4.2: Urnik po zamenjavi vrstnega reda operacij $3c$ in $1c$.

Opazimo, da nova kritična pot ne poteka več skozi operacijo $1c$, zato tudi v okolici novega urnika, ki se glasi $(4b, 2a)$, $(2a, 1b)$ in $(1b, 3b)$, ne obstaja reverzni premik, ki bi povrnil prvotni vrstni red opazovanima operacijama. Če se v nadaljevanju optimizacije izkaže, da je obravnavani premik zgrešen, ga bo zelo težko popraviti. V tipičnem primeru bo do zamenjave prišlo preko sekvence nekoreliranih premikov, ki bodo zopet restavrirali kritično pot skozi operaciji $1c$ in $3c$, pri tem pa bodo povzročene dodatne spremembe na urniku, kar bo zelo verjetno imelo negativne posledice.

Zaradi nesimetričnosti okolice je problem Π_J skrajno neugoden za realizacijo lokalnega iskanja oziroma vseh aproksimativnih postopkov, ki v zasnovi uporabljajo kritično pot. Kolonko (1998) je demonstriral težavo na primeru preproste instance dimenzij 2×3, kjer se postopno ohlajanje izredno hitro ujame v lokalni minimum, verjetnost pobega pa je zanemarljivo majhna, s čimer je podaljševanje optimizacije izguba časa.

4.6.4 Iskanje s tabu seznamom

Iskanje s tabu seznamom (ang. *tabu search*) (Glover 1989, Glover 1995) je novejša in bistveno bolj uspešna metoda v primerjavi s postopnim ohlajanjem. V delovanju ji je na določen način komplementarna. Prvi razlog za neučinkovitost postopkov postopnega ohlajanja je v zahtevi po počasnem manjšanju temperature, s čimer postanejo izvršni časi optimizacije nesprejemljivo dolgi. Drugi razlog se skriva v dejstvu, da so napredujoči premiki izbrani z verjetnostjo ena; v okolici lokalnega minimuma so to ravno tisti premiki, ki preiskovanje vračajo nazaj v lokalni minimum, zato je pobeg iz njega težaven.

Tabu seznam uporablja za pobeg iz lokalnega minimuma inherentno drugačen mehanizem, zaradi katerega ni samo možno ampak tudi zaželeno, da se lokalnim minimumom približamo čim hitreje, ponavadi zgolj z izvedbo premikov, ki izvršni čas urnika manjšajo. Vsakič, ko premik izvedemo, uvrstimo na tabu seznam njemu inverzen premik, s čimer le-ta postane prepovedan za izvajanje toliko iteracij, kolikor je (vnaprej določena fiksna ali spremenljiva) dolžina seznama (ang. *tabu tenure*).

Postopki lokalnega iskanja so ponavadi implementirani tako, da na tabu seznam ne uvrščajo samih inverznih premikov, ampak atribute trenutne rešitve, ki jih premiki ne smejo restavrirati. Primer: operacija x se ne sme nahajati na stroju pred operacijo y. Na ta način se bolj približamo pravi semantiki zastavljenih prepovedi, ki jih tabu seznam vsebuje.

Postopek optimizacije je naslednji. Izhajamo iz začetnega (tipično slabega) urnika in izvedemo tisti premik v njegovi okolici, ki kriterijsko funkcijo najbolj

izboljša[13], inverzni premik pa uvrstimo na tabu seznam[14]. V novem urniku postopek ponovimo, dokler imamo na voljo napredujoče premike. Ko dospemo do lokalnega minimuma, se postopek konceptualno ne spremeni. Iz okolice izberemo tisti premik izmed ne-tabu premikov, ki kriterijsko funkcijo najmanj poslabša. Ob izvedbi premika, ki urnik poslabša, se velikokrat v okolici pojavi inverzni premik, ki nas vrne v lokalni minimum; brez uporabe tabu seznama bi postopek tak premik na slepo izbral in obtičal v lokalnem minimumu. S pomočjo tabu seznama, ki učinkuje kot kratkotrajni pomnilnik (Glover 1989)[15] oziroma zemljevid pravkar prehojene poti, postopek ve, da takih premikov ne sme izbrati; izbere drug premik in se s tem oddaljuje od lokalnega minimuma ter si tako zagotovi relativno veliko verjetnost, da iz njega pobegne.

Mehanizem tabu seznama se praktično vedno uporablja v kombinaciji z *aspiracijsko funkcijo* (ang. *aspiration function*), ki lahko ukine tabu status določenim premikom, predvsem takrat, ko po izvedbi premika dobimo boljši urnik od trenutno najboljšega.

Eno prvih implementacij iskanja s tabu seznamom za reševanje problema Π_J je objavil Taillard (1994). Uporabil je okolico $\mathcal{H}_B$. Novost je bila estimacija izvršnih časov urnikov po izvedbi premikov in ne dejanska evaluacija, ki zahteva zamudno preračunavanje glav operacij. Na ta način ocene premikov sicer niso bile natančne, dobljene pa so bile dosti hitreje, kar je omogočilo izvedbo bistveno večjega števila iteracij iskanja v enakem časovnem intervalu. V prispevku so navedene zanimive izkušnje z optimizacijo velikega števila problemov različnih dimenzij. Čim število opravil močno preseže število strojev, postanejo instance lažje. Instance z milijon operacijami ne predstavljajo velikega problema, če število strojev ni večje od deset. Dejstvo gre pripisati velikemu številu operacij, ki se izvajajo na posameznih strojih. Njihova prisotnost zagotavlja, da je praktično za

[13]Ker je začetni urnik slabe kvalitete, napredujoči premiki v okolici skoraj zanesljivo obstajajo.

[14]V Mattfeld (1996) obstaja primerjava uspešnosti raznih strategij izbiranja naslednjih premikov, kot so najboljši premik, naključno izbran napredujoči premik in prvi najdeni napredujoči premik. Empirično je pokazano, da pristop izbire najboljšega premika daje v povprečju najboljše rezultate.

[15]V citiranem prispevku ter v Glover (1990) je prikazan tudi pomen srednjeročnega in dolgoročnega pomnilnika pri izvajanju optimizacije, vendar njun opis presega obseg tega dela.

vsak neizkoriščen časovni interval stroja možno najti operacijo, ki le-tega zapolni. Na ta način skoraj vedno naletimo na urnik, kjer je kritični stroj (to je stroj, na katerem se konča vsaj ena kritična pot) optimalno zapolnjen, kar pomeni, da je zaradi povsem zapolnjenih zmogljivostnih omejitev dobljena rešitev zanesljivo optimalna; to pojasnjuje, zakaj so bile nekatere instance velikih dimenzij rešene zelo hitro (poglavje 4.3). Ne preseneča dejstvo, da so najtežje instance kvadratnih ali blizu kvadratnih dimenzij.

Dell'Amico in Trubian (1993) sta izdelala kvalitetno metodo za izdelavo začetne rešitve, ki urnik gradi hkrati od spredaj in od zadaj. Samo iskanje s tabu seznamom uporablja več okolic, od katerih smo eno predhodno podrobneje opisali ($\mathcal{H}_D$). Ker njune okolice vsebujejo potencialno neizvedljive premike, detekcija neizvedljivosti pa je zamudna, sta avtorja uporabila hiter postopek napovedovanja neizvedljivosti; na ta način je neizvedljivost izvedljivega premika včasih lažno signalizirana, važno pa je, da obratno ne drži (neizvedljiv premik ni nikoli prepoznan kot izvedljiv). Avtorja sta tudi generalizirala postopek ocenjevanja izvršnih časov premikov, saj je pristop Taillard (1994) uporaben le z okolico $\mathcal{H}_B$ ali s katero drugo, ki vsebuje samo zamenjave sosednjih operacij na kritični poti.

Barnes in Chambers (1995) sta upoštevala pomen različnih začetnih rešitev in sta pred pričetkom izvajanja tabu iskanja generirala več urnikov na podlagi različnih prioritetnih pravil.

Na prvi pogled izredno učinkovito iskanje s tabu seznamom sta razvila avtorja Nowicki in Smutnicki (1996), ki uporabljata izredno majhno okolico $\mathcal{H}_N$. V rezultatih sta objavila, da njun postopek reši problem **ft10** v 30 sekundah na računalniku 386DX, vendar je podatek zavajujoč. Kot izhodišče za tabu iskanje je uporabljena dobra rešitev, dobljena s postopkom Werner in Winkler (1995), katere časovna kompleksnost je enaka $O(n^3m^5)$, pri čemer le-ta v prikazanih izvršnih časih ni bila upoštevana. Kljub temu se je smatralo, da je njun postopek zelo učinkovit (Jain 1998), dokler niso podrobnejše študije pokazale, da je okolica $\mathcal{H}_N$ premajhna (Jain in sod. 1999). Dobljene rešitve so močno korelirane z izhodiščno rešitvijo, kar pomeni, da uporabljena okolica ne omogoča omembe vrednega oddaljevanja izhodiščnega urnika k ostalim delom prostora rešitev, ki so potencialno obetajoči.

Ten Eikelder in sod. (1997) so razvili tako imenovani *metuljček* (ang. *bowtie*) algoritem, s pomočjo katerega je bilo možno hitrejše eksaktno preračunavanje glav in repov operacij po izvedbi premikov na urniku. Postopek uporablja topološko zaporedje in temelji na dejstvu, da se glave in repi nekaterih operacij po izvedbi premika ne spremenijo. Za primer si zopet oglejmo sliki 4.1 in 4.2, ki prikazujeta urnik pred in po izvedbi zamenjave vrstnega reda izvajanja operacij $3c$ in $1c$. Opazimo, da se večina operacij ni premaknila. Natančneje, glave operacij, ki se v rezultirajočem topološkem zaporedju nahajajo pred premaknjeno operacijo (v primeru premika v levo), se ne spremenijo. Analogna ugotovitev obstaja tudi za repe operacij. Podobno je pri premiku operacije v desno.

Omenimo še prispevek Pezzella and Merelli (2000), kjer je uspešno demonstrirana integracija postopka premikanja ozkega grla v iskanje s tabu seznamom. Vsakič, ko iskanje s tabu seznamom naleti na boljšo rešitev od dosedaj znane, je le-ta reoptimirana s postopkom premikanja ozkega grla.

Danes velja tabu iskanje za zelo uspešno meta-hevristično metodo. Njena slabost je lokalno delovanje v okolici trenutne rešitve, zato se večkrat kombinira z genetskimi algoritmi, ki te lastnosti nimajo.

4.6.5 Genetski algoritmi

Genetski algoritmi (ang. *genetic algorithms*) (Goldberg 1989) so zasnovani na abstraktnem modelu naravne evolucije . Rešitve problema[16], ki jim pravimo *fenotipi* (ang. *phenotypes*), so zakodirane v *kromosomih* (ang. *chromosomes*); kodi, ki predstavlja določeno rešitev, pravimo *genotip* (ang. *genotype*). Vsakemu kromosomu pripada vrednost *prilagoditvene* (ang. *fittnes*) funkcije, ki govori o tem, kako dobro je kromosom prilagojen na okolje (v primeru Π_J problema bi večja vrednost prilagoditvene funkcije pripadala kromosomom, ki vsebujejo urnik s krajšim izvršnim časom).

Optimizacija se izvaja v obliki evolucije. Na začetku se naključno ali na drug način inicializira *populacija* (ang. *population*) kromosomov. Njena velikost ozi-

[16]Razlage v tej fazi ne želimo omejiti na kombinatorične optimizacijske probleme, zato uporabljamo splošni pojem *rešitev problema*.

roma število kromosomov se tipično določi vnaprej. Kromosomom se priredi vrednost prilagoditvene funkcije (pri tem je potrebno genotip odkodirati v fenotip in kvaliteto rešitve ovrednotiti); kromosomi, ki kodirajo fenotip z boljšo vrednostjo kriterijske funkcije, imajo večjo vrednost prilagoditvene funkcije. V naslednjem koraku nastopi *selekcija* (ang. *selection*) (analogija naravnega izbora), s katero se tvori nova populacija. V le-to vstopajo kromosomi naključno, s tem, da se verjetnost njihove izbire veča, z večjo vrednostjo prilagoditvene funkcije.

Nova populacija je podvržena dvema genetskima operatorjema, ki ju imenujemo *križanje* (ang. *crossover*) in *mutacija* (ang. *mutation*). Cilj križanja je združevanje lastnosti dveh kromosomov med seboj v nov genotip (in njemu ustrezni fenotip). Vloga mutacije je v vpeljavi novih oziroma s selekcijo izgubljenih, genetskih vzorcev v populacijo, s čimer imamo možnost preiskovanja širšega področja prostora rešitev. Kromosomi vstopajo v oba operatorja naključno na podlagi vrednosti njihovih prilagoditvenih funkcij. Genetskim operatorjem zopet sledi selekcija. Celoten postopek evolucije se ponavlja, dokler ni izpolnjen terminacijski pogoj, ki je ponavadi predefinirano število evolucijskih ciklov brez izboljšave najboljšega kromosoma.

Ideja genetskih algoritmov je v vzpostavitvi generičnega optimizacijskega orodja, s katerim naj bi bilo možno reševati velik spekter optimizacijskih problemov. Osnovni skelet je vedno enak; vse kar je potrebno storiti ob prilagoditvi algoritma na specifičen optimizacijski problem, je definirati ustrezen genski zapis fenotipov in izbrati ustrezno prilagoditveno funkcijo, ki uteleša kriterijsko funkcijo problema. V uvodnem poglavju knjige *Handbook of Genetic Algorithms* (Davis 1991) je poudarjeno, da predstavlja prilagoditvena funkcija edino znanje, ki ga genetski algoritmi potrebujejo pri reševanju specifičnega problema. Sam genski zapis in z njim povezano dekodiranje kromosomov v fenotipe sicer izkazujeta specifičnost problema, vendar je pri vsakem problemu možno najti več kodnih shem, s čimer pomen le-teh ni v ospredju. Drugi avtorji (na primer Rawlins 1991) poudarjajo, da je tako razmišljanje zavajujoče, saj je uspeh genetskih algoritmov močno odvisen od izbire kodne sheme. Izkaže se, da uspešne kodne sheme vključujejo več specifičnega znanja o problemu. Rawlins (1991) podaja mnenje,

da so genenetski algoritmi brez vgrajenega specifičnega znanja o problemu enako uspešni kot naključno preiskovanje prostora rešitev.

Ob vpeljavi genetskih algoritmov kot orodju za optimizacijo, so bile raziskave intenzivno usmerjene v probleme iskanja minimuma ali maksimuma določene funkcije neodvisnih spremenljivk. Kot primer si poglejmo naslednjo funkcijo:

$$f = a_1x_1 + a_2x_2 + \cdots + a_nx_n.$$

Oznake $a_1, \ldots, a_n$ predstavljajo koeficiente, $x_1, \ldots, x_n$ pa binarne spremenljivke. Cilj optimizacije je poiskati tako konfiguracijo spremenljivk x_i, da bo vrednost funkcije f največja.

V takem primeru so kromosomi praktično vedno nizi n števk, ki lahko zavzamejo vrednost 0 ali 1. Kromosomom priredimo vrednost prilagoditvene funkcije tako, da izračunamo funkcijo f s konfiguracijo spremenljivk, ki je v kromosomu zapisana[17]. Ob križanju izberemo dva kromosoma in novega tvorimo tako, da določene odseke genskega zapisa preslikamo vanj iz prvega roditelja, ostale odseke pa iz drugega. Ob mutaciji nekaj naključno izbranim genom zamenjamo vrednosti.

Opisani postopek se zdi kar preveč enostaven, da bi v praksi lahko deloval. V Goldberg (1989) si lahko ogledamo teoretično utemeljitev delovanja takega koncepta optimizacije s pomočjo močno citiranega *teorema o shemah* (ang. *schema theorem*). Ob uporabi neodvisnih spremenljivk (v gornjem primeru) naj bi teorem o shemah teoretično utemeljil preiskovanje prostora rešitev z genetskimi algoritmi kot optimalnega v smislu verjetnosti odkritja najboljše rešitve v omejenem času.

Pri uporabi genetskih algoritmov za reševanje kombinatoričnih optimizacijskih problemov (Gen in Cheng 1997) se stvari močno zapletejo. Rešitev večine problemov v tej skupini ni mogoče zakodirati z nizom neodvisnih simbolov. Za problem Π_J navedimo naslednji primer. Denimo, da posamezni geni vsebujejo indekse operacij, katere le-ti ponazarjajo. Vsaka operacija nastopa v urniku samo enkrat, torej lahko vsak indeks nastopa samo enkrat v genskem zapisu. Tako geni

[17]Ponavadi ob tem opravimo tudi skaliranje vrednosti prilagoditvene funkcije, ki prepreči izrazito odstopanje manjšega števila izredno dobrih kromosomov, s čimer bi ostali del populacije ob selekciji odmrl (Goldberg 1989).

med seboj niso neodvisni (Davis 1985a, 1985b), s čimer odpade osnovna predpostavka, na kateri sloni teorem o shemah. Ustrezna teorija, ki bi obravnavala probleme, ki jih lahko zakodiramo samo z medsebojno odvisnimi geni, danes še ne obstaja.

Tudi realizacija genetskih operatorjev se zaradi odvisnosti genov močno zaplete. Mutacija ne sme izbranemu genu naključno spremeniti vrednosti, saj bi s tem urnik postal neizvedljiv, ker bi večkrat vseboval isto operacijo. Ena od možnosti v takem primeru je zamenjava vrednosti dveh genov, s čimer izpolnimo zahtevo, da nastopa indeks vsake operacije v kromosomu samo enkrat. Pri tem pa zopet nimamo prostih rok, saj operacije v kromosomu ne smemo premakniti pred njeno tehnološko predhodnico ali za njeno tehnološko naslednico. Izvedba ustreznega operatorja križanja je še bistveno zahtevnejša.

Izkaže se, da različni kombinatorični optimizacijski problemi, zahtevajo različno zapletene kodne sheme. V primeru trgovskega potnika so se uspešne kodne sheme pojavile relativno hitro (Grefenstette in sod. 1985, Goldberg in Lingle 1985). Problem Π_{J} se je tudi v tem pogledu izkazal za mnogo težjega. Iskanju ustreznega genskega zapisa in njemu primernih genetskih operatorjev je namenilo pozornost veliko raziskovalcev. Rezultat prizadevanj je večje število kodnih shem, ki jih lahko razdelimo v devet skupin (Cheng in sod. 1999a, 1999b).

Eno najzgodnejših kodnih shem je razvil Davis (1985b). Njegov pristop je imel veliko slabost, da so kromosomi lahko ponazarjali samo bez-čakalne urnike, ki ne vsebujejo vedno optimalne rešitve. Della Croce in sod. (1995) so skušali slabost omiliti z uporabo *tehnike gledanja naprej* (ang. *look-ahead technique*), s katero so brez-čakalni urnik skušali pretvoriti v aktivnega. Pristop še vedno ni jamčil, da je možno zakodirati vse aktivne urnike. Kljub temu so avtorji s tem postopkom generirali urnik z izvršnim časom 946 za problem **ft10**.

Nakano in Yamada (1991) sta uporabila binarno kodno shemo, kjer vsakemu paru operacij w_a in w_b na istem stroju priredimo binarni gen. Ko se operacija w_a izvaja pred operacijo w_b, je ustrezni gen nastavljen, v nasprotnem primeru je izbrisan. Slabost takega zapisa je veliko število potrebnih genov, saj v primeru pravokotne instance dimenzij $n \times m$ potrebujemo $mn(n-1)/2$ binarnih simbo-

lov. S takim zapisom lahko trivialno predstavimo vse pol-aktivne urnike, kar je bistvena prednost pred pristopom Della Croce in sod. (1995). Velika težava proste izbire vrednosti vsakega gena je možnost kodiranja neizvedljivih urnikov. Avtorja sta razvila postopek imenovan *harmonizacija* (ang. *harmonization*), s katerim sta skušala neizvedljivi kromosom pretvoriti v kar se da podoben izvedljivi kromosom. Pri reševanju problema **ft10** sta generirala urnik z izvršnim časom 965.

Ista avtorja sta leto kasneje (Nakano in Yamada 1992) predlagala kodni zapis, kjer posamezni geni predstavljajo končni čas izvajanja ustrezne operacije, zaradi česar so kromosomi bistveno krajši (mn genov namesto $mn(n-1)/2$). Genetske operatorje sta kombinirala z algoritmom Giffler in Thompson (1960) za izgradnjo aktivnih urnikov, s čimer sta odpravila možnost neizvedljivih kromosomov. S tem pristopom sta zgradila urnik z izvršnim časom 930 za instanco problema **ft10**.

Fand in sod. (1993) so opazili zanimiv fenomen. Med potekom evolucije začetni deli kromosoma konvergirajo hitreje od zadnjih. Posledica tega je, da izvajanje križanja v začetnem delu kromosomov vodi v veliko število lažnih križanj, saj med seboj križamo identične dele kromosomov. Poleg tega so mutacije v zadnjih delih kromosomov manj učinkovite, ker se kromosomi tam tako ali tako močno spreminjajo. Avtorji so zaradi tega porazdelili večji delež križanj v zadnji del kromosoma, mutacije pa so pogosteje izvajali v sprednjem delu. Empirično so demonstrirali večjo učinkovitost takega izvrševanja genetskih operatorjev v primerjavi z uniformno porazdeljenim delovanjem. Za problem **ft10** so uspeli zgraditi urnik z izvršnim časom 947.

Do danes najuspešnejšo kodno shemo je predlagal Bierwirth (1995). Kromosom je niz n_{tot} genov, ki vsebujejo opravilo, kateremu pripada operacija, ki jo ponazarja gen. Urnik se gradi na pol-aktiven način od začetka kromosoma naprej. Ob dekodiranju gena se ugotovi, katera operacija v vsebovanem opravilu je naslednja za uvrstitev v urnik; uvrsti se jo na mesto, ki ga določata njena tehnološka predhodnica in predhodnica na stroju. Najpomembnejša lastnost te sheme je, da je z njo možno zakodirati vse pol-aktivne urnike. Njena dodatna dobra stran je relativno preprosta izvedba genetskih operatorjev (Bierwirth in

sod. 1996). Mutacijo lahko izvedemo s preprosto zamenjavo vsebine dveh genov; na tehnološke predhodnice in naslednice operacij ni potrebno paziti, ker geni ne vsebujejo operacij, ampak le informacijo o pripadnosti opravilu. Pri reševanju problema **ft10** so Bierwirth in sod. (1996) dosegli rezultat 936.

Zanimiv poskus z genetskimi algoritmi so izvedli Mattfeld in sod. (1994). Kromosomom so priredili določene atribute socialnega obnašanja. Posamezni kromosomi v populaciji so bili prostorsko razporejeni, s čimer je vsak kromosom dobil določeno število sosedov[18]. Križanje poteka samo med kromosomi v njihovi okolici, s čimer je delno preprečena prezgodnja konvergenca populacije k izrazito izstopajočim posameznikom, kar je velik problem klasičnih genetskih algoritmov (Baker 1985). Poleg tega so kromosomi primerjali svoje prilagoditvene funkcije s sosedi in se obnašali različno glede na njihovo relativno uspešnost proti okolici. Zadovoljni kromosomi so težili k temu, da ohranijo svoj genetski material, medtem ko so slabši primerki težili k spremembam. Empirično so demonstrirali, da je tak pristop zelo robusten in se dobro obnese tudi na večjih problemih od **ft10**, za katerega so uspeli dobiti urnik 930.

Podoben eksperiment močnega posnemanja naravnega dogajanja sta izvedla Feyzbakhsh in Matsui (1999), ki ga omenjamo, čeprav ni namenjen reševanju problema Π_J. Kromosomom sta priredila nov atribut starosti ter posameznikom določila življensko dobo. Uvedla sta genetski operator *smrt* (ang. *death*), ki v vsaki populaciji odstrani nekaj kromosomov, ki so izbrani naključno na podlagi njihove starosti. V odvisnosti od starosti se tudi izbirajo kromosomi za križanje, za razliko od klasičnih implementacij genetskih algoritmov, kjer sta kromosoma za križanje izbrana zgolj na podlagi prilagoditvene funkcije. Mutaciji so podvrženi samo novo nastali kromosomi po križanju.

4.6.6 Genetsko lokalno iskanje

Predhodno smo spoznali nekaj primerov implementacije genetskih algoritmov za reševanje Π_J problema. Izkušnje na podlagi empiričnih rezultatov kažejo, da

[18]Do sedaj ni položaj v populaciji igral nobene vloge. Kromosomi so bili izbrani za delovanje genetskih operatorjev naključno, zgolj na podlagi njihove prilagoditvene funkcije.

ti algoritmi preiskujejo prostor rešitev robustno in niso podvrženi problemu lokalnih minimumov. Kljub temu je z njimi težko doseči solidne blizu-optimalne rešitve, ker tak način preiskovanja ni primeren za natančno približevanje optimalnim rešitvam (Grefenstette 1987). Rešitev ponujajo hibridne metode, kjer genetske algoritme združimo z eno od oblik lokalnega iskanja. Tem metodam, ki izkazujejo zelo solidne rezultate, pravimo *genetsko lokalno iskanje* (ang. *genetic local search*). Njihova uspešnost ne preseneča, saj lokalno iskanje integrira zajetno količino specifičnega znanja o optimizacijskem problemu v postopek optimizacije. Podroben pregled genetskega lokalnega iskanja podaja Mattfeld (1996). Na tem mestu omenimo naslednje prispevke.

Grefenstette (1987) je pri reševanju problema trgovskega potnika vsak kromosom, ki je bil rezultat genetskih operatorjev, uporabil kot izhodiščno rešitev za izvedbo lokalnega iskanja. Na ta način populacija vsebuje samo rešitve, ki se nahajajo v močnih lokalnih minimumih, v katere gravitirajo iz širokega okoliškega področja. Verjetnost odkritja optimalnih ali blizu-optimalnih rešitev je tako precej večja kot pri klasični izvedbi genetskih algoritmov.

Za reševanje problema Π_J sta Dorndorf in Pesch (1995) predlagala dva postopka genetskega lokalnega iskanja. V prvi izvedbi so v genih zakodirana prioritetna pravila, s katerimi se določa izgradnja urnika. Uporabljen je algoritem generiranja aktivnih urnikov (Giffler in Thompson 1960), kjer se na podlagi prioritetnih pravil izbirajo operacije ob konfliktih. S to izvedbo sta sestavila urnik z izvršnim časom 960 pri reševanju problema **ft10**. Druga izvedba uporablja kromosome, ki imajo toliko genov, kolikor strojev problem vsebuje. Le-ti določajo zaporedje izbire stoja, ki predstavlja ozko grlo pri optimizaciji s premikanjem ozkega grla (Adams in sod 1988). Pri reševanju problema **ft10** sta sestavila urnik z izvršnim časom 938.

Yamada in Nakano (1996a,b) sta izvedla zanimiv operator križanja, ki je v zasnovi iterativen in poteka na naslednji način. Izbrana sta dva kromosoma. Pred začetkom križanja je rezultirajoči kromosom enak enemu od staršev. V posameznih korakih križanja se temu kromosomu določi okolico. Posamezne rešitve v njej se indeksirajo na podlagi razdalje (oziroma mere različnosti) do drugega

starša. Iz okolice se naključno izbere eno od rešitev, vendar s tendenco, da so pogosteje izbrane tiste rešitve, ki so bližje drugemu staršu. Sprejetje nove rešitve je nadalje odvisno od rezultirajočega izvršnega časa. Rezultat križanja je najboljši kromosom, ki je generiran po vnaprej določenem številu iteracij. Avtorja sta uspela zgraditi urnik z izvršnim časom 930 za problem **ft10**.

