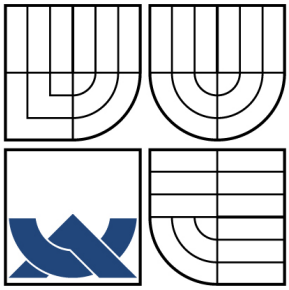
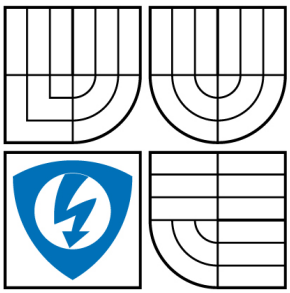


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

METODY DEKÓDOVÁNÍ KONVOLUČNÍCH KÓDŮ

METHODS DECODING CONVOLUCIONAL CODE

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — PETER VLČEK

AUTHOR

VEDOUCÍ PRÁCE — Ing. VÍTĚZSLAV KŘIVÁNEK

SUPERVISOR

BRNO 2008

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami:

1. Pan/paní

Jméno a příjmení: Peter Vlček
Bytem: Západná 11/42, 91108, Trenčín
Narozen/a (datum a místo): 16.10.1985, Ilava

(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta elektrotechniky a komunikačních technologií
se sídlem Údolní 244/53, 60200 Brno 2
jejímž jménem jedná na základě písemného pověření děkanem fakulty:
prof. Ing. Kamil Vrba, CSc.

(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   - ☐ disertační práce
   - ☐ diplomová práce
   - ☑ bakalářská práce

   jiná práce, jejíž druh je specifikován jako .....

(dále jen VŠKP nebo dílo)

Název VŠKP: Metody dekódování konvolučních kódů
Vedoucí/školitel VŠKP: Ing. Vítězslav Křivánek
Ústav: Ústav telekomunikací
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

- ☑ tištěné formě - počet exemplářů 1
- ☑ elektronické formě - počet exemplářů 1

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2

Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizovaní výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti
   - ☑ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3

Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Konvolučné kódy patria medzi lineárne stromové kódy. Sú to kódy používané pre kanálové kódovanie. Pri vypracovaní projektu, teda porovnaní jednotlivých algoritmov boli brané do úvahy Viterbiho algoritmus, Stack algoritmus, Stack-bucket algoritmus, Fano algoritmus a Feedback dekódovania. Bol zostavený návrh konvolučného kodéra a dekodéra, ktorý pri svojej činnosti využíva Viterbiho algoritmus a Stack algoritmus, čo slúži pre porovnanie oboch metód. Ukázalo sa, že Viterbiho algoritmus je vhodný hlavne pre kódy s menšou nútenou dĺžkou, pretože má pevné množstvo počítaní. Pri Stack algoritme je množstvo počítaní závislé na dĺžke prijatej sekvencie, čo môže mať za následok oveľa menší počet počítaní, avšak pri veľmi hlučnom kanáli sa môže stať, že počítaní bude omnoho viac ako u Viterbiho algoritmu. Ničmenej je Stack algoritmus alebo jeho variácia Stack-bucket algoritmus používaný hlavne na kódy s väčšou nútenou dĺžkou. Feedback dekódovanie je zase pre jednoduchú konštrukciu dekodéra vhodnou alternatívou k dekódovaniu s najväčšou pravdepodobnosťou a k sekvenčnému dekódovaniu.

Kľúčové slová

Viterbiho a Stack algoritmus pre dekódovanie konvolučných kódov

Abstract

Convolutional codes belongs to among linear tree codes. There are codes used for channel coding. At build - up project, then comparison separate algorithms were gateway into the account Viterbiho algorithm, Stack algorithm, Stack- bucket algorithm, Fano algorithm and Feedback decoding. Was built-up suggestion convolutional encoder and decoder, which at his activities make use of Viterbiho algorithm and Stack algorithm, what serves for contrast of both methods. It turned out, that the Viterbiho algorithm get past in the main for codes with smaller compulsory longitude, because has stand-by cost quantity calculated effect. At Stack algorithms is quantity calculated effect dependent on longitude received sequence, what may result in much smaller number calculated effect, however at very noisy channel is able to state, that the calculated effect will much more as with Viterbiho algorithm. Nothing less is Stack algorithm or his variation Stack- bucket algorithm used primarily on codes with bigger compulsory longitude. Feedback decoding is gainst for simple construction decoder acceptable alternative toward decoding most likely and to sequential decoding.

Keywords

Viterbi and Stack algorithms for decoding of convolutional codes

Prohlášení

Prohlašuji, že svůj semestrální projekt na téma „Metody dekódování konvolučních kódů“ jsem vypracoval samostatně pod vedením vedoucího semestrálního projektu a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedeného semestrálního projektu dále prohlašuji, že v souvislosti s vytvořením tohoto projektu jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne …….. ……...

podpis autora

Poďakovanie

Ďakujem vedúcemu bakalárskej práce Ing. Vítězslavu Křivánkovi za veľmi užitočnú metodickú pomoc a cenné rady pri spracovaní bakalárskej práce.

V Brne dňa …….. ……...

podpis autora

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam skratiek a symbolov

L	-	dĺžka informačnej sekvencie
u	-	informačná sekvencia
$\tilde{\mathbf{v}}$	-	konečný survivor
r	-	prijatá sekvencia
v	-	zakódovaná sekvencia
survivor	-	cesta s najlepšou metrikou
MLD	-	maximum likehood decoder (dekodér s najväčšou pravdepodobnosťou)
M	-	metrická hodnota
M_F	-	metrická hodnota pokračujúceho preskúmaného uzlu
M_B	-	metrická hodnota spätného preskúmaného uzlu
Δ	-	prahové zvýšenie
T	-	prah

Úvod

Úlohou tejto bakalárskej práce je zoznámiť sa s dekódovacími metódami konvolučných kódov, preštudovať základné dekódovavie algoritmy, porovnať ich výhody a nevýhody a nájsť najvýhodnejšie riešenie pre prax. Súčasťou práce je navrhnúť na základe získaných poznatkov model dekodéru a simuláciou preveriť jeho funkčnosť. A nakoniec ho zrelizovať.

Konvolučné kódy patria medzi lineárne stromové kódy. Sú to kódy používané pre kanálové kódovanie. Svoj názov odvodzujú od latinského slova konvolut – prepletenec.

Pre dekódovanie konvolučných kódov je známych viacero metód. Dekódovacie spôsoby stromových kódov môžme rozdeliť podla spôsobu, ktorý na ich riešenie používame, na syndromové dekódovanie a postupné dekódovanie.

Syndromové dekódovanie vychádza z existencie syndrómu, ako výsledku kontroly správnosti prenesenej postupnosti bitov.

Postupné dekódovanie využíva odchyliek medzi prijatou postupnosťou bitov a možnými postupnosťami bitov.

1 Dekódovanie konvolučných kódov s najväčšou pravdepodobnosťou

V roku 1967, Viterbi predstavil dekódovací algoritmus pre konvolučné kódy, ktorý je až doteraz známy ako Viterbiho algoritmus. Neskôr, Omura ukázal, že Viterbiho algorimus bol rovnocenný k dynamickému programovému riešeniu problému nájdutia najkratšej cesty cez vážený obrazec. Nakoniec, Forney uznal, že je to v skutočnosti najlepší dekódovací algoritmus pre konvolúčné kódy, to znamená že vybraný výstup dekodéru je vždy kódové slovo, ktoré dáva najväčšiu hodnotu logickej funkcie.

