OFF-LINE NALOGA — NAJKRAJŠI SKUPNI NADNIZ

Opis problema. Danih je k vhodnih nizov, ki jih označimo s $t^1, \ldots, t^k$. Množico vseh znakov, ki se pojavijo v vsaj enem vhodnem nizu, imenujmo *abeceda* in jo označimo s Σ, njeno velikost pa z $a = |\Sigma|$. Iščemo čim krajši skupni nadniz vseh vhodnih nizov, torej tak niz s, ki vsebuje vse nize $t^1, \ldots, t^k$ kot podnize. Pri tem so dovoljene tudi nestrnjene pojavitve podniza v nadnizu, torej take, pri katerih znaki podniza ne stojijo skupaj, ampak so med njimi še kakšni drugi znaki. (Na primer: niz *baa* se pojavlja kot podniz v nizu *banana*, in to celo na tri načine: $\underline{b}\underline{a}n\underline{a}na$, $\underline{b}\underline{a}nan\underline{a}$ in $\underline{b}an\underline{a}n\underline{a}$.)

Uvod. Problem iskanja najkrajšega skupnega nadniza je NP-težak [11, 16] — še več, tak ostane celo, če ga na razne načine omejimo, npr. če se omejimo na abecedo z dvema znakoma, na vhodne nize dolžine največ 3 ali pa če prepovemo, da bi se v vhodnih nizih kdaj pojavljali po dve zaporedni enaki črki [18, 13, 10]. Za iskanje optimalne rešitve (najkrajši skupni nadniz sploh) poznamo zato le algoritme z eksponentno časovno zahtevnostjo (v odvisnosti od števila vhodnih nizov), ki so primerni za manjše testne primere, za večje pa so prepočasni. Obstaja pa tudi več algoritmov, ki poskušajo najti čim krajši (vendar ne nujno najkrajši) skupni nadniz in ki so dovolj hitri, da so uporabni tudi na večjih testnih primerih. V splošnem je težko reči, kateri od njih je boljši (saj dajejo na nekaterih testnih primerih boljše rezultate eni algoritmi, na drugih drugi; poleg tega so nekateri od teh algoritmov za največje testne primere vendarle prepočasni), zato je v praksi najbolje, če jih preizkusimo več in pri vsakem testnem primeru obdržimo najboljšega izmed tako dobljenih nadnizov.

Preden nadaljujemo, še opomba glede notacije. Za poljuben niz u naj pomeni $|u|$ njegovo dolžino, u_i ali $u[i]$ njegov i-ti znak (za $i = 1, \ldots, |u|$), $u_{i..j}$ ali $u[i..j]$ pa podniz $u_i u_{i+1} \ldots u_{j-1} u_j$. Dolžine vhodnih nizov $t^1, \ldots, t^k$ bomo označili z $n_1, \ldots, n_k$.

Optimalna rešitev za dva niza. Recimo, da imamo samo dva vhodna niza, s in t. Označimo njuni dolžini z $n = |s|$ in $m = |t|$. Razmišljamo lahko takole: najkrajši skupni nadniz nizov s in t se mora končati bodisi na zadnji znak s-ja bodisi na zadnji znak t-ja. V prvem primeru imamo torej nadniz oblike us_n, pri čemer mora biti u najkrajši skupni nadniz nizov $s[1..n-1]$ in t, v drugem primeru pa imamo nadniz oblike ut_m, pri čemer mora biti u najkrajši skupni nadniz nizov s in $t[1..m-1]$. Če pa sta zadnja znaka obeh nizov enaka, $s_n = t_m$, je dovolj že, če za u vzamemo najkrajši skupni nadniz nizov $s[1..n-1]$ in $t[1..m-1]$. V vsakem primeru smo se torej znašli pred problemom enake oblike kot na začetku, le da namesto nizov s in t zdaj gledamo neka prefiksa (začetka) teh nizov. S takšnim razmišljanjem lahko nadaljujemo proti vse krajšim nizom, dokler ne pridemo do trivialno preprostih problemov, ki jih znamo rešiti; na primer, če je eden od nizov prazen, je najkrajši skupni nadniz kar enak drugemu nizu.

Označimo s $f(i,j)$ dolžino najkrajšega skupnega nadniza nizov $s[1..i]$ in $t[1..j]$. Potem lahko razmislek iz prejšnjega odstavka povzamemo takole:

$$f(i,j) = 1 + \begin{cases} f(i-1, j-1), & \text{če } s_i = t_j \\ \min\{f(i-1,j), f(i,j+1)\} & \text{sicer.} \end{cases}$$

Robni primeri (ko je eden od nizov prazen) pa so $f(0, j) = j$ in $f(i, 0) = i$. Vrednosti funkcije f bi lahko računali z rekurzivnim podprogramom (funkcijo); da ne bomo računali enih in istih vrednosti po večkrat, si vsako vrednost $f(i, j)$, ko jo prvič izračunamo, zapomnimo v neki tabeli, odkoder jo bomo lahko kasneje prebrali, kadarkoli jo bomo spet potrebovali. Še bolj elegantno kot z rekurzivnim podprogramom pa je, če računamo f sistematično po naraščajočih i in pri vsakem i po naraščajočih j. Tako bomo vedno, ko računamo funkcijo za nek (i, j), že izračunane rešitve za vse tiste podprobleme, ki jih pri njem potrebujemo. Prišli smo do naslednje rešitve z dinamičnim programiranjem:

algoritem O_2
1 **for** $j := 0$ **to** m **do** $f[0, j] := j$;
2 **for** $i := 1$ **to** n:
3 $\quad f[i, 0] := i$;
4 $\quad$ **for** $j := 1$ **to** m:
5 $\quad\quad$ **if** $s_i = t_j$ **then** $q := f[i-1, j-1]$;
6 $\quad\quad$ **else** $q := \min\{f[i-1, j], f[i, j-1]\}$;
7 $\quad\quad f[i, j] := q + 1$;

Za f torej potrebujemo tabelo $(n+1) \times (m+1)$ elementov. Na koncu tega postopka imamo v $f[n, m]$ dolžino najkrajšega skupnega nadniza nizov s in t. Naloga zahteva, da izpišemo primeren nadniz, ne le izračunamo njegovo dolžino. Pri sestavljanju takega nadniza lahko razmišljamo takole: recimo spet, da iščemo najkrajši skupni nadniz nizov $s[1..i]$ in $t[1..j]$. Če je $s_i = t_j$, je zadnji znak nadniza pač ta znak, pred tem pa imamo najkrajši skupni nadniz nizov $s[1..i-1]$ in $[1..j-1]$; torej zmanjšamo i in j za 1 in nadaljujemo s podobnim postopkom tudi tam. Če pa sta znaka s_i in t_j različna, moramo pogledati, katera od vrednosti $f[i-1, j]$ in $f[i, j-1]$ je manjša: če je $f[i-1, j] < f[i, j-1]$, moramo na zadnje mesto nadniza postaviti s_i (in pri nadaljevanju postopka zmanjšati i za 1), sicer pa t_j (in pri nadaljevanju postopka zmanjšati j za 1). (Če je $f[i-1, j] = f[i, j-1]$, je načeloma vseeno, ali za zadnji znak nadniza uporabimo s_i ali t_j; v obeh primerih bi nastal enako dolg nadniz, vendar pa sta tako dobljena nadniza različna.) Zapišimo s psevdokodo še ta del postopka:

algoritem O_2 (*nadaljevanje*)
8 $i := n$; $j := m$; alocirajmo izhodni niz u dolžine $f[n, m]$;
9 **while** $i > 0$ **or** $j > 0$:
10 $\quad q := f[i, j]$;
11 $\quad$ **if** $i = 0$ **then** $u_q := t_j$; $j := j - 1$;
12 $\quad$ **else if** $j = 0$ **then** $u_q := s_i$; $i := i - 1$;
13 $\quad$ **else if** $s_i = t_j$ **then** $u_q := s_i$; $i := i - 1$; $j := j - 1$;
14 $\quad$ **else if** $f[i-1, j] < f[i, j-1]$ **then** $u_q := s_i$; $i := i - 1$;
15 $\quad$ **else** $u_q := t_j$; $j := j - 1$;
16 **return** u;

Mimogrede, najkrajši skupni nadniz dveh nadnizov seveda ni nujno enolično določen. (Na primer, iz *miza* in *zima* lahko dobimo *mizima* ali pa *zimiza*.) V našem gornjem postopku se to pokaže v vrsticah 14 in 15: to, ali za u_q vzamemo s_i ali t_j, je odvisno od tega, katera od vrednosti $f[i-1, j]$ in $f[i, j-1]$ je manjša. Če sta obe enaki, pa se lahko odločimo za eno ali za drugo možnost; tako lahko dobimo različne, vendar enako dolge nadnize. (V zgornji psevdokodi se v takih primerih vedno odločimo za t_j. Še ena možnost bi bila, da bi to odločitev prepustili generatorju naključnih števil.) To, katerega od več možnih enako dolgih najkrajših skupnih nadnizov dobimo, ni nujno čisto vseeno: nekateri od njih utegnejo dajati boljše, nekateri pa slabše rezultate v nadaljevanju postopka (npr. če bomo kasneje iskali skupni nadniz med dosedanjim nadnizom in kakšnim tretjim vhodnim nizom).

Doslej opisani postopek vrne najkrajši skupni podniz dveh vhodnih nizov in pri tem porabi $O(nm)$ pomnilnika (za tabelo f) in časa. Če sta vhodna niza zelo dolga, si toliko pomnilnika ne moremo privoščiti: na primer, pri nekaterih naših testnih primerih smo imeli dva niza dolžine približno 500 000 znakov; takrat bi imela tabela f približno 250 milijard elementov in če je vsak dolg 4 byte, je to skupaj približno 1 TB, toliko pomnilnika pa običajni računalniki dandanes nimajo.

Našo dvodimenzionalno tabelo f si lahko predstavljamo kot razdeljeno na vrstice in stolpce, tako da je indeks i številka vrstice, indeks j pa številka stolpca. Porabo pomnilnika lahko zmanjšamo z naslednjim opažanjem: ko naš gornji postopek računa vrednosti v i-ti vrstici tabele f, uporablja pri tem že izračunane vrednosti iz iste vrstice in iz prejšnje, torej $(i-1)$-ve vrstice; ne uporablja pa več vrednosti iz vrstic $i-2$, $i-3$ in tako naprej. Zato lahko vsebino teh vrstic sproti pozabljamo in tako sprostimo pomnilnik, ki bi ga sicer potrebovali za hranjenje teh vrstic. Za izračun funkcije f je dovolj že, če hranimo v pomnilniku le dve vrstici naenkrat (vrstico i, ki jo trenutno računamo, in poleg nje še prejšnjo vrstico $i-1$). Težava pa je, da se bomo morali na koncu sprehoditi nazaj po tabeli, da bomo sestavili primeren nadniz u. Takrat bomo morali torej vrstice, ki smo jih prej zavrgli in pozabili, izračunati ponovno.