Genetsko lokalno iskanje je zelo uspešna aproksimativna metoda reševanja kombinatoričnih optimizacijskih problemov, vendar je teoretično še neraziskana. Mattfeld (1996) poudarja, da teorem o shemah za tak pristop ne velja, ker lokalno iskanje uniči mnogo shem v populaciji s tem, ko kromosome oziroma pripadajoče rešitve potiska v lokalne minimume. Empirični rezultati kažejo, da se s tem kvaliteta rešitev močno poveča, čas optimizacije pa občutno zmanjša, vendar ustrezna teorija, ki bi ta fenomen pojasnila, še ne obstaja.

5. Popravljalna tehnika

V predhodnih poglavjih je bilo omenjenih veliko postopkov aproksimativnega reševanja Π_J problema. Na podlagi empiričnih rezultatov, ki so jih avtorji dosegli, lahko zaključimo, da so najuspešnejši tisti postopki reševanja, ki uporabljajo tako ali drugačno obliko tehnike lokalnega iskanja.

Kljub nesporni perspektivnosti takega pristopa za reševanje številnih kombinatoričnih optimizacijskih problemov, naleti lokalno iskanje na veliko oviro prav pri problemu Π_J. Težava nastane zaradi obstoja neizvedljivih urnikov, o čemer smo govorili v poglavjih 2.1, 3.2, 3.7 in 4.6.1. Vzrok za neizvedljivost določenih kombinacij izvajanja operacij po strojih so izključno tehnološke omejitve med operacijami. Pri problemih, kot so trgovski potnik in večina problemov razvrščanja opravil na enem stroju, neizvedljivosti rešitev v tem smislu ne poznamo.

Delo je usmerjeno v omejevanje težav, ki jih neizvedljivost urnikov povzroča postopkom lokalnega iskanja. V nadaljevanju so najprej opisani ustaljeni prijemi preprečevanja neizvedljivosti in njihove slabosti (poglavje 5.1), nato pa sledi opis popravljalne tehnike, ki je naš odgovor na obravnavani problem. Ključna novost je vpeljava dodatnega nivoja v postopke lokalnega iskanja, s čimer je omogočeno varno izvajanje poljubnih premikov na urniku (poglavje 5.2).

5.1 Dosedanji pristopi k izogibanju neizvedljivosti

Če naredimo pregled nad dosedanjimi postopki lokalnega iskanja za reševanje Π_J problema, ugotovimo, da so se skozi zgodovino razvoja formirali trije pristopi izogibanja neizvedljivim urnikom. Prva možnost je uporaba okolice, za katero izvedljivost inherentno velja; z nobenim premikom v njej ne moremo izvedljivega urnika spremeniti v neizvedljivega. Primeri takih okolic so $\mathcal{H}_B$, $\mathcal{H}_M$ in $\mathcal{H}_N$. Okolica $\mathcal{H}_M$ ni več atraktivna (poglavje 4.6.1), tako da ostajata v igri le preostali dve. Kot smo omenili v poglavju 4.6.4, je bilo za okolico $\mathcal{H}_N$ ugotovljeno, da je premajhna in da ne vsebuje mnogih premikov, ki so za optimizacijo potencialno koristni. Kljub temu pa ostaja privlačna zaradi svoje majhnosti in veliko raziskovalcev ravno zaradi te lastnosti intenzivno dela z njo.

Druga do sedaj implementirana možnost izogibanja neizvedljivim urnikom je delo s potencialno neizvedljivo okolico, iz katere neizvedljive premike sproti odstranjujemo. Tak primer je okolica $\mathcal{H}_V$. Slaba stran tega pristopa je, da zavržemo premike, ki so lahko za optimizacijo koristni. Premik, ki je v tem trenutku neizvedljiv, se lahko v nadaljevanju izkaže za nujno potrebnega pri doseganju optimalnega urnika. Kot smo videli v poglavju 2.1, je neizvedljivost premika na nekem stroju pogojena z razporeditvijo operacij na ostalih strojih, zato neizvedljivost sama zase ne more biti kriterij pri selekciji nekoristnih premikov. S tem, ko neizvedljive premike odstranimo iz okolice, zmanjšamo moč lokalnega iskanja, ki ni več sposobno tako učinkovitega preiskovanja prostora rešitev, kot bi lahko bilo, če bi imelo na voljo celotno okolico.

Tretji pristop pri izogibanju neizvedljivosti je modifikacija neizvedljivih premikov, ki rezultira v povrnitvi izvedljivosti. Omenjeno idejo uteleša okolica $\mathcal{H}_D$, ki se izkaže za boljšo od ostalih obravnavanih okolic.

Slabosti okolic $\mathcal{H}_N$ in $\mathcal{H}_V$ so dovolj očitne, da smo jih lahko opisali v tem kratkem pregledu. Okolicama $\mathcal{H}_B$ in $\mathcal{H}_D$ pa posvečamo podrobnejšo diskusijo, ki sledi v nadaljevanju.

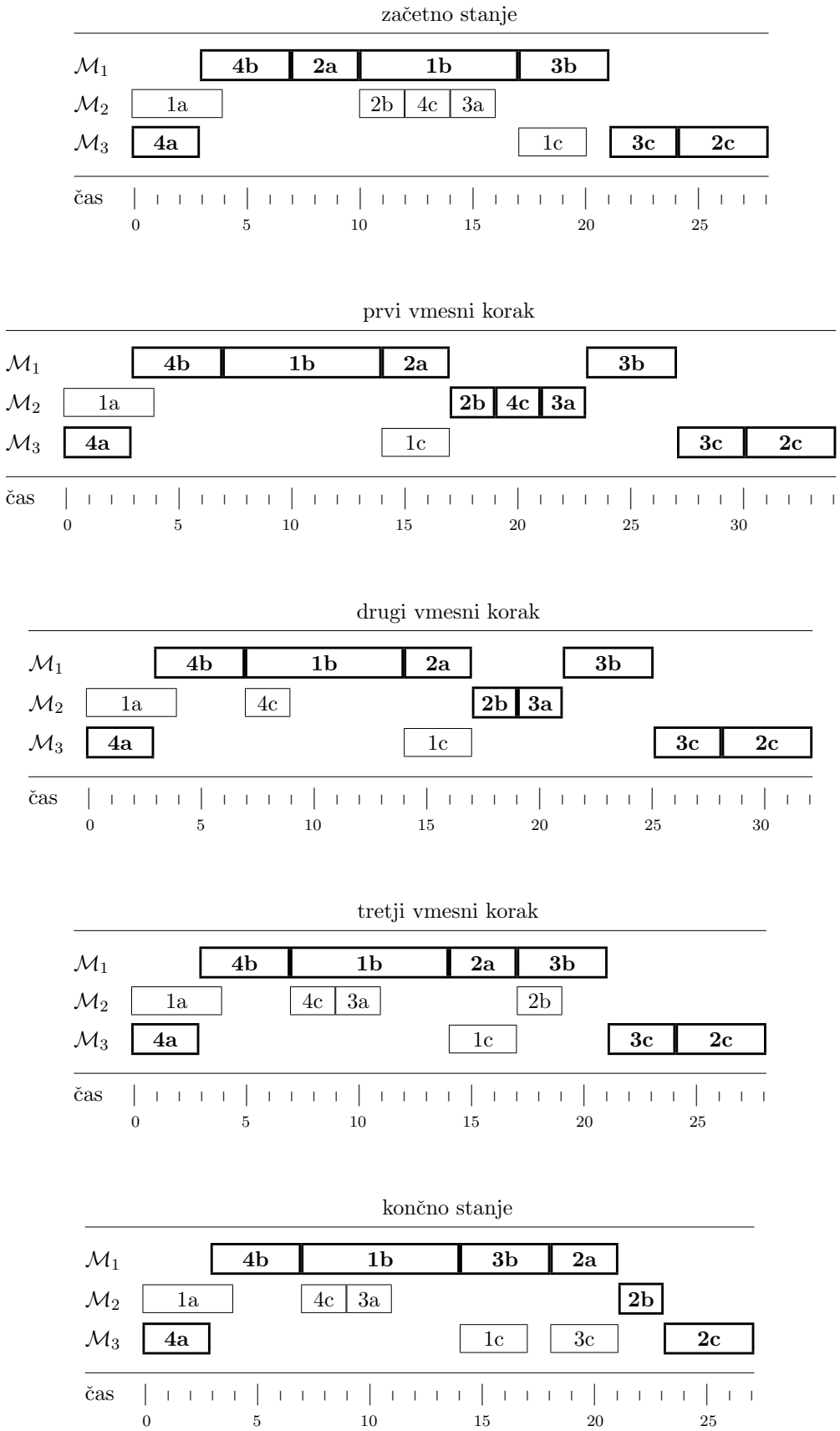
5.1.1 Slabosti okolice $\mathcal{H}_B$

Okolica $\mathcal{H}_B$ vsebuje bistveno več premikov od okolice $\mathcal{H}_N$ in izkazuje lastnost povezljivosti, zato ji ne moremo očitati, da je premajhna. Njena izrazito slaba lastnost je doseganje povezljivosti z množico premikov, ki za optimizacijo niso najbolj primerni. Videli smo, da postopek lokalnega iskanja izbira premike iz okolice v odvisnosti od dejanskih ali ocenjenih izvršnih časov urnika. Tu nastopi težava, ker zaradi veljavnosti teorema 4 (poglavje 3.7.2) noben premik v množici $\mathcal{H}_B \backslash \mathcal{H}_N$ ne more zmanjšati izvršnega časa sam zase. Posledica je, da so premiki v tej skupini pri izvajanju optimizacije potisnjeni v ozadje. Še zdaleč pa ta množica premikov ni nekoristna, saj brez nje v večini primerov optimalnih ali blizu-optimalnih rešitev ne moremo doseči, četudi poznamo najboljše zaporedje izvajanja premikov, ki obstaja.

Druga šibka točka okolice $\mathcal{H}_B$ je, da postopek lokalnega iskanja, ki je zasnovan na njeni podlagi, ni usmerjen v premike ne-prvih (ne-zadnjih) operacij v kritičnih blokih na začetek (konec) kritičnega bloka, kar bi moralo držati zaradi veljavnosti teorema 4. Če bi postopek vseeno uspeli zasnovati na način, ki to dejstvo upošteva, bi za dosego cilja potrebovali nesmotrno veliko premikov zaradi vsakokratnega preračunavanja glav in repov operacij ter iskanja kritične poti v grafu, kar pomeni počasno izvajanje optimizacije. Za ilustracijo si oglejmo naslednji primer na sliki 5.1, kjer želimo v začetnem urniku premakniti operacijo $2a$ na stroju $\mathcal{M}_1$ za operacijo $3b$. S pomočjo slike 5.1, kjer je prikazan podroben potek dogajanja, se lahko prepričamo, da je potrebnih premikov nepričakovano veliko.

Obravnavani premik sam zase povzroči neizvedljivost urnika iz istega razloga, kot se to zgodi pri premiku operacije $3b$ pred operacijo $2a$ (poglavje 2.1); zato so pri izvedbi premika potrebni vmesni koraki. V izhodiščnem urniku vsebuje okolica $\mathcal{H}_B$ naslednje zamenjave: $Q(4b, 2a)$, $Q(2a, 1b)$, $Q(1b, 3b)$ in $Q(3c, 2c)$. K cilju se približamo z izvedbo zamenjave $Q(2a, 1b)$, s čimer pridemo do prvega vmesnega koraka. Novo nastala kritična pot poteka preko operacij $2b$ in $3a$, zaradi česar se neizvedljiva zamenjava $Q(2a, 3b)$ ne nahaja v okolici. Če želimo operacijo $2a$ približati operaciji $3b$, s ciljem da bomo zamenjali njun vrstni red, nam ne preostane drugega, kot da premaknemo operacijo $3a$ v levo ali operacijo $2b$ v desno. Odločimo se za slednjo pot in izvedemo zamenjavo $Q(2b, 4c)$ ter tako dosežemo drugi vmesni korak. Operaciji $3b$ in $2a$ še vedno nista sosednji na kritični poti, zato nadaljujemo s premikom operacije $2b$ v desno; izvedemo zamenjavo $Q(2b, 3a)$, s čimer smo dobili urnik tretjega vmesnega koraka. Operaciji $2a$ in $3b$ sta sedaj sosednji na kritični poti, zato je njuna zamenjava varna in ne more povzročiti neizvedljivega urnika.

Zgled nazorno demonstrira, kako sprememba kritične poti ob vmesnih premikih poskrbi, da operaciji, katerih zamenjava bi povzročila neizvedljivost urnika, nista na njej nikoli sosednji; s površnim pogledom na urnik v začetnem stanju, bi sklepali, da moramo opraviti samo zamenjavi $Q(2a, 1b)$ in nato $Q(2a, 3b)$, vendar je po prvi zamenjavi nastal med operacijama $2a$ in $3b$ nov kritični blok $\{2b, 4c, 3a\}$, ki je neizvedljivo zamenjavo preprečil.



Slika 5.1: Premik operacije $2a$ za operacijo $3b$ z uporabo okolice $\mathcal{H}_B$.

Veliko število vmesnih premikov ima za posledico neučinkovito izvajanje optimizacije. Pri tipični implementaciji lokalnega iskanja so posamezne iteracije med seboj neodvisne, zgled pa nazorno kaže, da bi morale delovati sinergično. V dejanskih razmerah nimamo nobenega zagotovila, da bo postopek optimizacije opravil celotno zaporedje premikov, ki je prikazano. Bolj realistična je situacija, kjer v katerem od vmesnih korakov dobi nek nekorelirani premik boljšo oceno in je zaradi tega izbran za izvedbo, s tem pa postopek optimizacije zavije na stransko pot, kjer prvotna zamenjava $Q(2a, 1b)$ ne igra nobene vloge in v veliko primerih pomeni izgubo časa.

Nadalje lahko ugotovimo, da ima končni urnik krajši izvršni čas, urnik v prvem vmesnem koraku pa daljšega v primerjavi z začetnim urnikom. Zato je malo verjetno, da bi se postopek lokalnega iskanja sploh odločil za zamenjavo $Q(2a, 1b)$, čeprav preko nje lahko dosežemo boljši urnik[1]. Zaradi kratkovidnosti delovanja bi do izvedbe te ali podobne zamenjave prišlo šele, ko bi bile ostale možnosti izčrpane; na primer, ko bi imeli vsi premiki v množici premikov $\mathcal{H}_N$ tabu status.

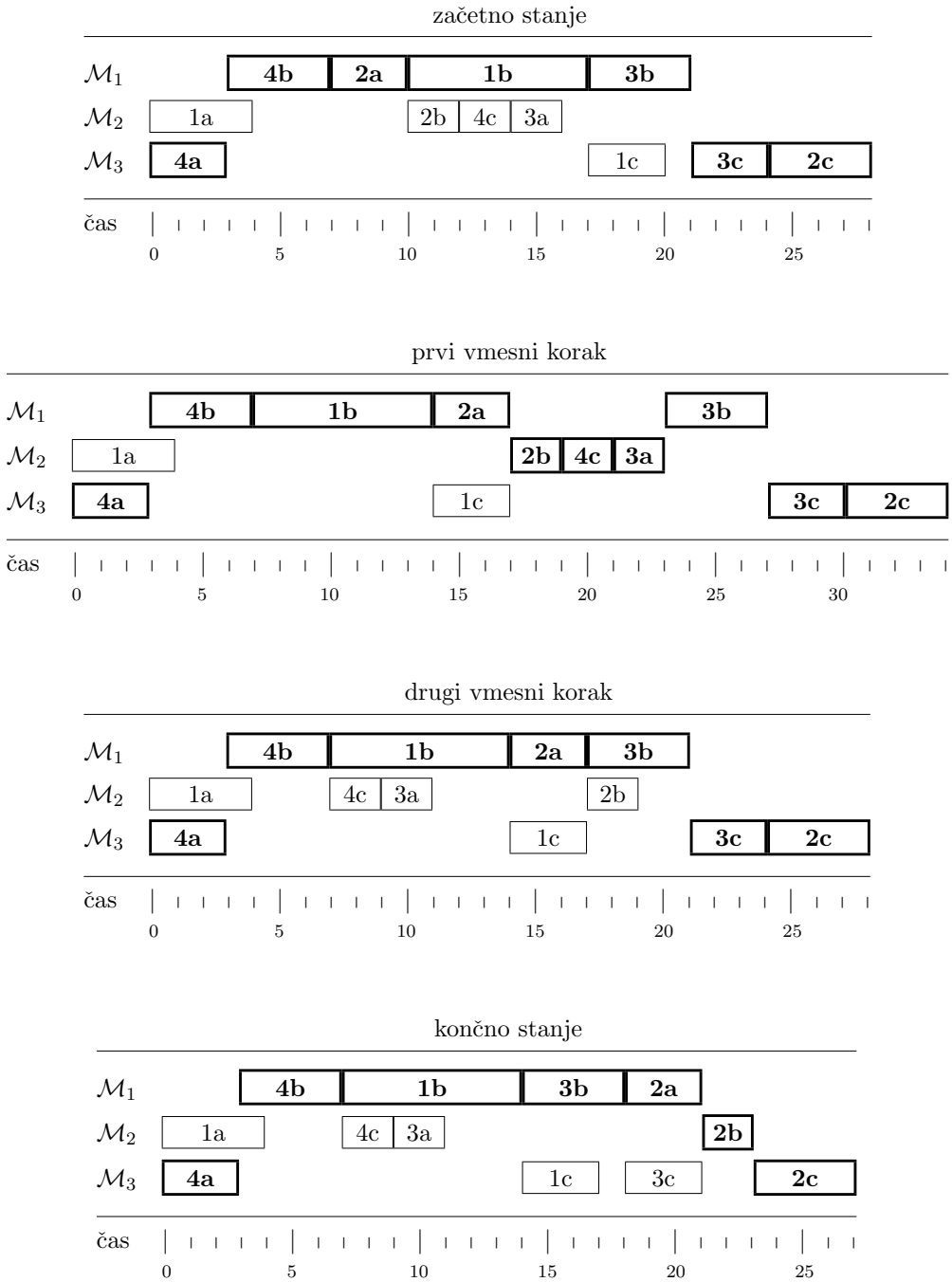
Dejstvo, da noben premik v množici $\mathcal{H}_B \backslash \mathcal{H}_N$ ne more zmanjšati izvršnega časa sam zase, ima za posledico, da je okolica $\mathcal{H}_B$ močno podvržena problemu lokalnih minimumov, iz katerih je pobeg težaven, ker postopek optimizacije nima jasne predstave o tem, na kakšen način naj se izvršni čas zmanjša.

5.1.2 Prednosti in slabosti okolice $\mathcal{H}_D$

Okolica $\mathcal{H}_D$ rešuje problem neizvedljivosti bistveno učinkoviteje od okolice $\mathcal{H}_B$. Ker je tendenca vsakega vsebovanega premika umik določene kritične operacije izven kritičnega bloka, so vsi premiki v zasnovi napredujoči. V primeru, da premik ni izvedljiv, se le-tega omeji na mesto, kjer je izvedljivost ohranjena. Tak premik izgubi možnost, da bi sam zase zmanjšal izvršni čas urnika, zato zanj veljajo enake ugotovitve, kot smo jih v prejšnjem razdelku navedli za premike v množici $\mathcal{H}_B \backslash \mathcal{H}_N$. Izkaže se, da je takih premikov bistveno manj kot v primeru okolice $\mathcal{H}_B$, zaradi česar je tudi uspeh lokalnega iskanja z okolico $\mathcal{H}_D$ večji.

[1] V tem primeru je sicer boljša zamenjava $Q(4b, 2a) \in \mathcal{H}_N$, ki sama zase zmanjša izvršni čas urnika, vendar je to zgolj posledica dejstva, da se izhodiščni urnik ne nahaja v lokalnem minimumu glede na okolico $\mathcal{H}_N$; v nasprotnem primeru tak premik ne bi obstajal.

Vrnimo se na predhodno obravnavani primer, kjer želimo v začetnem urniku izvesti premik operacije $2a$ za operacijo $3b$. Slika 5.2 prikazuje potek dogajanja pri doseganju zastavljenega cilja.



Slika 5.2: Premik operacije $2a$ za operacijo $3b$ z uporabo okolice $\mathcal{H}_D$.

Okolica $\mathcal{H}_D$, ki pripada začetnemu urniku, je v osnovi in brez upoštevanja neizvedljivosti premikov naslednja: $Q_{\mathrm{L}}(2a, 4b)$, $Q_{\mathrm{L}}(1b, 4b)$, $Q_{\mathrm{L}}(3b, 4b)$, $Q_{\mathrm{D}}(1b, 3b)$, $Q_{\mathrm{D}}(2a, 3b)$, $Q_{\mathrm{D}}(4b, 3b)$ in $Q(3c, 2c)$. Premika $Q_{\mathrm{D}}(2a, 3b)$ in $Q_{\mathrm{L}}(3b, 4b)$ sta neizvedljiva zaradi medsebojnega odnosa operacij $2b$ in $3a$ na stroju $\mathcal{M}_2$; ta premika spremenimo v $Q_{\mathrm{D}}(2a, 1b)$ in $Q_{\mathrm{D}}(3b, 1b)$, s čimer so vsi premiki v okolici izvedljivi. Spremenjena premika sama zase nista potencialno napredujoča, ostali pa to lastnost imajo. V okolici $\mathcal{H}_D$ začetnega urnika se tako nahaja pet potencialno napredujočih in dva nenapredujoča premika. Za okolico $\mathcal{H}_B$ bi podobno ugotovili, da vsebuje tri potencialno napredujoče in en nenapredujoči premik.

Na prvi pogled okolica $\mathcal{H}_B$ ne zaostaja bistveno za okolico $\mathcal{H}_D$, saj je razmerje napredujočih premikov proti nenapredujočim celo bolj ugodno zanjo. Vendar je treba upoštevati, da pri večjih instancah problema postanejo kritični bloki daljši, s tem pa okolica $\mathcal{H}_B$ pridobiva samo nenapredujoče premike, medtem ko se pri okolici $\mathcal{H}_D$ razmerje ne spreminja bistveno. Kljub temu je bolj važno samo število napredujočih premikov, saj večja množica le-teh pomeni manj lokalnih minimumov s stališča implementirane okolice, zato je tudi v našem primeru, ki obravnava pretirano majhno instanco problema, okolica $\mathcal{H}_D$ brezpogojno boljša od okolice $\mathcal{H}_B$.

Zasledujmo sedaj dogajanje na urniku pri izvajanju zastavljenega cilja. Prvi vmesni korak dosežemo z izvedbo premika $Q_{\mathrm{D}}(2a, 1b)$, kar je povsem enaka sprememba na urniku, kot smo jo naredili z uporabo okolice $\mathcal{H}_B$. V naslednjem koraku pa opazimo pomembno razliko. Nova okolica vsebuje premik $Q_{\mathrm{D}}(2b, 3a)$, s čimer odpade vmesni korak $Q_{\mathrm{D}}(2b, 4c)$. Tokrat smo prihranili samo en premik, vendar bi v primeru večjih instanc z večjimi kritičnimi bloki razlika postala močno opazna, saj lahko operacija ob uporabi okolice $\mathcal{H}_D$ v najboljšem primeru preskoči celotni kritični blok, pri okolici $\mathcal{H}_B$ pa mora vedno preskakovati po eno operacijo.

Kljub nespornim prednostim okolice $\mathcal{H}_D$ pred okolico $\mathcal{H}_B$ in ostalimi omenjenimi okolicami, problem neizvedljivosti še vedno ni rešen. Prva težava nastane pri detekciji neizvedljivosti premikov, ki jo moramo nujno opraviti pri vsakokratni izgradnji okolice. Za vsak premik moramo ugotovili, ali je izvedljiv in če ni, do katerega mesta ga moramo omejiti, da se mu izvedljivost vrne.

Eksaktno ugotavljanje izvedljivosti oziroma neizvedljivosti premika $Q_{\mathrm{D}}(u, v)$ ali $Q_{\mathrm{L}}(u, v)$ zahteva časovno kompleksnost $O(|\mathcal{T}(u, v)|)$, za kar sta Dell'Amico in Trubian (1993) ugotovila, da je prepotratno. Namesto tega sta razvila hitro oceno neizvedljivosti, ki včasih povsem izvedljiv premik prepozna kot neizvedljivega. To dejstvo že samo zase govori, da problem neizvedljivosti z uporabo okolice $\mathcal{H}_D$ še zdaleč ni rešen.

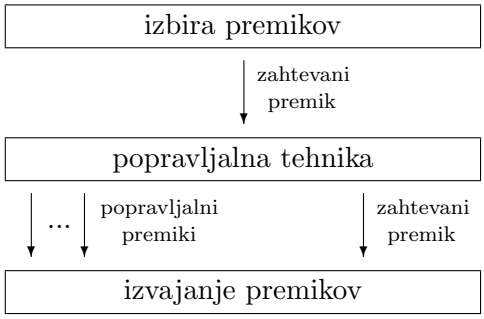
5.2 Ideja popravljalne tehnike

Opazovanje dogajanja ob zadnjih dveh primerih premikanja kritične operacije izven kritičnega bloka nakazuje vsaj eno možnost izboljšave dosedanjih postopkov lokalnega iskanja, ki so zasnovani na dvonivojski ureditvi, kakor je prikazano na sliki 5.3.

Slika 5.3: Dvonivojska zasnova lokalnega iskanja.

Višji nivo izbira premike v okolici, ki se na urniku izvršijo, nižji nivo pa izbrane premike dejansko izvede. Problem take zasnove je v nezmožnosti ustreznega reagiranja ob uporabi neizvedljivih premikov, kjer je potrebno, kakor smo predhodno videli, izvršiti daljšo sekvenco premikov, ki med seboj niso neodvisni. Tipičen scenarij pri uporabi okolice $\mathcal{H}_B$ je, da postopek lokalnega iskanja prične pomikati določeno operacijo proti robu bloka, v naslednji iteraciji pa zastavljenega cilja ne nadaljuje, ampak izbere naslednji premik iz novo nastale okolice neodvisno od trenutnega stanja pričete sekvence sprememb.

Opisano situacijo lahko izboljšamo z uvedbo dodatnega člena med opisana nivoja, kar je prikazano na sliki 5.4.



Slika 5.4: Zasnova lokalnega iskanja z uporabo popravljalne tehnike.

Dodani člen imenujemo *popravljalna tehnika*. Njena naloga je iz višjega nivoja sprejeti navodilo o izvedbi kateregakoli premika (izvedljivega ali neizvedljivega) in posredovati nižjemu nivoju ustrezno sekvenco premikov, med katerimi je tudi originalno zahtevani premik. Ostali premiki so popravljalni. Izvršiti jih moramo v primeru neizvedljivega zahtevanega premika, da le-temu povrnemo izvedljivost.

Na ta način odpade vsaka obremenitev višjega nivoja s problemom izvedljivosti. Pri definiciji okolice ni več potrebno odstranjevati neizvedljivih premikov ali jih omejevati v delovanju. Vsakič, ko se izvede zahtevani premik skupaj z dodatno sekvenco popravljalnih premikov, katerih se višji nivo sploh ne zaveda, se urnik nahaja v končnem zahtevanem stanju. Če je okolica pravilno definirana, bo postopek lokalnega iskanja vedno usmerjen v pomikanje kritičnih operacij izven kritičnih blokov in se nikoli ne bo mogel med začeto sekvenco izgubiti v množici ostalih premikov, ki se ponujajo kot obetavni.

Za uspešno implementacijo popravljalne tehnike morata biti izpolnjena dva ključna pogoja. Prvič, obstajati mora dovolj učinkovit način eksaktne detekcije neizvedljivosti premikov, s pomočjo katerega ugotovimo, kdaj so popravljalni premiki potrebni. Obstoj takega postopka ni sam po sebi umeven, saj smo že omenili, da sta se Dell'Amico in Trubian (1993) raje odločila za hitrejši aproksimativni postopek detekcije neizvedljivosti, ker je v nasprotnem primeru njun optimizacijski postopek potekal prepočasi.