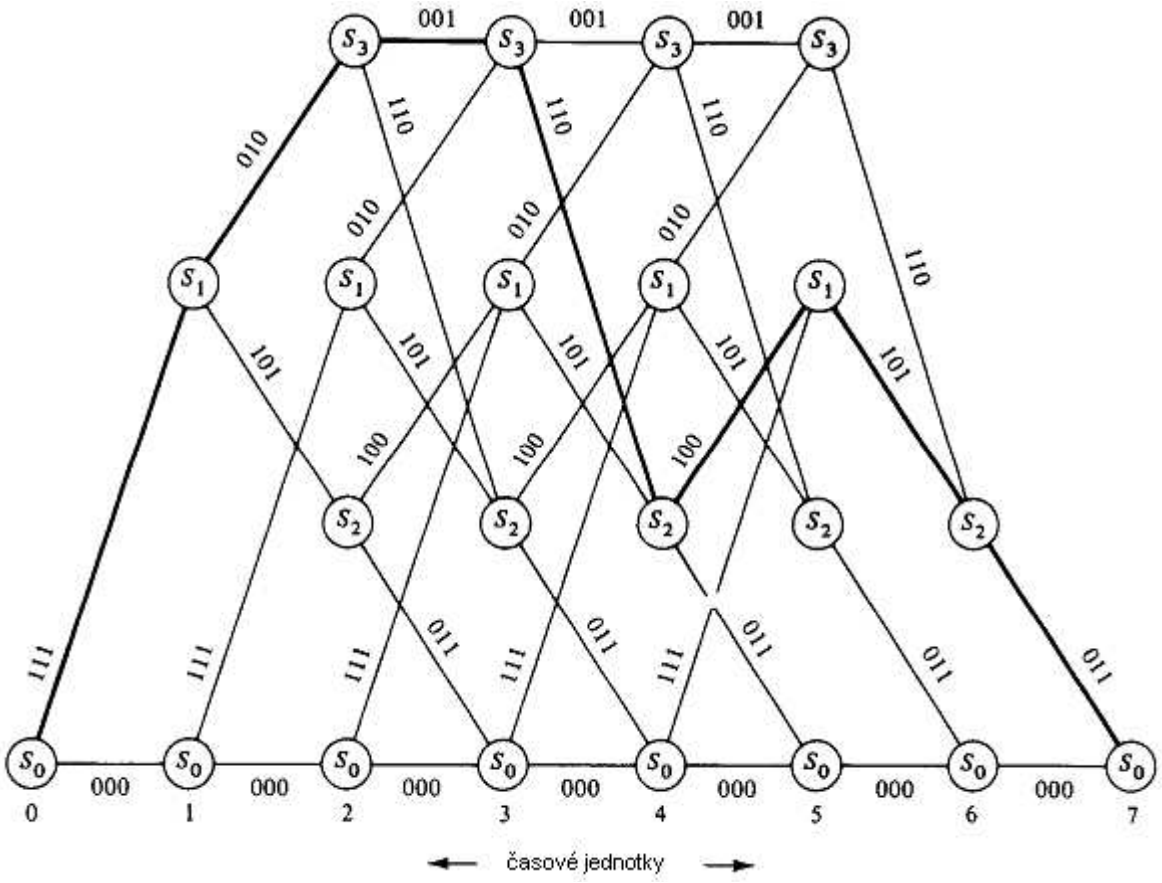
Forney však bol prvý, ktorý upozornil, že Viterbiho algorimus môže byť použitý, aby produkoval maximálny odhad pravdepodobnosti prenesenej sekvencie cez skupinový kanál s medzisymbolovou interferenciou. Toto pozorovanie je zjavné, keď sa sleduje, že kanálová pamäť, ktorá produkuje medzisymbolovú interferenciu je podobná jako pamäť kodéru v konvolučnom kóde.

1.1 Viterbiho algoritmus

K porozumeniu Viterbiho dekódovacieho algoritmu, je třeba rozšíriť stavový diagram kodéru v čase (tj. Predstaviť každú časovú jednotku v separátnom časovom diagrame). Vyplývajúca štruktúra sa nazýva schéma latkovej mreže, a je ukázaná na obrázku 1 pre (3, 1, 2) kódu je s

$$\mathbf{G}(£>) = [1 + D, 1 + D^2, 1 + D + D^2]$$

s informačnou sekvenciou L = 5. A schéma latkovej mreže obsahuje L +



Obrázok 1.1:schéma latkovej mreže pre (3, 1, 2) kód s L = 5

m+1 časových jednotiek alebo úrovní a tieto sú označené od 0 až do L + m v obrázku 1. Prijímanie, ktoré kodér začína v stave S_0 a vracia sa do stavu S_0, prvé m časové jednotky zodpovedajú odchodu kodéru zo stavu S_0 , a posledné m časové jednotky zodpovedajú návratu kodéru do stavu S_0. Z toho vyplýva, že nie všetky stavy môžu byť dosiahnuté v prvej m alebo v poslednej m časovej jednotke. Avšak, v centrálnej časti latkovej mreže sú všetky stavy možné, a každá časová jednotka obsahuje repliku stavového diagramu. Sú tam dve vetvy opustenia a vstupovania každého stavu. Horná vetva opustenia každého stavu v časovej jednotke *i* predstavuje vstup $u_i = 1$, zatial čo nižšia vetva predstavuje $u_i = 0$. Každá vetva je označená n odpovedajúcimi

výstupmi v_i a každé z 2^L kódových slov dĺžky $N = n(L + m)$ je reprezentované jedinečnou cestou cez latkovú mrežu. Napríklad kódové slovo odpovedajúce informačnej sekvencii je zvýraznemé na obrázku 1. V hlavnom prípade (n, k, m) kódu a informačnej sekvencie dĺžky kL, je tam 2^k vetví opustenia a vtupovania každého stavu, a 2^{kL} zretelných ciest cez latkovú mrežu odpovedajúcich 2^{kL} kódovým slovám.

Teraz predpokladajme, že informačná sekvencia $u = (u_0, \ldots, u_{L-1})$ o dĺžke kL je kódovaná v kódovom slove $v = (v_0, v_1, \ldots, v_{L+m-1})$ dĺžky $N = n(L + m)$ a tak Q-*ary* sekvencia $r = (r_0, r_1, \ldots, r_{L+m-1})$ je prijatá cez binárny vstup, Q-*ary* výstup diskrétny bezpamäťový kanál (DMC). Alternatívne, tieto sekvencie môžu byť napísané ako $u = (u_0, u_1, \ldots, u_{kL-1})$, $v = (v_0, v_1, \ldots, v_{N-1})$ a $\mathbf{r} = (r_0, r_1, \ldots, r_{N-1})$, kde dolné indexy teraz jednoducho predstavujú objednanie symbolov v každej sekvencii. Dekodér musí produkovať približné $\tilde{\mathbf{v}}$ z kódového slova v založené na prijatej sekvencii *r.* Dekodér s maximálnou pravdepodobnosťou (*maximum likehood decoder (MLD))* pre DMC vybera $\tilde{\mathbf{v}}$ ako kódové slovo *v*, ktoré maximalizuje v logickej pravdepodobnostnej funkcii log P (r | v). Odvtedy pre DMC

$$P(\mathbf{r} \mid \mathbf{v}) = \prod_{i=0}^{L+m-1} P(\mathbf{r}_i \mid \mathbf{v}_i) = \prod_{i=0}^{N-1} P(r_i \mid v_i), \tag{1.1}$$

z toho vyplýva

$$\log P(\mathbf{r} \mid \mathbf{v}) = \sum_{i=0}^{L+m-1} \log P(\mathbf{r}_i \mid \mathbf{v}_i) = \sum_{i=0}^{N-1} \log P(r_i \mid v_i), \tag{1.2}$$

kde P (r_i | v_i) je kanálová pravdepodobnosť prechodu. Toto je dekódovacie pravidlo s minimálnou pravdepodobnosťou chýb kedy všetky kódové slová sú rovnako pravdepodobné.