Tabelo f razdelimo v mislih na *bloke* po b vrstic. Že izračunane vrstice tabele f bomo sproti pozabljali, le prvo vrstico vsakega bloka (torej $i = 0, b, 2b, 3b, \ldots$) si bomo zapomnili. V drugem delu postopka, ko sestavljamo nadniz u in se pri tem premikamo po tabeli navzgor (zmanjšujemo i), lahko vsakič, ko pri premiku navzgor dosežemo nov blok, ta blok v celoti izračunamo iz njegove prve vrstice (ki jo imamo še shranjeno v pomnilniku); spodnji blok, iz katerega smo ravnokar prišli, pa lahko zdaj pozabimo, saj ga ne bomo več potrebovali. Poraba pomnilnika je torej zdaj takšna: imamo približno n/b blokov in od vsakega si moramo zapomniti po eno vrstico, poleg tega pa še celoten trenutni blok s približno b vrsticami. Tako torej porabimo $O((b + n/b)m)$ pomnilnika. Vidimo lahko, da bo ta poraba najnižja pri $b \approx \sqrt{n}$; takrat dobimo $O(m\sqrt{n})$. Če sta vhodna niza s in t različno dolga, je koristno za s vzeti daljšega od njiju. Časovna zahtevnost tega postopka je še vedno le $O(nm)$, vendar moramo zdaj vsak blok izračunati dvakrat (najprej pri premikanju dol po tabeli, ko prvič računamo tabelo f, nato pa še enkrat pri premikanju gor po tabeli, ko sestavljamo nadniz u), zato si lahko predstavljamo, da se bo izvajal približno dvakrat dlje kot prvotni postopek. Za naše testne primere s po dvema nizoma dolžine 500 000 je to čisto primerna rešitev (porabi približno 2,6 GB pomnilnika). Zapišimo ta postopek še s psevdokodo:

algoritem O_2-S
  for $i := 0$ **to** n:
    izračunaj vrstico $f[i]$ s pomočjo vrednosti iz vrstice $f[i-1]$;
    pozabi vrstico $f[i-1]$, razen če je $(i-1)$ večkratnik b;

$i := n$; $j := m$; alocirajmo izhodni niz u dolžine $f[n, m]$;
while $i > 0$ **or** $j > 0$:
  if $i > 0$ in je i večkratnik b in vrstice $i - 1$ še nimamo v pomnilniku:
    (* *Izračunajmo ponovno zgornji blok.* *)
    for $i' := i - b + 1$ **to** $i - 1$:
      izračunaj vrstico $f[i']$ s pomočjo vrednosti iz vrstice $f[i' - 1]$;
    (* *Pozabimo spodnji blok, ki ga ne bomo več potrebovali.* *)
    for $i' := i + 1$ **to** $\min\{i + b, n\}$:
      pozabi vrstico $f[i']$;
  določi trenutni znak niza u enako kot v vrsticah 10–15 prejšnjega algoritma;
return u;

Za naše namene je ta rešitev čisto dovolj dobra, kot zanimivost pa omenimo, da obstajajo še varčnejši algoritmi, na primer Hirschbergov, ki porabi $O(nm)$ časa (enako kot naš gornji algoritem), vendar le $O(m + n)$ prostora [6]. Težje pa je zmanjšati porabo časa; pokazati je mogoče, da v najslabšem primeru za iskanje najkrajšega skupnega nadniza dveh nizov nujno potrebujemo $O(nm)$ časa [1], obstaja pa precej algoritmov, ki skušajo zmanjšati porabo časa pod to mejo vsaj za nekatere pare nizov (glej npr. pregled v [18] in tam navedeno literaturo), na primer take, kjer je pogoj $s_i = t_j$ izpolnjen le pri majhnem deležu parov (i, j).

Optimalna rešitev za več nizov. Dosedanje rešitve za dva niza ni težko posplošiti na več nizov. Recimo, da imamo k vhodnih nizov $t^1, \ldots, t^k$ z dolžinami $n_1, \ldots, n_k$. Podobno kot prej si zastavimo podproblem: naj bo $f(i_1, \ldots, i_k)$ dolžina najkrajšega skupnega nadniza nizov $t^1[1..i_1], \ldots, t^k[1..i_k]$. Tak nadniz se mora končati na enega od znakov $t^r[i_r]$ (za $r = 1, \ldots, k$); če se konča recimo na znak c, smo s tem pokrili $t^r[i_r]$ pri tistih r, kjer je $t^r[i_r] = c$, zato se moramo pri tistih nizih premakniti za eno mesto nazaj. Lažje kot s formulo je to opisati s postopkom:

funkcija $f(i_1, \ldots, i_k)$:
  $C := \{\}$; **for** $r := 1$ **to** k **do if** $i_r > 0$ **then** dodaj $t^r[i_r]$ v C;
  if C je prazna **then return** 0;
  $g := \infty$;
  za vsak $c \in C$:
    for $r := 1$ **to** k:
      if $i_r > 0$ **and** $t^r[i_r] = c$
      then $j_r := i_r - 1$ **else** $j_r := i_r$;
      $g := \min\{g, f(j_1, \ldots, j_k)\}$;
  return $g + 1$;

Vidimo lahko, da ko rešujemo podproblem $(i_1, \ldots, i_k)$, si pri tem pomagamo z rešitvami takšnih podproblemov $(j_1, \ldots, j_k)$, pri katerih je vsaj j_r enak bodisi i_r bodisi $i_r - 1$. Podobno kot pri rešitvi za dva niza je torej koristno reševati te podprobleme po naraščajočih indeksih in si rezultate shranjevati v tabelo; ta bo zdaj k-dimenzionalna in velika $(i_1 + 1) \times (i_2 + 1) \times \ldots \times (i_k + 1)$ elementov.

algoritem O_k
1 alociraj tabelo f;
2 **for** $r := 1$ **to** k **do** $i_r := 0$;

```
  while true:
    izračunaj f(i_1, ..., i_k) in ga shrani v ustrezni element tabele;
    r := 1;
    while r ≤ k:
      if i_r < n_r then i_r := i_r + 1; break
      else i_r := 0; r := r + 1;
    if r > k then break;
```

V vsaki iteraciji glavne zanke torej izračunamo vrednost f za trenutni nabor indeksov (pri tem bodo prišle prav že shranjene rešitve prejšnjih podproblemov, ki jih imamo v tabeli), nato pa se premaknemo na naslednji nabor. Ta premik je zelo podoben kot pri povečevanju števila za 1: najprej povečujemo i_1, ko pa ta doseže n_1, ga postavimo nazaj na 0 in povečamo i_2; ko sčasoma i_2 doseže n_2, postavimo tudi njega na 0 in povečamo i_3 in tako naprej.

Ko imamo enkrat potabelirane vse vrednosti funkcije f, lahko najkrajši podniz sestavimo podobno kot pri rešitvi za dva niza:

algoritem O_k (*nadaljevanje*)

```
  for r := 1 to k do i_r := n_r;
  q := f[i_1, ..., i_k]; alocirajmo izhodni niz u dolžine q;
  while q > 0:
    ponovi izračun f(i_1, ..., i_k), vendar si tudi zapomni,
      pri katerem c je vrednost g dosegla svoj minimum;
    u_q := c; q := q − 1;
    for r := 1 to k do if i_r > 0 and t^r[i_r] = c then i_r := i_r − 1;
  return u;
```

Ta postopek torej porabi $O(\prod_{r=1}^{k}(n_r+1))$ prostora; ta produkt nastopa tudi v porabi časa, kjer pa mu moramo dodati še vsaj en faktor $O(k)$, ker se v vsakem izračunu funkcije f skrivajo zanke po vseh vhodnih nizih.

Doslej opisani postopek je primeren za nekatere naše testne primere z majhnim številom kratkih nizov; trije testni primeri pa so zanj že neugodno veliki. Pri enem imamo na primer šest nizov dolžine 17 in tri nize dolžine 18, torej potrebujemo tabelo z $18^6 \cdot 19^3$ elementi, kar je približno 233 milijard. Ker so nizi tako kratki, je tudi njihov nadniz kratek in za posamezni element tabele zadošča že 1 byte; vseeno pa je dobrih 217 GB velika tabela za glavni pomnilnik običajnega dandanašnjega osebnega računalnika že prevelika. Razmislimo torej o tem, kako porabo pomnilnika še zmanjšati.

Podobno kot smo si pri rešitvi za $k = 2$ predstavljali dvodimenzionalno tabelo f kot sestavljeno iz $n + 1$ vrstic (= enodimenzionalnih tabel), si lahko zdaj našo k-dimenzionalno tabelo f predstavljamo kot sestavljeno iz $n_1 + 1$ manjših, $(k-1)$-dimenzionalnih podtabel: $f[0], f[1], \ldots, f[n_1]$. Podobno kot prej tudi zdaj vidimo, da ko računamo vrednosti v $f[i_1]$, potrebujemo pri tem nekaj že prej izračunanih vrednosti iz $f[i_1]$ in pa vrednosti iz $f[i_1 - 1]$, ne potrebujemo pa več vrednosti iz $f[i_1 - 2]$, $f[i_2 - 3]$ in tako naprej.

Na primer, če pri testnem primeru, o katerem smo govorili malo prej (šest nizov dolžine 17 in trije nizi dolžine 18) za prvi niz vzamemo enega od daljših ($n_1 = 18$), nam tabela f (dolga približno 217 GB) razpade na 19 podtabel, od katerih je vsaka dolga dobrih 11 GB. Recimo, da si lahko v pomnilniku privoščimo imeti dve taki podtabeli naenkrat, več kot toliko pa že težko.

