KRPALNIK (PATCHER)

Ko uporabniki želimo naložiti nov računalniški program, to pogosto naredimo s prenosom programa s spletne strani ponudnika ali pa morda z USB ključa. Tako prenesemo na naš računalnik celotno, ponavadi najnovejšo različico programa. Vendar ker razvijalci programske opreme ne počivajo, se bo prej ali slej pojavila nova različica programa. Kako pridemo do najnovejše različice? Najbolj preprosta rešitev bi bila, da novi program ponovno prenesemo in presnamemo starejšo različico. Ampak če je število teh programov veliko, nam bo vse to prenašanje vzelo veliko časa, prav tako pa dodatno obremenimo našo internetno povezavo (v kolikor novi program dobivamo po tem viru).

To se nam zdi potratno, še posebej ker se novi program pogosto zelo malo razlikuje od starejšega. Ko popravimo nekaj vrstic v programu, to še zdaleč ne pomeni popolnoma drugačnega prevedenega programa.

Tukaj se pokaže koristnost krpalnikov. Glavna ideja je, da namesto prenašanja celotne datoteke raje generiramo manjšo datoteko imenovano krpa (patch), ki hrani strnjen zapis sprememb med staro in novo različico. Krpalnik pa je program, ki zna iz krpe razbrati in uveljaviti spremembe na starejši različici datoteke. Tako smo sedaj namesto celotne datoteke prenesli zgolj zelo zgoščen zapis razlike med datotekama.

Torej,

- Krpa (patch) je:
  - Kompakten zapis navodil krpalniku, s katerimi le-ta posodobi datoteko na novejšo različico
- Krpalnik (patcher) je :
  - Program, ki z navodili v krpi, posodobi izbrano datoteko

Kako ustvarimo krpo?

Krpo na nivoju tekstovnih datotek v okolju Linux zna ustvariti program diff.

- Diff:
  - Je program za primerjavo dveh datotek. Diff je vrstično orientiran, torej ne primerja med datotekama vseh znakov med seboj, ampak med seboj primerja zgolj vrstice. Vseeno pa poskuša določiti najmanjšo množico izbrisov in vnosov med datotekama, da spremeni prvo datoteko v drugo. Izhod programa se nato lahko uporabi za krpanje.

Primer kako z diff dobimo razliko med datotekama data1.txt in data2.txt:

```
$ diff   data1.txt   data2.txt
```

Primer kako z diff generiramo krpo patchfile.patch

```
$ diff   data1.txt  data2.txt  >  patchfile.patch
```

Sedaj ko imamo željeno krpo, si poglejmo še program patch.

- Patch:
  - Je program, ki vzame krpo s seznamom razlik, ki jih je generiral diff, in jo uporabi na originalni datoteki ter tako ustvari pokrpano verzijo. Patch deluje na tekstovnih datotekah.

Na data1.txt uporabimo patchfile.patch tako:

```
$ patch  data1.txt  patchfile.patch
```

Uporabimo pa lahko krpo tudi v obratni smeri (torej iz data2.txt dobimo data1.txt)

```
$ patch  -R  data2.txt  patchfile.patch
```

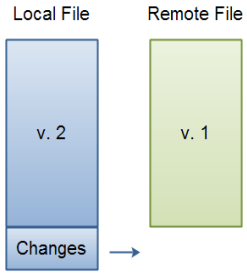
Originalni patch ni podpiral binarnih datotek. Lahko pa uporabimo trik za generiranje krp binarnih datotek kljub tej omejitvi, tako da sprva binarno datoteko pretvorimo v tekstovno, na primer z ukazom hexdump, nato pa na tej datoteki izvedemo diff. Seveda pa so sedaj na voljo že boljša orodja (npr. xxd(za delo z binarnimi datotekami), na Windows sistemih WinMerge).

Pogledali smo si primer programske opreme s katero ustvarimo in uveljavimo krpe. Sedaj pa si bolj podrobno poglejmo splošno logiko povezano s krpalniki.