Pravdepodobnostná funkcia log P (r | v) je nazvaná metrika s pridruženou cestou *v* a je označená M(r | v). Podmienky P (r_i | v_i) sú nazývané metrika vetvy a označené M(r_i | v_i), zatialčo podmienky log P (r_i | v_i) sú nazývané bitová metrika a sú označené M(r_i | v_i). Z toho dôvodu, metrika pridružená s cestou M(r | v) môže byť zapísaná ako

$$M(\mathbf{r} \mid \mathbf{v}) = \sum_{i=0}^{L+m-1} M(\mathbf{r}_i \mid \mathbf{v}_i) = \sum_{i=0}^{N-1} M(r_i \mid v_i), \tag{1.3}$$

A čiastočná cestová metrika pre prvé *j* vetvy cesty môže byť vyjadrená ako

$$M([\mathbf{r} \mid \mathbf{v}]_j) = \sum_{i=0}^{j-1} M(\mathbf{r}_i \mid \mathbf{v}_i). \tag{1.4}$$

Následne algoritmus, keď aplikujeme na prijatú sekvenciu *r* z DMC, nachádza cestu cez latkovú mrežu s najväčšou metrikou (t.j. Maximálna cesta pravdepodobnosti) . Algoritmus spracováva *r* opakovacím spôsobom. V každom kroku porovnáva metriku všetkých vstupujúcich ciest každého stavu a skladuje cestu s najväčšou metrikou, nazývanú preživsí (survivor)[1], spoločne s jeho metrikou.

Viterbiho algoritmus

krok 1. Začiatok na časovej jednotke j = m, vypočíta čiastočnú metriku pre jednu vstupujúcu cestu každého stavu. Skladuje cestu (survivor) a jeho metriku pre každý stav.

krok 2. Zvýšiť ju o 1. Vypočíta čiastočnú metriku pre všetky cesty vstupujúcich stavov pridávaním metriky stupujúcej vetvy toho stavu k spojovacej metrike survivor v predchádzajúcej časovej jednotke. Pre každý stav skladuje cestu s najväčšou merikou (survivor), spoločne s jeho metrikou a odstráni všetky ostatné cesty.

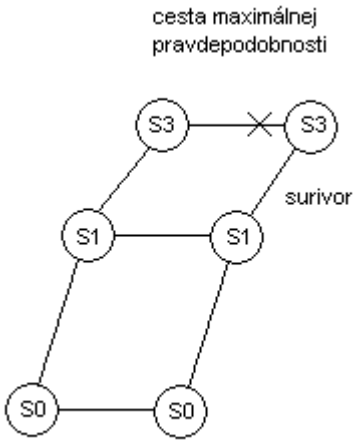
krok 3. Pokial j < L + m, opakuje krok 2. Inak skončí.

Je tam 2^k survivors z časovej jednotky *m* cez časovú jednotku *L*, Jeden pre každý z 2^k stavov. Po časovej jednotke *L* je tam menej survivors, pretože je tam menej stavov zatial čo sa kodér vracia do celonulového stavu. Nakoniec v časovej jednotke L + r je tam len jeden stav, celonulový stav, a z toho dôvodu je len jeden survivor a algoritmus sa ukončí. Teraz ukážeme, že tento konečný survivor ja maximálna cesta pravdepodobnosti.

Tvrdenie 1 : Konečný survivor vo Viterbiho algoritme je cesta s maximálnou pravdepodobnosťou [1], to je

$$M(r / \tilde{v}) \geq M(r / v), \qquad \text{pre všetky } v \neq \tilde{v}$$

Dôkaz: Povedzme, že maximálna cesta pravdepodobnosti je vyradená z algoritmu v časovej jednotke *j* a zobrazená na obrázku 1.2. Z toho vyplýva, že čiastočná metrika cesty survivora poračuje od maximálnej cesty pravdepodobnosti v tomto bode. Teraz, ak je zostávajúca časť maximálnej cesty pravdepodobnosti pripojená na survivora v časovej jednotke *j*, celková metrika cesty bude prevyšovať celkovú metriku maximálnej cesty pravdepodobnosti. Ale toto popiera definíciu maximálnej cesty pravdepodobnosti jako cestu s najvyššou metrikou. Z toho dôvodu maximálna cesta pravdepodobnosti nemôže byť vyradená algoritmom, a musí byť konečný survivor.



Obrázok 1.2: Vylúčenie cesty maximálnej pravdepodobnosti

Preto, Viterbiho algoritmus je optimálny v zmysle, že vždy nachádza maximálnu cestu pravdepodobnosti cez latkovú mrežu.

Z hladiska implementácie je lepšie používať metriku prirodzených čísel, než aktuálnu bitovú metriku. Bitová metrika $M(r_i \mid v_i) = \log P(r_i \mid v_i)$ môže byť nahradená $c_2[\log P(r_i \mid v_i) + c_1]$, kde c_1 je niake reálne číslo a c_2 ja niake kladné reálne číslo. To môže byť ukázané v ceste, ktorá maximalizuje $M(\mathbf{r} \mid \mathbf{v}) = \sum_{i=0}^{N-1} M(r_i \mid v_i) = \sum_{i=0}^{N-1} \log P(r_i \mid v_i)$

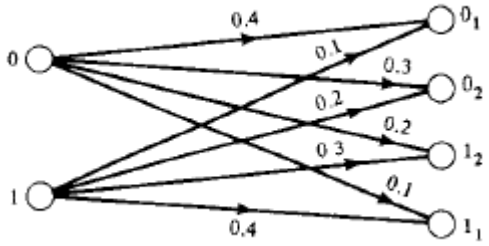
tiež maximalizuje

$$\sum_{i=0}^{N-1} c_2[\log P(r_i \mid v_i) + c_1],$$

a z toho dôvodu upravená metrika môže byť použitá bez ovlyvnenia výkonu Viterbiho algoritmu. Pokial c_1 je vybrané aby urobilo najmenšiu metriku 0, potom c_2 môže byť vybrané tak, aby celá metrika mohla byť aproximovaná celými číslami. Je mnoho celočíselných metrík pre dané DMC v závislost na volbe c_2. Výkon Vitebiho algoritmu je teraz mierne znížený kôli upravenej metrickej aproximácii celými číslam, ale zníženie je spravidla velmi nepatrné.

Príklad 1.1: Ako príklad zvažujme dvojkový vstup, kvartérny výstup (Q = 4)DMC ukázaný v obrázku 1.3. Použitím logaritmu o základe 10, bitová metrika pre tento kanál je zobrazená v metrickej tabulke v tabulke 1.1a. Vybraním $c_1 = 1$ a $c_2 = 17{,}3$ vynáša celočíselnú metrickú tabulku zobrazenú v tabulke 1.1b. Teraz predpokladajme, že kódové slovo zo chématu latkovej mreže (3, 1, 2) kódu zobrazeného v obrázku 1 je prenesené cez DMC z obrázku 3 a že kvartérna doručená sekvecia je :

$$\mathbf{r} = (1_1 1_2 0_1,\ 1_1 1_1 0_2,\ 1_1 1_1 0_1,\ 1_1 1_1 1_1,\ 0_1 1_2 0_1,\ 1_2 0_2 1_1,\ 1_2 0_1 1_1).$$