Zdaj bi lahko te podtabele v mislih združevali v bloke, podobno kot smo naredili pri algoritmu O_2-S, vendar bi se bilo pri tem težko izogniti potrebi po tem, da hranimo v pomnilniku več kot dve podtabeli hkrati. Raje si pomagajmo z naslednjo, še preprostejšo rešitvijo: ko podtabele ne potrebujemo več, jo preprosto odložimo na disk.

algoritem O_k-S
  for $i_1 := 0$ **to** n_1:
    if $i_1 \geq 2$:
      shrani podtabelo $f[i_1 - 2]$ na disk in jo zavrzi iz glavnega pomnilnika;
    izračunaj vse vrednosti f v podtabeli $f[i_1]$;
  nadaljuj enako kot v vrsticah 10–16 algoritma O_k
    z naslednjo razliko: ko se i_1 zmanjša z x na $x - 1$,
    zavrzi podtabelo $f[x]$ iz glavnega pomnilnika
    in (če je $x \geq 2$) naloži $f[x - 2]$ z diska v glavni pomnilnik;

Časovna zahtevnost tega algoritma je še vedno taka kot prej, le za nek konstantni faktor večja (ker moramo večino tabele f enkrat zapisati na disk in enkrat prebrati z njega). Tudi prostorska zahtevnost je enaka kot prej, vendar se zdaj nanaša na porabo prostora na disku; v glavnem pomnilniku pa hranimo le $2\prod_{r=2}^{k}(n_r + 1)$ elementov tabele f naenkrat. Pri največjem izmed naših majhnih testnih primerov pri letošnji off-line nalogi bomo tako porabili približno 22 GB glavnega pomnilnika in 217 GB prostora na disku, kar je za današnje računalnike še obvladljivo.

Požrešni algoritem po znakih. Če je vhodnih nizov malo več in so malo daljši (recimo vsaj nekaj deset nizov, dolgih po vsaj nekaj deset ali sto znakov), si doslej omenjenih optimalnih rešitev ne moremo privoščiti, saj nimamo niti dovolj časa niti dovolj pomnilnika, zato moramo poseči po raznih hevristikah, s katerimi lahko poskusimo najti čim boljšo rešitev, nimamo pa zagotovil o tem, kako blizu najboljše možne bomo na ta način prišli.

Oglejmo si algoritem, ki se je pri naših poskusih dobro obnesel pri srednje velikih testnih primerih — takšnih, kjer obstaja nek skupni nadniz, dolg največ kakšnih 1000 znakov. Nadniz bomo gradili postopoma, znak za znakom; začeli bomo s praznim nizom in vanj v vsakem koraku vrinili po eno črko. Vprašanje je, katero črko vriniti in kam. Kako obetaven je niz v, ki ga dobimo po nekem takem vrivanju? Ker smo začeli s praznim nizom in vanj postopoma dodajamo črke, niz v v resnici najbrž sploh še ni nadniz vseh vhodnih nizov. Toda kako blizu je temu, da bi vendarle bil nadniz vseh vhodnih nizov? To lahko ocenimo takole: predstavljajmo si najdaljši skupni podniz nizov t^r in v; označimo ga z $\textsc{nsp}(t^r, v)$. Razlika $|v| - |\textsc{nsp}(t^r, v)|$ nam pove, kolikšno je najmanjše število dodatnih znakov, ki bi jih bilo treba vriniti v v, da bi postal nadniz niza t^r; če je razlika 0, to pomeni, da je v že zdaj nadniz niza t^r. Torej je v tem bolj obetaven, čim daljši je $\textsc{nsp}(t^r, v)$. Dolžino slednjega seštejmo po vseh $r = 1, \ldots, k$, pa dobimo oceno obetavnosti v-ja:

$$J(v) := \sum_{r=1}^{k} |\textsc{nsp}(t^r, v)|.$$

Največja možna vrednost te ocene nastopi takrat, ko je v že nadniz vseh $t^1, \ldots, t^k$; takrat imamo oceno $J^\star = \sum_{r=1}^{k} |t^r|$. Osnovna ideja našega algoritma je torej takšna:

algoritem G:
$u :=$ prazen niz; $q := 0$;
while $J(u) < J^\star$:
  for $i := 1$ **to** $|u| + 1$:
    za vsako črko c, ki se pojavlja v kakšnem vhodnem nizu:
      $v :=$ niz, ki ga dobimo, če v u vrinemo c na indeks i;
      izračunaj oceno $J(v)$;
  $u :=$ tisti med vsemi tako pregledanimi v, ki je imel največjo $J(v)$;

Postopek je torej požrešen v tem smislu, da vsakič vrine tak znak, ki najbolj poveča oceno $J(u)$. (Ni pa na primer toliko požrešen, da bi poskušal graditi niz u od leve proti desni in torej vedno dodajati črke le na konec u-ja. Pri naših poskusih je tak algoritem dajal precej slabše rezultate, je pa res, da deluje veliko hitreje.)

V vsaki iteraciji zunanje zanke tega postopka moramo oceniti kar precej nizov v; razmislimo o tem, kako lahko to naredimo dovolj učinkovito, da bo ta postopek uporaben tudi v praksi. Ko ocenjujemo nek v (ki smo ga dobili tako, da smo v u vrinili c na indeksu i), nas zanimajo najdaljši skupni podnizi med njim in vsemi t^r. Tak podniz lahko v nizu v zajame tudi pravkar vrinjeni c (ki je v nizu v na indeksu i) ali pa ga ne. Če ga ne, je ta podniz to hkrati tudi podniz u-ja in je neodvisen od c in i, zato tega ne bo treba računati pri vsakem v posebej. Druga možnost pa je, da najdaljši skupni podniz nizov v in t^r zajame tudi tudi c na mestu $v[i]$; tedaj se mora ta c pojaviti tudi v pojavitvi tega podniza v t^r, recimo na nekem indeksu j (torej imamo $t^r[j] = c$). Vse tri nize — v, t^r in njun najdaljši skupni podniz — lahko zdaj razdelimo vsakega na tri dele: levi del (vse pred c-jem), c sam in desni del (vse za c-jem). Tako imamo:

$$\begin{array}{rrcl} v = & v[1..i-1] & c & v[i+1..q+1] \\ = & u[1..i-1] & c & u[i..q]\,, \\ t^r = & t^r[1..j-1] & c & t^r[j+1..n_r] \text{ in} \\ \textsc{nsp}(v, t^r) = & x & c & y \end{array}$$

za neka niza x in y. Naš $\textsc{nsp}(v, t^r)$ ne bi mogel biti res najdaljši skupni podniz nizov v in t^r, če ne bi bil hkrati tudi njegov levi (oz. desni) del najdaljši skupni podniz levih (oz. desnih) delov nizov v in t^r. Torej mora biti $x = \textsc{nsp}(u[1..i-1], t^r[1..j-1])$ in $y = \textsc{nsp}(u[i..q], t^r[j+1..n_r])$. Ker se v splošnem lahko c pojavi na več mestih j v nizu t^r, moramo preizkusiti vse te j in med njimi uporabiti tistega, ki pripelje do najdaljšega podniza. Tako smo dobili:

$$\begin{aligned} |\textsc{nsp}(v, t^r)| = \max\{\, & |\textsc{nsp}(u, t^r)|, \\ & \max\{\, |\textsc{nsp}(u[1..i-1], t^r[1..j-1])| + 1 + \\ & \qquad |\textsc{nsp}(u[i..q], t^r[j+1..n_r])| : 1 \le j \le n_r,\ t^r[j] = c \,\}\,\}. \end{aligned}$$

Če torej izračunamo dolžino najdaljših skupnih podnizov med vsemi začetki (prefiksi) nizov u in t^r in podobno še med vsemi končnicami (sufiksi) teh nizov, bomo lahko s pomočjo teh dolžin precej hitro izračunali $|\textsc{nsp}(v, t^r)|$ za poljuben v (torej za poljubna i in c). Tega pa ni težko računati z dinamičnim programiranjem, podobno kot smo že prej videli za najkrajši skupni nadniz. Pišimo $f(i,j) = |\textsc{nsp}(u[1..i], t^r[1..j])|$ in $g(i,j) = |\textsc{nsp}(u[i..q], t^r[j..n_r])|$. Funkcijo f lahko računamo takole:

```
for j := 0 to n_r do f[0, j] := 0;
for i := 1 to q:
  f[i, 0] := 0;
  for j := 1 to n_r:
    if u_i = t^r[j] then c := f[i − 1, j − 1]
    else c := min{f[i − 1, j], f[i, j − 1]};
    f[i, j] := 1 + c;
```

Zelo podobno pa tudi g:

```
for j := n_r + 1 downto 1 do g[q + 1, j] := 0;
for i := q to 1:
  g[i, n_r + 1] := 0;
  for j := n_r downto 1:
    if u_i = t^r[j] then c := g[i − 1, j + 1]
    else c := min{g[i − 1, j], f[i, j + 1]};
    g[i, j] := 1 + c;
```

Med drugim se v teh dveh tabelah skriva tudi dolžina $|\textsc{nsp}(u, t^r)|$, in sicer v $f[q, n_r]$ in v $g[1, 1]$.

S pomočjo tabel f in g lahko zdaj dovolj poceni izračunamo $|\textsc{nsp}(v, t^r)|$ za poljuben v. Zdaj imamo vse, kar potrebujemo, da lahko učinkovito ocenimo vse možne u'. Njihove ocene bomo počasi računali v tabeli, v kateri $J[i, c]$ predstavlja oceno $J(v)$ tistega niza v, ki ga dobimo z vrivanjem znaka c na indeks i v nizu u.

```
(* Ta postopek naredi to, kar je bilo nakazano v vrsticah 3–6 algoritma G. *)
for i := 1 to q:
  za vsak znak abecede c: J[i, c] := 0;
for r := 1 to k:
  pripravi tabeli f in g po zgoraj opisanem postopku;
  for i := 1 to q:
    za vsak znak abecede c ∈ Σ:
      (* V d izračunajmo |NSP(v, t^r)|. Ena možnost za to je
         |NSP(u, t^r)|, ki jo imamo na primer v g[1, 1]. *)
      d := g[1, 1];
      (* Druga možnost je, da podniz pokrije c na mestu v[i],
         torej mora biti c prisoten tudi v t^r na nekem indeksu j. *)
      for j := 1 to n_r do if t^r[j] = c then
        d := max{d, f[i − 1, j − 1] + 1 + g[i, j + 1] };
    J[i, c] := J[i, c] + d;
```

Notranjo zanko po j lahko še malo izboljšamo, če si vnaprej pripravimo za vsako možno črko c in vsak vhodni niz t^r seznam indeksov, kjer se ta črka pojavlja v t^r. Tako bo morala iti naša notranja zanka le po tistih j, ki nas zanimajo. Pri vsakem konkretnem i se tako, pri vseh c-jih skupaj, notranja zanka po j izvede največ n_r-krat, ker se z vsakim j srečamo pri največ enem c (namreč pri $c = t^r[j]$).