Drugič, obstajati mora učinkovit postopek izbiranja popravljalnih premikov, ki se v našem predlogu določajo na nivoju popravljalne tehnike in ne več na nivoju lokalnega iskanja, s čimer tudi niso več vezani na določeno okolico. Izbrani so

neodvisno od tega, katero množico premikov uporablja višji nivo lokalnega iskanja pri izvajanju optimizacije. S tem imamo možnost uvesti v lokalno iskanje dodatno prednost, saj so popravljalni premiki konceptualno drugačni od optimizacijskih, kar je bilo pri dosedanjih postopkih spregledano. Cilj le-teh ni več premik kritičnih operacij izven kritičnih blokov, ampak odpravljanje poti v kompletni izbiri, ki preprečujejo izvedljivost zahtevanega premika. Zato pri teh premikih operacij ni potrebno umikati izven kritičnih blokov, ampak samo toliko, da preskočijo ustrezne tehnološke predhodnice ali naslednice operacije, kateri zahtevani premik spreminja položaj. Na ta način se ob izvedbi zahtevanih premikov urnik manj spreminja, s čimer smo izboljšali lastnost koreliranosti okolice (poglavje 4.6.1).

V nadaljevanju bomo pokazali, da je možno oba pogoja uspešno izpolniti. Uporabo popravljalne tehnike bomo utemeljili tako teoretično, kakor tudi na podlagi empiričnih rezultatov.

6. Implementacija popravljalne tehnike

V prejšnjem poglavju smo opisali zamisel popravljalne tehnike in nakazali vsaj nekatere možnosti izboljšav postopkov lokalnega iskanja z njeno uporabo. Na tem mestu bomo predstavili implementacijo popravljanja urnikov, ki jo predlagamo za uporabo v praksi.

6.1 Detekcija ciklov

Detekcija ciklov, ki jo opisuje poglavje 6.1, ne predstavlja izvirnega prispevka. Klub temu predstavlja ta tematika temelje na katerih je implementirana popravljalna tehnika, zato jo obravnavamo na tem mestu.

Omenili smo že, da sta Dell'Amico in Trubian (1993) razvila hitro aproksimativno metodo napovedovanja ciklov, ki je sposobna v času $O(|\mathcal{O}(u,v)|)$ napovedati neizvedljivost premika pred njegovo dejansko izvedbo. Njuna metoda neizvedljivega premika nikoli ne prepozna kot izvedljivega, zato je varna za uporabo; njena slabost pa je, da v nekaterih primerih povsem izvedljiv premik prepozna za neizvedljivega, s čimer je moč lokalnega iskanja zmanjšana. Naš cilj je razviti splošno uporabno popravljalno tehniko, ki ni vezana na določeno definicijo okolice, zato se moramo odpovedati hitrim ocenam neizvedljivosti. Ostati moramo pri eksaktnih metodah, kjer je neizvedljivost premikov $Q_{\mathrm{D}}(u,v)$ ali $Q_{\mathrm{L}}(u,v)$ možno napovedati v času $O(|\mathcal{T}(u,v)|)$.

Popravljalna tehnika temelji na odkritju, da je možno združiti tri postopke, ki jih moramo ob izvajanju premikov opraviti, v enega samega; to so detekcija ciklov, iskanje popravljalnih premikov in preurejanje topološkega zaporedja.

6.1.1 Preureditev topološkega zaporedja

Vpeljimo naslednje oznake. Urnik pred izvedbo premika ali urniku pripadajočo kompletno izbiro označimo z že ustaljenim simbolom $\mathcal{G}$. Urnik po izvedbi premika ali njemu pripadajoča kompletna izbira ima oznako $\mathcal{G}^{\prime}$. Topološko zaporedje po izvedbi premika ponazarja simbol $\mathcal{T}(\mathcal{G}^{\prime})$.

V našem primeru poteka preureditev topološkega zaporedja v dveh korakih. Najprej označimo operacije, ki jih je potrebno v $\mathcal{T}(\mathcal{G}^{\prime})$ postaviti na novo mesto. V drugem koraku preureditev označenih operacij dejansko opravimo, v primeru da se premik na urniku resnično izvede.

V primeru izvedbe premika $Q_{\mathrm{D}}(u, v)$ bo operacija u postala neposredna topološka naslednica operacije v. Razlog je v tem, da bo po izvedbi premika veljala zveza $S_{\mathrm{M}}(v) = u$, zaradi česar bo v grafu $\mathcal{G}^{\prime}$ obstajala povezava $v \xrightarrow{\mathcal{E}} u$, ki zahteva, da se operacija u nahaja v $\mathcal{T}(\mathcal{G}^{\prime})$ za operacijo v. Operacija $S_{\mathrm{J}}(u)$ in vse njene naslednice, ki so vsebovane v $\mathcal{T}(u, v)$, se morajo v $\mathcal{T}(\mathcal{G}^{\prime})$ nahajati za operacijo u v enakem vrstnem redu, kot se nahajajo v $\mathcal{T}(\mathcal{G})$. Premik $Q_{\mathrm{D}}(u, v)$ namreč ne odpravi permanentne povezave $u \xrightarrow{\mathcal{A}} S_{\mathrm{J}}(u)$, poleg tega ne spremeni odnosa med operacijo $S_{\mathrm{J}}(u)$ in njenimi naslednicami, zato se mora v $\mathcal{G}^{\prime}$ ohraniti obstoj poti od $S_{\mathrm{J}}(u)$ do vsake njene naslednice, ki se nahaja v $\mathcal{T}(u, v)$. Operacija $S_{\mathrm{M}}(u)$ in njene naslednice, ki niso hkrati naslednice operacije $S_{\mathrm{J}}(u)$, morajo v $\mathcal{T}(\mathcal{G}^{\prime})$ ohraniti staro pozicijo, da odražajo novo zaporedje izvajanja na stroju, kjer se vse operacije $\mathcal{O}(u, v) \backslash \{u\}$ izvedejo v $\mathcal{G}^{\prime}$ pred operacijo u.

S podobnim razmišljanjem ugotovimo, da se mora v primeru izvedbe premika $Q_{\mathrm{L}}(u, v)$ operacija v nahajati v $\mathcal{T}(\mathcal{G}^{\prime})$ pred operacijo u. Operacija $P_{\mathrm{J}}(v)$ in vse njene predhodnice, ki pripadajo $\mathcal{T}(u, v)$, se morajo v $\mathcal{T}(\mathcal{G}^{\prime})$ nahajati pred v v enakem zaporedju, kot so se pred premikom nahajale v $\mathcal{T}(\mathcal{G})$.

Označitev operacije je v praksi izvedena z nastavitvijo vrednosti ustrezne binarne spremenljivke, ki operaciji pripada. Kljub temu v tem delu opisujemo označevanje operacije kot uvrščanje le-te v množico označenih operacij $\mathcal{L}$, s čimer dosežemo nazornejšo diskusijo. Poudarimo naj, da se testiranje označenosti operacije izvrši v času $O(1)$ na podlagi izvedbe preprostega logičnega pogoja, ki testira vrednost ustrezne binarne spremenljivke, in ne v času $O(|\mathcal{L}|)$ ali mogoče $O(\lceil \log_2 |\mathcal{L}| \rceil)$, ki bi bil potreben za preiskavo množice $\mathcal{L}$.

Predhodno opisano množico operacij, katere moramo v $\mathcal{T}(\mathcal{G}^{\prime})$ premakniti, označimo z algoritmom 1, ki ga podaja slika 6.1.

Algoritem 1. Označevanje operacij

pri izvedbi premika $Q_{\mathrm{D}}(u, v)$
1. $a := S_{\mathrm{J}}(u)$; $\mathcal{L} := \{u, a\}$;
2. **repeat**
3. **if** $a \in \mathcal{L}$ **then**
4. $\mathcal{L} := \mathcal{L} \bigcup \{S_{\mathrm{J}}(a), S_{\mathrm{M}}(a)\}$;
5. **end if**
6. $a := S_{\mathrm{T}}(a)$;
7. **until** $a = v$;
8. **if** $\circledast \in \mathcal{L}$ **then** $\mathcal{L} := \mathcal{L} \backslash \{\circledast\}$;

pri izvedbi premika $Q_{\mathrm{L}}(u, v)$
1. $a := P_{\mathrm{J}}(v)$; $\mathcal{L} := \{v, a\}$;
2. **repeat**
3. **if** $a \in \mathcal{L}$ **then**
4. $\mathcal{L} := \mathcal{L} \bigcup \{P_{\mathrm{J}}(a), P_{\mathrm{M}}(a)\}$;
5. **end if**
6. $a := P_{\mathrm{T}}(a)$;
7. **until** $a = u$;
8. **if** $\circledcirc \in \mathcal{L}$ **then** $\mathcal{L} := \mathcal{L} \backslash \{\circledcirc\}$;

Slika 6.1: Algoritem označevanja operacij, ki jih je potrebno v $\mathcal{T}(\mathcal{G}^{\curvearrowright})$ preurediti.

Algoritem 1 označi vse operacije, za katere smo predhodno ugotovili, da morajo v $\mathcal{T}(\mathcal{G}^{\curvearrowright})$ zamenjati položaj. O pravilnosti te trditve se lahko prepričamo s pomočjo naslednjega teorema.

Teorem 7. *V primeru izvedbe desnega premika* $Q_{\mathrm{D}}(u, v)$ *označi algoritem 1 vse naslednice operacije* $S_{\mathrm{J}}(u)$*, ki se nahajajo na seznamu* $\mathcal{T}(u, v)$*. Pri izvedbi levega premika* $Q_{\mathrm{L}}(u, v)$ *so označene vse predhodnice operacije* $P_{\mathrm{J}}(v)$*, ki pripadajo seznamu* $\mathcal{T}(u, v)$*.*

DOKAZ. V primeru premika $Q_{\mathrm{D}}(u, v)$ je operacija $S_{\mathrm{J}}(u)$ označena v koraku 1, zaradi česar sta njena neposredna tehnološka naslednica in njena neposredna naslednica na stroju označeni v koraku 4 ob prvem prehodu skozi **repeat** zanko. Ker sta to edini neposredni naslednici operacije $S_{\mathrm{J}}(u)$, velja trditev, da so v tem

trenutku vse naslednice operacije $S_{\text{J}}(u)$ na seznamu $\mathcal{T}(u, a)$ označene. V koraku 6 postopek nadaljuje preiskovanje seznama $\mathcal{T}(u, v)$ s premikom delovne spremenljivke a na naslednje mesto v $\mathcal{T}(u, v)$. Takoj ko postopek naleti na označeno operacijo, v naslednjem prehodu skozi **repeat** zanko označi obe njeni neposredni naslednici. Tako velja trditev, da so kadarkoli med izvajanjem algoritma 1 označene vse naslednice operacije $S_{\text{J}}(u)$, ki se nahajajo v $\mathcal{T}(u, a)$. Ko je izpolnjen terminacijski pogoj **repeat** zanke, je operacija a enaka operaciji v, kar pomeni, da so v tem trenutku označene vse naslednice operacije $S_{\text{J}}(u)$, ki se nahajajo v $\mathcal{T}(u, v)$. Dokaz v primeru premika $Q_{\text{L}}(u, v)$ je povsem analogen. □

V koraku 8 odstrani algoritem 1 oznako ustrezni fiktivni operaciji $\circledcirc$ ali $\circledast$, kar je potrebno za pravilno delovanje nadaljnjih algoritmov.

Ob izvedbi zahtevanega ali popravljalnega premika ni dovolj, da operacije označimo za preurejanje, ampak moramo preureditev dejansko opraviti. To storimo z algoritmom 2 na sliki 6.2. Algoritem je zelo preprost, zato ga na tem mestu ne bomo podrobno razlagali, saj brez težav vidimo, da se v primeru desnega (levega) premika, z njim premaknejo vse označene operacije v $\mathcal{T}(u, v)$ za operacijo v (pred operacijo u) v enakem zaporedju, kot so se prvotno nahajale v $\mathcal{T}(\mathcal{G})$.

6.1.2 Detekcija ciklov na podlagi označenih operacij

Oznake operacij, katerih namen je preureditev topološkega zaporedja, lahko uporabimo za eksaktno detekcijo izvedljivosti zahtevanega premika s časovno kompleksnostjo $O(1)$. Postopek detekcije podajata naslednja dva teorema.

Teorem 8. *Vse operacije, ki pripadajo seznamu $\mathcal{O}(u, v)$, pripadajo tudi seznamu $\mathcal{T}(u, v)$.*

DOKAZ. Ker velja zveza $\mathcal{O}(u, v) \in \mathcal{O}_i$, obstaja povezava $u \xrightarrow{\mathcal{E}} x$ za vsako operacijo $x \in \mathcal{O}(u, v) \backslash \{u\}$. Operacija u se zato v $\mathcal{T}(\mathcal{G})$ nahaja pred katerokoli operacijo x. Iz razloga $\mathcal{O}(u, v) \in \mathcal{O}_i$ obstaja tudi povezava $y \xrightarrow{\mathcal{E}} v$ za vsako operacijo $y \in \mathcal{O}(u, v) \backslash \{v\}$. Operacija v se zato v $\mathcal{T}(\mathcal{G})$ nahaja za katerokoli operacijo y. Če obe ugotovitvi združimo, vidimo, da se v nahaja v $\mathcal{T}(\mathcal{G})$ za u, vse operacije $\mathcal{O}(u, v) \backslash \{u, v\}$ pa se v $\mathcal{T}(\mathcal{G})$ nahajajo med njima. □

Algoritem 2. Preureditev $\mathcal{T}(\mathcal{G})$

pri izvedbi premika $Q_{\mathrm{D}}(u, v)$

1. $a := u$; $b := v$;
2. **while** $a \neq v$
3. **if** $a \in \mathcal{L}$ **then**
4. $n := S_{\mathrm{T}}(a)$;
5. izbriši a iz $\mathcal{T}(\mathcal{G})$;
6. vstavi a v $\mathcal{T}(\mathcal{G})$ za b;
7. $b := a$; $a := n$;
8. **else**
9. $a := S_{\mathrm{T}}(a)$;
10. **end if**
11. **end while**

pri izvedbi premika $Q_{\mathrm{L}}(u, v)$

1. $a := v$; $b := u$;
2. **while** $a \neq u$
3. **if** $a \in \mathcal{L}$ **then**
4. $n := P_{\mathrm{T}}(a)$;
5. izbriši a iz $\mathcal{T}(\mathcal{G})$;
6. vstavi a v $\mathcal{T}(\mathcal{G})$ pred b;
7. $b := a$; $a := n$;
8. **else**
9. $a := P_{\mathrm{T}}(a)$;
10. **end if**
11. **end while**

Slika 6.2: Algoritem za preureditev topološkega zaporedja.

Naslednji teorem predstavlja potrebno teoretično osnovo za eksaktno detekcijo ciklov, ki je sestavni del predlagane popravljalne tehnike.

Teorem 9. *Če nad urnikom $\mathcal{G}$ izvedemo premik $Q_{\mathrm{D}}(u, v)$, bo rezultirajoči urnik $\mathcal{G}^{\Pi}$ neizvedljiv takrat in samo takrat, ko algoritem 1 označi operacijo v. V primeru premika $Q_{\mathrm{L}}(u, v)$ velja ista trditev za operacijo u.*

DOKAZ. V primeru izvedbe premika $Q_{\mathrm{D}}(u, v)$ bo v $\mathcal{G}^{\prime}$ obstajal cikel takrat in samo takrat, ko obstaja v grafu $\mathcal{G}$ pot od operacije $S_{\mathrm{J}}(u)$ do $P_{\mathrm{J}}(x)$ za katerokoli operacijo $x \in \mathcal{O}(u, v) \backslash \{u\}$ (teorem 6, poglavje 3.7.2). Obstoj take poti pomeni, da je operacija $P_{\mathrm{J}}(x)$ naslednica operacije $S_{\mathrm{J}}(u)$. Zaradi veljavnosti teorema 7 algoritem 1 označi vse naslednike operacije $S_{\mathrm{J}}(u)$, ki se nahajajo v $\mathcal{T}(u, v)$. Sledi, da so zaradi veljavnosti teorema 8 označeni vsi nasledniki operacije $S_{\mathrm{J}}(u)$, ki se nahajajo v $\mathcal{O}(u, v)$. V primeru da za katerikoli x velja $P_{\mathrm{J}}(x) \in \mathcal{L}$, mora nujno veljati tudi $x \in \mathcal{L}$, ker je x neposredni tehnološki naslednik operacije $P_{\mathrm{J}}(x)$, zaradi česar je x tudi naslednik operacije $S_{\mathrm{J}}(u)$ (poglavje 3.5). Ker je operacija v enaka operaciji x, ali pa je njena naslednica na stroju, velja, da je v tudi naslednica operacije $S_{\mathrm{J}}(u)$, iz česar sledi $v \in \mathcal{L}$. Če v grafu $\mathcal{G}$ ne obstaja pot od $S_{\mathrm{J}}(u)$ do nobene operacije $P_{\mathrm{J}}(x)$, tudi obstoj poti od $S_{\mathrm{J}}(u)$ do v ni mogoč, ker velja zveza $\mathcal{J}_{\mathrm{v}} \neq \mathcal{J}_{\mathrm{u}}$. Analogen dokaz lahko izvedemo tudi za premik $Q_{\mathrm{L}}(u, v)$. □

Pri uporabi predloga Dell'Amico in Trubian (1993) je eksaktno ugotavljanje izvedljivosti premikov z algoritmom 1 časovno potratno, ker ga moramo izvesti za vsak premik v okolici in to v vsaki iteraciji, saj se po vsaki izvedbi premika na urniku okolica zgradi na novo. V primeru, da je premik neizvedljiv, bi bila zanj potrebna dodatna izvajanja algoritma 1, da bi ugotovili, na katero mesto lahko premikajočo operacijo varno premaknemo.

Z uporabo popravljalne tehnike to odpade, saj izvedljivosti premikov v okolici ni potrebno preverjati vnaprej. Šele ko se postopek lokalnega iskanja odloči, kateri premik se bo na urniku izvršil, uporabimo algoritem 1, da ugotovimo, katere operacije je pri izvedbi premika potrebno v $\mathcal{T}(\mathcal{G}^{\prime})$ preurediti. Stranski učinek tega je, da lahko povsem eksaktno detektiramo izvedljivost zahtevanega premika z izvedbo preprostega logičnega pogoja, ki testira vsebovanost operacije v ali u v množici $\mathcal{L}$; s tem postane časovna zahtevnost detekcije neizvedljivosti praktično nepomembna.

Oglejmo si primer delovanja algoritma 1 pri premiku operacije $2a$ za operacijo $3b$ na istem urniku, kot smo ga uporabili v poglavjih 5.1.1 in 5.1.2. Topološko zaporedje z označenimi operacijami je prikazano v tabeli 6.1.

1					u		$S_{\mathrm{J}}(u)$			v			
začetno stanje													
2	$\mathcal{T}(\mathcal{G})$	1a	4a	4b	2a	1b	2b	4c	3a	<u>3b</u>	1c	3c	2c
3	oznaka				•		•	•	•	•		∘	∘
4	razlog				u		$S_{\mathrm{J}}(2a)$	$S_{\mathrm{M}}(2b)$	$S_{\mathrm{M}}(2b)$	$S_{\mathrm{J}}(3a)$		$S_{\mathrm{J}}(3b)$	$S_{\mathrm{J}}(2b)$

Tabela 6.1: Potek označevanja operacij z algoritmom 1.

Prva vrsta označuje tri ključne operacije: operacijo $2a$, ki je premikajoča operacija opazovanega premika u, operacijo $2b$, ki je njena tehnološka naslednica $S_{\mathrm{J}}(u)$, ter operacijo $3b$, ki je ciljna operacija premika v. Druga vrsta prikazuje enega od možnih topoloških zaporedij urnika v začetnem stanju. Tretja vrsta indicira operacije, ki jih je algoritem 1 označil; razlog za označevanje operacij je podan v četrti vrsti.

V prvem koraku izvajanja algoritma sta označeni operaciji $2a$ in $2b$. Pri prvem prehodu skozi **repeat** zanko sta nadalje označeni operaciji $2c = S_{\mathrm{J}}(2b)$ in $4c = S_{\mathrm{M}}(2b)$. Opazimo, da je oznaka pod operacijo $2c$ predstavljena s prazno piko, ker se le-ta ne nahaja v $\mathcal{T}(2a, 3b)$ in v nadaljevanju postopka ne igra nobene vloge; razlog za njeno označitev je izključno v tem, da bi preverjanje pripadnosti operacij v množico $\mathcal{T}(u, v)$ predstavljalo nepotrebno porabo časa. V koraku 6 se premaknemo na naslednjo operacijo v $\mathcal{T}(\mathcal{G})$, ki je v tem primeru $4c$ in je označena, kar povzroči označitev $\circledast = S_{\mathrm{J}}(4c)$ (kar ni bistveno, zaradi česar tudi ni prikazana v tabeli) in $3a = S_{\mathrm{M}}(4c)$. Korak 6 nas sedaj premakne na operacijo $3a$, ki je označena, s čimer dobita oznako operaciji $3b = S_{\mathrm{J}}(3a)$ in $\circledast = S_{\mathrm{M}}(3a)$[1]. V koraku 8 odstranimo oznako z operacije $\circledast$.

Vidimo, da je operacija $3b = v$ označena, kar pomeni, da je zahtevani premik neizvedljiv, zato ga ne smemo izvesti, dokler predhodno ne izvedemo dovolj popravljalnih premikov, s katerimi odpravimo moteča razmerja med operacijami na drugih strojih. Popravljalna tehnika, ki jo predlagamo, vedno izvaja popravljalne premike pred zahtevanim, s čimer se izvedljivost urnika ohranja v vsakem koraku. V nasprotnem primeru bi bilo nemogoče zgraditi veljavno topološko zaporedje in predpostavke, zaradi katerih veljajo teoremi od 7 do 9, ne bi bile izpolnjene, s tem pa popravljanje urnikov v nadaljevanju ne bi bilo pravilno.

[1]Večkratno označevanje iste operacije ne predstavlja napake, ker gre za idempotenten postopek.

Na podlagi primera, ki je bil podan v poglavju 2.1, vemo da je vzrok neizvedljivosti zaporedje izvajanja operacij na stroju $\mathcal{M}_2$, kjer se operacija $2b$ izvede pred operacijo $3a$. V grafu $\mathcal{G}$ torej obstaja pot $2b \xrightarrow{\mathcal{E}} 4c \xrightarrow{\mathcal{E}} 3a$ med operacijo $2b = S_{\mathrm{J}}(2a) = S_{\mathrm{J}}(u)$ in operacijo $3a = P_{\mathrm{J}}(3b) = P_{\mathrm{J}}(v)$, kar je po teoremu 6 zadosten pogoj[2] za neizvedljivost obravnavanega premika.

Preden lahko izvedemo zahtevani premik $Q_{\mathrm{D}}(2a, 3b)$, moramo poskrbeti, da se operacija $3a$ izvede pred operacijo $2b$. To lahko storimo z enim od popravljalnih premikov $Q_{\mathrm{D}}(2b, 3a)$ ali $Q_{\mathrm{L}}(2b, 3a)$. Če bi bila operacija $3a$ neposredni naslednik operacije $2b$ na stroju, bi bila oba premika enaka. V našem primeru in v splošnem pa temu ni tako. Pri implementaciji popravljalne tehnike smo uporabili pristop, kjer imajo popravljalni premiki enako smer kakor zahtevani premik, v upanju, da s tem prihranimo delček časa pri večkratnem preurejanju topološkega zaporedja: v primeru, da vsi premiki pomikajo operacije v isto smer, je manj verjetno, da bi neka operacija pri različnih popravljalnih premikih potovala naprej in nazaj. Če bi obstajal boljši razlog za drugačno izbiro smeri popravljalnih premikov, bi zanesljivo imel prednost pred našo izbiro, vendar v tem trenutku takega razloga ne poznamo.

6.2 Izbira popravljalnih premikov

V primeru desnega (levega) premika pomeni označena operacija v (operacija u) neizvedljivost zahtevanega premika. Iz tega sledi, da lahko določimo popravljalne premike na podlagi ugotavljanja vzroka za označitev ustrezne operacije u ali v. Idejo implementira algoritem 3 na sliki 6.3.

V primeru desnega zahtevanega premika $Q_{\mathrm{D}}(u, v)$ je razlaga naslednja. Zasledovanje vzroka pričnemo pri operaciji v, zato jo v koraku 1 priredimo delovni spremenljivki a. Korak 2 preveri označenost operacije $P_{\mathrm{J}}(a)$ z namenom, da ugotovi potrebno smer zasledovanja. Če velja $P_{\mathrm{J}}(a) \in \mathcal{L}$, nadaljujemo zasledovanje preko tehnološke predhodnice operacije a, v nasprotnem primeru pa preko njene predhodnice na stroju.

[2] V tem primeru to ni potreben pogoj, ker bi neizvedljivost obravnavanega premika povzročila tudi pot $2b \xrightarrow{\mathcal{E}} 1a$, če bi obstajala.

Algoritem 3. Popravljalni premiki

pri izvedbi premika $Q_{\mathrm{D}}(u, v)$

1. $a := v$;
2. **if** $P_{\mathrm{J}}(a) \notin \mathcal{L}$ **then go to** 6;
3. **repeat**
4. $\quad a := P_{\mathrm{J}}(a)$;
5. **until** $P_{\mathrm{J}}(a) \notin \mathcal{L}$;
6. $b := a$;
7. **repeat**
8. $\quad a := P_{\mathrm{M}}(a)$;
9. $\quad$ **if** $P_{\mathrm{J}}(a) \in \mathcal{L}$ **then**
10. $\quad\quad$ **if** $a \in \mathcal{J}_{\mathrm{u}}$ **then go to** 15;
11. $\quad\quad$ **go to** 3;
12. $\quad$ **end if**
13. **until** $P_{\mathrm{M}}(a) \notin \mathcal{L}$;
14. **if** $a \notin \mathcal{J}_{\mathrm{u}}$ **then go to** 3;
15. { *popravljalni premik je* $Q_{\mathrm{D}}(a, b)$ }

pri izvedbi premika $Q_{\mathrm{L}}(u, v)$

1. $a := u$;
2. **if** $S_{\mathrm{J}}(a) \notin \mathcal{L}$ **then go to** 6;
3. **repeat**
4. $\quad a := S_{\mathrm{J}}(a)$;
5. **until** $S_{\mathrm{J}}(a) \notin \mathcal{L}$;
6. $b := a$;
7. **repeat**
8. $\quad a := S_{\mathrm{M}}(a)$;
9. $\quad$ **if** $S_{\mathrm{J}}(a) \in \mathcal{L}$ **then**
10. $\quad\quad$ **if** $a \in \mathcal{J}_{\mathrm{v}}$ **then go to** 15;
11. $\quad\quad$ **go to** 3;
12. $\quad$ **end if**
13. **until** $S_{\mathrm{M}}(a) \notin \mathcal{L}$;
14. **if** $a \notin \mathcal{J}_{\mathrm{v}}$ **then go to** 3;
15. { *popravljalni premik je* $Q_{\mathrm{L}}(a, b)$ }

Slika 6.3: Algoritem za izbiro popravljalnih premikov.

Razlog za tako odločitev je naslednji. V primeru da velja $P_{\mathrm{J}}(a) \in \mathcal{L}$, moramo zagotoviti, da ta pogoj preneha delovati: operacija a ne bo označena, le takrat, ko ne bodo označene vse njene predhodnice. Ker je operacija $P_{\mathrm{J}}(a)$ predhodnica operacije a zaradi nespremenljive povezave $P_{\mathrm{J}}(a) \xrightarrow{\mathcal{A}} a$ v grafu $\mathcal{G}$, nam ne preostane drugega, kot da raziščemo in odpravimo vzrok relacije $P_{\mathrm{J}}(a) \in \mathcal{L}$. Koraki od 3 do 5 poskrbijo, da poteka zasledovanje preko tehnoloških predhodnic toliko časa, dokler omenjena relacija velja. Rešitev problema, kako odpraviti relacijo $a \in \mathcal{L}$ v primeru, ko velja $P_{\mathrm{J}}(a) \notin \mathcal{L}$, je naloga preostalega dela algoritma.