Obrázok 1.3: Binárny vstup, kvartérny výstup DMC

Tabulka 1.1: Metrické tabulky pre kanál z obrázku 1.3

$\mathbf{r_i}$	$\mathbf{0_1}$	$\mathbf{0_2}$	$\mathbf{1_1}$	$\mathbf{1_2}$
$\mathbf{v_i}$	-	-	-	-
0	- 0,4	- 0,52	- 0,7	-1
1	-1	- 0,7	- 0,52	- 0,4

a

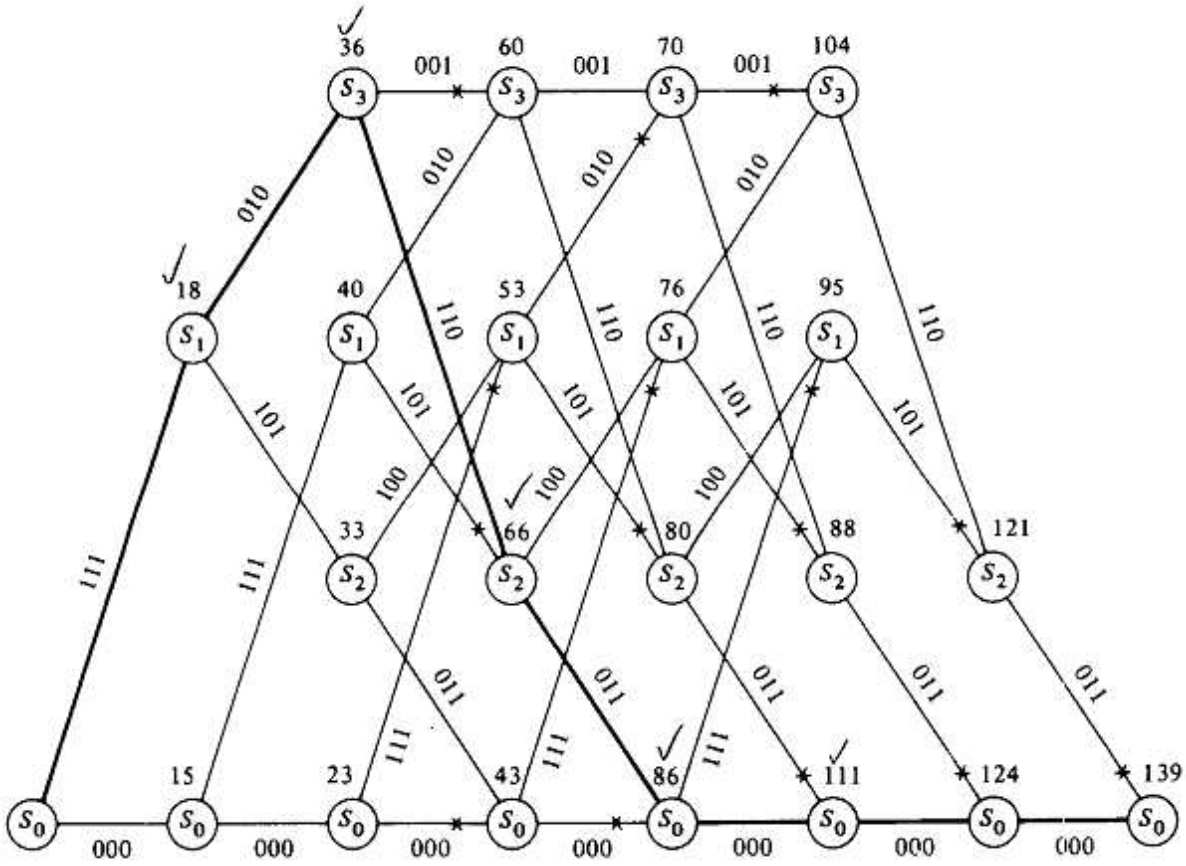
$\mathbf{r_i}$	$\mathbf{0_1}$	$\mathbf{0_2}$	$\mathbf{1_1}$	$\mathbf{1_2}$
$\mathbf{v_i}$	-	-	-	-
0	10	8	5	0
1	0	5	8	10

b

Použitie Viterbiho algoritmu je zobrazené na obrázku 1.4. Čísla nad každým stavom predstavujú metriku survivor pre ten stav, a cesty vyradené z každého stavu sú zobrazené ako vyškrtnuté z diagramu latkovej mreže. Konečný survivor,

$$\tilde{\mathbf{v}} = (1\ \ 1\ \ 1,\ \ 0\ \ 1\ \ 0,\ \ 1\ \ 1\ \ 0,\ \ 0\ \ 1\ \ 1,\ \ 0\ \ 0\ \ 0,\ \ 0\ \ 0\ \ 0,\ \ 0\ \ 0\ \ 0),$$

je zobrazený ako zvýraznená cesta. Toto zodpovedá dekódovanej sekvencii informácií.



R = ($1_1 1_2 0_1$, $1_1 1_1 0_2$, $1_1 1_1 0_1$, $1_1 1_1 1_1$, $0_1 1_2 0_1$, $1_2 0_2 1_1$, $1_2 0_1 1_1$)

Obrázok 1.4 : Viterbiho algoritmus pre DMC

ű = (1 1 0 0 0). Všimnite si, že posledné *m* vetvy v každej časti latkovej mreže zodpovedajú nulovým svtupom a preto ich nepovažujeme za časť informačnej sekvencie.

V špeciálnom prípade binárneho symetrického kanála BSC s prevdepodobnosťou prechodu p < ½, je prijatá sekvencia *r* binárna (Q = 2) a logická funkcia pravdepodobnosti sa stáva:

$$\log P(\mathbf{r}\,|\mathbf{v}) = d(\mathbf{r}, \mathbf{v}) \log \frac{p}{1-p} + N \log (1-p), \tag{1.5}$$

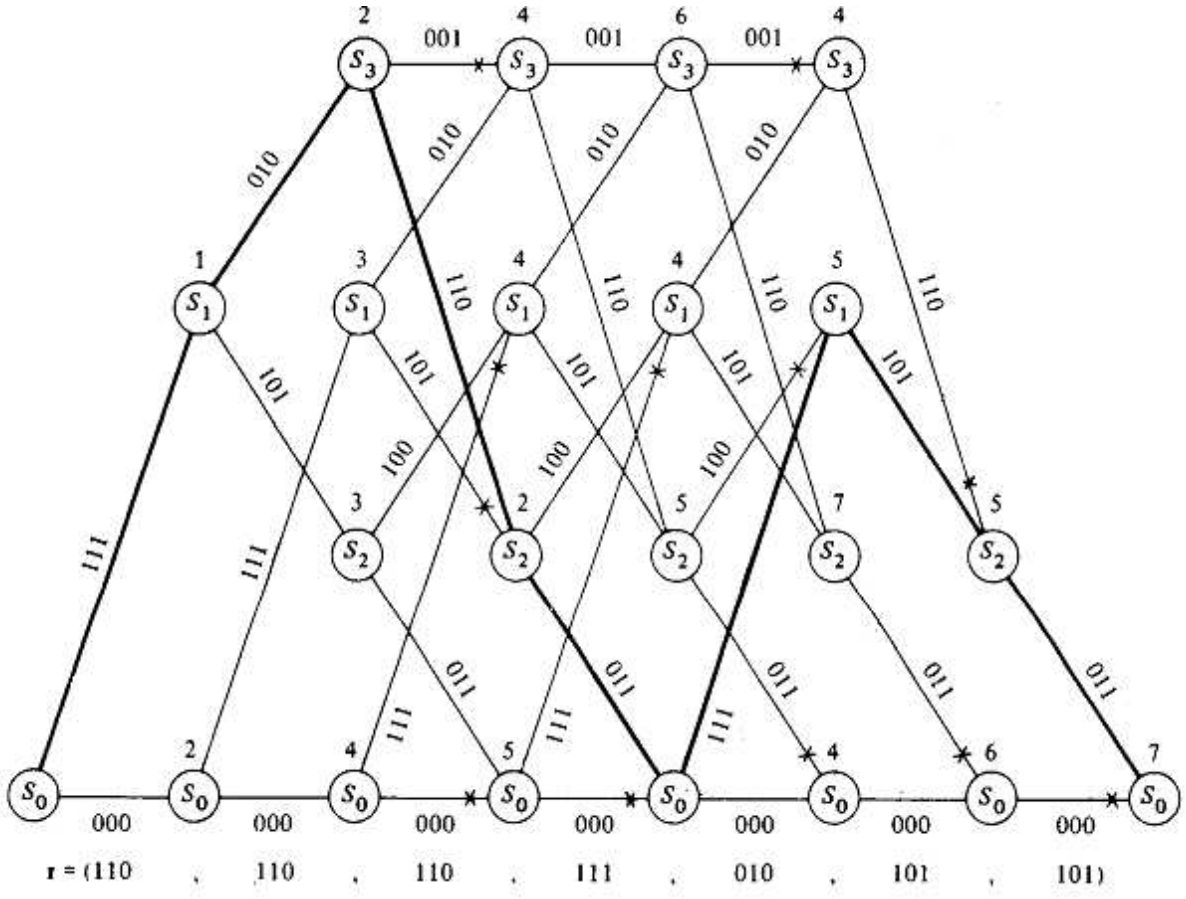
kde $d(\mathbf{r}, \mathbf{v})$ je Hammingova vzdialenosť medzi *r* a *v.* Od log $[p/(1-p)] < 0$ a N log $(1-p)$ je konštantná pre všetky *v,* MLD pre BSC si vybere *v* ako kódové slovo $\tilde{v}$ tak minimalizuje hamingovu vzdialenosť