Kakšna je časovna zahtevnost tega postopka? Pri vsakem r (v zanki v vrsticah 3–10) porabimo najprej $O(q \cdot n_r)$ časa za izračun tabel f in r (vrstica 4), nato pa imamo še q iteracij zanke po i (vrstice 5–10) in v vsaki od njih a iteracij zanke po c (npr. tolikokrat se izvede vrstica 7; pri tem je $a = |\Sigma|$ velikost naše abecede) ter vsega skupaj (po vseh c pri tem i) največ n_r iteracij zanke po j (vrstica 9); to je za celotno zanko po i skupaj $O(q(a + n_r))$ časa. Za vrstice 4–10 imamo tako skupaj $O(q(a + n_r))$ časa in za celoten postopek računanja vseh ocen $J(v)$ dobimo skupaj $O(q(ka + \sum_{r=1}^{k} n_r))$. Če označimo povprečno dolžino vhodnih nizov z $\bar{n}$, lahko ta izraz poenostavimo v $O(qk(a + \bar{n}))$.

Zdaj lahko razmislimo še o časovni zahtevnosti algoritma G kot celote. Pravkar opisani postopek pokrije vrstice 3–6 tistega algoritma; nato pa porabimo (v vrstici 7) še $O(qa)$ časa za izbor najbolj obetavnega q in $O(q)$ časa, da se premaknemo vanj. Če je nadniz, ki ga na koncu dobimo, dolg d znakov, to pomeni, da glavna zanka algoritma (vrstica 2) naredi d iteracij, pri katerih se q počasi povečuje od 0 do d, torej je časovna zahtevnost celotnega algoritma G reda $O(d^2 k(a + \bar{n}))$. Zaradi te kvadratne zahtevnosti v odvisnosti od d je ta algoritem primeren le za tiste testne primere, pri katerih obstaja nek dovolj kratek skupni nadniz vseh vhodnih nizov (recimo do $d \approx 1000$).

Omenimo še naslednje: v vrstici 7 algoritma G se pogosto zgodi, da niz v z najvišjo $J(v)$ ni en sam, ampak je takih nizov več (vsi pa imajo enako oceno) in se bomo morali naključno odločiti za enega od njih. Ker pa niso nujno vsi ti nizi enako primerni za nadaljevanje postopka, se zato lahko zgodi, da bo na koncu dobljeni nadniz malo krajši ali pa malo daljši, odvisno od naših naključnih odločitev. Zato je v takem primeru koristno naš algoritem pognati po večkrat in si zapomniti najkrajšega od tako dobljenih nadnizov.

Sestavljanje nadniza od leve proti desni. Naš pravkar opisani požrešni algoritem je dodajal znake v nadniz enega po enega, vendar jih je bil pripravljen vrivati kjerkoli v nadnizu. Videli smo, da nas je to pripeljalo do neugodno velike časovne zahtevnosti, ker smo morali vsakič oceniti vse možne položaje, kamor bi se dalo vriniti naslednji znak v naš nastajajoči nadniz. Hitrejši (in preprostejši), vendar malo slabši algoritem dobimo, če se odločimo, da bomo znake vedno dodajali na konec nadniza. Tako nam vprašanje o tem, *kam* vriniti naslednji znak, odpade in razmišljati moramo le še o tem, *kateri* znak bi dodali.

algoritem W:
$u :=$ prazen niz; **for** $r := 1$ **to** k **do** $i_r := 1$;
ponavljaj, dokler pri kakšnem r še velja $i_r \le n_r$:
  izberi naslednjo črko $c \in \Sigma$ in jo dodaj na konec niza u;
  for $r := 1$ **to** k **do if** $i_r \le n_r$ **and** $t^r[i_r] = c$
  then $i_r := i_r + 1$;

Postopek se torej z indeksi $i_1, \ldots, i_r$ premika naprej po vhodnih nizih; v vsaki iteraciji glavne zanke si izbere naslednji znak c in se premakne za eno mesto naprej po tistih vhodnih nizih, ki so imeli na trenutnem mestu c (torej $t^r[i_r] = c$). Ko pridemo do konca vseh vhodnih nizov ($i_r > n_r$ pri vseh r), se lahko ustavimo; takrat je u nek skupni nadniz vseh vhodnih nizov.[1]

[1] O pravilnosti tega postopka se lahko prepričamo s pomočjo naslednje invariante: na začetku

Obstaja več različic tega postopka, ki se razlikujejo po tem, kako si v vrstici 3 izberejo naslednjo črko:

- Ena možnost je, da si izberemo naslednjo črko tako, da se bomo premaknili naprej po tistem vhodnem nizu, iz katerega smo doslej pobrali najmanj črk. Med vsemi r, ki imajo $i_r \leq n_r$, torej vzamemo tistega z najmanjšim i_r in za ta r potem vzamemo $c := t^r[i_r]$. Če obstaja več enako dobrih r-jev (z enakim i_r), si enega od njih izberemo naključno. Tej hevristiki v literaturi ponavadi pravijo *min-height* [9], vendar se pri naših poskusih na naših testnih primerih ni dobro obnesla.

- Naslednjo črko si lahko izberemo tako, da se bomo premaknili naprej po čim več vhodnih nizih. Vzamemo torej tisto c, za katero velja pogoj $t^r[i_r] = c$ pri največ r-jih (izmed tistih $r \in \{1, \ldots, k\}$, pri katerih je $i_r \leq n_r$, torej da še nismo na koncu tistega vhodnega niza). Če je po tem kriteriju več črk c enako dobrih, si eno od njih izberemo naključno. Tej hevristiki ponavadi pravijo *sum-height* ali *majority merge* (MM) [7].[2]

- Prejšnjo hevristiko (MM) lahko posplošimo tako, da vsakemu znaku vsakega vhodnega niza pripišemo neko utež: naj bo $w(r, i)$ utež i-tega znaka niza t^r. Ko se potem odločamo, kateri c bi dodali v naš izhodni niz, izračunajmo $\omega(c) := \sum_r w(r, i_r)$, pri čemer gre vsota po tistih r, pri katerih je $i_r \leq n_r$ in $t^r[i_r] = c$. V izhodni niz dodamo tisti c, ki ima največjo $\omega(c)$. Če postavimo vse $w(r, i)$ na 1, smo na istem kot pri MM. Vprašanje je seveda, kako si izbrati te uteži; nekateri avtorji so to počeli z znanimi (in pogosto precej zamudnimi) postopki za naključno preiskovanje prostora, na primer genetskimi algoritmi [4] ali kolonijami mravelj [15].

- Eleganten in poceni razmislek, s katerim lahko pridemo do dobrega nabora uteži, pa je naslednji [5]: lahko se zgodi, da nam je od nekaterih vhodnih nizov ostalo še veliko znakov, od drugih pa malo (ali pa smo celo prišli pri njih že do konca niza); in tedaj je koristno, če se osredodotočamo predvsem na črke iz tistih nizov, pri katerih nam je ostalo še veliko znakov (kajti če je od nekega niza ostalo le še malo znakov, si lahko mislimo, da jih bomo najbrž brez težav prej ali slej pobrali spotoma, ne da bi se posebej osredotočali nanje). Tako lahko za $w(r, i)$ vzamemo kar dolžino tistega dela niza t^r, ki ga še nismo pokrili z doslej sestavljenim delom nadniza: torej $w(r, i) := n_r - i + 1$. Tej hevristiki pogosto pravijo *weighted majority merge* (WMM).

- Pri naših poskusih so je pogosto še bolje obnesle uteži oblike $w(r, i) := (n_r - i + 1)^\alpha$ za neko konstanto $\alpha > 1$. To, katera α je dajala najboljše rezultate, je pri različnih testnih primerih različno. Učinek takšnega potenciranja je, da algoritem še bolj prisilimo k osredotočanju na tiste nize, od katerih nam

---

vsake iteracije glavne zanke velja, je u nek skupni nadniz nizov $t^r[1..i_r - 1]$ za vse $r = 1, \ldots, k$.

[2] Ime *majority merge* (večinsko zlivanje) je sicer rahlo zavajajoče, saj ni nujno, da se c pojavlja na trenutnem položaju pri *večini* vhodnih nizov, pač pa le, da se nobena druga črka ne pojavlja na trenutnem položaju pri več nizih kot c.

je ostalo še veliko znakov. To različico bomo v nadaljevanju označevali s PMM(α).[3]

- Če se v PMM splača vzeti $\alpha > 1$ in s tem v vsoti $\sum_r (n_r - i + 1)^\alpha$ še bolj poudariti tiste r, pri katerih je dolžina ostanka $n_r - i + 1$ velika, nam lahko pride na misel, da bi šli s tem razmislekom do konca in gledali le tisti r, pri katerem je vsota $n_r - i$ največja; za tisti r bi potem vzeli $c = t^r[i_r]$. Če je takih r-jev več (enako dobrih), si izberimo tisti c, ki se pojavlja pri največ teh r-jih. Vendar pa se ta hevristika pri naših poskusih ni dobro obnesla (pa tudi pri poskusih s PMM se je pokazalo, da prevelika vrednost α začne škodovati).

Ko enkrat poznamo uteži $w(r, i)$ (oz. znamo poljubno utež izračunati v $O(1)$ časa), opisanega požrešnega algoritma ni težko implementirati tako, da ima časovno zahtevnost $O(d \cdot a + \sum_r n_r)$, če je d dolžina nadniza, ki ga vrne na koncu.