Najprej se vprašajmo, kako sploh zaznamo spremembe med datotekami?

Odgovor je trivialen, v primeru ko se datoteki nahajata na isti napravi. Kaj pa če imamo veliko datoteko na računalniku, kjer jo spreminjamo, njeno kopijo pa imamo shranjeno na strežniku. Želimo, da je datoteka na strežniku vedno posodobljena, ampak je ne želimo pošiljati vedno ko naredimo kakšno majhno spremembo.

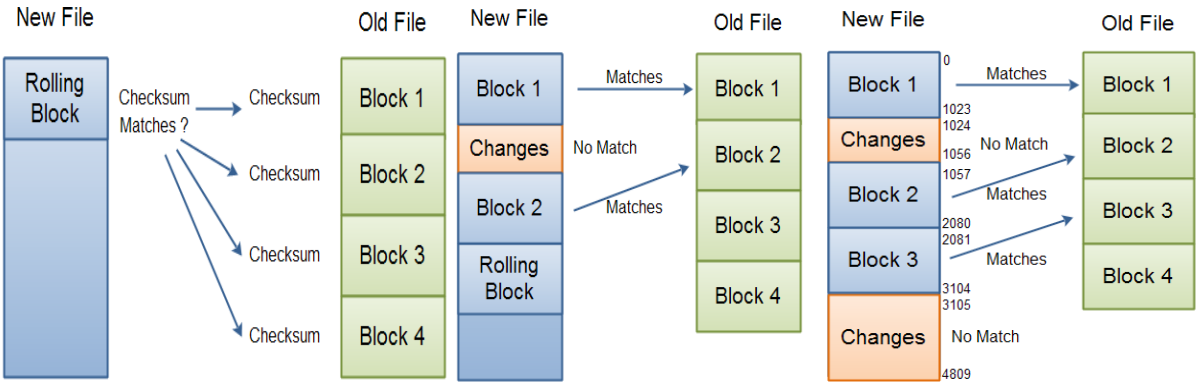
Pogledali si bomo RSync, ki je protokol za usklajevanje datotek na daljavo. RSync v zgoraj opisanem primeru poskrbi da je datoteka na strežniku vedno posodobljena (to mu uspe s pošiljanjem le okoli 2% celotne datoteke).



Local and Remote File Versions

Poglejmo si kako RSync zazna razlike med datotekama brez da bi imel obe na isti napravi. Označimo računalnik z R (ima novo datoteko), strežnik pa s S (ima staro datoteko).

S razdeli datoteko na bloke (recimo 1 ali 2kB) za vsak blok S-ja izračunamo kontrolno vsoto. Te so poslane R. R išče po svoji datoteki bloke, ki bi imeli enako kontrolno vsoto. Najprej generira kontrolno vsoto za prvi blok, če se vsota ne ujema z nobenim blokom S-ja, potem se premakne za 1B po datoteki in tam zračuna novo kontrolno vsoto (torej lahko generira kontrolno vsoto za vse možne bloke svoje datoteke).



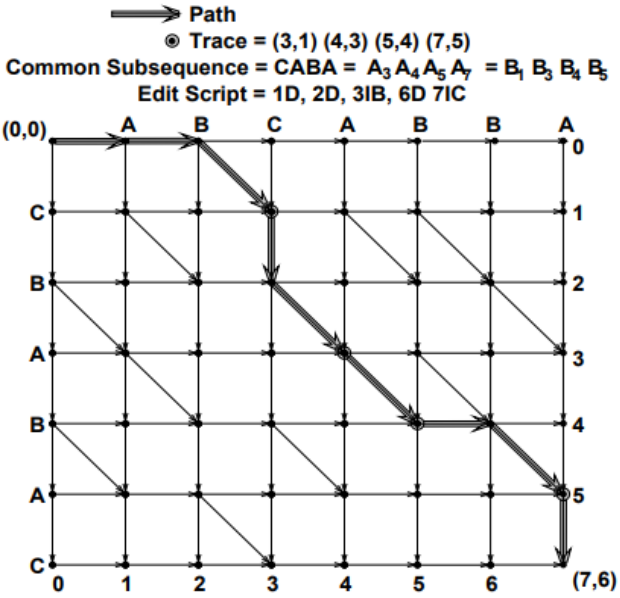
Če R najde blok z isto kontrolno vsoto, kot jo ima nek blok od S, potem to obravnava, kot da ta blok obstaja pri S. Tukaj ni pomembno, da si bloki sledijo v istem vrstnem redu kot pri S. R v tem primeru skoči na konec bloka in tam nadaljuje z iskanjem enakih kontrolnih vsot.