$$d(\mathbf{r}, \mathbf{v}) = \sum_{t=0}^{L+m-1} d(\mathbf{r}_t, \mathbf{v}_t) = \sum_{t=0}^{N-1} d(r_t, v_t). \tag{1.6}$$

Z toho dôvodu, pri aplikovaní Viterbiho algoritmu na BSC, $d(r_i, v_i)$ sa stáva metrikou vetvy, $d(r_i, v_i)$ sa stáva bitovou metrikou a algoritmus musí nájsť cestu skrz latkovú mrežu s najnižšou metrikou (t.j. Cesta najbližšia k *r* v Hammingovej vzdialenosti). Detaily algoritmu sú v skutku rovnaké, ibaže Hammindova vzdialenosť nahrádza logickú pravdepodobnostnú funkciu ako metriku a survivor v každom stave je cesta s najmenšou metrikou.

Príklad 2: Príklad použitia Viterbiho algoritmu k BSC je zobrazený na obrázku 1.5, Predpokladajme, že kódové slovo z diagramu latkovej mreže (3, 1,2) kódu z obrázku 1 je prenesený cez BSC a prijatá sekvencia je:

$$\mathbf{r} = (1\ \ 1\ \ 0,\ \ 1\ \ 1\ \ 0,\ \ 1\ \ 1\ \ 0,\ \ 1\ \ 1\ \ 1,\ 0\ \ 1\ \ 0,\ \ 1\ \ 0\ \ 1,\ \ 1\ \ 0\ \ 1).$$



Obrázok 1.5: Viterbiho algoritmus pre BSC

Posledný survivor

$$\tilde{\boldsymbol{v}} = (1\ \ 1\ \ 1,\ \ 0\ \ 1\ \ 0,\ \ 1\ \ 1\ \ 0,\ \ 0\ \ 1\ \ 1,\ \ 1\ \ 1\ \ 1,\ \ 1\ \ 0\ \ 1,\ \ 0\ \ 1\ \ 1),$$

je zobrazený ako zýraznená cesta na obrázku a dekódovaná informačná sekvencia je $\tilde{u} = (1\ 1\ 0\ 0\ 1)$. Skutočnosť, že survivor má metriku 7 znamená, že žiadna iná cesta cez latkovú mrežu odlišná od *r* nemá menej ako 7 pozícií. Všimnite si, že v niektorých stavoch

žiadna cesta nieje vyškrtnutá. To signalizuje väzbu v metrických hodnotách dvoch ciest vstupujúcich do stavu. Pokial survivor prechádza akýmkolvek z týchto stavov, je tam viac než jedna maximálna cesta pravdepodobnosti (t.j. Viac než jedna cesta, ktorej vzdialenosť od *r* je minimálna). Z hladiska implementácie, kedykolvek nastáva väzba v metrických hodnotách, jedna cesta je lubovolne vybraná ako survivor, kôli nepraktičnosti skladovania premenného množstva ciest. Toto lubovolné riešenie väzieb nemá žiadny účinok na dekódujúcu pravdepodobnosť chýb.

Robenie mäkkých rozhodnutí demodulátora *(Q* > 2) má za následok výkonovú výhodu nad robením tvrdých rozhodnutí (*Q* = 2) [2]. Dva príklady použitia Viterbiho algoritmu vyššie to pomáhajú ilustrovať. Pokial kvartérne výstupy 0_1 a 0_2 splynú do jednoho výstupu 0, a 1_1 a 1_2 do jednoho výstupu 1, mäkké rozhodnutie DMC je prevedené na tvrdé rozhodnutie BSC s pravdepodobnosťou prechodu *p* = 0,3. Vo vyššie uvedenom príklade, sekvencia *r* v prípade tvrdého rozhodnutia je rovanká ako v prípade mäkkého rozhodnutia 0_1 a 0_2 prevedené na 0 a1_1 a 1_2 prevedené na 1. Ale Viterbiho algoritmus vynáša rôzne výsledky v dvoch prípadoch. V prípade mäkkého rozhodnutia (*Q* = 4), sekvencia informácií *u* = (1 1 0 0 0) produkuje maximálnu cestu pravdepodobnosti, ktorá má konečnú metriku 139. V prípade tvrdého rozhodnutia (*Q* = 2), maximálna cesta pravdepodobnosti je *u* = (1 1 0 0 1). Metrika tejto cesty na kvarternom výstupe kanálu je 135 a zretelne to neni maximálna cesta pravdepodobnosti v prípade mäkkého rozhodnutia. Ale, odkedy tvrdé rozhodnutie maskuje rozdiel medzi jasnými mäkkými rozhodnutiami (tzn. výstupy 0_1 a 0_2 sú považované za ekvivalentné výstupy v prípade tvrdého rozhodovania), hard decision dekodér robí odhad, ktorý by nebol urobený keby nebolo k dospozícii viac informácií o kanále (t.j. Keď v mäkkom rozhodnutí boli urobené).

Oba kanály popísané vyššie môžu byť klasifikované ako “velmi hlučné“ kanály. Kódový pomer R = ½ prekračuje kapacitu koryta C v oboch prípadoch. Z toho dôvodu by sme neočakávali, že výkon tohto kódu môže byť velmi dobrý s kanálom. Toto sa odráža v relatívne nízkej hodnote (139) konečnej metriky maximálnej cesty pravdepodobnosti v prípade DMC, v porovnaní s maximom možnej metriky cesty (210) ktorá súhlasí úplne s *r.* Taktiež, v BMC prípade, konečná Hammingova vzdialenosť 7 pre maximálnu cestu pravdepodobnosti je velká pre cesty dlhé len 21 bitov. Pre dosiahnutie dobrého výkonu cez tieto kanály by boli potrebné nižšie kódové sadzby.