Videli smo, da se včasih lahko zgodi, da se mora algoritem odločiti med več možnimi c, ki so z vidika njegove hevristike videti enako dobri, zato si enega od njih izbere naključno. To pomeni, da če algoritem poženemo po večkrat, lahko nastanejo različni (in tudi različno dolgi) nadnizi, zato ga je koristno (če imamo čas) pognati večkrat in si zapomniti najkrajši tako dobljeni nadniz.[4]

Doslej opisani postopek je nekoliko kratkoviden: ko razmišlja o tem, kateri znak $c \in \Sigma$ bi zdaj dodal na konec izhodnega niza, gleda le na to, katere vhodne znake bi z njim pokrili (torej sešteje uteži $w(r, i_r)$ za trenutne indekse i_r pri tistih vhodnih nizih, ki imajo $t^r[i_r] = c$). Nič pa ne razmišlja o tem, kaj se bo zgodilo v nadaljevanju postopka; mogoče bi bilo bolje zdajle vzeti kakšen drug znak, ki je sicer na prvi pogled manj obetaven, vendar nam bo kasneje omogočil priti do boljše rešitve. Postopek lahko zato izboljšamo tako, da naj gleda več znakov naprej (*lookahead*), recimo ℓ znakov. Namesto vsakega znaka c moramo zdaj pregledati vse možne „podaljške“ — vse nize ℓ znakov iz abecede Σ, s katerimi bi se dalo zdaj podaljšati naš dosedanji izhodni niz u. Enako kot pri prvotnem postopku tudi tu vsak tak podaljšek ocenimo z vsoto uteži tistih znakov, ki jih ta podaljšek pokrije v vhodnih nizih. Na koncu potem podaljšamo izhodni niz s prvim znakom tistega podaljška, ki je imel najvišjo oceno:

algoritem W-L(ℓ)
$u :=$ prazen niz; **for** $r := 1$ **to** k **do** $i_r := 1$;
ponavljaj, dokler pri kakšnem r še velja $i_r \le n_r$:
  $\omega^\star := 0$; (* *ocena najboljšega podaljška doslej* *)
  za vsak niz v dolžine ℓ (nad abecedo Σ):
    for $r := 1$ **to** k **do** $j_r := i_r$;
    $\omega := 0$; (* *ω bo ocena trenutnega podaljška, v* *)
    for $\lambda := 1$ **to** ℓ:
      for $r := 1$ **to** k **do if** $j_r \le n_r$ **and** $t^r[j_r] = v[r]$

[3] Ideja potenciranja uteži se pojavi v [15], kjer so uporabili $\alpha = 9$. Kasneje še mnogi avtorji omenjajo WMM, potenciranja pa ne, kar je škoda, ker je videti, da se res dobro obnese.

[4] Lahko pa gremo z naključnostjo še malo dlje: v vsaki iteraciji se najprej z verjetnostjo q odločimo, ali bi uporabili tisti c, ki maksimizira $\omega(c)$ (tako, kot smo to počeli doslej), ali pa bi (z verjetnostjo $1 - q$) si c izbrali naključno (z verjetnostjo, ki je sorazmerna $\omega(c)$). Avtorja, ki opisujeta ta pristop [15], sta sicer uporabila $q = 0{,}9$, torej večinoma vendarle uporabita tisti c, ki maksimizira $\omega(c)$.

then $\omega := \omega + w(r, j_r)$; $j_r := j_r + 1$;
if $\omega \geq \omega^\star$ **then** $\omega^\star := \omega$; $c := v[1]$;
for $r := 1$ **to** k **do if** $i_r \leq n_r$ **and** $t^r[i_r] = c$
then $i_r := i_r + 1$;
dodaj c na konec niza u;

Ta postopek je seveda precej počasnejši od prvotnega, saj mora v vsaki iteraciji glavne zanke pregledati kar a^ℓ različnih podaljškov. Večji ko je a (velikost abecede Σ), hitreje narašča ta časovna zahtevnost v odvisnosti od ℓ in krajše podaljške si bomo lahko privoščili. Pri naših testnih primerih smo imeli ponekod $a = 5$ (tu gremo lahko brez težav do $\ell = 7$ ali 8), ponekod pa $a = 22$ in 26 (tu gremo lahko vsaj do $\ell = 2$, z nekaj potrpežljivosti tudi do $\ell = 3$). Že $\ell = 2$ je pri naših poskusih dajal precej boljše rezultate kot $\ell = 1$ (torej prvotni postopek brez gledanja naprej), pri večjih ℓ pa so se sicer rezultati še izboljševali, vendar vse manj.

Tudi pri tem postopku se lahko zgodi, da ima več možnih nadaljevanj enako oceno in se moramo naključno odločiti za enega od njih; zato lahko tudi ta postopek poženemo po večkrat, dobimo različno dolge nadnize in na koncu obdržimo najkrajšega od njih. Vendar pa se je pri naših poskusih izkazalo, da je, če imamo dovolj časa, bolje le-tega porabiti tako, da postopek poženemo enkrat z malo večjim ℓ kot pa večkrat z manjšim ℓ.

Najkrajši skupni nadniz, ki se čim bolj prekriva z ostalimi vhodnimi nizi. Pri postopku O_2 smo že videli, da v splošnem lahko obstaja več različnih, vendar enako dolgih najkrajših skupnih nadnizov za dana dva niza s in t. V nadaljevanju bomo O_2 uporabljali kot osnovni gradnik postopkov za iskanje najkrajšega skupnega podniza več kot dveh nizov. Ko se mora O_2 odločiti, katerega od več enako dolgih najkrajših skupnih podnizov s in t naj vrne, se je dobro zavedati naslednjega: čeprav so vsi ti nadnizi enako dolgi, pa niso nujno vsi enako primerni za nadaljevanje postopka, npr. ko bomo poskušali poiskati nadniz tega nadniza in kakšnega tretjega vhodnega niza.

Zato nam lahko pride na misel, da bi pri sestavljanju najkrajšega skupnega nadniza dveh nizov s in t uporabili še nek dodaten kriterij, ki bi nam pomagal izbrati med najkrajšimi nadnizi takega, ki bo imel v nekem smislu čim več skupnega z ostalimi vhodnimi nizi, tako da se bo kasneje čim manj podaljšal, ko bomo iskali skupne nadnize med njim in temi ostalimi vhodnimi nizi. Označimo ostale vhodne nize z $u^1, \ldots, u^\kappa$. Zdaj bi lahko na primer rekli, da bi radi med najkrajšimi skupnimi nadnizi nizov s in t poiskali tak nadniz z, ki maksimizira vrednost $J(z) := \sum_{r=1}^{\kappa} |\textsc{nsp}(z, u^r)|$ (torej vsoto dolžin najdaljših skupnih podnizov med z in vsemi dodatnimi nizi u^r). Vendar pa se računanje te hevristike izkaže za precej zamudno in celoten postopek iskanja skupnega nadniza vseh vhodnih nizov bi bil s to hevristiko počasnejši, kot bi si bilo želeti. Zato uporabimo raje naslednjo hevristiko, ki se jo bo dalo računati hitreje, pripelje pa do bolj ali manj podobno dobrih rezultatov: $\hat{J}(z) := \sum_{r=1}^{\kappa} |\textsc{npp}(z, u^r)|$, pri čemer je $\textsc{npp}(z, u^r)$ najdaljši tak začetek (prefiks) niza u^r, ki se pojavlja kot podniz (lahko tudi nestrnjen) v z. Med najkrajšimi skupnimi nadnizi nizov s in t torej iščemo takega z najmanjšo $\hat{J}(z)$.

S to zamislijo je nekaj težav. Naš algoritem O_2 se je opiral na dejstvo, da lahko najkrajši skupni nadniz nizov $s[1..i]$ in $t[1..j]$ dobimo tako, da z enim znakom podaljšamo nek najkrajši skupni nadniz nizov $s[1..i-1]$ in $t[1..j]$ ali pa nizov $s[1..i]$

in $t[1..j-1]$ (ali pa, če je $s_i = t_j$, nek najkrajši skupni nadniz nizov $s[1..i-1]$ in $t[1..j-1]$). Zaradi tega dejstva je bilo pri O_2 dovolj že, da smo za vsak par (i, j) izračunali dolžino $f(i, j)$ najkrajšega skupnega nadniza nizov $s[1..i]$ in $t[1..j]$; te dolžine so zadoščale tako za računanje podobnih dolžin pri večjih i in j kot tudi za to, da smo na koncu sestavili nek konkreten primer nadniza.

Ta prikladni razmislek pa ne deluje več, če bi se radi med vsemi najkrajšimi skupnimi nadnizi omejili le na tistega z največjo vrednostjo ocene $\hat{J}$. Na primer: recimo, da imamo niza $s =$ `b` in $t =$ `aa`, dodatna niza pa sta dva ($\kappa = 2$), in sicer $u^1 =$ `ab` in $u^2 =$ `ba`. Pri $i = j = 1$, torej ko gledamo najkrajše skupne nadnize nizov $s[1..1] =$ `b` in $t[1..1] =$ `a`, je dolžina takih nadnizov $f(1, 1) = 2$ in obstajata dva nadniza te dolžine: `ab` in `ba`. Oba imata $\hat{J} = 3$, torej si bo naš algoritem kot rešitev podproblema $i = j = 1$ naključno izbral enega od njiju. Če ima smolo, si bo izbral `ba`. Kmalu zatem se bo začel ukvarjati s podproblemom $i = 1, j = 2$, ko nas zanima najkrajši skupni nadniz nizov $s[1..1] =$ `b` in $t[1..2] =$ `aa`. Naš algoritem bo imel na izbiro, da bodisi rešitev podproblema $i = 0, j = 2$ (to je lahko le niz `aa`) podaljša z znakom s_1 — tako nastane `aab`, ki ima $\hat{J} = 3$ — ali pa da rešitev podproblema $i = 1, j = 1$ (za tega pa smo malo prej rekli, da imamo `ba`) podaljša z znakom t_2 — tako pa nastane `baa`, ki ima tudi $\hat{J} = 3$. V vsakem primeru torej dobimo nek skupni nadniz dolžine 3 z oceno $\hat{J} = 3$. Toda obstaja tudi skupni nadniz s-ja in t-ja z dolžino 3 in oceno $\hat{J} = 4$, namreč `aba`. Do njega bi prišli, če bi si bili pri $i = j = 1$ izbrali rešitev `ab` namesto `ba`, pa si je po nesrečnem naključju pač nismo.

Tako torej vidimo, da če si pri vsakem podproblemu (i, j) zapomnimo le eno rešitev, se bomo včasih prisiljeni naključno odločati med več enako dobrimi kandidati (z enako dolžino in enakim $\hat{J}$) in tedaj se lahko (če imamo smolo) zgodi, da si izberemo med njimi takega, zaradi katerega kasneje pri nekem večjem podproblemu ne bomo mogli priti do tistega najkrajšega skupnega nadniza, ki ima tam največji $\hat{J}$. Temu bi se lahko izognili, če bi si pri vsakem podproblemu zapomnili vse rešitve (vse najkrajše skupne nadnize nizov $s[1..i]$ in $t[1..j]$), toda to ne gre, saj jih je lahko eksponentno mnogo (v odvisnosti od i in j). Sprijazniti se moramo torej s tem, da se lahko sicer trudimo dobiti najkrajši skupni nadniz s čim večjim $\hat{J}$, ne bo pa to nujno tisti z največjim $\hat{J}$. V nadaljevanju bomo tisti konkretni nadniz, ki ga dobimo pri podproblemu (i, j), označili z $z(i, j)$ (njegova dolžina je seveda $f(i, j)$).