Ugotovili smo, da ob izvedbi koraka 6 velja zveza $P_{\mathrm{J}}(a) \notin \mathcal{L}$. Relacija $a \in \mathcal{L}$ je zanesljivo posledica veljavnosti $P_{\mathrm{M}}(a) \in \mathcal{L}$, ker drugih neposrednih predhodnic operacija a nima. V tem primeru oznako lahko odstranimo s spremembo zaporedja izvajanja operacij na stroju. Operacija a je možen kandidat za izvedbo popravljalnega premika $Q_{\mathrm{D}}(?, a)$, zaradi česar jo shranimo v pomožno spremenljivko b.

V koraku 8 nadaljujemo zasledovanje preko predhodnic operacije a na stroju. Ko naletimo na operacijo, za katero velja $P_{\mathrm{J}}(a) \in \mathcal{L}$, testiramo, če pripada istemu opravilu kot operacija u. V tem primeru je označitev operacije v posledica obstoja poti $a \xrightarrow{\mathcal{E}} \cdots \xrightarrow{\mathcal{E}} b$. Popravljalni premik, s katerim motečo relacijo odstranimo, se glasi $Q_{\mathrm{D}}(a, b)$, zato postopek nemudoma skoči na korak 15, kjer se konča.

Če zveza $P_{\mathrm{J}}(a) \in \mathcal{J}_{\mathrm{u}}$ ne velja, bi bilo z izvedbo premika $Q_{\mathrm{D}}(a, b)$ še vedno možno odpraviti razlog (splošneje: enega od razlogov) za relacijo $v \in \mathcal{L}$, vendar takega premika ne storimo. Razlog je v tem, da z vztrajanjem na zahtevi $\mathcal{J}_{\mathrm{a}} = \mathcal{J}_{\mathrm{u}}$ ob izvedbi popravljalnega premika, postane veljaven teorem 10, ki je opisan v nadaljevanju. Izvajanje algoritma preusmerimo na korak 3, kjer ves postopek ponovimo s ciljem ugotoviti in odpraviti razlog za veljavnost relacije $a \in \mathcal{L}$. Vse poteka enako kot v prvem prehodu, le da sedaj operacija a ni enaka v, ampak je predhodnica le-te in zato predstavlja posreden razlog za veljavnost relacije $v \in \mathcal{L}$. Podobna razlaga velja v primeru, ko se na korak 3 vračamo s koraka 14. Takrat zanesljivo veljata zvezi $\mathcal{J}_{\mathrm{a}} \neq \mathcal{J}_{\mathrm{u}}$ in $P_{\mathrm{M}}(a) \notin \mathcal{L}$. Ker je operacija a označena (drugače je z zasledovanjem ne bi mogli doseči), mora nujno veljati $P_{\mathrm{J}}(a) \in \mathcal{L}$, zato lahko postopek varno nadaljujemo s korakom 3.

Povsem analogno razlago bi lahko podali za izvajanje v primeru levega zahtevanega premika $Q_{\mathrm{L}}(u, v)$. Algoritem 3 izkazuje dve pomembni lastnosti, ki ju podajata naslednja teorema.

Teorem 10. *Popravljalni premik, ki ga vrne algoritem 3, je vedno izvedljiv in ga lahko varno izvedemo brez bojazni, da bomo z njim povzročili neizvedljivost urnika.*

DOKAZ. Veljavnost teorema je rezultat dejstva, da se vedno odločimo za zasledovanje oznak preko tehnološke predhodnice (naslednice v primeru levega zahtevanega premika) namesto preko predhodnice na stroju v primeru, ko sta obe predhodnici označeni (pogojna stavka v korakih 2 in 9). Urnik je pred izvedbo popravljalnega premika izvedljiv, ker noben premik še ni bil izveden; do tega trenutka smo opravili samo označevanje operacij. V primeru desnega popravljalnega premika $Q_{\mathrm{D}}(a, b)$ teorem velja, če v grafu $\mathcal{G}$ ne obstaja pot od $S_{\mathrm{J}}(a)$ do katerekoli operacije $P_{\mathrm{J}}(x)$, kjer je $x \in \mathcal{O}(a, b) \backslash \{a\}$. Operacija $S_{\mathrm{J}}(a)$ je tehnološka naslednica operacije u (korak 10 ali 14), kar pomeni, da obstoj poti od $S_{\mathrm{J}}(a)$ do katerekoli operacije $z \in \mathcal{T}(S_{\mathrm{J}}(a), b) \backslash \{S_{\mathrm{J}}(a)\}$ ni možen, če $z \notin \mathcal{L}$, ker sta operaciji a in b predhodnici operacije v ter veljata teorema 7 in 8. Ob izvršitvi koraka 6 velja $P_{\mathrm{J}}(b) \notin \mathcal{L}$, s čimer je onemogočen obstoj poti od operacije $S_{\mathrm{J}}(a)$ do operacije $P_{\mathrm{J}}(b)$. Preden algoritem preneha z izvajanjem, se v koraku 9 zagotovi veljavnost relacije $P_{\mathrm{J}}(y) \notin \mathcal{L}$ za vsak $y \in \mathcal{O}(a, b) \backslash \{a, b\}$. V primeru da naletimo na operacijo, ki tega pogoja ne izpolnjuje, algoritem nemudoma skoči na izvajanje koraka 3, s čimer se izbereta nova kandidata a in b za izvedbo popravljalnega premika $Q_{\mathrm{D}}(a, b)$. To je zadosten pogoj, da obstoj poti od operacije $S_{\mathrm{J}}(a)$ do katerekoli operacije $P_{\mathrm{J}}(x)$ ni možen, kar pomeni, da je premik $Q_{\mathrm{D}}(a, b)$ zanesljivo izvedljiv. Dokaz v primeru izvedbe levega popravljalnega premika je povsem analogen. □

Teorem 11. *V primeru da zahtevani premik $Q_{\mathrm{D}}(u, v)$ vodi v neizvedljivost urnika, zanesljivo velja, da popravljalni premik $Q_{\mathrm{D}}(a, b)$ odpravi vsaj eno relacijo v grafu $\mathcal{G}$, ki preprečuje izvedljivost premika $Q_{\mathrm{D}}(u, v)$.*

DOKAZ. V primeru desnega premika $Q_{\mathrm{D}}(u, v)$ moramo odstraniti relacijo $v \in \mathcal{L}$. Vsakič, ko algoritem 3 izvede korak 6, je operacija b označena predhodnica operacije v. Ker vsaka označena predhodnica operacije v povzroči relacijo $v \in \mathcal{L}$, moramo odstraniti oznako z vsake od njih. Če to uspemo narediti za eno od njih, smo s tem odpravili vsaj eno relacijo v grafu $\mathcal{G}$, ki je povzročila označitev operacije v. Da katerakoli operacija x ne bo označena, morata veljati obe relaciji $P_{\mathrm{J}}(x) \notin \mathcal{L}$ in $P_{\mathrm{M}}(x) \notin \mathcal{L}$. Če velja $P_{\mathrm{M}}(x) \in \mathcal{L}$ in $P_{\mathrm{J}}(x) \notin \mathcal{L}$, lahko odstranimo oznako z operacije x z zamenjavo vrstnega reda izvajanja operacij na stroju. Če velja $P_{\mathrm{J}}(x) \in \mathcal{L}$, je edina možnost za dosego cilja zasledovanje oznak preko tehnoloških predhodnic x (koraki od 3 do 5), dokler ne dosežemo operacije $y \in \mathcal{J}_{\mathrm{x}}$, za katero velja $P_{\mathrm{J}}(y) \notin \mathcal{L}$. Sedaj lahko odstranimo oznako z operacije y s spremembo vrstnega reda izvajanja operacij na stroju $\mathcal{M}_{\mathrm{y}}$. S tem ima operacija x razlog manj, da je označena. Posledica je, da imajo vse naslednice operacije x v $\mathcal{T}(x, v)$, preko katerih smo izvedli zasledovanje oznak, razlog manj za označitev. Ker se je zasledovanje pričelo pri operaciji v, velja ugotovitev tudi zanjo. Dokaz v primeru levega premika je povsem simetričen. □

Teorem 10 zagotavlja, da med izvajanjem popravljalnih premikov ni potrebno rekurzivno klicati algoritma 3 in s tem ni potrebno, da bi bil algoritem 3 implementiran za večkratno sočasno izvajanje (ang. *reentrant*), s čimer lahko pridobimo na hitrosti izvajanja. Med raziskavami smo testirali podoben algoritem, kjer je zasledovanje favoriziralo potovanje po stroju namesto po opravilu; sprememba je povzročila pogoste rekurzivne klice, ki so malenkostno upočasnjevali izvajanje premikov. Teorem 11 nam daje garancijo, da je popravljalni proces vedno končen.

Opazujmo izvajanje algoritma na našem konkretnem primeru. Zasledovanje se prične z operacijo $3b$. Pogoj v koraku 2 ni izpolnjen, ker velja $P_{\mathrm{J}}(3b) = 3a \in \mathcal{L}$ (tabela 6.2), zato nadaljujemo zasledovanje preko operacije $3a$ (korak 4). Neposredna tehnološka predhodnica operacije $3a$ je fiktivni $\circledcirc$, ki ni označen, zato nadaljujemo izvajanje v koraku 6, kjer si operacijo $3a$ zapomnimo v pomožni spremenljivki b. V koraku 8 se premaknemo po stroju do operacije $4c$. Zanjo velja $P_{\mathrm{J}}(4c) = 4b \notin \mathcal{L}$, zato nadaljujemo zasledovanje po stroju. Operacija a je sedaj enaka operaciji $2b$, za katero velja $P_{\mathrm{J}}(2b) = 2a \in \mathcal{L}$; poleg tega je izpol-

njena relacija $2b \in \mathcal{J}_{2a} = \mathcal{J}_u$, s čimer je zadoščeno izstopnemu pogoju algoritma. Popravljalni premik se glasi $Q_D(2b, 3a)$, kot smo pričakovali.

6.3 Celotni postopek izvajanja premikov

Tabela 6.2 prikazuje stanje po izvedbi popravljalnega premika in po ponovni uporabi algoritma 1 za označevanje operacij pri premiku $Q_D(2a, 3b)$. Opazimo, da operacija $3b$ ni več označena, ker je popravljalni premik odpravil motečo relacijo med operacijama $2b$ in $3a$ na stroju $\mathcal{M}_2$. Sedaj lahko zahtevani premik dejansko izvedemo.

Celoten postopek izvajanja premikov s pomočjo popravljalne tehnike, ki smo ga v tem poglavju predstavili, prikazuje algoritem 4 na sliki 6.4. Z njegovo uporabo lahko izvedemo poljuben premik na poljubnem stroju brez bojazni, da bi rezultirajoči urnik postal neizvedljiv. Varne premike, ki se izvajajo s pomočjo popravljalne tehnike, označujemo s $Q^*_D(u, v)$ v primeru desnega premika in s $Q^*_L(u, v)$, ko je premik levi. Če nas smer premika ne zanima, uporabljamo oznako $Q^*(u, v)$.

Odgovorimo na vprašanje, kaj s predlagano popravljalno tehniko pridobimo. V poglavju 5.1.2 smo videli, da sta Dell'Amico in Trubian (1993) uspela definirati okolico urnika, ki ima hkrati lastnost povezljivosti in izvedljivosti; niso pa vsi njeni premiki potencialno napredujoči. Težava njunega pristopa je v tem, da bi moral postopek lokalnega iskanja večkrat opraviti serijo premikov, ki so med seboj semantično povezani; pri tem pa nimamo nikakršnega zagotovila, da se to v resnici zgodi. Poleg tega smo na preprostem primeru videli, da je včasih za izvedbo neizvedljivega premika potrebno večje število izvedljivih premikov, kar upočasnjuje postopek optimizacije.

1					u		$S_J(u)$			v			
začetno stanje													
2	$\mathcal{T}(\mathcal{G})$	1a	4a	4b	2a	1b	2b	4c	3a	<u>3b</u>	1c	3c	2c
3	oznaka				•		•	•	•			∘	∘
4	razlog				u		$S_J(2a)$	$S_M(2b)$	$S_M(2b)$	$S_J(3a)$		$S_J(3b)$	$S_J(2b)$
po izvedbi popravljalnega premika													
5	$\mathcal{T}(\mathcal{G})$	1a	4a	4b	2a	1b	**4c**	**3a**	**2b**	<u>3b</u>	1c	3c	2c
6	oznaka				•				•				∘
7	razlog				u				$S_J(2a)$				$S_J(2b)$

Tabela 6.2: Topološko zaporedje po izvedbi popravljalnega premika.

Algoritem 4. Varni premiki

pri izvedbi premika $Q_{\mathrm{D}}(u, v)$

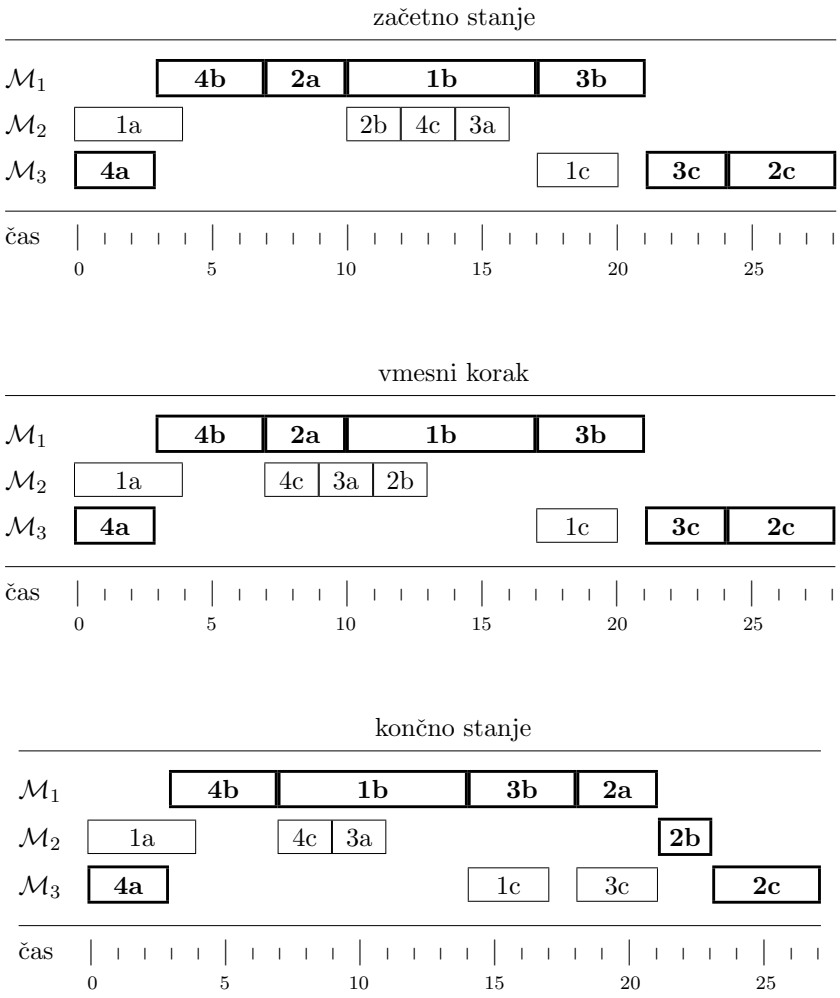
1. označevanje operacij za premik $Q_{\mathrm{D}}(u, v)$;
2. **if** $v \in \mathcal{L}$ **then** {*popravljanje*}
3. iskanje popravljalnega premika $Q_{\mathrm{D}}(b, a)$;
4. označevanje operacij za premik $Q_{\mathrm{D}}(b, a)$;
5. izvedba premika $Q_{\mathrm{D}}(b, a)$;
6. preureditev $\mathcal{T}(\mathcal{G})$;
7. **go to** 1;
8. **end if**
9. izvedba premika $Q_{\mathrm{D}}(u, v)$;
10. preureditev $\mathcal{T}(\mathcal{G})$;
11. izračun glav in repov ter iskanje kritične poti;

pri izvedbi premika $Q_{\mathrm{L}}(u, v)$

1. označevanje operacij za premik $Q_{\mathrm{L}}(u, v)$;
2. **if** $u \in \mathcal{L}$ **then** {*popravljanje*}
3. iskanje popravljalnega premika $Q_{\mathrm{L}}(b, a)$;
4. označevanje operacij za premik $Q_{\mathrm{L}}(b, a)$;
5. izvedba premika $Q_{\mathrm{L}}(b, a)$;
6. preureditev $\mathcal{T}(\mathcal{G})$;
7. **go to** 1;
8. **end if**
9. izvedba premika $Q_{\mathrm{L}}(u, v)$;
10. preureditev $\mathcal{T}(\mathcal{G})$;
11. izračun glav in repov ter iskanje kritične poti;

Slika 6.4: Algoritem za izvajanje varnih premikov.

Za primerjavo si oglejmo sliko 6.5, kjer je prikazan potek izvajanja premika $Q_{\mathrm{D}}(2a, 3b)$ ob uporabi popravljalne tehnike. Že na pogled je postopek hitrejši, ker za dosego želenega cilja porabimo samo en vmesni korak, medtem ko postopek Dell'Amico in Trubian (1993) potrebuje dva. Naš pristop prihrani po en vmesni korak za vsak popravljalni premik, ki ga je potrebno izvesti; postopek Dell'Amico in Trubian (1993) mora premikajočo operacijo u najprej premakniti znotraj njenega kritičnega bloka na ustrezno mesto, da nova kritična pot poteka



Slika 6.5: Premik operacije $2a$ za operacijo $3b$ z uporabo okolice $\mathcal{H}_D$.

preko njene tehnološke naslednice $S_J(u)$, s čimer se lahko izvede premik le-te izven njenega kritičnega bloka. To se lahko večkrat ponavlja, saj premik operacije $S_J(u)$ ni nujno izvedljiv sam zase.

Prihranek časa z uporabo popravljalne tehnike je dejansko še večji, saj med vmesnimi koraki ni potrebno preračunavati glav in repov operacij ter iskati nove kritične poti. Vsi ostali postopki izvajanja premikov ta relativno zamuden korak potrebujejo.

Dodaten prihranek časa nastane zato, ker postopek Dell'Amico in Trubian (1993) izvaja pomožne premike[3] na tak način, da skuša premikajočo operacijo premakniti izven njenega kritičnega bloka, kar morda sploh ni potrebno. To

[3]Namenoma nismo napisali, da gre za popravljalne premike, ker pri tem postopku ni semantične razlike med obema vrstama premikov; pojem popravljalni premik se uvede šele s popravljalno tehniko.

je slabo zaradi dveh razlogov. Premik operacije dlje, kot je potrebno, ni zaželen, ker je le-ta bolj verjetno neizvedljiv, zaradi česar potrebujemo za dosego cilja dodatne premike, ko so le-ti povsem nepotrebni. Druga slabost pristopa je, da se urnik spremeni bolj, kot je potrebno za izvedbo zahtevanega premika, s čimer slabšamo lastnost koreliranosti okolice (poglavje 4.6.1). V nasprotju s tem popravljalna tehnika določa popravljalne premike s tehniko zasledovanja označenih operacij, kar ji omogoča, da izvor neizvedljivosti locira povsem precizno, zato lahko premika operacije ravno dovolj daleč, da moteče relacije na strojih odpravi.

Pričakujemo tudi, da popravljalna tehnika zmanjša dovzetnost lokalnega iskanja za lokalne minimume. Kot smo videli v poglavju 5.1.2, mora postopek Dell'Amico in Trubian (1993) pri izvedbi premika $Q_{\mathrm{D}}(2a, 3b)$ opraviti dva vmesna koraka, pri čemer ima urnik v prvem koraku daljši izvršni čas, zaradi česar postopek lokalnega iskanja takega premika ne bi prepoznal kot napredujočega. Nasprotno pa je z uporabo popravljalne tehnike prehod iz začetnega v končno stanje direkten, brez kakršnegakoli določanja vmesnih izvršnih časov urnika; zato postopek lokalnega iskanja obravnavani premik prepozna kot napredujoči. Večje število napredujočih premikov v okolici pomeni manjšo dovzetnost optimizacijskega postopka za ujetje v lokalne minimume.

Na podlagi gornjih ugotovitev sklepamo, da prinaša predlagana popravljalna tehnika precej pomembnih prednosti pri snovanju postopkov lokalnega iskanja. Da bi trditev tudi empirično preizkusili, moramo razviti postopek lokalnega iskanja, ki je njene prednosti sposoben izkoristiti. Tej nalogi posvečamo pozornost v poglavjih, ki sledijo.

7. Okolica urnika

Okolica urnika je ključni sestavni del vsake izvedbe lokalnega iskanja za reševanje problema Π_{J}. V poglavju 3.7.2 je bila teoretično utemeljena uporaba omejene množice premikov na urniku pri izvedbi optimizacije. Poglavje 4.6.1 opisuje lastnosti, ki naj jih uporabljena množica premikov izkazuje. Na teh osnovah so se skozi zgodovino razvoja formirale razne okolice, ki smo jih opisali v istem poglavju; njihove slabosti smo podrobneje preučili v poglavju 5.1.

Uvedba popravljalne tehnike potencialno odpira nove možnosti pri definiciji okolice, s katerimi lahko zaobidemo vsaj en kompromis, kateremu se dosedanje okolice niso mogle izogniti. Bistvena novost je v tem, da se sedaj lahko pri definiciji okolice osredotočimo izključno na vprašanje, kateri premiki so za optimizacijo koristni, ob tem pa nam ni potrebno posvečati nikakršne pozornosti njihovi izvedljivosti. Kot smo opisali v poglavju 4.6.1, je izogibanje neizvedljivim premikom zaznamovalo prav vse dosedaj uporabljene okolice, kar govori o pomembnosti vpeljave popravljalne tehnike.

V tem delu predlagamo uporabo okolice $\mathcal{H}$, ki je konceptualno podobna okolici $\mathcal{H}_D$ avtorjev Dell'Amico in Trubian (1993). Bistvena razlika je v tem, da pri uporabi okolice $\mathcal{H}$ neizvedljivih premikov ne obravnavamo ločeno od izvedljivih. Vsak premik v okolici $\mathcal{H}$ spremeni začetek ali konec vsaj enega kritičnega bloka, kar favorizira teorem 4 (poglavje 3.7.2). Ob tem upoštevamo izsledke teorema 5, s katerim iz okolice izločimo premike, ki spremenijo samo začetek prvega ali samo konec zadnjega kritičnega bloka.

7.1 Formalna definicija predlagane okolice

Predlagano okolico formalno definiramo na naslednji način. Vsakemu kritičnemu bloku B_i priredimo množico desnih varnih premikov $\mathcal{H}_{\mathrm{D}i}$ in množico levih varnih premikov $\mathcal{H}_{\mathrm{L}i}$ na naslednji način:

$$\mathcal{H}_{\mathrm{D}i} = \begin{cases} Q^*_{\mathrm{D}}(B^1_i, B^2_i) & ; \text{če velja } (r > 1) \wedge (|B_i| = 2) \\ \bigcup_{j=1}^{|B_i|-1} Q^*_{\mathrm{D}}(B^j_i, B^{|B_i|}_i) & ; \text{če velja } (i \neq r) \wedge (|B_i| > 2) \\ \emptyset & ; \text{v vseh drugih primerih} \end{cases}$$

$$\mathcal{H}_{\mathrm{L}i} = \begin{cases} \bigcup_{j=2}^{|B_i|} Q^*_{\mathrm{L}}(B^1_i, B^j_i) & ; \text{če velja } (i \neq 1) \wedge (|B_i| > 2) \\ \emptyset & ; \text{v vseh drugih primerih} \end{cases}$$

Okolica $\mathcal{H}$ je unija vseh teh množic: $\mathcal{H} = \bigcup_{i=1}^{r} (\mathcal{H}_{\mathrm{D}i} \cup \mathcal{H}_{\mathrm{L}i})$.

Vidimo, da kritične bloke, ki vsebujejo dve operaciji, obravnavamo s posebnim pogojem, kar je potrebno iz dveh razlogov. Zamenjava operacij v takem bloku vedno spremeni hkrati začetek in konec bloka, zato je tak premik potencialno napredujoč tudi, če je kritični blok prvi ali zadnji na kritični poti. Izjema je situacija, kjer kritična pot vsebuje samo en kritični blok, kar pomeni, da je urnik optimalen (teorem 13 v nadaljevanju). Drugi razlog, da take bloke obravnavamo posebej, je v tem, da je pri njih vseeno, ali izvedemo desni ali levi premik; preprečiti pa želimo, da bi v okolico vključili oba.

Kritični bloki, ki vsebujejo samo eno operacijo, ne prispevajo v okolico nobenega premika, ker ne vsebujejo ne-prvih in ne-zadnjih operacij.

Ostalim kritičnim blokom (razen zadnjemu) so prirejeni desni varni premiki, ki vsako ne-zadnjo operacijo premaknejo za zadnjo operacijo v bloku. Poleg tega so jim (razen prvemu) prirejeni varni premiki, ki premaknejo vsako ne-prvo operacijo pred prvo operacijo v bloku.

7.2 Lastnosti predlagane okolice

Okolica $\mathcal{H}$ izkazuje tri zanimive lastnosti, ki nakazujejo, da je le-ta močnejša od dosedaj uporabljenih okolic.

Teorem 12. *Vsi premiki v okolici $\mathcal{H}$ so potencialno napredujoči: vsak premik je v določenih situacijah zmožen sam zase zmanjšati izvršni čas urnika.*

DOKAZ. Da bi bil premik v okolici potencialno napredujoč, morata biti izpolnjena dva pogoja: (1) premik mora spremeniti začetek ali konec vsaj enega kritičnega bloka (teorem 4), (2) pri tem se ne sme spremeniti samo začetka prvega ali samo konca zadnjega kritičnega bloka na kritični poti (teorem 5). Vsi premiki v okolici $\mathcal{H}$ izpolnjujejo oba pogoja. □

Od obravnavanih okolic enako lastnost izkazujeta še okolici $\mathcal{H}_N$ in $\mathcal{H}_B$. Da bi katerikoli premik v okolici dejansko zmanjšal izvršni čas urnika, morajo biti izpolnjeni dodatni pogoji, ki so povezani z medsebojnim odnosom ostalih (nekritičnih) operacij v urniku. Na primer, pri levem premiku ne-prve operacije pred kritični blok se lahko zgodi, da tehnološka predhodnica premaknjene operacije le-tej prepreči dovolj zgodnje izvajanje, s čimer se ob premiku kritična pot poveča.

Teorem 13. *Če okolica $\mathcal{H}$ ne vsebuje nobenega premika, je urnik optimalen.*

DOKAZ. Tak primer nastopi, ko vsebujejo vsi kritični bloki po eno operacijo, ali ko kritična pot vsebuje samo en poljubno velik kritični blok. S pomočjo definicije se lahko prepričamo, da v vseh ostalih primerih okolica $\mathcal{H}$ ni prazna.

Če kritična pot vsebuje samo kritične bloke z eno operacijo, to pomeni, da kritično pot sestavljajo izključno operacije enega opravila, ki se izvaja optimalno brez zastojev od začetka do konca. Kritično pot je nemogoče nadalje zmanjšati zaradi tehnoloških omejitev, s čimer je izvršni čas urnika dosegel spodnjo mejo možnega izvršnega časa, ki je določena z vsoto izvršnih časov operacij opravila, ki se izvaja optimalno.