2 Sekvenčné dekódovanie konvolučných kódov

Hlavný problém s Viterbiho dekódovaním konvolučných kódov je, že lubovolne malé pravdepodobnosti chýb slúbené náhodnou kódovacou hranicou niesu dosiahnutelné v praxi. Toto je kôli tomu, že len malá nútená dĺžka môže byť použitá kôli limitovanej pamäti kodéru *K*. Ďalší problém s Viterbiho algoritmom je fixované počítanie čísla 2^k , ktoré musí byť prevedené za dekódovaný blok informácií. Toto množstvo počítania není vždy potrebné, zvlášť keď je šum lahký. Vezmime príklad, že celá zakódovaná sekvencia dĺžky *N* je prenesená cez BSC (tj. $r = v$). Viterbiho algoritmus ešte vykonáva 2^k počítaní za dekódovaným blokom informácií, ktoré sú v tomto prípade všetky zbytočné. Inými slovami je žiaduce mať dekódujúcu procedúru, ktorej dekódujúce úsilie je prispôsobivé hladine hluku. Sekvenčné dekódovanie je taká procedúra. Jako uvidíme, dekódujúce úsilie sekvenčného dekódovania je nezávislé na *K*, takže môžu byť použité velké nútené dĺžky a z toho dôvodu môžu byť dosiahnuté lubovolne nízke pravdepodobnosti chýb. Temná stránka je, že šumové rámce zaberú vela počítania a dekódujúce časy niekedy prevýšia horný limit spôsobujúci, aby bola informácia vymazaná alebo stratená.

Historicky, sekvenčné dekódovanie bolo predstavené Wozencraftom ako prvá praktická metóda dekódovania konvolučných kódov [1]. V roku 1963, Fano, uviedol novú verziu sekvenčného dekódovania, následne označovanú jako Fanov algoritmus. O pár rokov neskôr bola nezávisle od seba Zingagirovom a Jelínkom objavená nová verzia sekvenčného dekódovania nazvaná stack (skladisko) alebo ZJ algoritmus. Tomuto algoritmu sa dá najjednoduchšie porozumieť.

2.1 Stack algoritmus

V diskusii sekvenčného dekódovania sa predstavuje 2^{kL} kódové slovo o dĺžke $N = n(L + m)$ pre (n, k, m) kód a informačnú sekvenciu dĺžky kL jako cestu cez kódový strom obsahujúci $(L + m + 1)$ časových jednotiek alebo úrovní. Kódový strom je len rozšírená verzia schémy latkovej mreže, v ktorom každá cesta je totálne odlišná od každej ďalšej cesty [5]. Kódový strom pre (3, 2, 1) kóduje s

$$\boldsymbol{G}(\mathrm{D}) = [1 + D, 1 + D^2, 1 + D + D^2],$$

je ukázaný na obrázku 2.1 pre informačnú sekvenciu dĺžky 5. Latkový diagram pre tento kód bol zobrazený na obrázku 1.1. L + m + 1 stromových úrovní je označených od 0 až po L + m v obrázku 2.1. Lavý krajný uzol stromu je nazvaný pôvodný uzol a predstavuje počiatočnú stanicu *S* kodéru. Je tam 2^k vetiev opustenia každého uzlu v prvých L úrovniach pre (L, k, m) kód. Toto je nazvané deliaca časť stromu. V obrázku 2.1, horná vetva opustenia každého uzlu v deliacej časti stromu predstavuje vstup $u_i = 1$, zatialco nižšia vetva predstavuje $u_i = 0$. Po *L* časových jednotkách tam je už len jedna vetva opustenia každého uzlu. Toto predstavuje vstupy $u_i = 0$. pre i = L, L+1, L+m-1, a odpovedá to návratu dekodéru na nulový stav S_0. Z toho dôvodu sa to nayýva ocas stromu a 2^{kL} pravé krajné uzly sa nazývajú koncové uzly. Každá vetva je označená *n* výstupmi v_i odpovedajúcimi vstupnej sekvencii, a každé z 2^{kL} kódových slov o dĺžke *N* je reprezentované totálne zretelnou cestou cez strom. Napríklad, kódové slovo odpovedajúce informačnej sekvencii *u* = (1 1 1 0 1) je ukázané zvýraznené na obrázku 2.1. je dôležité si uvedomiť , že kódový strom obsahuje úplne rovnakú informáciu o kóde ako diagram laťkovej mreže. Ako je vidieť, strom je lepšie vhodný k pochopeniu operácie sekvenčného dekodéra.

Sú rôznorodé algoritmy stromového vyhľadávania, ktoré sú zaradené pod sekvenčné dekódovanie. Budeme sa zaoberať stack alebo ZJ algoritmom trochu detailnejšie. Účel algoritmu sekvenčného dekódovania je hľadať cestu skrz uzly kódového stromu účinným spôsobom (t.j. bez toho, aby skúmal príliš mnoho uzlov.) v pokuse nájsť maximálnu cestu pravdepodobnosti. Každý preskúmaný uzol predstavuje cestu cez časť stromu. Či špecifická cesta bude časť maximálnej cesty pravdepodobnosti závisí na metrickej hodnote pridruženej k tejto ceste. Metrika je miera blízkosti cesty k prijatej sekvencii.

Pre binárny vstup, Q-ary výstup DMC, metrika vo Viterbiho algoritme pravdepodobnostnú funkciu (1.1). Toto je optimálna metrika pre Viterbiho algoritmus od ciest porovnávaných v nejakom dekódujúcom kroku, ktoré majú všetky rovnakú dĺžku. Avšak v sekvenčnom dekódovaní sada ciest, ktoré boli preskúmané v určitom dekódovacom kroku majú obecne rôznu dĺžku. Pokiaľ

je metrika logickej funkcie použitá pre tieto cesty, doručenej sekvencii vyplýva skreslený obraz blízkosti ciest.

Príklad 2.1:

Uvažujme kódový strom z obrázku 2.1. Predpokladajme, že kódové slovo je prenesené z tohoto kódu cez BSC, a že sekvencia

$\boldsymbol{r}$ = (0 1 0, 0 1 0, 0 0 1, 1 1 0, 1 0 0, 1 0 1, 0 1 1).

je prijatá. Pre BSC, hodnota metriky pre cestu *v* vo Viterbiho algoritme je daná *d(r, v)*, s maximálnou cestou pravdepodobnosti je ten s najmenšou metrikou. Teraz zvažujeme porovnávanie čiastočnej metriky cesty pre dve cesty rôznych dĺžok [t.j. Skrátené kódové slová $[v_5]$ = (1 1 1, 0 1 0, 0 0 1, 1 1 0, 1 0 0, 1 0 1) a $[v_0]$ = (0 0 0)]. Čiastočné metriky ciest sú $d([r_5], [v_5])$ = 2 a $d([r_0], [v_0])$ = 1, čo dáva najavo, že $[v_0]$ je lepšia z dvoch ciest. Avšak, naša intuícia nám hovorí, že cesta $[v_5]$ má väčšiu pravdepodobnosť byť časťou maximálnej cesty pravdepodobnosti ako cesta $[v_0]$, do dokončenia cesty začínajúcej s $[v_0]$ vyžaduje 18 prídavných bitov, v porovnaní s len 3 prídavnými bitmi požadovanými k tomu, aby dokončili cestu začínajúcu s $[v_5]$. Inými slovami, cesta začínajúca s $[v_0]$ pravdepodobne nahromadí väčšiu vzdialenosť od *r* ako cesta začínajúca s $[v_5]$.