Ko R konča s svojim iskanjem, bo imel nekaj blokov, ki ustrezajo stari datoteki. Podatke med bloki pošlje S, saj se razlikujeta zgolj po njih. Skupaj z njimi pošlje navodila kako sestaviti novo datoteko.

Ampak kontrolna vsota sama po sebi ne zagotavlja, da sta bloka zares enaka, vendar je verjetnost, da imata dva bloka enako kontrolno vsoto majhna. Po drugi strani pa vemo, če bloka nimata enake kontrolne vsote, zagotovo nista enaka. Kontrolne vsote so narejene tako, da se majhna razlika v podatkih odraža v povsem drugačni kontrolni vsoti. V praksi datoteke spreminjamo malo.

RSync uporablja rolling checksum(32 bit) algoritem. V opisu delovanja pa smo zaradi jasnosti izpustili, da se poleg kontrolne vsote S izračuna še hash z MD5(128 bit), ki ga izračuna tudi R, takrat ko dobi ujemajoč checksum, s tem pa z večjo verjetnostjo potrdi enakost blokov.

Sedaj ko smo ugotovili, kako zaznamo spremembe med različicami datotek, nastopi najpomembnejši del, kako te spremembe najti, in zapisati v datoteko z navodili v čim bolj kompaktni obliki.



Recimo da imamo dve zaporedji,

$A = a_1 a_2 \ldots a_n$ in $B = b_1 b_2 \ldots b_m$. Graf urejanja za A in B ima vozlišča na mreži (x,y), $x \in [0,n]$ and $y \in [0,m]$. Vodoravne povezave so $(x-1,y) \rightarrow (x,y)$ za $x \in [1,n]$ in $y \in [0,m]$. Navpične povezave so $(x,y-1) \rightarrow (x,y)$ za $x \in [0,n]$ in $y \in [1,m]$. Če je $a_x = b_y$ potem ima graf tudi diagonalno povezavo med (x-1,y-1) in (x,y). Točkam (x,y), kjer $a_x = b_y$ rečemo točke ujemanja.

(na sliki prva koordinata predstavlja stolpec druga pa vrstico)

Na sliki je narisan graf za zaporedji
A = abcabba and B = cbabac.

Sled dolžine L je zaporedje L točk $(x_1, y_1)\ (x_2, y_2) \ldots (x_L, y_L)$, da $x_i < x_{i+1}$ and $y_i < y_{i+1}$ za zaporedne točke $i \in [1, L-1]$.

Podzaporedje je poljubno zaporedje, ki ga dobimo tako, da izbrišemo nič ali več elementov iz osnovnega zaporedja. Skupno podzaporedje dveh zaporedij A in B je tako, ki je podzaporedje obeh. Vsaka sled definira določeno podzaporedje in obratno.

Navodila urejanja so množica vstavljanj in brisanj elementov, da iz A dobimo B. Ukaz »x_iD« izbriše element x_i iz A in »x I $b_1 \ldots b_n$« vstavi za a_x elemente $b_1 \ldots b_n$.