Druga težava pa je, da je računanje ocene $\hat{J}(z)$ zamudno početje. Dolžino $\text{NPP}(z, u^r)$ lahko izračunamo tako, da se v zanki sprehodimo po znakih z-ja in gledamo, koliko prvih znakov niza u^r smo že videli. To bo vzelo $O(|z| + |u^r|)$ časa in če to naredimo za vse r, bomo $\hat{J}(z)$ izračunali v $O(\kappa|z| + \sum_{r=1}^{\kappa} |u^r|)$ časa. Na srečo gre tudi bolje. Spomnimo se, da je $z(i, j)$ vedno oblike xc, pri čemer je x eden od nizov $z(i-1, j)$, $z(i, j-1)$ in $z(i-1, j-1)$, znak c pa je bodisi s_i bodisi t_j. V vsakem primeru je naš $z(i, j) = xc$ podaljšek nekega malo krajšega niza x, za katerega smo nekoč prej že izračunali $\hat{J}(x)$.

Ko hočemo nato izračunati $\text{NPP}(xc, u^r)$, lahko razmišljamo takole: če se nek prefiks u^r-ja pojavlja kot podniz v xc, se pojavlja (1) bodisi v celoti znotraj x (2) bodisi zajame tudi znak c na koncu. V primeru (1) je torej to tudi podniz niza x, najdaljši tak prefiks u^r-ja pa je kar $\text{NPP}(x, u^r)$. V primeru (2) pa mora biti torej naš $\text{NPP}(xc, u^r)$ oblike pc, pri čemer je p nek prefiks u^r-ja, ki se pojavlja kot podniz v x. Če je p kaj krajši od najdaljšega takega prefiksa (torej od $\text{NPP}(x, u^r)$), bo pc

kvečjemu tako dolg kot tisti najdaljši prefiks, torej takrat s primerom (2) ne bomo prišli do daljšega prefiksa kot z (1). Edini način, da nas (2) pripelje do še daljšega prefiksa, je ta, da je p kar enak $\text{NPP}(x, u^r)$; da pa bo tedaj pc sploh res prefiks u^r-ja, mora biti seveda naslednji znak u^r-ja (za p) ravno c, torej $u^r[|\text{NPP}(x, u^r)|] = c$. Torej lahko dolžino $\text{NPP}(xc, u^r)$ računamo takole:

$$|\text{NPP}(xc, u^r)| = |\text{NPP}(x, u^r)| + \begin{cases} 1, & \text{če } u^r[|\text{NPP}(x, u^r)|] = c \\ 0 & \text{sicer.} \end{cases}$$

Nato moramo to le še sešteti po vseh r, pa dobimo $\hat{J}(xc)$.

Iz tega razmisleka torej vidimo, da je koristno, če pri vsakem (i, j) hranimo ne le $\hat{J}(z(i, j))$, pač pa tudi dolžine posameznih prefiksov $|\text{NPP}(z(i, j), u^r)|$ za vse $r = 1, \ldots, \kappa$. Za vsak (i, j) imamo zdaj takšno tabelo s κ elementi; da pa ne bomo po nepotrebnem zapravljali pomnilnika, lahko te tabele sproti pozabljamo, ko jih ne potrebujemo več: ko je glavna zanka našega postopka pri nekem konkretnem i, potrebujemo te tabele za trenutni i in še za $i - 1$, ne pa več tistih za $i - 2$, $i - 3$ in tako nazaj.

Še en način, kako lahko prihranimo veliko časa, pa je tale: $\hat{J}(z(i, j))$ nas pravzaprav zanima le pri tistih (i, j), ki se jih bo dalo nekako podaljšati v nek najkrajši skupni nadniz celotnih nizov s in t. Niz $z(i, j)$ je najkrajši skupni nadniz nizov $s[1..i]$ in $[1..j]$; če hočemo iz njega narediti čim krajši skupni nadniz celotnih nizov s in t, ga moramo podaljšati še z najkrajšim skupnim nadnizom nizov $s[i+1..n]$ in $t[j+1..m]$. Označimo dolžino slednjega s $\hat{f}(i+1, j+1)$; vse te dolžine lahko izračunamo in potabeliramo na zelo podoben način, kot smo v prvotnem O_2 izračunali tabelo f. Zdaj torej vidimo, da je najkrajši tak skupni nadniz nizov s in t, ki ga je mogoče dobiti s podaljševanjem niza $z(i, j)$, dolg $f(i, j) + \hat{f}(i+1, j+1)$. Če je ta vsota enaka dolžini najkrajšega skupnega nadniza nizov s in t sploh (to dolžino najdemo v $f(n, m)$, pa tudi v $\hat{f}(1, 1)$), potem je (i, j) dovolj obetaven, da moramo zanj izračunati $\hat{J}(z(i, j))$, sicer pa ga lahko preskočimo.

Zapišimo tako dobljeni algoritem še s psevdokodo. Dolžine $|\text{NPP}(z(i, j), u^r)|$ hranimo v tabeli $P_{ij}[1..\kappa]$; nizov $z(i, j)$ ni treba hraniti eksplicitno, pač pa je dovolj že, če si pri vsakem (i, j) zapomnimo, iz katerega od sosednjih podproblemov smo ta niz dobili: iz $(i - 1, j)$, iz $(i, j - 1)$ ali iz $(i - 1, j - 1)$. V ta namen imamo tabelo ζ, ki za vsak (i, j) hrani eno od vrednosti $\{\leftarrow, \uparrow, \nwarrow\}$. Preden si ogledamo glavni del algoritma, zapišimo še tale pomožni podprogram, ki za dani znak c izračuna iz tabele P z vrednostmi $|\text{NPP}(x, u^r)|$ za nek niz x tabelo P' z vrednostmi $|\text{NPP}(xc, u^r)|$ za niz xc in vrne njihovo vsoto, torej $\hat{J}(xc)$:

funkcija Oceni(P, c, P'):
  $\hat{J} := 0$;
  for $r := 1$ **to** κ:
    $p := P[r]$;
    if $p < |u^r|$ **and** $u^r[p + 1] = c$ **then** $p := p + 1$;
    $P'[r] := p$; $\hat{J} := \hat{J} + p$;
  return $\hat{J}$;

Zdaj imamo vse, kar potrebujemo, da zapišemo glavni del našega algoritma:

```
algoritem O₂-P(s, t, {u¹, ..., u^κ})
  (* Izračunaj f. *)
  for i := 0 to n do for j := 0 to m:
    c := i + j;
    if i > 0 then c := min{c, f[i − 1, j] + 1};
    if j > 0 then c := min{c, f[i, j − 1] + 1};
    if i > 0 and j > 0 and s_i = t_j then c := min{c, f[i − 1, j − 1] + 1};
    f[i, j] := c;
  (* Izračunaj f̂. *)
  for i := n + 1 downto 1 do for j := m + 1 downto 1:
    c := (n + 1 − i) + (m + 1 − j);
    if i < n then c := min{c, f̂[i + 1, j] + 1};
    if j < m then c := min{c, f̂[i, j + 1] + 1};
    if i < n and j < m and s_i = t_j then c := min{c, f̂[i + 1, j + 1] + 1};
    f̂[i, j] := c;
  (* Izračunaj P, Ĵ in ζ. *)
  for i := 0 to n:
    if i > 1 then pozabi tabele P_{i−2,j} za vse j = 0, ..., m;
    for j := 0 to m:
      if f[i, j] + f̂[i + 1, j + 1] > f[n, m] then continue;
      if i = 0 and j = 0:
        for r := 1 to κ do P_ij[r] := 0;
        continue;
      naj bodo P←, P↑, P↖ tri pomožne tabele s po κ elementi;
      if i > 0 and f[i − 1, j] + 1 = f[i, j] then Ĵ← := Oceni(P_{i−1,j}, s_i, P←)
                                              else Ĵ← := −∞;
      if j > 0 and f[i, j − 1] + 1 = f[i, j] then Ĵ↑ := Oceni(P_{i,j−1}, t_j, P↑)
                                              else Ĵ↑ := −∞;
      if i > 0 and j > 0 and s_i = t_j and f[i − 1, j − 1] + 1 = f[i, j] then
                                 Ĵ↖ := Oceni(P_{i−1,j−1}, s_i, P↖) else Ĵ↖ := −∞;
      d := tista izmed {←, ↑, ↖}, ki dá največjo Ĵ_d;
      P_ij := P_d; ζ[i, j] := d;
  (* Sestavi primeren izhodni niz. *)
  i := n; j := m; alociraj izhodni niz u dolžine f[n, m];
  while i > 0 or j > 0:
    q := f[i, j]; d := ζ[i, j];
    if d = ← then u_q := s_i; i := i − 1;
    else if d = ↑ then u_q := t_j; j := j − 1;
    else (* torej je d = ↖ *) u_q := s_i; i := i − 1; j := j − 1;
  return u;
```

V naslednjem razdelku si bomo ogledali, kako lahko ta algoritem s pridom uporabimo pri iskanju čim krajšega skupnega nadniza več vhodnih nizov.

Dodajanje vhodnih nizov po vrsti. Zelo preprost način, da pridemo do skupnega nadniza vseh vhodnih nizov, je ta, da začnemo z enim od njih in nato v vsakem koraku poiščemo najkrajši skupni nadniz med dosedanjim nadnizom in naslednjim vhodnim nizom. Dolžina nadniza, ki nam na koncu nastane, je odvisna od tega, v kakšnem vrstnem redu jemljemo vhodne nize. Ta vrstni red lahko opišemo s permutacijo π nad indeksi $\{1, \ldots, k\}$. Tako dobimo naslednji postopek:[5]

algoritem R:
1 $s :=$ prazen niz;
2 **for** $r := 1$ **to** k:
3 $\quad s :=$ najkrajši skupni nadniz nizov s in $t^{\pi(r)}$;
4 **return** s;

V vrstici 3 lahko najkrajši skupni nadniz dveh nizov poiščemo z algoritmom O_2, še boljše rezultate (krajše nadnize) pa bomo dobili, če uporabimo O_2-P; pokličemo ga kot $\mathrm{O}_2\text{-P}(s, t^{\pi(r)}, \{t^{\pi(r+1)}, \ldots, t^{\pi(k)}\})$.

Dolžina nadniza, ki ga dobimo na koncu tega postopka, je odvisna od vrstnega reda π, v katerem smo dodajali nadnize, pa tudi od tega, kako smo si v posameznem izvajanju vrstice 3 izbrali najkrajši skupni nadniz nizov s in $t^{\pi(r)}$. Kot smo videli že zgoraj, se namreč lahko zgodi, da ta najkrajši skupni nadniz ni enoličen, ampak obstaja več različnih (enako dolgih) najkrajših skupnih nadnizov; čeprav so enako dolgi, pa niso nujno vsi enako primerni za nadaljevanje postopka (v naslednjih iteracijah glavne zanke).