Preto je nutné nastaviť metriku použitú v sekvenčnom dekódovaní k tomu, aby brala do úvahy dĺžky rôznych ciest, ktoré boli porovnané. Pre binárny vstup a Q-ary výstup DMC, najlepšia bitová metrika použitá pri porovnávaní ciest rôznej dĺžky je:

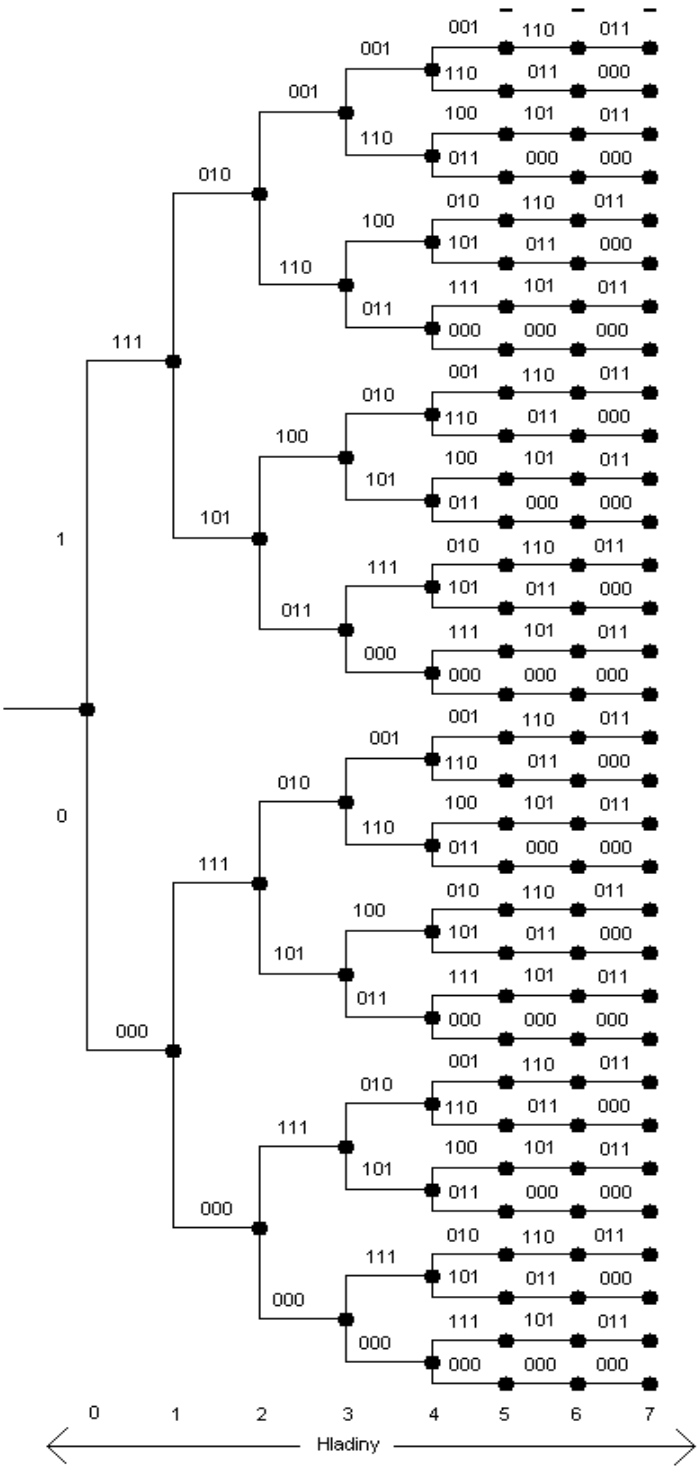
$$M(r_i \mid v_i) = \log_2 \frac{P(r_i \mid v_i)}{P(r_i)} - R, \tag{2.1}$$

Kde $P(r_i, v_i)$ je kanálová pravdepodobnosť prechodu, $P(r_i)$ je symbol pravdepodobnosti výstupu kanálu, a *R* je kódový pomer [5]. Čiastočná metrika cesty pre prvých *l* vetiev cesty *v* je daná:

$$M([\mathbf{r} \mid \mathbf{v}]_{l-1}) = \sum_{j=0}^{l-1} M(\mathbf{r}_j \mid \mathbf{v}_j) = \sum_{i=0}^{nl-1} M(r_i \mid v_i), \tag{2.2}$$

kde $M(r_j / v_j)$ metrika vetvy pre *j*-tu vetvu je počítaná pridaním bitovej metriky pre *n* bitov na tejto vetve. Skombinovaním 2.1 a 2.2 dostaneme:

$$M([\mathbf{r} \mid \mathbf{v}]_{l-1}) = \sum_{i=0}^{nl-1} \log_2 P(r_i \mid v_i) - \sum_{i=0}^{nl-1} \log_2 P(r_i) - nlR. \tag{2.3}$$



Obrázok 2.1: Kódový strom pre (3, 1, 2)kód s L = 5

dvojkový vstup a Q-ary výstup hovorí, že je symetrický ak:

$$P(j\,|\,0) = P(Q-1-j\,|\,1), \qquad j = 0, 1, \ldots, Q-1. \tag{2.4}$$

Pre symetrický kanál s rovnako pravdepodobnými vstupnými symbolmi, symboly výstupu kanálu musia uspokojiť $P(r_i = j) = P(r_i = Q - 1 - j) \leq ½$ pre $0 \leq j \leq Q - 1$ a všetky i.A (2.3) sa zredukuje na:

$$M([\mathbf{r}\,|\,\mathbf{v}]_{l-1}) = \sum_{i=0}^{nl-1} \log_2 P(r_i\,|\,v_i) - \sum_{i=0}^{nl-1} [\log_2 P(r_i) + R]$$
$$= \sum_{i=0}^{nl-1} \log_2 P(r_i\,|\,v_i) + \sum_{i=0}^{nl-1} \left[\log_2 \frac{1}{P(r_i)} - R\right]. \qquad (2.5)$$

Prvý názov v (2.5) je metrika pre Viterbiho algoritmus. Druhý termín predstavuje pozitívne (od $1/P(ri) \geq 2$ *a* $R \leq 1$) dispozície ktoré sa lineárne zvyšujú s optickou dĺžkou. Z tohoto dôvodu, dlhšie cesty majú väčšie dispozície než kratšie cesty, odráža skutočnosť, že sú bližšie ku koncu stromu a preto pravdepodobnejšie budú časť maximálnej cesty pravdepodobnosti. Bitová metrika bola poprvý krát predstavená Fanom na intuitívnych základoch a preto sa nazýva Fanova metrika. Je to metrika najčastejšie používaná pre sekvenčné dekódovanie, aj keď boli navrhované aj niake iné metriky. Keď sa porovnávajú cesty o rôznych dĺžkach, Cesta s najväčšou Fanovou metrikou je považovaná za najlepšiu cestu (t.j. Najpravdepodobnejšie to je časť maximálnej cesty pravdepodobnosti).

Príklad 2.2

Pre BSC (Q = 2) s pravdepodobnosťou prenosu *p*, Fanova metrika pre skrátené kódové slová $[v_5]$ a $[v_1]$ uvedené v príklade 2.1 je daná:

$$M([\mathbf{r}\,|\,\mathbf{v}]_5) = 16 \log_2 (1 - p) + 2 \log_2 p + 18(1 - 1/3)$$
$$= 16 \log_2 (1 - p) + 2 \log_2 p + 12$$

a

$$M([\mathbf{r}\,|\,\mathbf{v}']_0) = 2 \log_2 (1 - p) + \log_2 p + 3(1 - 1/3)$$
$$= 2 \log_2 (1 - p) + \log_2 p + 2.$$

pre P = 0,10

$$M([\mathbf{r}\,|\,\mathbf{v}]_5) = 2.92 > M([\mathbf{r}\,|\,\mathbf{v}']_0) = -1.63,$$

Zobrazujúca, že $[v_5]$ je lepšia z dvoch ciest. Toto sa líši od výsledkov získaných používaním metriky Viterbiho algoritmu, od termínu dispozícia vo Fanovej metrike odráža rozdiel medzi dĺžkami ciest.