Vsaka sled določa unikatna navodila urejanja. Naj bo $(x_1, y_1)(x_2, y_2) \ldots (x_L, y_L)$ sled. Naj bo $y_0 = 0$ in $y_{l+1} = m+1$. Sled določa ukaze »xD« za $x \in / \{x_1, x_2, \ldots, x_L\}$, in »$x_k$ I $b_{(yk)+1}, \ldots, b_{(yk+1)-1}$« za k, za katerega velja $y_k + 1 < y_{k+1}$. V navodilih torej izbrišemo N – L simbolov in vstavimo M – L simbolov. Torej za vsako sled dolžine L, obstaja navodilo dolžine D = N+M-2L.

Opazimo, da je dolžina navodil enaka številu premikov po poti navpično in vodoravno. Za vsak vzporeden premik dobimo ukaz tipa D (briši), za premik navpično pa ukaz tipa I (vstavi). Število diagonalnih premikov je dolžina podzaporedja, skupna dolžina povezav na poti pa je N+M-L.

Če damo diagonalnim povezavam ceno 0, ostalim pa ceno 1, lahko problem iskanja najkrajšega navodila prevedemo na problem iskanja najcenejše poti skozi graf.

S tako formulacijo problema in dolgim razmislekom so uspeli pokazati, da obstaja požrešen algoritem z zahtevnostjo O((M+N)D), kjer je D število nediagonalnih povezav v najdeni poti. S probabilistično analizo ima pokazano pričakovano zahtevnost O(M + N + D*2). Pomembno je poudariti, da je pri problemih, ko iščemo razlike v datotekah, pogosto D veliko manjši od M in N. Pogosto se še uporablja hevristika, da se izognemo robnim primerom.

Seveda pa morda nimamo časa implementirati kompliciranih algoritmov, in bi raje prišli do te rešitve kako drugače. Če bi se lotili problema kar z preizkušanjem rešitev (n je dolžina prvega zaporedja, m dolžina drugega), bi nas to pripeljalo do časovne zahtevnosti $O(m * 2^n)$, kjer $O(2^n)$ čas za generiranja vseh podzaporedij prvega, pri vsakem pa tudi preverimo, ali je podzaporedje drugega zaporedja.

Lahko pa se zadeve lotimo z dinamičnim programiranjem. S kratkim razmislekom pridemo do rekurzivne zveze:

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \hat{} x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Razmislek: Če sta X in Y zaporedji, potem je v primeru, da je eno prazno najdaljše skupno podzaporedje prazna množica (prva možnost). Če se njun končen element ujema, potem je zagotovo del najdaljšega skupnega podzaporedja (je končen element rešitve), zato lahko nadaljujemo z iskanjem najdaljšega podzaporedja X in Y brez končnih elementov (2.možnost). Drugače pa najdaljše podzaporedje ne more vsebovati končnih elementov X-a ali Y-a. V tem primeru iščemo najdaljše podzaporedje naprej, ločeno vsakič brez enega izmed teh elementov. Za rešitev vzamemo daljše podzaporedje.

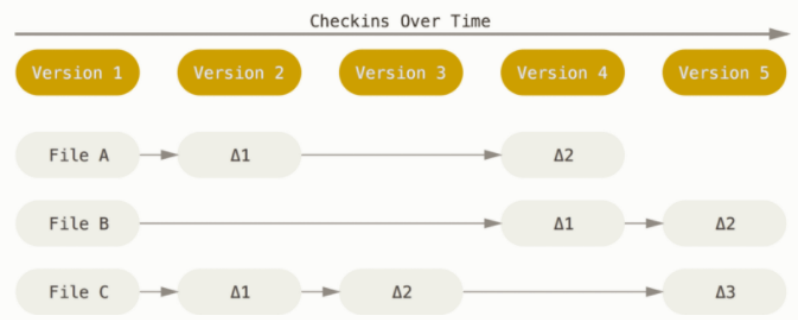
Sam sem se lastne implementacije algoritma lotil z dinamičnim programiranjem od spodaj navzgor, ki najprej s pomočjo rekurzivne zveze izračuna dolžino najdaljšega podzaporedja. To naredimo tako, da se z gnezdenima zankama sprehajamo po matriki velikosti (n+1)*(m+1) (n, m sta dolžini zaporedij A in B, +1 v vsaki dimenziji imamo zaradi praznih zaporedij) in vpisujemo najdaljše podzaporedje glede na že prej izračunane vrednosti v matriko. Na primer na lokacijo [a][b], zapišemo vrednost najdaljšega podzaporedja prvih a elementov A in prvih b elementov B. Nato lahko s pomočjo omenjene rekurzivne zveze razberemo rešitev.