Če kritična pot vsebuje samo en kritični blok, to pomeni, da kritično pot sestavljajo operacije, ki se izvajajo na enem stroju; operacije na tem stroju se izvajajo optimalno brez zastojev od začetka do konca. Kritično pot je nemogoče nadalje zmanjšati zaradi zmogljivostnih omejitev optimalno razporejenega kritičnega stroja. Izvršni čas urnika je dosegel spodnjo mejo možnega izvršnega časa, ki je določena z vsoto izvršnih časov operacij, ki se izvajajo optimalno na kritičnem stroju. □

V splošnem velja, da pri aproksimativnem reševanju problemov optimalnosti rešitev ne moremo dokazati. Izjema so nekateri primeri izredno lahkih instanc, kjer je možno doseči eno od zgoraj opisanih enostavno določljivih spodnjih mej izvršnega časa. Torej lahko v redkih primerih na enostaven način detektiramo optimalnost dobljenih rešitev. To lastnost izkazuje tudi okolica $\mathcal{H}_N$, s čimer sta Nowicki in Smutnicki (1996) detektirala optimalnost rešitev v primeru približno 20% lahkih instanc. Velja, da so instance, kjer to lastnost lahko izkoristimo, redke in zelo enostavne za reševanje, s tem pa tudi nezanimive za raziskave. Pri vseh instancah, ki niso trivialno rešljive, te lastnosti okolice ne moremo uporabiti, saj tudi okolice optimalnih urnikov vsebujejo večje ali manjše število premikov.

Oglejmo si naslednji teorem, katerega veljavnost potrebujemo v nadaljevanju pri opisu zadnje lastnosti predlagane okolice.

Teorem 14. *Dan imamo optimalen urnik $\mathcal{G}^*$ z izvršnim časom $C^*_{\max}$ in izvedljiv urnik $\mathcal{G}$ z izvršnim časom $C_{\max}$. Zadosten pogoj za optimalnost urnika $\mathcal{G}$ je, da je za vsak njegov kritični blok izpolnjena ustrezna, v nadaljevanju navedena trditev. Za prvi kritični blok B_1: vse ne-zadnje operacije $B_1 \backslash \{B_1^{|B_1|}\}$ se v $\mathcal{G}^*$ izvedejo pred zadnjo operacijo $B_1^{|B_1|}$ v poljubnem zaporedju. Za zadnji kritični blok B_r: vse ne-prve operacije $B_r \backslash \{B_r^1\}$ se v $\mathcal{G}^*$ izvedejo za prvo operacijo B_r^1 v poljubnem zaporedju. Za vse ostale kritične bloke B_i, $2 \leq i \leq (r-1)$: vse notranje operacije bloka $B_i \backslash \{B_i^1, B_i^{|B_i|}\}$ se v $\mathcal{G}^*$ izvedejo za operacijo B_i^1, vendar pred operacijo $B_i^{|B_i|}$ v poljubnem zaporedju.*

DOKAZ. Graf $\mathcal{G}$ ima naslednje lastnosti. V prvem kritičnem bloku B_1 obstaja v množici $\mathcal{E}$ povezava od vsake ne-zadnje operacije $B_1 \backslash \{B_1^{|B_1|}\}$ do zadnje operacije $B_1^{|B_1|}$ (za vsak $x \in B_1 \backslash \{B_1^{|B_1|}\}$ obstaja povezava $x \xrightarrow{\mathcal{E}} B_1^{|B_1|}$). V zadnjem kritičnem bloku B_r obstaja v množici $\mathcal{E}$ povezava od prve operacije B_r^1 do vsake ne-prve operacije $B_r \backslash \{B_r^1\}$ (za vsak $x \in B_r \backslash \{B_r^1\}$ obstaja povezava $B_r^1 \xrightarrow{\mathcal{E}} x$). V vseh ostalih kritičnih blokih B_i $\big(2 \leq i \leq (r-1)\big)$ obstaja v množici $\mathcal{E}$ povezava od prve operacije B_i^1 do vseh ne-prvih operacij $B_i \backslash \{B_i^1\}$ (za vsak $x \in B_i \backslash \{B_i^1\}$ obstaja povezava $B_i^1 \xrightarrow{\mathcal{E}} x$); nadalje obstaja v množici $\mathcal{E}$ povezava od vseh ne-zadnjih operacij $B_i \backslash \{B_i^{|B_i|}\}$ do zadnje operacije $B_i^{|B_i|}$ (za vsak $x \in B_i \backslash \{B_i^{|B_i|}\}$ obstaja povezava $x \xrightarrow{\mathcal{E}} B_i^{|B_i|}$). Poleg tega obstajajo v množici $\mathcal{A}$ povezave od

zadnje operacije $B_i^{|B_i|}$ vseh kritičnih blokov razen zadnjega do prve operacije B_{i+1}^1 naslednjega kritičnega bloka (zadnji odstavek poglavja 3.6). Če so s teoremom zahtevani pogoji izpolnjeni za vsak kritični blok B_i, potem obstajajo vse opisane povezave tudi v grafu $\mathcal{G}^*$. Posledica je, da graf $\mathcal{G}^*$ vsebuje vsaj eno pot od $\circledcirc$ do $\circledast$, ki je vsaj tako dolga kot kritična pot grafa $\mathcal{G}$; zato optimalni izvršni čas $C^*_{\max}$ ne more biti krajši do izvršnega časa urnika $\mathcal{G}$. □

Teorem 15. *Okolica $\mathcal{H}$ izkazuje lastnost povezljivosti.*

DOKAZ. Izberimo optimalni urnik $\mathcal{G}^*$ z izvršnim časom $C^*_{\max}$ in poljuben izvedljiv urnik $\mathcal{G}$ z izvršnim časom $C_{\max}$. Izbrana urnika se razlikujeta samo po usmeritvah povezav v množici $\mathcal{E}$. Število povezav, ki so v $\mathcal{G}$ usmerjene drugače kot v $\mathcal{G}^*$, naj nam predstavlja mero razdalje med urnikoma. Če velja $C_{\max} > C^*_{\max}$, potem obstaja v $\mathcal{G}$ vsaj en kritični blok B_i in vsaj ena kritična operacija B_i^j, za katero pogoj, ki ga postavlja teorem 14, ni izpolnjen. Natančneje, operacija B_i^j se v $\mathcal{G}^*$ izvaja pred operacijo B_i^1 ali za operacijo $B_i^{|B_i|}$. Obravnavajmo samo primer, ko se mora B_i^j izvesti za operacijo $B_i^{|B_i|}$; povsem analogno razmišljanje lahko uporabimo v tudi prvem primeru.

S pomočjo definicije okolice $\mathcal{H}$ ugotovimo, da le-ta vsebuje varni premik $Q^*_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$, s katerim je možno v $\mathcal{G}$ vzpostaviti enako relacijo med operacijama B_i^j in $B_i^{|B_i|}$, kot obstaja v $\mathcal{G}^*$. Ko tak premik izvedemo, je povezava $B_i^j \overset{\mathcal{E}}{\longleftrightarrow} B_i^{|B_i|}$ usmerjena v obeh urnikih enako, in sicer $B_i^{|B_i|} \overset{\mathcal{E}}{\longrightarrow} B_i^j$. Ni rečeno, da se s takim premikom zmanjša razdalja med obema urnikoma, saj se ob izvedbi premika spremenijo tudi vse povezave, ki obstajajo med B_i^j in vsemi operacijami $B_i^{j+1}, \ldots, B_i^{|B_i|-1}$. Če pa za premik vedno izberemo najbližjo operacijo B_i^j, ki jo je potrebno premakniti za operacijo $B_i^{|B_i|}$ (vse operacije $B_i^{j+1}, \ldots, B_i^{|B_i|-1}$ se v $\mathcal{G}^*$ izvajajo pred operacijo $B_i^{|B_i|}$), so po izvedbi premika $Q^*_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ vse povezave med B_i^j in $B_i^{j+1}, \ldots, B_i^{|B_i|}$ v obeh urnikih enako usmerjene. Če je premik $Q_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ izvedljiv, se ob izvedbi premika $Q^*_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ razdalja med $\mathcal{G}^*$ and $\mathcal{G}$ zanesljivo zmanjša. V nasprotnem primeru so za izvedbo premika $Q_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ potrebni popravljalni premiki, kar zahteva nadaljnjo diskusijo.

Popravljalni premiki so potrebni, če pred izvedbo premika $Q^*_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ obstaja v grafu $\mathcal{G}$ povezava med operacijo $S_{\mathrm{J}}(B_i^j)$ in katerokoli operacijo $P_{\mathrm{J}}(x)$, kjer je $x \in \{B_i^{j+1}, \ldots, B_i^{|B_i|}\}$. Za tako operacijo x velja, da je povezava $S_{\mathrm{J}}(B_i^j) \xleftarrow{\mathcal{E}} P_{\mathrm{J}}(x)$ usmerjena v $\mathcal{G}$ drugače kot v $\mathcal{G}^*$, saj v nasprotnem prmeru graf $\mathcal{G}^*$ ne bi bil acikličen (operacije $B_i^{j+1}, \ldots, B_i^{|B_i|}$ se v $\mathcal{G}^*$ izvajajo pred operacijo B_i^j). Torej vsak popravljalni premik usmeri vsaj eno dodatno povezavo v $\mathcal{G}$ enako, kot je usmerjena v $\mathcal{G}^*$. Razdalja med urnikoma se ob izvedbi popravljalnega premika lahko poveča samo v primeru, ko obstajajo med premikajočo operacijo u in ciljno operacijo v operacije z_k, ki se v $\mathcal{G}^*$ izvajajo za operacijo u. Če katerakoli operacija z_k vpliva na kritično pot novo nastalega grafa $\mathcal{G}$ takoj po izvedbi premika $Q^*_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ ali v katerikoli naslednji iteraciji lokalnega iskanja, se bo v okolici $\mathcal{H}$ nahajal premik, s katerim bo mogoče napačno relacijo med operacijami popraviti. Zaporedje izvajanja operacij $B_i^{|B_i|}$ in B_i^j, ki je nastalo po izvedbi premika $Q^*_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ ne bo več potrebno spremeniti, da bi bil katerikoli premik $Q^*_{\mathrm{D}}(z_k, u)$ izvedljiv. V $\mathcal{G}^*$ se namreč vse operacije z_k izvajajo za operacijo u, prav tako kot se operacija B_i^j izvaja za vsemi operacijami $B_i^{j+1}, \ldots, B_i^{|B_i|}$.

Lahko se torej zgodi, da premik $Q^*_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ poveča razdaljo med $\mathcal{G}$ in $\mathcal{G}^*$. Vendar se ob izvedbi vsakega takega premika poveča tudi število usmeritev v množici $\mathcal{E}$, ki jih z nobenim nadaljnjim premikom ne bo več potrebno spremeniti. Število takih usmeritev je pri vsakem premiku $Q^*_{\mathrm{D}}(B_i^j, B_i^{|B_i|})$ enako ena plus število popravljalnih premikov, ki so bili za izvedbo zahtevanega premika potrebni. Zaključimo, da je s pravilno izbiro premikov možno spremeniti poljuben začetni izvedljiv urnik $\mathcal{G}$ v optimalnega $\mathcal{G}^*$ z največ $|\mathcal{E}|$ premiki v okolici $\mathcal{H}$. □

Poleg okolice $\mathcal{H}$ velja lastnost povezljivosti tudi za okolici $\mathcal{H}_D$ in $\mathcal{H}_B$, vendar je slabost teh okolic doseganje povezljivosti s premiki, ki sami zase ne morejo zmanjšati izvršnega časa urnika, kar slabo vpliva na lokalno iskanje (poglavji 5.1.1 in 5.1.2). Okolica $\mathcal{H}$ je prva in do sedaj edina okolica, ki vsebuje samo potencialno napredujoče premike (teorem 12) in hkrati izkazuje lastnost povezljivosti (teorem 15). To je pomembna izboljšava v primerjavi z dosedanjimi postopki lokalnega iskanja, kjer je vedno bilo treba narediti kompromis med obema lastnostima. S tem se je lokalno iskanje za reševanje problema Π_{J} približalo im-

plementacijam lokalnega iskanja za reševanje problemov, ki neizvedljivosti rešitev ne poznajo (na primer, problem trgovskega potnika in problem Φ), zato je zanje relativno enostavno definirati okolice, ki vsebujejo samo potencialno napredujoče premike in izkazujejo lastnost povezljivosti. Obstoj neizvedljivih urnikov v primeru problema Π_J uporabo takih okolic onemogoča, vsaj dokler ne integriramo popravljalne tehnike v postopke lokalnega iskanja.

8. Smernice za izvajanje empiričnih testov

V predhodnih poglavjih smo predstavili popravljalno tehniko in njeno uporabo tudi teoretično utemeljili. Nadalje smo definirali okolico urnika, za katero menimo, da je vsaj delno sposobna izkoristiti nove možnosti, ki nam jih popravljalna tehnika odpira. Teoretično dokazane lastnosti predlagane okolice dajejo slutiti, da je le-ta močnejša od vseh do sedaj predlaganih okolic. Vendar nima teoretični napredek pri razvoju postopkov za reševanje problema Π_J nobene vrednosti, če z njegovo pomočjo v praksi ne moremo doseči boljših rešitev in/ali hitrejšega pridobivanja le-teh. Zaradi tega je nujno preizkusiti predlagane postopke s pomočjo primerjalnih empiričnih testov, ki so tako zasnovani, da se iz pridobljenih razultatov jasno vidi prednosti in slabosti uporabe našega postopka glede na ustaljene pristope.

Splošno uveljavljenega stališča, katera metoda optimizacije je najboljša za reševanje Π_J problema, ni. Razširjeno neuradno mnenje je, da ta naziv pripada iskanju s tabu seznamom (poglavje 4.6.4) ter genetskim algorimom v kombinaciji z lokalnim iskanjem (poglavje 4.6.6), kar pričajo reference, kot so Jain in Meeran (1999), Jain (1998), Jain in sod. (2000), Błażewicz in sod. (1996), Vaessens in sod. (1996) ter Yamada in Nakano (1996b). Na tej predpostavki so zasnovani tudi empirični testi, ki smo jih opravili.

V nadaljevanju sta najprej opisani uporabljeni implementaciji tabu iskanja in genetskega lokalnega iskanja, ki smo ju pri testiranju uporabili. Njun opis je namenoma dolg in natančen zaradi ponovljivosti rezultatov, saj noben znanstveni poskus nima uporabne vrednosti, če ga na drugem mestu in ob drugem času ne more ponoviti drug izvajalec. Sledijo empirični rezultati, kjer primerjamo zmožnosti naše okolice (in s tem popravljalne tehnike) s štirimi najpomembnejšimi okolicami v zgodovini problema Π_J.

9. Iskanje s tabu seznamom

Do danes je bilo za reševanje Π_J problema razvitih precej implementacij iskanja s tabu seznamom. Nekatere od njih so zelo preproste in uporabljajo samo osnovno idejo tabu iskanja. Obstajajo tudi bolj dodelane izvedbe, kot je predlog avtorjev Nowicki in Smutnicki (1996), ki vsebuje relativno zapleten mehanizem detektiranja *kroženja* (ang. *cycling*) (stanje, ko se postopek ciklično giblje skozi isti prostor rešitev, s čimer samo izgublja čas).

Implementacija iskanja s tabu seznamom, ki jo predlagamo v nadaljevanju, spada med preprostejše izvedbe. Razlog je v tem, da se ne naslanjamo na tabu seznam kot na edini mehanizem preiskovanja prostora rešitev, ampak predlagani postopek integriramo v genetski algoritem na način, ki je opisan v naslednjem poglavju. Za razliko od takega pristopa je implementacija iskanja s tabu seznamom, ki jo predlagata Nowicki in Smutnicki (1996), samostojna, kar opravičuje njeno večjo kompleksnost.

Algoritem iskanja s tabu seznamom, ki ga predlagamo, je prikazan na sliki 9.1. Poljuben začetni izvedljiv urnik $\mathcal{G}$ je vhodni podatek v algoritem skupaj s spremenljivko *mstag*, ki določa število iteracij brez izboljšave, preden se postopek prekine. Začetni urnik v koraku 1 priredimo delovni spremenljivki $\mathcal{G}_{\text{naj}}$, ki hrani trenutno znano najboljšo rešitev. Inicializiramo spremenljivko *iter*, ki pomni vrednost trenutne iteracije, ter izbrišemo matrično spremenljivko *TM*, v kateri pomnimo tabu status premikov. Elementu matrike *TM[a,b]* je prirejen varni premik $Q^*(w_a, w_b)$, s čimer lahko ustrezna vrednost elementa označuje tabu status prirejenega premika.

V koraku 2 zgradimo okolico urnika. Oznaka $\mathcal{H}_X$ pomeni okolico, ki jo uporabljamo za iskanje. Pri empiričnih testih, ki so podani v nadaljevanju, izvedemo primerjavo naše okolice z ostalimi, zato se na tem mestu ne omejujemo zgolj na okolico $\mathcal{H}$.

V koraku 3 preverimo, ali je okolica urnika $\mathcal{G}$ prazna. V tem primeru postopek prekinemo. Pri uporabi okolice, kjer tako stanje pomeni optimalnost urnika $\mathcal{G}$ (ta lastnost je dokazana za okolici $\mathcal{H}_N$ in $\mathcal{H}$), lahko optimizacijo prekinemo.

Algoritem 5. Iskanje s tabu seznamom

{izvedljiv urnik $\mathcal{G}$ je vhodni podatek algoritma}

{število korakov brez izboljšave *mstag* je vhodni podatek algoritma}

1. $\mathcal{G}_{\text{naj}} := \mathcal{G}$; iter := 1; TM $[a, b] := 0$ za vse $1 \leq a, b \leq n_{\text{tot}}$;
2. izgradi okolico $\mathcal{H}_X$;
3. **if** $|\mathcal{H}_X| = 0$ **then return** $\mathcal{G}$**;** {pri nekaterih okolicah je $\mathcal{G}$ optimalen}
4. oceni izvršni čas $C_{\text{ocen}}(i)$ za vsak premik $Q^*(u_i, v_i) \in \mathcal{H}_X$;
5. **if** $\min_{i=1}^{|\mathcal{H}_X|} \left(C_{\text{ocen}}(i)\right) < C_{\max}(\mathcal{G})$ **then**
6. nad $\mathcal{G}$ izvedi poljuben premik $Q^*(u_i, v_i)$ z najmanjšim $C_{\text{ocen}}(i)$;
7. **go to** 2;
8. **end if**
9. **if** $C_{\max}(\mathcal{G}) < C_{\max}(\mathcal{G}_{\text{naj}})$ **then** $\mathcal{G}_{\text{naj}} := \mathcal{G}$;
10. stag := 0; tenure := $|C(\mathcal{G})|/2$; iter := iter + n_{tot};
11. $Q^*(u_x, v_x)$:= poljuben premik $Q^*(u_i, v_i)$ z najmanjšim $C_{\text{ocen}}(i)$;
12. $\mathcal{H}_X := \mathcal{H}_X \backslash \{Q^*(u_x, v_x)\}$;
13. **if** TM $[\text{num}(u_x), \text{num}(v_x)] \geq$ (iter − tenure) **and** $C_{\text{ocen}}(x) \geq C_{\max}(\mathcal{G}_{\text{naj}})$ **then**
14. **if** $|\mathcal{H}_X| = 0$ **then**
15. iter := iter + tenure/2;
16. izgradi okolico $\mathcal{H}_X$;
17. **end if**
18. **go to** 11;
19. **end if**
20. TM $[\text{num}(u_x), \text{num}(v_x)]$:= iter ; TM $[\text{num}(v_x), \text{num}(u_x)]$:= iter ;
21. nad $\mathcal{G}$ izvedi premik $Q^*(u_x, v_x)$;
22. **if** $C_{\max}(\mathcal{G}) < C_{\max}(\mathcal{G}_{\text{naj}})$ **then go to** 2;
23. izgradi okolico $\mathcal{H}_X$;
24. tenure := $|C(\mathcal{G})|/2$; iter := iter + 1; stag := stag + 1;
25. **if** stag < mstag **then go to** 11;
26. **return** $\mathcal{G}_{\text{naj}}$;

Slika 9.1: Algoritem za izvedbo iskanja s tabu seznamom.

V koraku 4 ocenimo rezultirajoči izvršni čas urnika pri izvedbi posameznih premikov v okolici. Postopek ocenjevanja smo opisali v poglavju 4.6.2.

V koraku 5 preverimo, ali je najmanjši ocenjeni izvršni čas kateregakoli premika v $\mathcal{H}_X$ manjši od trenutnega izvršnega časa urnika $\mathcal{G}$, kar pomeni, da se postopek verjetno ne nahaja v lokalnem minimumu glede na uporabljeno okolico. V tem primeru enostavno izvedemo premik (korak 6), za katerega smo ocenili, da izvršni čas urnika najbolj zmanjša; če ima več premikov isto najmanjšo oceno, izberemo kateregakoli izmed njih. Po izvedbi premika se vrnemo na korak 2, kjer se celotni postopek ponovi.

V koraku 9 se urnik nahaja v lokalnem minimumu glede na uporabljeno okolico. Sedaj preverimo, ali je izvršni čas urnika manjši od trenutno najboljšega urnika; v tem primeru si ga zapomnimo v spremenljivki $\mathcal{G}_{\text{naj}}$. Preverjanje tega pogoja po izvedbi koraka 6 bi bilo nesmotrno, saj bi to povzročilo veliko časovno zamudnih prirejanj vmesnih urnikov spremenljivki $\mathcal{G}_{\text{naj}}$ vsakič, ko bi urnik izboljšali, čeprav se le-ta še ne nahaja v lokalnem minimumu, s čimer je verjetnost nadaljnje izboljšave ob naslednji ponovitvi korakov od 2 do 8 velika.

Pravo iskanje s tabu seznamom se prične v koraku 10. Ker se urnik sedaj nahaja v lokalnem minimumu glede na uporabljeno okolico, moramo za nadaljnjo optimizacijo uporabiti meta-hevristični mehanizem pobega, ki ga v našem primeru uteleša tabu seznam. Na tem mestu inicializiramo spremenljivko *stag*, ki šteje število iteracij brez izboljšave trenutne rešitve. Spremenljivka *tenure* hrani dolžino trajanja (število iteracij) tabu statusa premikov. Vidimo, da leto ni konstantno, ampak se spreminja od urnika do urnika glede na trenutno število kritičnih operacij vzdolž obravnavane kritične poti. Tak postopek samoprilagajanja sta predlagala Mastrolilli in Gambardella (2000) za generalizirani Π_{J} problem; njuna izbira je $\mathit{tenure} := |C(\mathcal{G})|$, vendar smo mi dosegli boljše rezultate s prepolovitvijo te vrednosti. Spremenljivka *iter* hrani trenutno iteracijo, da lahko skupaj s *tenure* določimo, kateri premiki imajo tabu status. V tem koraku spremenljivko *iter* povečamo za število operacij v instanci, kar ima učinek preklica tabu statusa vsem premikom, saj kritična pot in s tem spremenljivka *tenure* ne more biti večja od n_{tot}.

V koraku 11 izberemo iz okolice premik, ki ima najmanjši ocenjeni izvršni čas; le-ta je tipično večji od trenutnega izvršnega časa, ker se nahajamo v lokalnem minimumu, ali izvajamo premike, ki izvršni čas urnika slabšajo. Izbrani premik odstranimo iz trenutne okolice (korak 12), ko zanj velja tabu status, da lahko pri ponovni izbiri določimo drugega kandidata za izvedbo.

V koraku 13 preverimo tabu status izbranega premika. Pri tem nam pomaga matrika *TM*, katere elementi hranijo zadnjo iteracijo, ob kateri je bil določeni premik izveden (iteracije se elementom priredijo v koraku 20, kot je opisano v nadaljevanju). V primeru da se je izbrani premik ali njegov inverzni premik izvršil manj kot *tenure* iteracij nazaj, ima izbrani premik tabu status in ga ne izvedemo; izjema je primer, ko je ocenjeni izvršni čas premika manjši od trenutno znanega najboljšega izvršnega časa (aspiracijski kriterij, poglavje 4.6.4), s čimer se tabu status ukine in nadaljujemo s korakom 20, kjer izbrani premik izvršimo.

V primeru tabu statusa izbranega premika obstajata dve možnosti (korak 14). Če trenutna okolica ni prazna, se v koraku 18 vrnemo na korak 11, kjer izberemo naslednji najboljši premik v okolici in poskusimo znova. Če je trenutna okolica prazna (vsi premiki imajo tabu status), povečamo spremenljivko *iter* za polovico trenutne vrednosti *tenure* (korak 15), s čimer se verjetno ukine tabu status določenim premikom. Ob tem ponovno zgradimo okolico trenutnega urnika (korak 16), saj smo predhodno posamezne premike iz nje izbrisali (korak 12). Nato se vrnemo na korak 11, kjer ponovno izbiramo premike iz okolice, vendar tokrat z manj restriktivnim tabu statusom. Če so zopet vsi premiki tabu, se v koraku 15 spremenljivka *iter* že drugič poveča za polovično vrednost spremenljivke *tenure*, s čimer je zanesljivo ukinjen tabu status vsem premikom v okolici in optimizacija se nadaljuje.

Korak 20 dosežemo, ko izbrani premik ni tabu in ga brezpogojno izvršimo. Najprej vpišemo v ustrezna elementa matrike *TM* trenutno iteracijo, da lahko izbranemu in njemu inverznemu premiku naknadno pripišemo tabu status.

V koraku 21 premik dejansko izvršimo na urniku. Če je dejanski (in ne ocenjeni) izvršni čas tako dobljenega urnika manjši od trenutno znanega najboljšega izvršnega časa (korak 22), se vrnemo na korak 2, kjer se celotni postopek ponovi

z novim najboljšim urnikom kot izhodiščem za preiskovanje. Pomembno je, da pri tem ne uporabljamo ocenjeni izvršni čas ampak dejanskega, saj bi se v nasprotnem primeru postopek lahko zavozlal, ker so ocenjeni izvršni časi največkrat bolj optimistični od dejanskih.

Če dobljeni urnik ni boljši od najboljšega znanega, se nadaljuje običajno tabu iskanje. V koraku 23 zgradimo okolico novega urnika, v koraku 24 določimo novo vrednost *tenure*, povečamo število iteracij *iter* ter števec stagnacij *stag*.

Če število stagniranih iteracij ni večje od maksimalne predpisane vrednosti *mstag* (korak 25), skočimo na korak 11, sicer postopek končamo in kot rezultat vrnemo najboljši urnik, ki smo ga v teku tabu iskanja našli.

Pri nekaterih testih v nadaljevanju uporabljamo predpisani čas izvajanja tabu iskanja kot terminacijski pogoj (korak 25) namesto števila iteracij brez izboljšave.

10. Genetski algoritem

V poglavju 4.6.5 smo opisali prednosti in slabosti genetskih algoritmov, ki diktirajo način njihove uporabe. Generski algoritmi preiskujejo prostor rešitev zelo na široko in neprecizno, zato je njihova prednost v tem, da niso občutljivi na lokalne minimume. Vendar genetski algoritmi sami zase niso sposobni natančnega preiskovanja prostora rešitev, zato z njihovo pomočjo zelo redko naletimo na rešitve velike kvalitete. Komplementarna ugotovitev velja za iskanje s tabu seznamom, ki preiskuje prostor rešitev relativno precizno v okolici začetne rešitve, ne more pa se bistveno odmakniti od lokalno omejenega teritorija, kjer se je preiskovanje pričelo. V praksi zato pogosto naletimo na kombinacijo obeh pristopov, kar imenujemo genetsko lokalno iskanje (Grefenstette 1987).

Naša implementacija genetskega lokalnega iskanja je naslednja. Vhodni podatki v algoritem so spremenljivke, ki določajo parametre iskanja, kakor je opisano v nadaljevanju.