Ker je vnaprej težko reči, kateri vrstni red π in kakšne odločitve glede izbiranja nadniza v vrstici 3 nam bodo dale na koncu postopka najkrajši niz, je še najpreprosteje, če poženemo postopek po večkrat za različne, naključno izbrane permutacije π (z drugimi besedami, pred vsakim poskusom naključno premešamo vrstni red vhodnih nizov), pa tudi nadniz v vrstici 3 vsakič izbiramo naključno; med vsemi tako dobljenimi rešitvami pa si zapomnimo najkrajšo.[6]

Lahko si zapomnimo tudi vrstni red π, pri katerem smo ta najkrajši znani nadniz (ki seveda v splošnem ni nujno tudi najkrajši nadniz sploh) našli. Ta vrstni red lahko potem poskušamo še izboljšati z neke vrste *lokalno optimizacijo*: poskusimo naš vrstni red malo spremeniti, na primer tako, da zamenjamo položaj dveh nizov v njem. Če tako spremenjeni vrstni red pripelje do krajšega nadniza, spremembo obdržimo, sicer pa jo zavržemo in poskusimo kakšno drugo. Postopek ustavimo, ko se naveličamo čakati ali pa če pri nekem vrstnem redu preizkusimo vse možne spremembe, pa nobena od njih ne pripelje do krajšega nadniza. Tako dobimo približno takšen postopek:

algoritem R-L:
1 večkrat poženi algoritem R za različne naključne vrstne rede π
in si zapomni tisti π, pri katerem je nastal najkrajši nadniz s;

[5] V literaturi o problemu najkrajšega skupnega nadniza temu algoritmu pogosto pravijo preprosto Greedy, kar je sicer malo neugodno, ker deluje po požrešnem načelu tudi več drugih algoritmov za ta problem (npr. tisti, ki smo jih že videli zgoraj in ki sestavljajo nadniz znak po znak).

[6] Omeniti pa velja, da četudi bi lahko preizkusili vseh $k!$ možnih vrstnih redov (kar je sicer praktično mogoče le pri majhnih k) in če bi pri vsakem izvajanju vrstice 3 lahko preizkusili vse možne najkrajše skupne nadnize nizov s in $t^{\pi(r)}$, to še ne pomeni, da bi nekoč dobili najkrajši možni skupni nadniz sploh (torej takega, kot ga dobimo z algoritmom O_k). Na primer, če imamo tri vhodne nize $\{abbaaa, aaabba, baaaab\}$, je mogoče pokazati, da algoritem R vedno najde nek nadniz dolžine 10, najboljša možna rešitev pa je dolga le 9 znakov (to je niz $abaabaaba$).

2 $T :=$ prazna množica;
3 **ponavljaj**, dokler ne mine dovolj časa:
4 izberi si naključna indeksa i in j tako, da bo $1 \le i < j \le k$ in $(i, j) \notin T$;
5 $\pi' := \pi$; $\pi'[i] := \pi[j]$; $\pi'[j] := \pi[i]$;
6 $s' :=$ nadniz, ki ga vrne algoritem R pri vrstnem redu π';
7 **if** $|s'| < |s|$ **then** $s := s'$; $\pi := \pi'$; $T :=$ prazna množica;
8 **else** dodaj (i, j) v T;
9 **return** s;

Neobetavne poskuse sprememb si torej zapisujemo v množico T (*tabu list*), da jih kasneje ne bomo preizkusili še enkrat; ko pa najdemo nek premik na bolje, spisek tabujev pobrišemo (vrstica 7). V vrstici 4 se lahko tudi zgodi, da vsebuje T že vse možne pare (i, j), kar pomeni, da smo našli lokalni minimum in moramo postopek ustaviti.

Ta postopek bi se dalo na razne načine še malo izboljšati. Na primer, prva sprememba v našem vrstnem redu nastopi na indeksu i, torej nadniza ne bi bilo treba računati vsakič od začetka, ampak bi lahko nadaljevali pri tistem nadnizu, ki je nastal iz prvih $i-1$ vhodnih nizov. Tako prihranimo nekaj časa, vendar povečamo porabo pomnilnika, saj si moramo poleg končnega nadniza zapomniti še vseh $k-1$ vmesnih. Še ena možna izboljšava je, da bi postopek včasih sprejel tudi spremembe na slabše, vendar s tem manjšo verjetnostjo, čim bolj poslabšajo rešitev. Temu prijemu pravimo *simulirano ohlajanje* (*simulated annealing*) in nam lahko pomaga, da se izognemo kakšnim neobetavnim lokalnim ekstremom.

Razbijanje vhodnih nizov na krajše kose. Zgoraj smo videli, da algoritem G pogosto daje dobre rezultate, vendar je uporabno hiter le, če so vhodni nizi (in njihov skupni nadniz) dovolj kratki. Če ima naš testni primer dolge vhodne nize, jih lahko poskusimo razbiti na več krajših. Izberimo si nek b in razbijmo vsak vhodni niz t^r na b približno enako dolgih kosov $t^{r,1}, \ldots, t^{r,b}$:

$$t^{r,i} = t^r[\lfloor n_r(i-1)/b \rfloor + 1..\lfloor n_r i/b \rfloor].$$

Istoležne kose vseh nizov združimo v nek (čim krajši) skupni nadniz in nato tako dobljene nadnize staknimo skupaj, pa imamo rešitev prvotnega problema:

algoritem C(b):
1 razbij vsak vhodni niz t^r na b kosov $t^{r,i}$, $i = 1, \ldots, b$;
2 **for** $i := 1$ **to** b:
3 $s^i :=$ čim krajši skupni nadniz nizov $t^{1,i}, t^{2,i}, \ldots, t^{k,i}$;
4 $s := s^1 s^2 \ldots s^b$; **return** s;

S tem, ko smo vhodne nize razcepili na b kosov in nato vsakič iskali nadniz le za istoležne kose, smo seveda vpeljali v postopek neko dodatno omejitev, ki je prvotna naloga ni imela, zato nas ne bo presenetilo, da so rešitve zaradi tega lahko malo slabše (nastane malo daljši nadniz). To poslabšanje je tem večje, čim večji je b (na čim več kosov razcepljamo vhodne nize), zato je pametno vzeti najmanjši b, ki si ga še lahko privoščimo. Glavna omejitev tu je, kdaj postanejo kosi vhodnih nizov dovolj kratki, da bomo v vrstici 3 lahko z nekim drugim algoritmom poiskali njihov (čim krajši) skupni nadniz. Na primer, če tam uporabimo algoritem G, si načeloma želimo, da obstaja nadniz s^i dolžine največ kakšnih 1000 znakov, sicer se začne izvajati neugodno dolgo.

Drugi algoritmi. Za konec le na kratko omenimo še nekaj drugih algoritmov, ki so jih v literaturi predlagali za iskanje čim krajših skupnih nadnizov.

Zapišimo po vrsti vse znake naše abecede Σ, vsakega po enkrat; dobimo nek niz dolžine a; staknimo skupaj d izvodov tega niza. Nastal je niz dolžine $d \cdot a$, ki ima za podnize prav vse nize (nad abecedo Σ) dolžine d ali manj. Če torej za d vzamemo $\max_r |t^r|$, imamo primeren skupni nadniz vseh naših vhodnih nizov. Ta algoritem se imenuje Alphabet in je zanimiv predvsem s teoretičnega vidika, ker daje neko zagotovilo o tem, kako dolg je (v najslabšem primeru) lahko najkrajši skupni nadniz: največ a-krat toliko, kolikor je dolg najdaljši od vhodnih nizov.

Zgoraj smo videli algoritem R, ki dodaja vhodne nize v nadniz enega po enega. Še ena podobna ideja je, da najprej združujemo vhodne nize po dva skupaj v nadnize; tako iz prvotnega nabora k nizov dobimo $\lceil k/2 \rceil$ malo daljših nadnizov. Na tem novem naboru nizov isti postopek ponovimo in tako nadaljujemo, dokler ne ostane en sam niz. Temu postopku pravijo „turnir“ (Tournament), vendar je pri naših poskusih na naših testnih primerih dajal veliko slabše rezultate (daljše nadnize) kot algoritem R.

Reduce-expand [2] najprej oklesti vhodne nize tako, da od več zaporednih enakih znakov obdrži le po enega; nato poišče skupni nadniz tako okleščenih nizov; nato pa ta nadniz počasi napihuje nazaj tako, da podvaja znake v njem in sproti briše tiste, ki jih ne potrebuje. Pri naših poskusih se ni preveč obnesel.

Deposition-reduction [8, 14] poskuša nek že znan skupni nadniz s izboljšati tako, da ga na vse možne načine razbije na dva dela, levega s_L in desnega s_D; nato še vsak vhodni niz t^r razbije na levi in desni del glede na to, v katerem delu nadniza leži; nato poskuša za leve dele vhodnih nizov najti nov skupni nadniz in če je ta kaj krajši od s_L, ga uporabi v s namesto dosedanjega s_L. V obliki, kot jo opisujejo avtorji, je videti ta postopek precej počasen, je pa s prijemi tega tipa vsekakor mogoče včasih še malo skrajšati nadnize, ki smo jih dobili s kakšnim drugim algoritmom.

Precej avtorjev uporablja tudi pristope, ki sestavljajo nadniz s kakšno od znanih hevristik za preiskovanje prostora, kot sta na primer iskanje v snopu (*beam search*) [3, 12] in optimizacija po zgledu kemijskih reakcij (*chemical reaction optimization*, cro) [17].