	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	↑←0	↖1	←1	←1	←1
A	0	↖1	↑←1	↑←1	↖2	←2
C	0	↑1	↑←1	↖2	↑←2	↑←2

Primer matrike imamo na sliki za zaporedji A =GAC in B=AGCAT. Puščice kažejo, iz katerih poti smo lahko prišli, da dobimo dano rešitev.

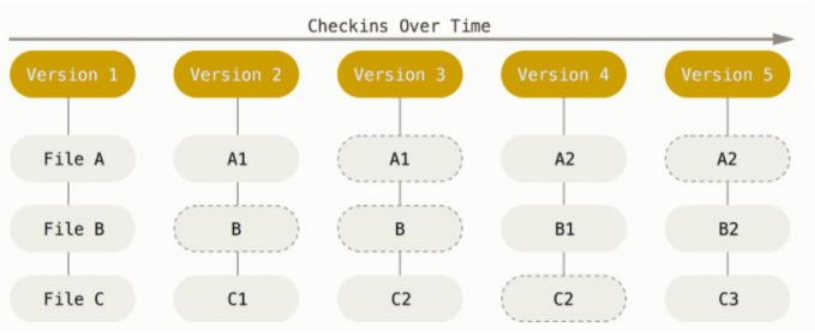
Časovna zahtevnost algoritma je O(n*m) (izpolnjevanje polj matrike, razbiranje rešitve je O(n+m)), prostorska pa prav tako O(n*m) (matrika velikosti n*m). Posamezni elementi zaporedij A in B so vrstice dveh različic datoteke.

Obstajajo tudi programi, ki so namenjeni zgolj hranjenju različnih verzij datotek. Rečemo jim sistemi za upravljanje z izvorno kodo (angl. VCS, *version control system*).



Običajni VCS-ji, kot je Subversion, shranijo začetno stanje datoteke, nato pa na vsaki novi verziji shranijo zgolj spremembe te datoteke. Takemu zapisu pravimo delta zapis podatkov.

Izjema med VCS-ji je Git, ki ne beleži sprememb na ta način, ampak gleda na vsako verzijo bolj kot posnetek malega datotečnega sistema. Vsakič ko pošljemo nove datoteke, si jih Git zapomni, in shrani referenco na posnetek oddanih datotek. V primeru da se kakšna datoteka ni spremenila, Git shrani zgolj povezavo na prejšnjo verzijo datoteke.



Glavna prednost Gita je predvsem preprosto vejanje sistema in lokalno skladišče - torej nimamo le ene točke okvare (kot npr. Subversion).

Krpalnike uporabljajo proizvajalci programske opreme, tako da uporabniku nameščene verzije programa pošljejo le krpe. Tako odpravljajo napake v programu. Krpalniki se med drugim uporabljajo za branje fizičnih podenot zunanjih pomnilniških naprav, berejo in spreminjajo logične podenote v zbirkah ali izvedljivih modulih. Krpalniki so v neposrednem stiku z zunanjimi pomnilniškimi mediji, ki jih morajo doseči tudi kadar so pokvarjeni. Njihovo delovanje pa mora biti čim bolj neodvisno od drugih programov. Ponavadi komunicirajo z že napisanimi rutinami operacijskega sistema za nadzor vhodno-izhodnih operacij. Te rutino ločijo med fizičnim (fizična podenota diska) in logičnim (sklicujemo se na ime zbirke) dostopom do podatkov.