Pred pričetkom izvajanja evolucije se vnaprej določeno število *ps* kromosomov inicializira tako, da vsak od njih predstavlja naključno generiran izvedljiv urnik. Urniki so zakodirani s kodno shemo, ki jo je predlagal Bierwirth (1995). Vsak kromosom je optimiran z iskanjem s tabu seznamom, kakor je bilo opisano v predhodnem poglavju. Vrednost *mstag* je enaka n_{tot}, s čimer se postopek delno avtomatično adaptira na različne velikosti instanc. Optimirani kromosomi vstopajo v populacijo vendar le, če je njihov izvršni čas različen od vseh izvršnih časov kromosomov, ki so že v populaciji; v nasprotnem primeru se naključna inicializacija in optimizacija kromosoma ponovi. Tak pristop naleti na oviro pri izredno lahkih instancah, kjer je nemogoče generirati dovolj optimiranih kromosomov z različnimi izvršnimi časi (vse generirane rešitve so zelo blizu optimalni rešitvi), s čimer se postopek zavozla. Da to preprečimo, se kromosoma ne inicializira več kot v m (število strojev v instanci) poskusih. Namesto tega se populacija (in s tem parameter *ps*) zmanjša na število dobljenih kromosomov z različnimi izvršnimi časi, ki smo jih do tega trenutka uspeli generirati. Ko je začetna populacija inicializirana, se prične prva iteracija evolucije.

Vsakemu kromosomu $c(i)$, $1 \leq i \leq ps$, z izvršnim časom $C_{\max}$(i) se priredi vrednost prilagoditvene funkcije $f(i)$ po obrazcu $f(i) = \overline{C_{\max}}/C_{\max}(i)$, kjer je $\overline{C_{\max}} = \sum_{i=1}^{ps} C_{\max}(i)/ps$. Sledi linearno skaliranje vrednosti prilagoditvene funkcije, na povsem enak način, kot je priporočeno v Goldberg (1989). Namen tega koraka je uravnoteženje populacije in preprečitev prehitrega izginotja večine kromosomov zaradi enega ali dveh izrazito močnih primerkov. Parameter skaliranja je vrednost $R_{B/A}$, ki predpisuje razmerje med najboljšo in povprečno prilagoditveno funkcijo v populaciji. S pomočjo skaliranih vrednosti prilagoditvene funkcije se vsakemu kromosomu izračuna kumulativna vrednost prilagoditvene funkcije $f_c(i) = \sum_{j=1}^{i} f_c(j)$, ki se uporablja pri izbiri kromosomov za delovanje genetskih operatorjev. Kromosomi se izbirajo na naslednji način: generira se naključno število rn med 0 in $f_c(ps)$ (vključno) ter izbere kromosom $c(s)$ z najmanjšim indeksom s, za katerega velja relacija $rn \leq f_c(s)$.

Za izvedbo križanja smo izbrali genetski operator GOX, kot je to predlagano v Mattfeld (1996), vendar smo v predlagano izvedbo vnesli majhno spremembo: priporočilo pravi, naj bo dolžina kromosomovega odseka velika med $n_{\text{tot}}/3$ in $n_{\text{tot}}/2$, vendar smo mi dosegli boljše rezultate z manj restriktivnim pogojem dolžine odseka med $n_{\text{tot}}/20$ in $n_{\text{tot}}/2$. Poleg tega izbrana kromosoma za križanje pregledamo, da ugotovimo njun prvi g_p in zadnji g_z gen pri katerem se razlikujeta. Če je dolžina odseka med g_p in g_z manjša od $n_{\text{tot}}/15$, se križanje prekliče, ker sta si kromosoma preveč podobna, zato je verjetnost, da s križanjem nastane drugačen urnik, relativno majhna. V teku evolucije celotna populacija izkazuje lastnost konvergence in kromosomi si postajajo med seboj zelo podobni, s čimer bi brez ugotavljanja njihove različnosti postopek optimizacije izgubljal čas s tabu iskanjem istih rešitev.

Število križanj v evolucijski iteraciji se izračuna kot $cprob \cdot ps$, kjer je $cprob$ ($0 \leq cprob \leq 1$) vnaprej predpisana verjetnost križanja. Pred vsako izvedbo križanja postopek izbere dva kromosoma $c(a)$ in $c(b)$, kot je bilo predhodno opisano. V primeru da je $a = b$, se izbira ponovi. Rezultat križanja se optimira s tabu seznamom in doda v populacijo, če sta izpolnjena naslednja pogoja: (1) izvršni čas dobljenega urnika je različen od vseh izvršnih časov urnikov v popu-

laciji, (2) izvršni čas dobljenega urnika je krajši od izvršnega časa najslabšega urnika v populaciji. Če kateri od pogojev ni izpolnjen, se rezultirajoči kromosom zavrne. Ko so izvedena vsa križanja v evolucijski iteraciji, se k vrednosti *cprob* prišteje $\triangle cprob$ ($-1 \leq \triangle cprob \leq +1$), ki je prav tako vnaprej predpisana vrednost. V primeru da vrednost *cprob* postane večja od 1 ali manjša od 0, se popravi na najbližjo od obeh vrednosti.

Drug genetski operator so mutacije. Mattfeld (1996) predlaga izvajanje mutacij z operatorjem PMB. Vendar ima izvajanje mutacij direktno na kromosomih slabo lastnost, in sicer je določen procent mutacij samo navideznih, ker je kodna shema redundantna. Zaradi tega naš postopek izvaja mutacije direktno na urniku in ne na samem kromosomu. Pred izvedbo mutacije se kromosom odkodira v urnik. Nato se naključno izbere eden od premikov v njegovi okolici $\mathcal{H}$ ter brezpogojno izvrši, ne glede na rezultirajoči izvršni čas. Na ta način zagotovimo, da vsaj en premik na urniku vpliva na njegovo kritično pot. Na urniku se nato izvrši še naključno izbrano število dodatnih premikov med 1 in $n_{\text{tot}} \cdot gprob$, kjer je *gprob* vnaprej predpisana vrednost med 0 and 1. Pri vsakem dodatnem premiku se naključno določi stroj $\mathcal{M}_{\text{i}}$ ter operaciji $w_a \in \mathcal{O}_i$ in $w_b \in \mathcal{O}_i$ ($a \neq b$), nakar se izvrši premik $Q^*(w_a, w_b)$. Po izvedbi vseh mutacijskih premikov, se rezultirajoči urnik optimira s tabu iskanjem in doda v populacijo, če sta izpolnjena ista pogoja, kot pri križanju.

Število mutacij v evolucijski iteraciji se izračuna kot $mprob \cdot ps$, kjer je *mprob* ($0 \leq mprob \leq 1$) vnaprej predpisana vrednost. Ko so vse mutacije opravljene, se vrednosti *mprob* prišteje vnaprej predpisana vrednost $\triangle mprob$ ($-1 \leq \triangle mprob \leq +1$), s čimer se število mutacij poveča ali zmanjša med posameznimi evolucijskimi iteracijami. Podobno se vrednosti *gprob* prišteje vnaprej predpisana vrednost $\triangle gprob$ ($-1 \leq \triangle mprob \leq +1$), ki poveča ali zmanjša število mutacijskih premikov. Če katera od vrednosti *mprob* ali *gprob* postane večja od 1 ali manjša od 0, se popravi na najbližjo mejno vrednost.

Ker kromosomi, ki so rezultat genetskih operacij, ne zamenjajo originalnih kromosomov, se populacija med posamezno iteracijo poveča. Po koncu vsake evolucijske iteracije se število kromosomov zmanjša na začetno vrednost *ps* tako,

da se iz populacije izbriše ustrezno število kromosomov, ki imajo najslabši izvršni čas. Evolucija se prekine, ko v zadnjih *siters* iteracijah ni prišlo do izboljšave najboljšega kromosoma v populaciji.

11. Rezultati pri iskanju s tabu seznamom

Prvi niz empiričnih testov uporablja samo iskanje s tabu seznamom, ki je bilo opisano v poglavju 4.6.4 Pri snovanju njihove izvedbe se je pojavilo vprašanje, na kakšen način primerjati različne okolice na objektiven način. Dilemo je predstavljala izbira terminacijskega kriterija, kjer število iteracij brez izboljšave ni najbolj primerna izbira. Problem je v tem, da popravljalna tehnika izvaja poleg zahtevanega premika tudi popravljalne premike, zato bi tak kriterij za našo okolico predstavljal prednost. Da bi opisano pristranskost preprečili, smo se odločili uporabiti dovoljeni čas tabu optimizacije kot terminacijski pogoj (korak 25 algoritma 5). Izbire so bile naslednje: 10 ms, 50 ms, 100 ms, 500 ms in 1000 ms. Izvajanje testov je potekalo na računalniku s procesorjem AMD Athlon 900 TB.

Primerjalne teste smo izvajali nad množico 12 izbranih testnih primerov, ki so znani kot težko rešljivi (poglavji 4.2 in 4.3). Za vsako instanco smo generirali 500 naključnih urnikov in jih optimirali s tabu seznamom z uporabo štirih najpomembnejših Π_J okolic (poglavje 4.6) in naše okolice (poglavje 7.1). Pri vseh okolicah smo uporabili iste začetne urnike. Rezultati so prikazani v tabelah od 11.1 do 11.10. Naša okolica je ločeno primerjana z vsako od ostalih obravnavanih okolic.

Posamezni stolpci v tabelah imajo naslednji pomen. V prvem stolpcu se nahaja ime testnega problema. Stolpec $C_{\max}(\mathcal{H})$ podaja povprečni izvršni čas urnikov ob uporabi naše okolice. Pod oznakami $C_{\max}(\mathcal{H}_N)$, $C_{\max}(\mathcal{H}_B)$, $C_{\max}(\mathcal{H}_D)$ in $C_{\max}(\mathcal{H}_V)$ se nahajajo rezultati dobljeni z uporabo pripadajočih okolic. V stolpcih $\Delta C_{\max}$ se nahaja razlika povprečnega rezultata med primerjano in našo okolico, kjer pozitivna vrednost pomeni, da smo z našo okolico dobili boljši rezultat $\Big(\Delta C_{\max} = C_{\max}(\mathcal{H}_X) - C_{\max}(\mathcal{H})\Big)$. Stolpca $\mathrm{N}^\circ(\mathcal{H} < \mathcal{H}_X)$ in $\mathrm{N}^\circ(\mathcal{H} < \mathcal{H}_X)$ podajata, kolikokrat smo z našo okolico dobili boljši rezultat kakor s primerjano in obratno. Dva stolpca sta potrebna zato, ker z enim ni mogoče prikazati nepristranskih rezultatov, ko z obema okolicama dobimo isti rezultat. Zadnji stolpec podaja razliko med obema številoma, kjer pozitivna vrednost pomeni, da smo z našo okolico večkrat dobili boljši rezultat kot s primerjalno.

11.1 Rezultati tabu iskanja pri času izvajanja 10 ms

Tabela 11.1 primerja našo okolico z okolicama $\mathcal{H}_N$ (Nowicki in Smutnicki 1996) ter $\mathcal{H}_B$ (Balas 1969) ob dovoljenem času izvajanja 10 ms. Vidimo, da so pri vseh instancah povprečni izvršni časi urnikov, dobljenih z našo okolico, boljši od primerjalnih okolic. Tudi število tabu iskanj, kjer je naša okolica dala boljši rezultat govori močno v našo korist. Edina izjema je instanca **ft10**, kjer smo z okolico $\mathcal{H}_N$ dobili boljši rezultat v 249 primerih, z našo okolico pa le v 242 primerih. Kljub temu je povprečni rezultat tudi pri tej instanci neznatno boljši ob uporabi naše okolice.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_N$ (Nowicki in Smutnicki 1996)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_N)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_N$)	N°($\mathcal{H} > \mathcal{H}_N$)	ΔN°
abz7	743,46	756,59	13,13	352	138	214
abz8	745,27	759,55	14,28	353	139	214
abz9	773,43	780,63	7,20	326	164	162
ft10	985,48	986,17	0,69	242	249	-7
la21	1090,13	1099,70	9,57	294	192	102
la24	959,98	970,17	10,19	351	140	211
la29	1235,59	1247,30	11,71	328	164	164
la40	1294,10	1304,28	10,18	317	178	139
yn1	1056,81	1089,29	32,48	411	86	325
yn2	1082,87	1108,18	25,31	387	110	277
yn3	1103,79	1131,54	27,75	379	117	262
yn4	1208,24	1234,42	26,18	374	122	252
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_B$ (Balas 1969)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_B)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_B$)	N°($\mathcal{H} > \mathcal{H}_B$)	ΔN°
abz7	743,46	818,22	74,76	495	5	490
abz8	745,27	808,19	62,92	485	12	473
abz9	773,43	825,33	51,90	489	10	479
ft10	985,48	1030,95	45,47	431	66	365
la21	1090,13	1160,10	69,97	465	29	436
la24	959,98	1006,30	46,32	421	73	348
la29	1235,59	1468,73	233,14	500	0	500
la40	1294,10	1349,30	55,20	450	45	405
yn1	1056,81	1103,16	46,35	439	58	381
yn2	1082,87	1131,74	48,87	448	50	398
yn3	1103,79	1161,38	57,59	441	56	385
yn4	1208,24	1310,30	102,06	472	26	446

Tabela 11.1: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 10 ms).

V tabeli 11.2 si lahko ogledamo primerjavo naše okolice z okolicama $\mathcal{H}_D$ (Dell'Amico in Trubian 1993) ter $\mathcal{H}_V$ (Yamada in sod. 1994) ob dovoljenem času izvajanja 10 ms. Tokrat so rezultati manj enolični. Ob uporabi okolice $\mathcal{H}_D$ smo pri instancah **yn1**, **yn2** in **yn3** dosegli boljše povprečne urnike v primerjavi z našo okolico. Pri instancah **yn1** in **yn3** je zaostanek naše okolice relativno velik.

Z okolico $\mathcal{H}_V$ smo dosegli še boljše rezultate, saj so bili z njeno pomočjo dobljeni urniki boljši od naših kar v sedmih od dvanajstih primerov; pri instancah **yn1**, **yn2** in **yn3** so rezultati znatno boljši. Okolica $\mathcal{H}_V$ precej zaostaja za našo samo pri instancah **ft10** in **la29**.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_D$ (Dell'Amico in Trubian 1993)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_D)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_D$)	N°($\mathcal{H} > \mathcal{H}_D$)	ΔN°
abz7	743,46	750,45	6,99	324	169	155
abz8	745,27	751,46	6,19	314	174	140
abz9	773,43	775,25	1,82	261	220	41
ft10	985,48	1001,63	16,15	355	135	220
la21	1090,13	1101,59	11,46	321	169	152
la24	959,98	964,93	4,95	298	184	114
la29	1235,59	1281,71	46,12	446	49	397
la40	1294,10	1300,68	6,58	294	203	91
yn1	1056,81	1046,77	-10,04	180	307	-127
yn2	1082,87	1080,10	-2,77	230	261	-31
yn3	1103,79	1092,81	-10,98	197	290	-93
yn4	1208,24	1225,35	17,11	309	184	125
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_V$ (Yamada in sod. 1994)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_V)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_V$)	N°($\mathcal{H} > \mathcal{H}_V$)	ΔN°
abz7	743,46	734,68	-8,78	143	346	-203
abz8	745,27	739,51	-5,76	181	309	-128
abz9	773,43	764,62	-8,81	143	352	-209
ft10	985,48	996,04	10,56	324	169	155
la21	1090,13	1095,79	5,66	247	235	12
la24	959,98	963,65	3,67	254	226	28
la29	1235,59	1261,58	25,99	366	128	238
la40	1294,10	1288,06	-6,04	187	295	-108
yn1	1056,81	1036,40	-20,41	122	376	-254
yn2	1082,87	1066,48	-16,39	140	353	-213
yn3	1103,79	1082,06	-21,73	144	352	-208
yn4	1208,24	1210,99	2,75	250	242	8

Tabela 11.2: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 10 ms).

11.2 Rezultati tabu iskanja pri času izvajanja 50 ms

Tabela 11.3 primerja našo okolico z okolicama $\mathcal{H}_N$ (Nowicki in Smutnicki 1996) ter $\mathcal{H}_B$ (Balas 1969) ob dovoljenem času izvajanja 50 ms. Z uporabo okolice $\mathcal{H}_N$ smo pri instanci **ft10** dobili nekoliko boljše rezultate kot z uporabo naše okolice. Pri ostalih instancah okolica $\mathcal{H}_N$ močno zaostaja.

Okolica $\mathcal{H}_B$ daje bistveno slabše rezultate, zato je praktično ni potrebno komentirati.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_N$ (Nowicki in Smutnicki 1996)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_N)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_N$)	N°($\mathcal{H} > \mathcal{H}_N$)	ΔN°
abz7	686,48	696,20	9,72	401	84	317
abz8	697,66	706,29	8,63	408	80	328
abz9	726,13	732,50	6,37	337	147	190
ft10	955,83	953,18	-2,65	239	245	-6
la21	1062,96	1072,14	9,18	281	195	86
la24	949,10	957,56	8,46	371	117	254
la29	1193,33	1200,91	7,58	337	153	184
la40	1246,53	1251,48	4,95	313	161	152
yn1	933,50	946,15	12,65	374	116	258
yn2	957,30	967,42	10,12	357	133	224
yn3	959,72	978,52	18,80	397	92	305
yn4	1049,88	1065,12	15,24	361	129	232
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_B$ (Balas 1969)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_B)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_B$)	N°($\mathcal{H} > \mathcal{H}_B$)	ΔN°
abz7	686,48	779,02	92,54	500	0	500
abz8	697,66	743,40	45,74	472	26	446
abz9	726,13	781,70	55,57	489	8	481
ft10	955,83	1027,63	71,80	447	49	398
la21	1062,96	1151,84	88,88	478	21	457
la24	949,10	982,73	33,63	405	84	321
la29	1193,33	1483,45	290,12	500	0	500
la40	1246,53	1282,96	36,43	461	34	427
yn1	933,50	953,61	20,11	431	61	370
yn2	957,30	985,79	28,49	450	44	406
yn3	959,72	1001,06	41,34	475	22	453
yn4	1049,88	1193,96	144,08	500	0	500

Tabela 11.3: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 50 ms).

V tabeli 11.4 si lahko ogledamo primerjavo naše okolice z okolicama $\mathcal{H}_D$ (Dell'Amico in Trubian 1993) ter $\mathcal{H}_V$ (Yamada in sod. 1994) ob dovoljenem času izvajanja 50 ms. Rezultati z uporabo naše okolice so se močno popravili glede na čas optimizacije 10 ms. Okolica $\mathcal{H}_D$ je bila pri vseh instancah slabša, še posebej izrazito pri primerih **ft10**, **la29** in **yn4**.

Tudi glede na okolico $\mathcal{H}_V$ so se naši rezultati popravili, saj je naša okolica slabša od $\mathcal{H}_V$ samo še na štirih od dvanajstih primerov. Naš zaostanek ni niti malenkosten niti izrazit. Nasprotno s tem je zaostanek okolice $\mathcal{H}_V$ zelo izrazit pri instancah **ft10**, **la29** in **yn4**. Tudi v primeru instance **la21** okolica $\mathcal{H}_V$ precej zaostaja. Pri reševanju instance **abz7** sta obe okolici dokaj enakovredni.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_D$ (Dell'Amico in Trubian 1993)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_D)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_D$)	N°($\mathcal{H} > \mathcal{H}_D$)	ΔN°
abz7	686,48	693,55	7,07	359	127	232
abz8	697,66	702,87	5,21	344	144	200
abz9	726,13	728,84	2,71	299	191	108
ft10	955,83	980,42	24,59	383	105	278
la21	1062,96	1074,98	12,02	341	145	196
la24	949,10	952,00	2,90	272	176	96
la29	1193,33	1252,36	59,03	474	26	448
la40	1246,53	1251,28	4,75	281	193	88
yn1	933,50	935,06	1,56	253	229	24
yn2	957,30	959,98	2,68	273	214	59
yn3	959,72	964,58	4,86	287	198	89
yn4	1049,88	1087,07	37,19	442	57	385
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_V$ (Yamada in sod. 1994)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_V)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_V$)	N°($\mathcal{H} > \mathcal{H}_V$)	ΔN°
abz7	686,48	687,09	0,61	232	244	-12
abz8	697,66	699,49	1,83	246	229	17
abz9	726,13	722,85	-3,28	185	298	-113
ft10	955,83	977,49	21,66	362	125	237
la21	1062,96	1078,57	15,61	349	139	210
la24	949,10	953,10	4,00	262	211	51
la29	1193,33	1237,30	43,97	439	58	381
la40	1246,53	1247,99	1,46	234	244	-10
yn1	933,50	930,94	-2,56	199	290	-91
yn2	957,30	954,41	-2,89	193	286	-93
yn3	959,72	957,11	-2,61	210	275	-65
yn4	1049,88	1080,94	31,06	398	98	300

Tabela 11.4: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 50 ms).

11.3 Rezultati tabu iskanja pri času izvajanja 100 ms

Tabela 11.5 primerja našo okolico z okolicama $\mathcal{H}_N$ (Nowicki in Smutnicki 1996) ter $\mathcal{H}_B$ (Balas 1969) ob dovoljenem času izvajanja 100 ms. Okolica $\mathcal{H}_N$ še vedno daje boljše rezultate pri instanci **ft10**, v ostalih primerih pa zaostaja.

Okolica $\mathcal{H}_B$ daje zelo slabe rezultate, zato je praktično ni potrebno komentirati.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_N$ (Nowicki in Smutnicki 1996)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_N)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_N$)	N°($\mathcal{H} > \mathcal{H}_N$)	ΔN°
abz7	678,21	685,58	7,37	394	76	318
abz8	690,47	699,28	8,81	416	68	348
abz9	713,15	719,70	6,55	353	125	228
ft10	951,60	949,05	-2,55	256	230	26
la21	1059,21	1069,82	10,61	340	126	214
la24	949,02	954,52	5,50	346	131	215
la29	1184,29	1190,76	6,47	363	126	237
la40	1239,91	1242,90	2,99	302	173	129
yn1	918,64	925,72	7,08	355	130	225
yn2	940,24	948,45	8,21	374	112	262
yn3	935,74	949,03	13,29	385	106	279
yn4	1020,02	1030,27	10,25	351	140	211
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_B$ (Balas 1969)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_B)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_B$)	N°($\mathcal{H} > \mathcal{H}_B$)	ΔN°
abz7	678,21	774,62	96,41	500	0	500
abz8	690,47	737,16	46,69	475	21	454
abz9	713,15	771,79	58,64	487	11	476
ft10	951,60	1019,56	67,96	455	42	413
la21	1059,21	1143,89	84,68	475	23	452
la24	949,02	983,50	34,48	377	103	274
la29	1184,29	1477,24	292,95	500	0	500
la40	1239,91	1271,15	31,24	420	67	353
yn1	918,64	931,08	12,44	402	84	318
yn2	940,24	961,47	21,23	450	47	403
yn3	935,74	973,76	38,02	472	24	448
yn4	1020,02	1174,67	154,65	499	1	498

Tabela 11.5: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 100 ms).

V tabeli 11.6 si lahko ogledamo primerjavo naše okolice z okolicama $\mathcal{H}_D$ (Dell'Amico in Trubian 1993) ter $\mathcal{H}_V$ (Yamada in sod. 1994) ob dovoljenem času izvajanja 100 ms. Naša okolica je boljša od okolice $\mathcal{H}_D$ na vseh testnih primerih. Pri instancah **ft10**, **la21**, **la29** in **yn4** je naša prednost izrazita.

Glede na okolico $\mathcal{H}_V$ naši rezultati zaostajajo pri instancah **abz9**, **la40**, **yn1**, **yn2** in **yn3**. V nobenem primeru zaostanek ni izrazit. Nasprotno pa okolica $\mathcal{H}_V$ znatno zaostaja pri reševanju problemov **ft10**, **la21**, **la29** in **yn4**.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_D$ (Dell'Amico in Trubian 1993)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_D)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_D$)	N°($\mathcal{H} > \mathcal{H}_D$)	ΔN°
abz7	678,21	681,89	3,68	318	153	165
abz8	690,47	694,97	4,50	329	145	184
abz9	713,15	716,27	3,12	303	171	132
ft10	951,60	977,13	25,53	400	87	313
la21	1059,21	1074,55	15,34	393	101	292
la24	949,02	951,64	2,62	260	198	62
la29	1184,29	1247,50	63,21	481	17	464
la40	1239,91	1242,06	2,15	263	212	51
yn1	918,64	919,43	0,79	253	232	21
yn2	940,24	941,96	1,72	264	218	46
yn3	935,74	941,39	5,65	318	170	148
yn4	1020,02	1058,02	38,00	446	48	398
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_V$ (Yamada in sod. 1994)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_V)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_V$)	N°($\mathcal{H} > \mathcal{H}_V$)	ΔN°
abz7	678,21	678,92	0,71	220	244	-24
abz8	690,47	693,05	2,58	261	207	54
abz9	713,15	711,87	-1,28	206	267	-61
ft10	951,60	977,43	25,83	387	101	286
la21	1059,21	1079,76	20,55	382	98	284
la24	949,02	951,53	2,51	247	213	34
la29	1184,29	1240,94	56,65	459	38	421
la40	1239,91	1239,46	-0,45	189	278	-89
yn1	918,64	917,14	-1,50	218	262	-44
yn2	940,24	939,44	-0,80	222	261	-39
yn3	935,74	935,71	-0,03	238	253	-15
yn4	1020,02	1054,40	34,38	398	93	305

Tabela 11.6: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 100 ms).

11.4 Rezultati tabu iskanja pri času izvajanja 500 ms

Tabela 11.7 primerja našo okolico z okolicama $\mathcal{H}_N$ (Nowicki in Smutnicki 1996) ter $\mathcal{H}_B$ (Balas 1969) ob dovoljenem času izvajanja 500 ms. Okolica $\mathcal{H}_N$ ponovno daje boljše rezultate pri instanci **ft10**, v ostalih primerih pa zaostaja.

Okolica $\mathcal{H}_B$ daje zelo slabe rezultate, zato je praktično ni potrebno komentirati.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_N$ (Nowicki in Smutnicki 1996)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_N)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_N$)	N°($\mathcal{H} > \mathcal{H}_N$)	ΔN°
abz7	669,79	674,60	4,81	418	56	362
abz8	680,33	690,02	9,69	461	25	436
abz9	696,28	702,77	6,49	405	73	332
ft10	951,09	945,49	-5,60	185	297	-112
la21	1058,45	1068,14	9,69	325	136	189
la24	949,46	954,41	4,95	307	167	140
la29	1174,53	1181,78	7,25	394	91	303
la40	1233,35	1235,68	2,33	296	93	203
yn1	903,35	907,21	3,86	360	111	249
yn2	924,99	929,61	4,62	362	121	241
yn3	911,17	916,37	5,20	366	109	257
yn4	990,94	994,20	3,26	315	166	149
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_B$ (Balas 1969)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_B)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_B$)	N°($\mathcal{H} > \mathcal{H}_B$)	ΔN°
abz7	669,79	776,03	106,24	500	0	500
abz8	680,33	724,29	43,96	477	21	456
abz9	696,28	763,66	67,38	498	2	496
ft10	951,09	1013,48	62,39	439	58	381
la21	1058,45	1145,68	87,23	469	30	439
la24	949,46	975,03	25,57	356	127	229
la29	1174,53	1469,27	294,74	500	0	500
la40	1233,35	1270,98	37,63	448	30	418
yn1	903,35	909,81	6,46	389	91	298
yn2	924,99	937,65	12,66	428	61	367
yn3	911,17	934,75	23,58	450	42	408
yn4	990,94	1153,71	162,77	498	2	496

Tabela 11.7: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 500 ms).

V tabeli 11.8 si lahko ogledamo primerjavo naše okolice z okolicama $\mathcal{H}_D$ (Dell'Amico in Trubian 1993) ter $\mathcal{H}_V$ (Yamada in sod. 1994) ob dovoljenem času izvajanja 500 ms. Naša okolica je boljša od okolice $\mathcal{H}_D$ na vseh testnih primerih. Pri instancah **ft10**, **la21**, **la29** in **yn4** je naša prednost izrazita.