Testni primeri

Pripravili smo 60 testnih primerov, ki pokrivajo širok razpon kombinacij števila vhodnih nizov, dolžine vhodnih nizov, dolžine skupnega nadniza in velikosti abecede. Vsi pa se držijo omejitve iz besedila naloge, namreč da skupna dolžina vhodnih nizov ne presega milijon znakov. Glede na način, kako smo pripravili vhodne nize, lahko ločimo naslednje tipe testnih primerov:

- $\mathrm{P}(d, x)$: s postopkom x smo najprej pripravili niz dolžine d, nato pa smo za vhodne nize uporabljali njegove naključne podnize. Posamezni podniz dobimo tako, da naključno izberemo množico različnih indeksov od 1 do d, jih uredimo naraščajoče, odčitamo s teh indeksov črke našega niza in jih staknemo skupaj. Za x smo uporabili naslednje možnosti:

- $x = \text{S}$: posamezne črke niza so izbrane naključno in neodvisno od ostalih črk iz abecede 5 znakov (vsi znaki so enako verjetni).
- $x = \text{V}$: kot $x = \text{M}$, le da imamo abecedo 22 znakov, ki niso vsi enako verjetni, ampak so verjetnosti porazdeljene približno po potenčni porazdelitvi (podobno kot v besedilih v naravnem jeziku).
- $x = \text{M}$: niz smo zgenerirali z markovskim modelom na črkah; to pomeni, da je verjetnostna porazdelitev naslednje črke odvisna od tega, kakšne so bile prejšnje 3 (že zgenerirane) črke. Te verjetnosti smo ocenili na slovarju približno 350 tisoč angleških besed.
- $x = \text{R}$: niz je „realističen“, dobili smo ga tako, da smo iz korpusa angleških besedil odstranili vse ne-črkovne znake.

• M(d): vsak vhodni niz posebej smo zgenerirali z enakim markovskim modelom, ki je bil že omenjen zgoraj; vhodne nize smo dodajali tako dolgo, dokler njihova skupna dolžina ni dosegla d.

• D(d): kot vhodne nize smo vzeli naključen izbor besed iz slovarja (že omenjeni seznam 350 tisoč angleških besed), pri čemer smo jih jemali tako dolgo, dokler njihova skupna dolžina ni dosegla d.

Podrobnejši opis posameznih testnih primerov je v tabeli z rezultati spodaj.

Rezultati

Tabela na naslednjih straneh podaja nekaj podrobnosti o testnih primerih in o najboljših nam znanih rešitvah za vsakega od njih. Stolpci od leve proti desni pomenijo: (1) # = zaporedna številka testnega primera; (2) k = število vhodnih nizov; (3) vsota dolžin vhodnih nizov; (4) povprečje dolžin vhodnih nizov; (5) postopek, s katerim je bil ta testni primer pripravljen; (6) dolžina najkrajšega nam znanega skupnega nadniza (zvezdica označuje tiste, pri katerih vemo, da je to res optimalna rešitev, torej najkrajši skupni nadniz sploh); (7) stopnja prekrivanja = razmerje med skupno dolžino vhodnih nizov in dolžino najkrajšega nam znanega skupnega nadniza; (8) postopek, s katerim je bil pridobljen najkrajši nam znani skupni nadniz. V zadnjem stolpcu zapis W(ℓ, α) pomeni, da smo uporabili postopek W-L(ℓ) z utežmi PMM z eksponentom α.

#	Št. nizov	Skupna dolž.	Povpr. dolž.	Kako pripravljen	Dolž. nadniza	Stop. prekr.	Katera rešitev
1	100	1 229	12,3	P(100, S)	50	24,6	G
2	300	3 669	12,2	P(100, S)	58	63,3	G
3	1 000	12 312	12,3	P(100, S)	65	189,4	G
4	100	1 501	15,0	P(1000, S)	58	25,9	G
5	300	4 505	15,0	P(1000, S)	65	69,3	G
6	30	224 722	7 491	P(10^5, S)	22 860	9,8	W(6, 6)
7	50	363 339	726	P(10^5, S)	23 904	15,2	W(7, 9)
8	100	774 567	7 746	P(10^5, S)	26 084	29,7	W(6, 11)
9	100	1 223	12,2	P(100, V)	83	14,7	G
10	300	3 648	12,2	P(100, V)	100	36,5	G
11	1 000	12 512	12,5	P(100, V)	101	123,9	G
12	100	1 523	15,2	P(1000, V)	131	11,6	G
13	300	4 490	15,0	P(1000, V)	153	29,3	G
14	1 000	14 876	14,9	P(1000, V)	169	88,0	G
15	100	9 520	95,2	P(1000, V)	816	11,7	G
16	300	31 893	106,3	P(1000, V)	959	33,3	G
17	1 000	15 115	15,1	P(10^4, V)	190	79,6	G
18	10 000	150 475	15,0	P(10^4, V)	227	662,9	G
19	50	51 551	1 031	P(10^4, V)	7 291	7,1	R-L + O_2-P
20	100	100 654	1 007	P(10^4, V)	8 113	12,4	G
21	30	234 545	7 818	P(10^5, V)	46 521	5,0	R-L + O_2-P
22	50	367 352	7 347	P(10^5, V)	51 310	7,2	R-L + O_2-P
23	100	751 859	7 519	P(10^5, V)	62 485	12,0	R-L + O_2-P
24	1 059	9 999	9,4	D(10^4)	150	66,7	G
25	10 603	100 000	9,4	D(10^5)	190	526,3	G
26	105 779	999 995	9,5	D(10^6)	226	4 424	G
27	1 111	9 999	9,0	M(10^4)	159	62,9	G
28	11 055	99 999	9,0	M(10^5)	196	510,2	G
29	109 650	999 994	9,1	M(10^6)	227	4 405	G
30	100	1 184	11,8	P(100, M)	106	11,2	G
31	300	3 867	12,9	P(100, M)	105	36,8	G
32	100	1 470	14,7	P(1000, M)	154	9,5	G
33	300	4 456	14,9	P(1000, M)	196	22,7	G
34	100	1 262	12,6	P(100, R)	102	12,4	G
35	300	3 813	12,7	P(100, R)	103	37,0	G
36	100	1 473	14,7	P(1000, R)	156	9,4	G
37	300	4 462	14,9	P(1000, R)	195	22,9	G
38	1 000	14 865	14,9	P(10^4, R)	239	62,2	G
39	10 000	150 145	15,0	P(10^4, R)	288	521,3	G
40	100	100 746	1 008	P(10^4, R)	10 518	9,6	C(6) + G
41	30	201 340	6 711	P(10^5, R)	48 937	4,1	R-L + O_2-P
42	50	370 670	7 413	P(10^5, R)	63 966	5,8	R-L + O_2-P
43	100	723 977	7 240	P(10^5, R)	76 145	9,5	R-L + O_2-P
44	2	995 531	497 766	P(10^6, V)	701 370	1,4	O_2-S

#	Št. nizov	Skupna dolž.	Povpr. dolž.	Kako pripravljen	Dolž. nadniza	Stop. prekr.	Katera rešitev
45	4	196	49,0	P(100, V)	89⋆	2,2	O_k
46	7	144	20,6	P(80, V)	53⋆	2,7	O_k
47	9	92	10,2	P(60, V)	32⋆	2,9	O_k
48	9	108	12,0	P(70, V)	38⋆	2,8	O_k
49	9	156	17,3	P(150, V)	60⋆	2,6	O_k-S
50	4	199	49,8	P(100, V)	78⋆	2,6	O_k
51	7	139	19,9	P(80, V)	40⋆	3,5	O_k
52	9	87	9,7	P(60, V)	23⋆	3,8	O_k
53	9	108	12,0	P(70, V)	29⋆	3,7	O_k
54	9	154	17,1	P(150, V)	42⋆	3,7	O_k-S
55	4	201	50,3	P(10^4, M)	126⋆	1,6	O_k
56	7	138	19,7	P(10^4, M)	70⋆	2,0	O_k
57	9	86	9,6	P(10^4, M)	39⋆	2,2	O_k
58	9	110	12,2	P(10^4, M)	51⋆	2,2	O_k
59	9	156	17,3	P(10^4, M)	72⋆	2,2	O_k-S
60	2	996 063	498 031	P(10^6, M)	786 274⋆	1,3	O_2-S

Literatura

[1] A. V. Aho, D. S. Hirschberg, J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *J. of the ACM*, 23(1):1–12 (Jan. 1976).

[2] P. Barone, P. Bonizzoni, G. D. Vedova, G. Mauri. An approximation algorithm for the shortest common supersequence problem: an experimental analysis. *Proc. of the 2001 ACM Symposium on Applied Computing*, pp. 56–60.

[3] C. Blum, C. Cotta, A. J. Fernández, F. Gallardo. A probabilistic beam search approach to the shortest common supersequence problem. *Proc. of the 7th European Conf. on Evolutionary Computation in Combinatorial Optimization* (EvoCOP 2007), pp. 36–47.

[4] Jürgen Branke, M. Middendorf. Searching for shortest common supersequences by means of a heuristic-based genetic algorithm. *Proc. of the 2nd Nordic Workshop on Genetic Algorithms and Their Applications*, pp. 105–114 (1996).

[5] Jürgen Branke, M. Middendorf, F. Schneider. Improved heuristics and a genetic algorithm for finding short supersequences. *OR Spektrum*, 20:39–45 (1998).

[6] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343 (June 1975).

[7] T. Jiang, M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal of Computing*, 24(5):1122–1139 (October 1995).

[8] K. Ning, K. P. Choi, H. W. Leong, L. Zhang. A post-processing method for optimizing synthesis strategy for oligonucleotide microarrays. *Nucleic Acids Research*, 33(17): e144 (2005).

[9] S. Kasif, Z. Weng, A. Derti, R. Beigel, C. DeLisi. A computational framework for optimal masking in the synthesis of oligonucleotide microarrays. *Nucleic Acids Research*, 30(20):e103 (October 15, 2002).

[10] A. Lagoutte, S. Tavenas. The complexity of Shortest Common Supersequence for inputs with no identical consecutive letters. `arXiv.org/abs/1309.0422v2`, January 9, 2015.

[11] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336 (April 1978).

[12] S. R. Mousavi, F. Bahri, F. S. Tabataba. An enhanced beam search algorithm for the shortest common supersequence problem. *Engineering Applications of Artificial Intelligence*, 25(3):457–67 (April 2012).

[13] M. Middendorf. More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science*, 125:205–28 (1994).

[14] K. Ning, H. W. Leong. Towards a better solution to the shortest common supersequence problem: the deposition and reduction algorithm. *BMC Bioinformatics*, 7 (Suppl 4): S12 (2006).

[15] M. René, M. Middendorf. An island model based ant system with lookahead for the shortest supersequence problem. *Parallel Problem Solving from Nature, 5th International Conference*, 1998. LNCS 1498, pp. 692–701.

[16] K.-J. Räihä, E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2):187–98 (1981).

[17] C. M. K. Saifullah, M. R. Islam. Solving shortest common supersequence problem using chemical reacton optimization. *Proc. 5th Int. Conf. on Informatics, Electronics and Vision* (ICIEV 2016), pp. 50–55.

[18] V. G. Timkovskii. Complexity of common subsequence and supersequence problems and related problems. Cybernetics 25(5):565–580, September 1989.