Krpalnik se uporablja tudi za testiranje delovanja diskovnih površin. To storijo tako prepišejo izbrano območje in primerjajo nato prebrano vsebino z pričakovano. V kolikor rezultati niso pričakovani, je izbrano območje verjetno okvarjeno. To metodo testiranja lahko uporabimo na neuporabljenih površinah, drugače pa preverjajo zgolj nemoteno branje iz površine. Pomembno je tudi, da operacijski sistem ne prekine delovanja krpalnika ko zazna napako na zunanji napravi, saj tako damo krpalniku možnost da se sam odloči kako bo odreagiral na napako. To naredimo tako, da prekinitvene vektorje za obravnavo napak preusmerimo iz prekinitvenih rutin operacijskega sistema na prekinitveno rutino znotraj krpalnika.

Razložili smo kako lahko zaznamo spremembe v tekstovnih datotekah(izvorni kodi programov). Vendar pa ta ni vedno dostopna. Razvijalci naredijo krpo na podlagi prevedenih verzij programa nato pa poiščejo razlike med različicama kode. V primeru ko dodajamo ukaze, se spremenijo naslovi kakšnih operandov, skočnih ukazov ali podatkovnih struktur, ki so vezani nanje. V kolikor nimamo dostopa do izvornega programa, moramo sami ugotoviti kakšne spremembe so potrebne na strojnem nivoju. Če je koda enake dolžine lahko program le prepišemo. Lahko tudi izločamo dele kode s skočnimi ukazi. Težje je dodajati ukaze, kjer moramo narediti nove ukazne predele in jih povezati s skočnimi ukazi. Npr. lahko zapišemo na izbrano mesto skočni ukaz na konec kode, kjer najprej dodamo povožen ukaz, na koncu predela pa imamo še skočni ukaz za povratni skok.

Poznamo tudi nekatere bolj neobičajne ampak uporabne načine krpanja. Na primer Monkey patching, ko program spreminjamo v pomnilniku medtem ko teče, namesto da spreminjamo njegov zapis na disku. Poznamo pa tudi dinamično krpanje kode, kjer popravimo program brez da bi pri tem ustavili in ponovno pognali program ali sistem.

Viri in literatura:

Zazula, D. in Lenič, M.(2006). *Principi sistemske programske opreme*. Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2006

Myers , E.(2020). *An O(ND) Difference Algorithm and Its Variations.* Pridobljeno: http://www.xmailserver.org/diff2.pdf

Diff.(2020). Pridobljeno s https://en.wikipedia.org/wiki/Diff

Patch (computing).(2020). Pridobljeno s https://en.wikipedia.org/wiki/Patch_(computing)

patch (Unix). (2020). Pridobljeno s https://en.wikipedia.org/wiki/Patch_(Unix)

rsync. (2020). Pridobljeno s https://en.wikipedia.org/wiki/Rsync

The rsync algorithm. (2020). Pridobljeno s https://rsync.samba.org/tech_report/

RSync - Detecting File Differences. (2020). Pridobljeno s http://tutorials.jenkov.com/rsync/checksums.html

Longest common subsequence problem . (2020). Pridobljeno s https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

Dynamic software updating. (2020). Pridobljeno s https://en.wikipedia.org/wiki/Dynamic_software_updating

Monkey patch. (2020). Pridobljeno s https://en.wikipedia.org/wiki/Monkey_patch

Git book. (2020). Pridobljeno s https://git-scm.com/book/en/v2

Patch(1) — Linux manual page. (2020). Pridobljeno s https://man7.org/linux/man-pages/man1/patch.1.html

diff(1) — Linux manual page. (2020). Pridobljeno s https://man7.org/linux/man-pages/man1/diff.1.html

git/xdiff/xdiffi.c. (2020). Pridobljeno s https://github.com/git/git/blob/master/xdiff/xdiffi.c