Glede na okolico $\mathcal{H}_V$ naši rezultati zaostajajo samo pri pri instancah **yn1**, **yn2**. V obeh primerih so zaostanki izredno majhni. Okolica $\mathcal{H}_V$ znatno zaostaja pri reševanju primerov **ft10**, **la21**, **la29** in **yn4**. Pri instanci **abz9** sta bili okolici enakovredni.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_D$ (Dell'Amico in Trubian 1993)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_D)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_D$)	N°($\mathcal{H} > \mathcal{H}_D$)	ΔN°
abz7	669,79	672,12	2,33	312	151	161
abz8	680,33	682,96	2,63	281	198	83
abz9	696,28	698,40	2,12	304	163	141
ft10	951,09	975,50	24,41	389	102	287
la21	1058,45	1070,61	12,16	360	111	249
la24	949,46	950,59	1,13	242	224	18
la29	1174,53	1250,98	76,45	490	10	480
la40	1233,35	1236,28	2,93	228	186	42
yn1	903,35	903,62	0,27	230	221	9
yn2	924,99	925,33	0,34	237	226	11
yn3	911,17	913,06	1,89	295	180	115
yn4	990,94	1023,73	32,79	439	54	385
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_V$ (Yamada in sod. 1994)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_V)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_V$)	N°($\mathcal{H} > \mathcal{H}_V$)	ΔN°
abz7	669,79	671,28	1,49	255	202	53
abz8	680,33	683,98	3,65	253	215	38
abz9	696,28	696,28	0,00	238	226	12
ft10	951,09	975,63	24,54	387	104	283
la21	1058,45	1075,87	17,42	387	105	282
la24	949,46	952,31	2,85	231	228	3
la29	1174,53	1238,65	64,12	484	15	469
la40	1233,35	1235,22	1,87	192	209	-17
yn1	903,35	902,65	-0,70	223	249	-26
yn2	924,99	924,81	-0,18	227	245	-18
yn3	911,17	914,02	2,85	247	224	23
yn4	990,94	1036,91	45,97	417	76	341

Tabela 11.8: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 500 ms).

11.5 Rezultati tabu iskanja pri času izvajanja 1000 ms

Tabela 11.9 primerja našo okolico z okolicama $\mathcal{H}_N$ (Nowicki in Smutnicki 1996) ter $\mathcal{H}_B$ (Balas 1969) ob dovoljenem času izvajanja 1000 ms. Okolica $\mathcal{H}_N$ še vedno daje boljše rezultate pri instanci **ft10**, v ostalih primerih pa zaostaja.

Okolica $\mathcal{H}_B$ daje zelo slabe rezultate, zato je praktično ni potrebno komentirati.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_N$ (Nowicki in Smutnicki 1996)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_N)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_N$)	N°($\mathcal{H} > \mathcal{H}_N$)	ΔN°
abz7	668,37	672,76	4,39	392	67	325
abz8	677,23	686,85	9,62	479	11	468
abz9	692,64	698,34	5,70	404	78	326
ft10	951,18	946,88	-4,30	178	298	-120
la21	1057,88	1067,22	9,34	326	136	190
la24	948,54	954,74	6,20	324	155	169
la29	1174,65	1178,72	4,07	388	97	291
la40	1232,51	1235,91	3,40	312	96	216
yn1	900,41	903,97	3,56	379	92	287
yn2	921,52	926,03	4,51	387	89	298
yn3	907,14	911,11	3,97	368	101	267
yn4	984,28	989,19	4,91	374	95	279
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_B$ (Balas 1969)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_B)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_B$)	N°($\mathcal{H} > \mathcal{H}_B$)	ΔN°
abz7	668,37	778,99	110,62	499	1	498
abz8	677,23	726,60	49,37	484	11	473
abz9	692,64	758,43	65,79	494	4	490
ft10	951,18	1015,48	64,30	456	40	416
la21	1057,88	1152,68	94,80	474	22	452
la24	948,54	978,67	30,13	369	111	258
la29	1174,65	1473,46	298,81	499	1	498
la40	1232,51	1263,74	31,23	460	28	432
yn1	900,41	907,50	7,09	387	82	305
yn2	921,52	933,68	12,16	431	50	381
yn3	907,14	930,90	23,76	446	37	409
yn4	984,28	1166,48	182,20	500	0	500

Tabela 11.9: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji s tabu seznamom (izvršni čas 1000 ms).

V tabeli 11.10 si lahko ogledamo primerjavo naše okolice z okolicama $\mathcal{H}_D$ (Dell'Amico in Trubian 1993) ter $\mathcal{H}_V$ (Yamada in sod. 1994) ob dovoljenem času izvajanja 1000 ms. Naša okolica je boljša od okolice $\mathcal{H}_D$ na vseh testnih primerih. Pri instancah **ft10**, **la21**, **la29** in **yn4** je naša prednost izrazita.

Glede na okolico $\mathcal{H}_V$ naši rezultati zaostajajo samo pri pri instancah **yn1**, **yn2**. V obeh primerih so zaostanki izredno majhni. Okolica $\mathcal{H}_V$ znatno zaostaja pri reševanju primerov **ft10**, **la21**, **la29** in **yn4**. Pri instanci **abz9** sta bili okolici skoraj enakovredni.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_D$ (Dell'Amico in Trubian 1993)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_D)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_D$)	N°($\mathcal{H} > \mathcal{H}_D$)	ΔN°
abz7	668,37	670,02	1,65	307	137	170
abz8	677,23	679,53	2,30	278	190	88
abz9	692,64	694,47	1,83	301	162	139
ft10	951,18	977,11	25,93	379	110	269
la21	1057,88	1073,54	15,66	376	98	278
la24	948,54	951,80	3,26	259	205	54
la29	1174,65	1252,16	77,51	487	13	474
la40	1232,51	1234,22	1,71	227	205	22
yn1	900,41	900,56	0,15	236	221	15
yn2	921,52	922,23	0,71	260	200	60
yn3	907,14	908,38	1,24	281	179	102
yn4	984,28	1025,73	41,45	442	49	393
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_V$ (Yamada in sod. 1994)						
problem	$C_{\max}(\mathcal{H})$	$C_{\max}(\mathcal{H}_V)$	$\Delta C_{\max}$	N°($\mathcal{H} < \mathcal{H}_V$)	N°($\mathcal{H} > \mathcal{H}_V$)	ΔN°
abz7	668,37	670,80	2,43	265	182	83
abz8	677,23	681,04	3,81	240	229	11
abz9	692,64	692,80	0,16	227	224	3
ft10	951,18	976,57	25,39	387	105	282
la21	1057,88	1078,08	20,20	386	96	290
la24	948,54	952,44	3,90	251	212	39
la29	1174,65	1242,84	68,19	480	20	460
la40	1232,51	1233,53	1,02	196	233	-37
yn1	900,41	900,33	-0,08	214	231	-17
yn2	921,52	921,23	-0,29	202	257	-55
yn3	907,14	908,21	1,07	263	191	72
yn4	984,28	1032,28	48,00	432	59	373

Tabela 11.10: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji s tabu seznamom (izvršni čas 1000 ms).

11.6 Komentar rezultatov tabu iskanja

Okolica $\mathcal{H}_B$ (Balas 1969) je najstarejša od vseh primerjanih okolic in se danes praktično ne uporablja več, saj njeni rezultati znatno zaostajajo za ostalimi. Empirični testi jasno kažejo, da lastnost povezljivosti, ki jo ta okolica ima, sama zase ne zadostuje za učinkovito izvedbo optimizacije. Kot je bilo opisano v poglavju 5.1.1, je pri tej okolici lastnost povezljivosti dosežena z vsebovanjem velikega števila premikov, ki sami zase ne morejo izboljšati izvršnega časa urnika. Kratkovidno delovanje lokalnega iskanja, ki predvideva dogajanje samo en premik vnaprej, nenapredujočih premikov nikakor ne more pravilno ovrednotiti in učinkovito uporabiti v teku optimizacije.

Okolica $\mathcal{H}_N$ izkazuje boljše rezultate. Vsebuje malo premikov, ki pa so izrazito potencialno napredujoči. Postopek optimizacije je zato dobro usmerjen v preiskovanje obetavnih delov prostora rešitev in hitro doseže lokalne minimume. Pri optimizaciji problema **ft10** okolica $\mathcal{H}_N$ naravnost blesti, saj je pri časih optimizacije, večjih od 10 ms, dosegala najboljše rezultate. To dejstvo dokazuje, da predlagani algoritem tabu iskanja zelo slabo izkorišča lastnost povezljivosti, ker je preveč kratkoviden. Ugotovitev v splošnem velja za vse postopke lokalnega iskanja, kar potrjujejo opažanja številnih avtorjev. Mastrolilli in Gambardella (2000) sta s svojim algoritmom preizkusila dve njuni okolici, od katerih je bila prva podmnožica druge. Za večjo od njiju je lastnost povezljivosti veljala, za manjšo pa ne, vendar sta z manjšo od obeh dobila boljše rezultate. Manjše okolice imajo večji delež potencialno napredujočih premikov, ki jih lokalno iskanje učinkovito izkoristi. V primerjavi z našo okolico je bila okolica $\mathcal{H}_N$ slabša na vseh ostalih testih, zato lahko nedvomno zaključimo, da je naša okolica v splošnem močnejša.

Okolica $\mathcal{H}_D$ (Dell'Amico in Trubian 1993) kaže dvolično sliko. Pri instancah **abz8**, **abz9**, **la24**, **la40**, **yn1**, **yn2** in **yn3** se je izkazala za relativno uspešno, drugje pa je močno zaostajala. Empirično dejstvo, da vsak postopek lokalnega iskanja optimira uspešno samo določeno skupino problemov, ne predstavlja novosti. Jain in Meeran (1999) sta celo razdelila množico testnih problemov v skupine glede na to, katerim algoritmom povzročajo težave. Okolica $\mathcal{H}_D$ izkazuje velik

razpon uspešnosti, zaradi česar je manj ugodna za realizacijo optimizacije, ker lahko sklepamo, da bi bili njeni rezultati pri optimizaciji novih instanc zelo nepredvidliji. Naša okolica je bila od okolice $\mathcal{H}_D$ boljša pri reševanju vseh instanc, če je bil čas izvajanja večji od 50 ms. Tudi pri najmanjšem času optimizacije je naša okolica bolje optimirala devet od dvanajstih primerov, zato lahko brez dvoma trdimo, da je naša okolica močnejša.

Okolica $\mathcal{H}_V$ (Yamada in sod. 1994) je uspešnejša od okolice $\mathcal{H}_D$, čeprav velja, da je slednja njena nadmnožica. Pri reševanju problemov **abz9**, **la40**, **yn1**, **yn2** in **yn3** smo z okolico $\mathcal{H}_V$ pogosto dobili boljše rezultate kakor z našo. Vendar so zlasti pri daljših časih izvajanja optimizacije (100 ms, 500 ms in 1000 ms), njeni rezultati zgolj malenkostno boljši od rezultatov naše okolice. Pri reševanju primerov **ft10**, **la21**, **la29** in **yn4** je okolica $\mathcal{H}_V$ izrazito zaostajala za našo, zato lahko nesporno trdimo, da je naša okolica močnejša od nje.

Na podlagi rezultatov tabu iskanja lahko zaključimo, da je naša okolica močnejša od vseh predhodno primerjanih okolic. Z njo smo v večini primerov dobili najboljše rezultate. V situacijah, ko to ne drži, je bil njen zaostanek zelo majhen (razen pri času optimizacije 10 ms in morda 50 ms). Pomembno je, da je njen zaostanek majhen pri velikih časih optimizacije, kjer so dobljeni urniki relativno kvalitetni. Uspešnost naše okolice je tudi konsistentna preko celotnega nabora preizkušenih instanc (kar do neke mere izkazuje samo še okolica $\mathcal{H}_N$), zaradi česar je tudi manj verjetno, da bo izkazovala slabe rezultate pri optimizaciji neznanih instanc.

12. Rezultati uporabe genetskega algoritma

Drugi niz empiričnih testov preizkuša uspešnost genetskega algoritma v kombinaciji s tabu iskanjem, kakor je bilo opisano v poglavju 10. Majhna razlika glede na opisani postopek je v tem, da smo mutacijo izvajali s pomočjo PMB operatorja (Mattfeld 1996) in ne s postopkom, ki je bil predhodno opisan. Razlog je v tem, da našega operatorja mutacije ni možno implementirati brez popravljalne tehnike. S testi smo hoteli ugotoviti doprinos popravljalne tehnike k zmogljivosti okolice, zato je nismo hoteli vključevati v ostale dele optimizacijskega postopka.

Opisan genetski algoritem vsebuje večje število parametrov, ki jih lahko nastavimo. Izbrane vrednosti, s katerimi smo izvajali teste, so povzete v tabeli 12.1.

parameter	vrednost
ps	30
$R_{B/A}$	5.0
$cprob$	1.0
$\triangle cprob$	-0.01
$mprob$	0.3
$\triangle mprob$	0.01
$gprob$	0.03
$\triangle gprob$	0.0011
$siters$	25

Tabela 12.1: Izbira parametrov genetskega algoritma.

Poudariti želimo, da smo vse instance optimirali z istimi vrednostimi parametrov. V literaturi pogosto zasledimo primerjalne rezultate, ki so dobljeni s skrbnim prilagajanjem parametrov vsaki testni instanci posebej. V našem primeru temu ni tako, zato so dobljeni rezultati kvečjemu preveč pesimistični in bi se jih dalo s skrbno izbiro parametrov dodatno izboljšati. Izvajanje testov je potekalo na računalniku s procesorjem AMD Athlon 900 TB. Rezultati so prikazani v tabelah 12.2 in 12.3.

Posamezni stolpci v tabelah imajo naslednji pomen. V prvem stolpcu se nahaja ime testnega problema ter v drugem pripadajoči najboljši rezultat, ki ga je do tega trenutka uspel kdorkoli dobiti; vrednost v oklepaju pomeni, da je problem še odprt in zanj ni dokazano, da najboljša znana rešitev predstavlja tudi optimalno rešitev problema. Naslednji trije stolpci (najmanjši $C_{\max}$) podajajo najmanjši dobljeni čas urnika z uporabo naše okolice ($\mathcal{H}$), z uporabo primerjane okolice ($\mathcal{H}_N$, $\mathcal{H}_B$, $\mathcal{H}_D$ ali $\mathcal{H}_V$) ter razliko obeh vrednosti (Δ), pri čemer pozitivna vrednost razlike pomeni, da smo z našo okolico dobili boljši rezultat. Naslednja kategorija stolpcev (povprečni $C_{\max}$) podaja povprečni izvršni čas urnikov z uporabo naše okolice ($\mathcal{H}$), z uporabo primerjane okolice ($\mathcal{H}_N$, $\mathcal{H}_B$, $\mathcal{H}_D$ ali $\mathcal{H}_V$) ter razliko obeh vrednosti (Δ); pozitivna vrednost razlike zopet pomeni, da smo z našo okolico dobili boljše rezultate. Zadnji trije stolpci (čas) podajajo povprečni čas v sekundah, v katerem je optimizacijski algoritem našel najboljšo rešitev z uporabo naše okolice ($\mathcal{H}$), z uporabo primerjane okolice ($\mathcal{H}_N$, $\mathcal{H}_B$, $\mathcal{H}_D$ ali $\mathcal{H}_V$) ter razliko obeh vrednosti (Δ); pozitivna vrednost razlike pomeni hitrejše delovanje ob uporabi naše okolice.

Poudarimo naj, da čas iskanja najboljše rešitve ni enak času optimizacije. Terminacijski pogoj prekine postopek optimizacije, ko zadnjih *siters* (v našem primeru 25) evolucijskih iteracij ni prišlo do izboljšave najboljše rešitve.

12.1 Primerjava naše okolice z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$

Tabela 12.2 primerja našo okolico z okolicama $\mathcal{H}_N$ (Nowicki in Smutnicki 1996) ter $\mathcal{H}_B$ (Balas 1969). Vidimo, da rezultati z uporabo naše okolice niso v nobenem primeru slabši od rezultatov primerjanih okolic. Opazimo, da so najboljši dobljeni urniki pri uporabi naše in primerjane okolice v nekaterih primerih enaki. To je posledica dejstva, da je predlagani genetski algoritem dovolj močan in uspe v obeh primerih poiskati globalni minimum problema, ki se ne more več izboljšati. To velja za instance **ft10**, **la21** in **la24**. Pri reševanju instanc **abz9** in **yn2** smo z našo okolico dobili boljša rezultata od najboljših do sedaj znanih. V primeru **yn2** smo dosegli malenkostno slabši rekord (urnik z izvršnim časom 908 enot proti 907) tudi ob uporabi okolice $\mathcal{H}_B$ (Balas 1969). V vseh ostalih primerih smo z našo

okolico vedno dobili boljše rezultate kakor s primerjano. Podobno sliko kaže tudi povprečna kvaliteta rešitev (povprečni $C_{\max}$), kjer je naša okolica boljša v vseh primerih.

Povprečni čas iskanja najboljše rešitve je ob uporabi naše okolice nekoliko večji v večini primerov, kar je razumljivo. Če sta obe okolici sposobni poiskati urnik z izvršnim časom 1500 v petih sekundah, nato pa eni od obeh uspe po nadaljnjih treh sekundah poiskati urnik z izvršnim časom 1200, bo čas iskanja v prvem primeru pet sekund, v drugem pa osem. Objektivna primerjava časov optimizacije je izredno težavna (Jain in Meeran 1999), zato je pomen prikazanih podatkov zgolj vpogled v okvirno oceno hitrosti algoritmov.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_N$ (Nowicki in Smutnicki 1996)										
		najmanjši $C_{\max}$			povprečni $C_{\max}$			čas (s)		
problem		$\mathcal{H}$	$\mathcal{H}_N$	Δ	$\mathcal{H}$	$\mathcal{H}_N$	Δ	$\mathcal{H}$	$\mathcal{H}_N$	Δ
abz7	656	658	662	4	667,48	672,79	5,31	3,91	3,45	-0,46
abz8	(665)	669	670	1	676,42	682,83	6,41	4,71	4,02	-0,69
abz9	(679)	**678**	683	5	689,25	695,80	6,55	4,67	3,90	-0,77
ft10	930	930	930	0	931,66	936,61	4,96	0,35	0,29	-0,06
la21	1046	1046	1046	0	1049,47	1052,29	2,82	0,57	0,58	0,01
la24	935	935	935	0	938,57	941,43	2,87	0,52	0,56	0,03
la29	1152	1157	1163	6	1167,18	1176,58	9,40	1,85	1,46	-0,38
la40	1222	1222	1224	2	1226,56	1230,19	3,63	1,63	1,39	-0,25
yn1	(888)	888	892	4	897,16	901,43	4,27	7,18	6,82	-0,35
yn2	(909)	**907**	909	2	917,69	923,84	6,15	8,99	7,10	-1,89
yn3	(893)	893	895	2	901,97	906,99	5,02	8,33	7,05	-1,28
yn4	(968)	969	973	4	980,58	986,67	6,09	9,27	7,58	-1,69
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_B$ (Balas 1969)										
		najmanjši $C_{\max}$			povprečni $C_{\max}$			čas (s)		
problem		$\mathcal{H}$	$\mathcal{H}_B$	Δ	$\mathcal{H}$	$\mathcal{H}_B$	Δ	$\mathcal{H}$	$\mathcal{H}_B$	Δ
abz7	656	658	665	7	667,48	677,70	10,21	3,91	3,76	-0,15
abz8	(665)	669	674	5	676,42	688,34	11,92	4,71	4,40	-0,31
abz9	(679)	**678**	685	7	689,25	704,19	14,94	4,67	4,47	-0,19
ft10	930	930	930	0	931,66	937,35	5,70	0,35	0,34	-0,02
la21	1046	1046	1046	0	1049,47	1057,71	8,24	0,57	0,75	0,18
la24	935	935	935	0	938,57	941,28	2,71	0,52	0,59	0,07
la29	1152	1157	1164	7	1167,18	1207,19	40,01	1,85	1,65	-0,20
la40	1222	1222	1224	2	1226,56	1232,88	6,32	1,63	1,72	0,08
yn1	(888)	888	890	2	897,16	902,48	5,32	7,18	7,29	0,11
yn2	(909)	**907**	**908**	1	917,69	927,35	9,66	8,99	8,29	-0,71
yn3	(893)	893	896	3	901,97	909,78	7,81	8,33	8,27	-0,06
yn4	(968)	969	980	11	980,58	1002,46	21,88	9,27	8,66	-0,61

Tabela 12.2: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_N$ in $\mathcal{H}_B$ pri optimizaciji z genetskim algoritmom.

12.2 Primerjava naše okolice z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$

Že pri tabu iskanju smo videli, da sta okolici $\mathcal{H}_D$ (Dell'Amico in Trubian 1993) ter $\mathcal{H}_V$ (Yamada in sod. 1994) močnejši od prejšnjih dveh, kar se je pokazalo tudi na tem testu. Z okolico $\mathcal{H}_D$ smo pri večini problemov dobili enake najboljše rezultate kakor z našo. Pri reševanju instance **la29** je bila okolica $\mathcal{H}_D$ malenkostno boljša od naše, pri instancah **abz9**, **yn2** in **yn4** pa nekoliko slabša. Dober rezultat **la29** preseneča, saj povprečna kvaliteta rešitev pri okolici $\mathcal{H}_D$ precej zaostaja v primerjavi z našo okolico; prav pri tej instanci je njen zaostanek največji. Naša okolica je v povprečju boljša od okolice $\mathcal{H}_D$ v vseh primerih, zaradi česar je presenetljivo, da je skoraj vedno tudi hitrejša pri iskanju najboljše rešitve.

primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_D$ (Dell'Amico in Trubian 1993)										
		najmanjši $C_{\max}$			povprečni $C_{\max}$			čas (s)		
problem		$\mathcal{H}$	$\mathcal{H}_D$	Δ	$\mathcal{H}$	$\mathcal{H}_D$	Δ	$\mathcal{H}$	$\mathcal{H}_D$	Δ
abz7	656	658	658	0	667,48	668,18	0,70	3,91	4,52	0,61
abz8	(665)	669	669	0	676,42	677,92	1,50	4,71	5,77	1,05
abz9	(679)	**678**	679	1	689,25	690,73	1,47	4,67	5,54	0,87
ft10	930	930	930	0	931,66	933,46	1,80	0,35	0,47	0,11
la21	1046	1046	1046	0	1049,47	1049,97	0,50	0,57	0,85	0,28
la24	935	935	935	0	938,57	938,60	0,03	0,52	0,64	0,12
la29	1152	1157	1156	-1	1167,18	1176,60	9,42	1,85	2,76	0,91
la40	1222	1222	1222	0	1226,56	1227,14	0,58	1,63	1,87	0,24
yn1	(888)	888	888	0	897,16	897,83	0,67	7,18	7,43	0,25
yn2	(909)	**907**	**908**	1	917,69	918,70	1,01	8,99	8,66	-0,33
yn3	(893)	893	893	0	901,97	903,25	1,27	8,33	8,94	0,61
yn4	(968)	969	973	4	980,58	985,78	5,20	9,27	11,06	1,79
primerjava okolice $\mathcal{H}$ z okolico $\mathcal{H}_V$ (Yamada in sod. 1994)										
		najmanjši $C_{\max}$			povprečni $C_{\max}$			čas (s)		
problem		$\mathcal{H}$	$\mathcal{H}_V$	Δ	$\mathcal{H}$	$\mathcal{H}_V$	Δ	$\mathcal{H}$	$\mathcal{H}_V$	Δ
abz7	656	658	658	0	667,48	668,22	0,74	3,91	3,33	-0,58
abz8	(665)	669	669	0	676,42	678,04	1,62	4,71	4,11	-0,60
abz9	(679)	**678**	**678**	0	689,25	690,80	1,55	4,67	3,96	-0,71
ft10	930	930	930	0	931,66	933,47	1,81	0,35	0,35	-0,01
la21	1046	1046	1046	0	1049,47	1049,97	0,50	0,57	0,54	-0,03
la24	935	935	935	0	938,57	938,54	-0,03	0,52	0,49	-0,04
la29	1152	1157	1157	0	1167,18	1174,62	7,44	1,85	1,86	0,01
la40	1222	1222	1222	0	1226,56	1226,88	0,32	1,63	1,49	-0,14
yn1	(888)	888	888	0	897,16	897,92	0,76	7,18	6,09	-1,08
yn2	(909)	**907**	**908**	1	917,69	919,04	1,35	8,99	7,83	-1,17
yn3	(893)	893	893	0	901,97	903,43	1,46	8,33	6,88	-1,45
yn4	(968)	969	971	2	980,58	986,38	5,80	9,27	8,44	-0,83

Tabela 12.3: Primerjava okolice $\mathcal{H}$ z okolicama $\mathcal{H}_D$ in $\mathcal{H}_V$ pri optimizaciji z genetskim algoritmom.

Tako okolica $\mathcal{H}_D$ kot tudi $\mathcal{H}_V$ sta uspeli doseči rekord pri optimizaciji problema **yn2** (urnik z izvršnim časom 908, ki zaostaja za urnikom 907, dobljenim z našo okolico). S pomočjo okolice $\mathcal{H}_V$ smo tudi dobili rekord pri reševanju problema **abz9** (tako kot v primeru naše okolice ima urnik izvršni čas 678 enot). Ta okolica je pri večini primerov uspela dobiti enake najboljše rezultate, kot naša. V primeru instanc **yn2** in **yn4** je rahlo zaostajala za našo, v nobenem primeru pa ni bila boljša od nje.

Povprečne vrednosti rešitev so pri okolici $\mathcal{H}_V$ v večini primerov nekoliko slabše od naših (pri instancah **la29** in **yn4** je zaostanek izrazit), neznatno boljše so le pri reševanju problema **la24**. Zato pa so njeni časi iskanja najboljše rešitve nekoliko krajši kot pri naši okolici.

12.3 Komentar rezultatov genetskega algoritma

Tudi pri optimizaciji testnih instanc z genetskimi algoritmi se nesporno kaže večja moč naše okolice v primerjavi z ostalimi. Primerjava najmanjših izvršnih časov urnikov, kaže enako ali boljše delovanje naše okolice v vseh razen v enem primeru (reševanje problema **la29** z okolico $\mathcal{H}_D$). Pri interpretaciji rezultatov je potrebno upoštevati, da so bili testi ponovljeni petstokrat in da najmanjši izvršni čas podaja najboljši urnik izmed vseh petsto dobljenih urnikov. Rezultat je enak neodvisno od tega, ali je določena okolica tak urnik dosegla v enem ali v vseh petstotih primerih. Namen teh rezultatov je prikazati največ, kar od določene okolice lahko pričakujemo.

Natančenejši vpogled v zmogljivost okolic nam podaja povprečni izvršni čas urnikov, kjer se bolje vidi, katere okolice pogosteje generirajo kvalitetne rešitve. Tu je naša okolica samo v enem primeru (reševanje problema **la24** z okolico $\mathcal{H}_V$) malenkostno zaostala za primerjano. V vseh ostalih primerih je bila boljša. To jasno kaže, da ima naša okolica večjo sposobnost generiranja optimalnih ali blizu–optimalnih rešitev kakor primerjalne okolice.

12.4 Dodatni testi z genetskim algoritmom

Zadnji rezultati, ki jih prikazujemo, demonstrirajo uporabo popravljalne tehnike pri realizaciji operatorja mutacije. Tokrat mutacij ne izvajamo več s PMB operatorjem, ampak s postopkom, ki je bil opisan v poglavju 10. Ostali parametri genetskega algoritma so ostali enaki. Rezultati so prikazani v tabeli 12.4.

Posamezni stolpci v tabeli imajo naslednji pomen. Podobno kot prej se v prvem stolpcu nahaja ime testnega problema ter v drugem pripadajoči najboljši znani rezultat. Naslednji trije stolpci (najmanjši $C_{\max}$) podajajo najmanjši dobljeni čas urnika z uporabo našega operatorja mutacije (NM), z uporabo PMB operatorja mutacije (PMB) ter razliko obeh vrednosti (Δ), pri čemer pozitivna vrednost razlike pomeni, da smo z našim operatorjem mutacije dobili boljši rezultat. Naslednja kategorija stolpcev (povprečni $C_{\max}$) podaja povprečni izvršni čas urnikov z uporabo našega operatorja mutacije (NM), z uporabo PMB operatorja mutacije (PMB) ter razliko obeh vrednosti (Δ); pozitivna vrednost razlike zopet pomeni, da smo z našim operatorjem mutacije dobili boljše rezultate. Zadnji trije stolpci (čas) podajajo povprečni čas v sekundah, v katerem je optimizacijski algoritem našel najboljšo rešitev z uporabo našega operatorja mutacije (NM), z uporabo PMB mutacije (PMB) ter razliko obeh vrednosti (Δ); pozitivna vrednost razlike pomeni hitrejše delovanje ob uporabi našega operatorja mutacije.

		najmanjši $C_{\max}$			povprečni $C_{\max}$			čas (s)		
problem		NM	PMB	Δ	NM	PMB	Δ	NM	PMB	Δ
abz7	656	658	658	0	667,91	667,48	-0,42	5,02	3,91	-1,11
abz8	(665)	669	669	0	676,53	676,42	-0,11	6,44	4,71	-1,73
abz9	(679)	**678**	**678**	0	689,91	689,25	-0,65	6,27	4,67	-1,60
ft10	930	930	930	0	932,02	931,66	-0,36	0,35	0,35	0,01
la21	1046	1046	1046	0	1049,08	1049,47	0,39	0,65	0,57	-0,08
la24	935	935	935	0	938,60	938,57	-0,03	0,60	0,52	-0,08
la29	1152	1157	1157	0	1167,94	1167,18	-0,76	2,40	1,85	-0,55
la40	1222	1222	1222	0	1226,68	1226,56	-0,12	1,97	1,63	-0,34
yn1	(888)	**886**	888	2	897,70	897,16	-0,53	10,65	7,18	-3,47
yn2	(909)	**907**	**907**	0	918,69	917,69	-1,00	12,88	8,99	-3,88
yn3	(893)	895	893	-2	902,68	901,97	-0,71	11,39	8,33	-3,06
yn4	(968)	969	969	0	978,50	980,58	2,08	12,00	9,27	-2,73

Tabela 12.4: Primerjava mutacijskih operatorjev.

Primerjava najmanjših izvršnih časov urnika ne kaže praktično nobene izboljšave našega operatorja mutacje glede na operator PMB. Pri reševanju problema **yn1** se je naš operator izkazal boljši, pri instanci **yn3** pa je deloval slabše. Rezultati ne bi bili nič posebnega, če pri instanci **yn1** ne bi dobili nov svetovni rekord: urnik z izvršnim časom 886 v primerjavi s sedanjim najboljšim rezultatom 888. To nakazuje, da naš operator mutacije preiskuje prostor rešitev na nekoliko drugačen način od dosedanjih, saj takega rezultata nismo uspeli dobiti na nobenem testu, ki uporablja PMB mutacijski operator.

Ostali rezultati kažejo nekoliko slabše delovanje našega operatorja mutacije. Povprečna kvaliteta urnikov se je poslabšala v vseh primerih razen pri reševanju instanc **la21** in **yn4**. Tudi povprečni čas, ki je bil potreben za generiranje najboljšega urnika, se je nekoliko povečal. Delni vzrok slabih rezultatov je dejstvo, da smo izvajali oba testa z istim naborom parametrov. Naš operator mutacije deluje precej drugače od operatorja PMB, zato tudi potrebuje drugačne nastavitve.

Kljub temu rezultati demonstrirajo, da uporaba popravljalne tehnike ni omejena na realizacijo učinkovite okolice. Z njeno pomočjo lahko izvedemo tudi ostale funkcije, kot so genetski operatorji. Prikazana izvedba mutacije je zgolj začetni poskus v tej smeri.

13. Diskusija

Problem Π_{J} spada med težje kombinatorične optimizacijske probleme. To dejstvo je nedvomno vplivalo na zgodovinski razvoj postopkov za njegovo reševanje. Velika kompleksnost problema je povzročila, da so se raziskovalci pri razvoju novih postopkov optimizacije najprej osredotočali na lažje probleme, kot so trgovski potnik, omejeni nahrbtnik in večje število tehnoloških optimizacij na enem proizvodnem resorju. Tehnika lokalnega iskanja ni bila izjema. Uspešne izvedbe le-te za reševanje omenjenih lažjih problemov so se pojavile pred izvedbami za reševanje problema Π_{J}.

Bistvena razlika med omenjenimi problemi in problemom Π_{J} je v tem, da pri slednjem poznamo neizvedljive rešitve v smislu, ki ga obravnava to delo. Pri lažjih problemih je relativno enostavno definirati okolico, v kateri so vsi premiki potencialno napredujoči in hkrati doseči lastnost povezljivosti. S tako okolico je moč zasnovati dokaj uspešno dvonivojsko lokalno iskanje. Za problem Π_{J} take okolice ni mogoče definirati na ekvivalenten način. Odločimo se lahko za lastnost povezljivosti ali za uporabo samo potencialno napredujočih premikov, ne moremo pa realizirati obeh lastnosti hkrati. Posledica se kaže v manjši učinkovitosti optimizacijskih algoritmov.

Ideja o popravljalni tehniki, ki semantično predstavlja dodaten sloj v zasnovi lokalnega iskanja, se nam je porodila na osnovi tega spoznanja. Pri snovanju novih postopkov optimizacije smo želeli imeti možnost izvedbe kateregakoli premika na urniku brez bojazni, da bi zašli v neizvedljivost. To je popolna novost v razmišljanju, saj noben postopek lokalnega iskanja za problem Π_{J} te svobode niti ni imel niti je ni nakazoval. Do tega trenutka so bile razprave usmerjene v problematiko učinkovitega omejevanja okolice na izvedljive premike in v primerno uporabo le-teh. Nihče pa ni pomislil, da bi okolico razširili na množico potencialno neizvedljivih premikov, s čimer je možno v hipu odpraviti kompromis med uporabo izključno napredujočih premikov in vztrajanjem pri povezljivosti.

Pri sami realizaciji ideje je bilo potrebno rešiti nemalo problemov. Prva izvedba popravljanja urnikov je bila časovno neučinkovita. Utelešena je bila kot razširitev postopka za preračunavanje glav operacij ob izvedbi premika. Klasični postopek preračunavanja se v primeru neizvedljivega urnika zavozla ali vrne nesmiselni rezultat. To se zgodi, ko skušamo izračunati glavo operacije, katere tehnološka predhodnica še ni preračunana, kar je posledica neizvedljivega zaporedja izvajanja operacij. Težavo smo preprečili z majhno razširitvijo postopka, kjer smo med preračunavanjem glav tako stanje preverjali za vsako operacijo posebej. Če ustrezne tehnološke predhodnice še niso bile preračunane, je v urniku obstajala neizvedljiva relacija in urnik je bilo potrebno popraviti. To je bil relativno preprost poseg, kjer smo vse še ne preračunane tehnološke predhodnice obravnavane operacije razvrstili za izvajanje neposredno za zadnjimi operacijami, ki smo jih na njihovih strojih že preračunali.

Kljub temu, da je tak pristop zelo enostaven, vsebuje pomembno lastnost, da se dodatni premiki na urniku ne izvršijo preko izbire v okolici, ampak med samo izvedbo preračunavanja glav operacij. To pomeni, da se glave preračunajo samo enkrat neodvisno od števila popravljalnih premikov, ki so potrebni za restavracijo izvedljivosti urnika, s čimer pridobivamo na hitrosti. Izrazito slabo stran pristopa pa je predstavljalo dejstvo, da je bilo potrebno pri vsaki operaciji preveriti pogoj izvedljivosti, kar je zamudno. Velika večina operacij (tipično 99% ali več) je vedno izvedljiva, zato je nujnost preverjanja omenjenega pogoja razsipanje z računalniškim časom. Še več, večina premikov v okolici je navadno izvedljivih, zato je pri njih preverjanje pogojev povsem nepotrebno.

Prva izboljšava se je nakazovala sama zase. Pred izvedbo premika smo preverili, ali je le-ta izvedljiv in v tem primeru testiranje posameznih operacij povsem opustili. Hitrost izvajanja se je povečala, čeprav je preračunavanje glav v primeru neizvedljivih premikov ostalo enako potratno.

Boljšo rešitev predstavlja drugačna zasnova popravljanja, ki temelji na zasledovanju označevanja operacij. Na ta način smo uspeli izvesti popravljalni sloj praktično brez vsakega opaznega večanja kompleksnosti izvajanja premikov zaradi upočasnjenega preračunavanja glav. Označevanje operacij moramo izvesti

neodvisno od tega, ali potrebujemo popravljalno tehniko ali ne, saj v nasprotnem primeru po izvedbi premika ne moremo preurediti topološkega zaporedja. Izvedljivost premika testiramo s preverjanjem prisotnosti oznake na eni sami operaciji, torej je preverjanje enega samega preprostega logičnega pogoja edina operacija, ki se izvede dodatno v primeru, da uporabljamo potencialno neizvedljive premike. V primeru dejansko neizvedljivega premika označene operacije izkoristimo za zasledovanje vzroka neizvedljivosti, s čimer dosežemo, da se vsaka operacija premakne v urniku natančno toliko, kolikor je resnično potrebno za restavracijo izvedljivosti. Na ta način je dobljeni urnik kolikor je možno podoben izhodiščnemu, kar izboljšuje lastnost koreliranosti premikov.

Dejstvo, da je možno uvesti dodatni sloj v postopek lokalnega iskanja na tako čist in brezkompromisen način, je po svoje presenetljivo, saj opisana popravljalna tehnika postavi problem Π_J ob bok trgovskemu potniku in ostalim problemom, ki neizvedljivosti ne poznajo. Z njeno uporabo se lahko pri snovanju lokalnega iskanja koncentriramo izključno na vprašanje, katere premike naj nad urnikom izvajamo, ne da se morali omejevati na za izvedbo optimizacije manj ugodno množico izvedljivih premikov.

Uporaba popravljalne tehnike ni omejena samo na izboljšavo okolice urnika. V tem delu smo izkoristili njene možnosti za izvedbo mutacijskega operatorja pri realizaciji genetskega algoritma. Zelo verjetno obstajajo tudi drugi načini njene uporabe, ki se bodo pokazali s časom.

14. Predlogi za nadaljevanje raziskav

Že dolgo je znano, da sedanji postopki lokalnega iskanja preiskujejo prostor rešitev v relativno majhni okolici začetne rešitve, zaradi česar je lastnost povezljivosti slabo izkoriščena. To so pokazali tudi empirični testi, ki smo jih predstavili v tem delu, saj so nepovezljive okolice velikokrat dajale boljše rezultate od povezljivih. Na tej osnovi predlagamo raziskave v smeri učinkovitega razširjanja preiskovalnega območja postopkov lokalnega iskanja na večji del prostora rešitev. Popravljalna tehnika zanesljivo odpira nove možnosti na tem področju, saj omogoča izvajanje širšega spektra modifikacij na urniku, kot je bilo omogočeno do sedaj.

V tem delu smo razvili popravljalno tehniko za implementacijo učinkovite okolice, ki je zasnovana izključno na analizi kritične poti in kritičnih blokov, tako kot vse dosedanje okolice. Koristno bi bilo raziskati prednosti in slabosti razširitve okolice z množico premikov, ki kritične poti same zase ne spreminjajo.

Dejstvo, da lahko varno izvedemo katerikoli premik na urniku, odpira povsem nove možnosti snovanja optimizacijskih postopkov. Predhodno smo nakazali možnost uporabe popravljalne tehnike pri realizaciji mutacijskega operatorja, s čimer njene možnosti zanesljivo še niso izkoriščene v celoti. Podobno bi lahko realizirali operator križanja, ki s staršev na potomce ne bi prenašal delov kromosomov ampak relacije med operacijami. Popravljalna tehnika bi poskrbela, da bi bili rezultirajoči urniki zanesljivo izvedljivi.

15. Izvirni prispevki

1. Naš postopek optimizacije Π_J problema je prvi, ki uporablja popravljalno tehniko za izvedbo poljubnih premikov na urniku (algoritma 3 na sliki 6.3 in 4 na sliki 6.4). Dosedanji postopki so se morali omejevati na množico izvedljivih premikov, s čimer je bil del zmožnosti lokalnega iskanja že v osnovi zavržen. Naš pristop teoretično utemeljujeta teorema 10 in 11 s pripadajočima dokazoma.

2. Dodatne premike na urniku določamo z zasledovanjem vzroka neizvedljivosti, kar je novost. Dosedanji postopki so vse premike izvajali preko izbire v okolici. Zasledovanje vzroka omogoča izbiro premikov brez predhodnega preračunavanja glav in repov operacij, kar postopek pohitri.

3. Prvi smo realizirali okolico Π_J problema z lastnostjo povezljivosti, v kateri so vsi premiki potencialno napredujoči. S tem je razrešen več desetletij prisoten kompromis med obema lastnostima. Dosedanje okolice so bile lahko povezljive ali so imele vse premike potencialno napredujoče, vendar niso mogle združiti obeh lastnosti. Našo okolico teoretično utemeljujejo teoremi 12, 13 in 15 s pripadajočimi dokazi.

4. Realizirali smo operator mutacije, ki izvaja spremembe direktno na urniku in ne na genskem zapisu. Empirično smo pokazali, da je z njegovo uporabo preiskovanje prostora rešitev vsaj v nekaterih primerih bolj učinkovito, kot s priporočenim operator PMB.

5. S postopkom optimizacije, ki ga predlagamo, smo dosegli rekordne rezultate na treh mednarodno razširjenih in intenzivno uporabljanih testnih Π_J instancah.

Literatura

[1] Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[2] David Applegate and William Cook. A computational study of job-shop scheduling. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[3] James Edward Baker. Adaptive selection methods for genetic algorithms. In John Grefenstette, editor, *Proceedings of the* 1^{st} *International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[4] K. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[5] Mahendra S. Bakshi and Sant Ram Arora. The sequencing problem. *Management Science*, 16(4):247–263, 1969.

[6] E. Balas, J.K. Lenstra, and A. Vazacopoulos. The one-machine problem with delayed precedence constraints and its use in job shop scheduling. *Management Science*, 41(1):94–109, 1995.

[7] Egon Balas. Machine scheduling via disjunctive graphs: An implicit enumeration algorithm. *Operations Research*, 17:941–957, 1969.

[8] Egon Balas and Alkis Vazacopoulos. Guided local search with shifting bottleneck for job shop scheduling. *Management Science*, 44:262–275, 1998.

[9] Jeffrey R. Barker and Graham B. McMahon. Scheduling the general job-shop. *Management Science*, 31(5):594–598, 1985.

[10] J. Wesley Barnes and John B. Chambers. Solving the job shop scheduling problem with tabu search. *IEEE Transactions*, 27:257–263, 1995.

[11] Christian Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *OR-Spektrum*, 17:87–92, 1995.

[12] Christian Bierwirth, Dirk Christian Mattfeld, and Herbert Kopfer. On permutation representations for scheduling problems. Technical report, University of Bremen, Dept. of Economics, D-28334 Bremen, Germany, 1996.

[13] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93:1–33, 1996.

[14] Peter Brucker, Bernd Jurisch, and Andreas Krämer. The job-shop problem and immediate selection. *Annals of Operations Research*, 50:73–114, 1994b.

[15] Peter Brucker, Bernd Jurisch, and Bernd Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete applied mathematics*, 49:107–127, February 1994a.

[16] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, February 1989.

[17] Jacques Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11:42–47, 1982.

[18] Yih-Long Chang, Toshiyuki Sueyoshi, and Robert S. Sullivan. Ranking dispatching rules by data envelopment analysis in a job shop environment. *IIE Transactions*, 28:631–642, 1996.

[19] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms—i. representation. *Computers & industrial engineering*, 30:983–997, 1999a.

[20] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms, part ii: hybrid genetic search strategies. *Computers & industrial engineering*, 36:343–364, 1999b.

[21] Nicos Christofides. *Graph Theory: An Algorithmic Approach*. Academic Press, London, 1975.

[22] Wallace Clark. *The Gantt Chart: A working tool of management*. The Ronald Press Company, New York, 1922.

[23] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts London, England, 1997.

[24] Federico Della Croce, R. Tadei, and G. Volta. A genetic algorithm for the job shop problem. *Computers Ops. Res.*, 22(1):15–24, 1995.

[25] S. Dauzère-Pérès and J.B. Lasserre. A modified shifting bottleneck procedure for job-shop scheduling. *International Journal of Production Research*, 31(4):923–932, 1993.

[26] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the* 9^{th} *International Joint Conference on Artificial Intelligence*, pages 162–164, 1985a.

[27] L. Davis. Job shop scheduling with genetic algorithms. In John Grefenstette, editor, *Proceedings of the* 1^{st} *International Conference on Genetic Algorithms and Their Applications*, pages 136–140. Lawrence Erlbaum Associates, Hillsdale, NJ, 1985b.

[28] Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[29] Mauro Dell'Amico and Marco Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[30] Ulrich Dorndorf and Erwin Pesch. Evolution based learning in a job shop scheduling environment. *Computers and Operations Research*, 22(1):25–40, 1995.

[31] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A promising genetic algorithm approach to job-shop scheduling, rescheduling and open-shop scheduling problems. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 375–382, 1993.

[32] S. Alireza Feyzbakhsh and Masayuki Matsui. Adam–Eve-like genetic algorithm: a methodology for optimal design of a simple flexible assembly system. *Computers & Industrial Engineering*, 36:233–258, 1999.

[33] H. Fisher and G.L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In *Industrial Scheduling*, pages 225–251, 1963.

[34] M. Florian, P. Trépant, and G. McMahon. An implicit enumeration algorithm for the machine sequencing problem. *Management Science Application Series*, 17:B782–B792, 1971.

[35] Simon French. *Sequencing and Scheduling: An introduction to the mathematics of the Job-Shop.* Ellis Horwood, John Willey & Sons, New York, 1982.

[36] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Franzisco, 1979.

[37] Mitsuo Gen and Runwei Cheng. *Genetic Algorithms and Engineering Design.* John Wiley & Sons, 1997.

[38] B. Giffler and G. L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8:487–503, 1960.

[39] Fred Glover. Tabu search—part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[40] Fred Glover. Tabu search—part ii. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[41] Fred Glover. Tabu search fundamentals and uses. Technical report, Working Paper, College of Bussines and Administration and Graduate School of Bussines Administration, 1995.

[42] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[43] David E. Goldberg and Robert Lingle. Alleles, loci, and the traveling salesman problem. In John Grefenstette, editor, *Proceedings of the* 1^{st} *International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[44] J. Grabowski, E. Nowicki, and S. Zdrzalka. A block approach for single machine scheduling with release dates and due dates. *European Journal of Operational Research*, 26:278–285, 1986.

[45] John Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 42–60. Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[46] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In John Grefenstette, editor, *Proceedings of the* 1^{st} *International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[47] Oliver Holthaus. Scheduling in job shops with machine breakdowns: an experimental study. *Computers & Industrial Engineering*, 36:137–162, 1999.

[48] J.R. Jackson. An extension of Johnson's result on job lot scheduling. *Naval Research Logistics Quarterly*, 3(3):201–203, 1956.

[49] Anant S. Jain. *A Multi-Level Hybrid Framework for the Deterministic Job-Shop Scheduling Problem*. PhD thesis, Department of Applied Physics and Electronic and Mechanical Engineering, University of Dundee, UK, 1998.

[50] A.S. Jain and S. Meeran. Deterministic job-shop scheduling: Past, present and future. *European journal of operational research*, 113:390–434, 1999.

[51] A.S. Jain, B. Rangaswamy, and S. Meeran. New and "stronger" job-shop neighbourhoods: A focus on the method of nowicki and smutnicki (1996). *Journal of Heuristics*, 6:457–480, 2000.

[52] S.M. Johnson. Optimal two- and three-stage production schedules with set-up times included. *Naval Research Logistics Quarterly*, 1:61–68, 1954.

[53] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[54] M. Kolonko. Some new results on simulated annealing applied to the job shop scheduling problem. *European Journal of Operational Research*, 113:123–136, 1999.

[55] Evald Mark Koren. *Magistrsko delo*. Fakulteta za elektrotehniko, Univerza Edvarda Kardelja v Ljubljani, 1985.

[56] K. Udaya Kumar. *Plama: an expert system for the design of galvanic process plants*. PhD thesis, Faculty of electrical engineering, Edvard Kardelj University of Ljubljana, Yugoslavia, 1983.

[57] Peter J. M. Van Laarhoven, Emile H. L. Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125, 1992.

[58] B.J. Lageweg. Private Communications with P.J.M. Van Laarhooven, E.H.L. Aarts and J.K. Lenstra, discussing the acchievement of a makespan of 930 for ft10, 1984.

[59] B.J. Lageweg, J.K. Lenstra, and A.H.G Rinnooy Kan. Job-shop scheduling by implicit enumeration. *Management Science*, 24(4), 1977.

[60] S. Lawrence. Supplement to resource constrained project scheduling: An experimental invastigation of heuristic scheduling techniques. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1984.

[61] J. K. Lenstra and A. H. G. Rinnooy Kan. Computational complexity of discrete optimization problems. *Annals of Discrete Mathematics*, 4:121–140, 1979.

[62] Paul Martin and David B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings of the Fifth International IPCO Conference*, 1996.

[63] Monaldo Mastrolilli and Luca Maria Gambardella. Effective neighbourhood functions for the flexible job shop problem. *Journal of Scheduling*, 3:3–20, 2000.

[64] Hirofumi Matsuo, Chank Juck Suh, and Robert S. Sullivan. A controlled search simulated annealing method for the general jobshop scheduling problem. Technical report, Department of Management, Graduate School of Business, The University of Texas at Austin, Austin, TX 78712, 1988.

[65] D.C. Mattfeld, C. Bierwirth, and H. Kopfer. A search space analysis of the job shop scheduling problem. *Annals of Operations Research*, 86:441–453, 1999.

[66] Dirk C. Mattfeld, Herbert Kopfer, and Christian Bierwirth. Control of parallel population dynamics by social-like behavior of ga-individuals. In *PPSN'3 Proceedings of the Third International Conference on Parallel Problem Solving bkMortonHeuristicfrom Nature*, 1994.

[67] Dirk Christian Mattfeld. *Evolutionary Search and the Job Shop: Investigations on Genetic Algorithms for Production Scheduling.* Physica-Verlag, Heidelberg, Germany, 1996.

[68] G.B. McMahon and M. Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 3:475–482, 1975.

[69] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21:1087–1092, 1953.

[70] Thomas E. Morton and David W. Pentico. *Heuristic Scheduling Systems.* John Wiley & Sons, 1993.

[71] J.K. Muth and G.L. Thompson. *Industrial Scheduling.* Prentice-Hall, Englewood Cliffs, New York, 1963.

[72] Ryohei Nakano and Takeshi Yamada. Conventional genetic algorithm for job shop problems. In M. K. Kenneth and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms and their Applications*, pages 474–479. San iego, USA, 1991.

[73] S.S. Panwalkar and W. Iskander. A survey of scheduling rules. *Operations Research*, 25:45–61, 1977.

[74] R.G. Parker. *Deterministic Scheduling.* Chapman and Hall, 1995.

[75] Ferdinando Pezzella and Emanuela Merelli. A tabu search method guided by shifting bottleneck for the job shop scheduling problem. *European Journal of Operational Research*, 120:297–310, 2000.

[76] Michael Pinedo. *SCHEDULING; Theory, Algorithms, and Systems.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1995.

[77] Gregory J.E. Rawlins. *Foundations of Genetic Algorithms.* Morgan Kaufmann Publishers, 1991.

[78] B. Roy and B. Sussmann. Les problèmes d'ordonnancement avec contraintes disjonctives. Note D.S. no. 9 bis, SEMA, Paris, France, Décembre, 1964.

[79] Norman M. Sadeh and Yoichiro Nakakuki. Focused simulated annealing search: An application to job shop scheduling. *Annals of Operations Research*, 63:77–103, 1996.

[80] Norman M. Sadeh and Yoichiro Nakakuki. Learning to recognize (un)promising simulated annealing runs: Efficient search procedure for job shop scheduling and vehicle routing. *Annals of Operations Research*, 75:189–208, 1997.

[81] Bruce Schneier. *Applied Cryphography, second edition.* John Wiley & Sons, Inc., 1996.

[82] R. H. Storer, S. D. Wu, and R. Vaccari. New search spaces for sequencing problems with applications to job-shop scheduling. *Management Science*, 38(10), 1992.

[83] É. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993.

[84] É. D. Taillard. Parallel taboo search techniques for the job shop scheduling problem. Technical report, International Research Report ORWP89/11, Départment de Mathématiques (DMA), École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 1989.

[85] É. D. Taillard. Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing*, 6(2):108–117, 1994.

[86] Huub M. M. ten Eikelder, Bas J. M. Aarts, Marco G. A. Verhoeven, and Emile H. L. Aarts. Sequential and parallel local search algorithms for job shop scheduling. In *MIC'97 second International Conference on Meta-heuristics*, pages 75–80, Sophia-Antipolis, France, Jul 1997.

[87] Peter Šuhel, Tomaž Slivnik, Evald M. Koren, and Marko Jagodič. *Integralni proizvodni sistemi*. TORI Ljubljana, 1989.

[88] R.J.M Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by local search. *Informs Journal on computing*, 8:302–317, 1996.

[89] Robert Johannes Maria Vaessens. *Generalized Job Shop Scheduling: Complexity and Local Search*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.

[90] V. Černy. Thermodynamical approach to the traveling salesman problem. *Journal of Optimization Theory Applications*, 45:41–51, 1985.

[91] F. Werner and A. Winkler. Insertion techniques for the heuristic solution of the job-shop problem. *Discrete Applied Mathematics*, 58:191–211, 1995.

[92] K. Preston White and Ralph V. Rogers. Job-shop scheduling: limits of the binary disjunctive formulation. *International Journal of Production Research*, 28(12):2187–2200, 1990.

[93] T. Yamada and R. Nakano. *Parallel Problem Solving from Nature, 2*, chapter A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems. Elsevier Science Publishers B. V., 1992.

[94] T. Yamada and R. Nakano. Job-shop scheduling by simulated annealing combined with deterministic local search. In *Metaheuristics International Conference (MIC'95)*, pages 344–349, Hilton, Breckenridge, Colorado, USA, 1995a.

[95] T. Yamada, B.E. Rosen, and R. Nakano. A simulated annealing approach to job-shop scheduling using critical block transition operators. In *IEEE International Conference on Neural Networks (ICNN'94)*, pages 4687–4692, Orlando, Florida, USA, 1994.

[96] Takeshi Yamada and Ryohei Nakano. A fusion of crossover and local search. In *IEEE International Conference on Industrial Technology Shanghai, China*, pages 426–430, 1996a.

[97] Takeshi Yamada and Ryohei Nakano. *Meta-heuristics: theory & applications*, chapter Job-shop Scheduling by Simulated Annealing Combined with Deterministic Local Search. Kluwer academic publishers, 1996a.

[98] Takeshi Yamada and Ryohei Nakano. Scheduling by genetic local search with multi-step crossover. In *The Fourth International Conference on Parallel Problem Solving from Nature*, pages 960–969, Berlin, Germany, September 1996b.

Izjava

Izjavljam, da sem doktorsko disertacijo izdelal samostojno pod vodstvom mentorja prof. dr. Petra Šuhla. Izkazano pomoč ostalih sodelavcev sem v celoti navedel v zahvali.

V Ljubljani, 22. avgust 2006 Boštjan Murovec