Obecne, pre BSC s pravdepodobnosťou prenosu *p*, bitová metrika je:

$$M(r_i|v_i) = \begin{cases} \log_2 2p - R & \text{if } r_i \neq v_i \\ \log_2 2(1-p) - R & \text{if } r_i = v_i. \end{cases} \tag{2.6}$$

Príklad 2.3

Ak $R = 1/3$ a $p = 0{,}10$

$$M(r_i|v_i) = \begin{cases} -2.65 & \text{if } r_i \neq v_i \\ 0.52 & \text{if } r_i = v_i, \end{cases} \tag{2.7}$$

a my máme metrickú tabulku zobrazenú v tabulke 2.1*a*. Je bežná prax, že stupnica metriky pozitívnych konštánt je stanovená tak, aby mohli byť bližšie aproximované ako celé čísla pre jednoduchosť implementácie. V tomto prípade, mierka z 1/0,52 výnosov je metrická tabulka ceých čísel zobrazená v tabulke 2.1*b*.

Tabulka 2.1: Metrické tabulky pre $R = 1/3$ kód a s BSC s $p = 0{,}10$.

$\mathbf{r_i}$	**0**	**1**
$\mathbf{v_i}$	-	-
0	0,52	- 2,65
1	- 2,65	0,52

a

$\mathbf{r_i}$	**0**	**1**
$\mathbf{v_i}$	-	-
0	1	-5
1	-5	1

b

V stack alebo ZJ algoritme, zoznam inštrukcií alebo vela predtým preskúmaných ciest rôznych dĺžok je držaných v sklade. Každý vstup skladiska obsahuje cestu spolu s jej metrikou, cesta s najväčšou metrikou je umiestnená na vrchu, a ostatné sú uvedené v zozname so znižujúcou sa metrikou. Každý krok dekódovania pozostáva z predlžovania vrchnej cesty v skladisku počítaním metriky vetiev z jeho 2^k nasledujúcich vetví a potom pridaním týchto k metrike vrchnej cesty k tvoreniu 2^k nových ciest, nazývaných *successors* (následníci) vrchnej cesty. Vrchná cesta je potom vymazaná zo skladiska, je vložených 2^k následníkov, A skladisko je preskupené podla poriadku so znižujúcou sa hodnotou metriky. Algoritmus končí, keď je najvyššia cesta v skladisku koniec stromu.

Stack algoritmus

Krok 1. Naložiť skladište s pôvodným uzlom stromu, ktorého metrika je vzatá nula.

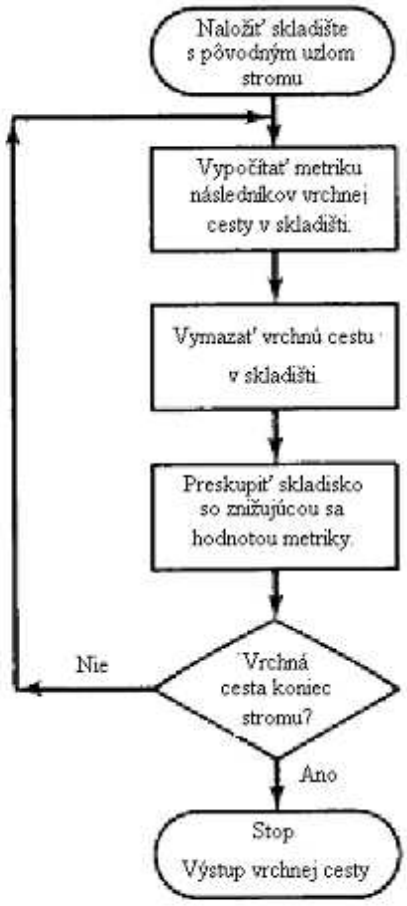
Krok 2. Vypočítať metriku následníkov vrchnej cesty v skladišti.

Krok 3. Vymazať vrchnú cestu v skladišti.

Krok 4. Vložiť nové cesty do skladišťa a preskupiť skladisko podla poriadku so znižujúcou sa hodnotou metriky.

Krok 5. Ak je vrchná cesty v skladisku v koncovom uzle stromu, koniec. Inak sa vrátime do bodu 2.

Keď algoritmus skončí, vrchná cesta v skladišti je vzatá ako dekódovaná cesta. Kompletný vývojový diagram pre stack algoritmus je ukázaný na obrázku 2.2.



Obrázok 2.2: Vývojový diagram pre stack algoritmus

V deliacej časti stromu je 2^k nových metrík vypočítaných v kroku 1. Na konci stromu je len jedna nová metrika vypočítaná. Všimnite si, že pre (n, l, m) kódy, sa velkosť skladiska zvyšuje o jedna pre každý dekódujúci krok v deliacej časti stromu, ale nezvýši sa vôbec, keď je dekodér na konci stromu. Od deliacej časti stromu je obvykle ovela dlhší ako ocas ($L >> m$), velkosť skladiska sa zhruba rovná počtu dekódujúcich krokov keď algoritmus končí.

Príklad 2.4

Zvažujeme použitie stack algoritmu na $R = 1/3$ kódový strom z obrázku 2.1. Predpokladajme, že kódové slovo je prenesené z tohoto kódu cez BSC s $p = 0{,}10$ a sekvencia

$\mathbf{r}$ = (0 1 0, 0 1 0, 0 0 1, 1 1 0, 1 0 0, 1 0 1, 0 1 1).

je doručená. Používaním celočíselnej metrickej tabuľky 2.1*b*, je obsah skladiska po každom kroku algoritmu ukázaný v tabuľke 2.2. Algoritmus sa ukončí po desiatich dekódujúcich krokoch a konečná dekódovaná cesta je:

$\tilde{\mathbf{v}}$ = (1 1 1, 0 1 0, 0 0 1, 1 1 0, 1 0 0, 1 0 1, 0 1 1).

zodpovedajúca informačnej sekvencii u = (1 1 1 0 1). V tomto príklade, väzby v metrických hodnotách boli rozhodnuté preložením najvyššej cesty navrch. Toto má efekt mierneho znižovania celkového počtu dekódujúcich krokov. Avšak riešenie väzieb je ľubovolné a nepôsobí na pravdepodobnosť chýb dekodéra.

To je zaujímavé pri porovnávaní počtu dekódujúcich krokov potrebných v stack algoritme s počtom požadovaným Viterbiho algoritmom. Dekódujúci krok alebo počítanie pre Viterbiho algoritmus je zrovnávanie a vyberanie operácie vykonanej pre každý stav v schéme laťkovej mreže za časovou jednotkou m. Z toho dôvodu by Viterbiho algoritmus požadoval 15 počítaní k tomu aby dekódoval prijatú sekvenciu v príklade 2.4(vidieť v schéme laťkovej mreže na obrázku.1.5). Aj keď dekódujúci krok alebo počítanie je v stack algoritme trošku zložitejšie, skladište musí byť preskupované po každom predĺžení cesty, stack algoritmus potrebuje len 10 krokov na dekódovanie

Tabuľka 2.2: Obsah skladiska v príklade 2.4

Krok 1	*Krok 2*	*Krok 3*	*Krok 4*	*Krok 5*
0(-3)	00(-6)	000(-9)	1(-9)	11(-6)
1(-9)	1(-9)	1(-9)	0001(-12)	0001(-12)
	01(-12)	01(-12)	01(-12)	01(-12)
		001(-15)	001(-15)	001(-15)
			0000(-18)	0000(-18)
				10(-24)

Krok 6	*Krok 7*	*Krok 8*	*Krok 9*	*Krok 10*
111(-3)	1110(0)	11101(+3)	111010(+6)	1110100(+9)
0001(-12)	0001(-12)	0001(-12)	0001(-12)	0001(-12)
01(-12)	01(-12)	01(-12)	01(-12)	01(-12)
001(-15)	001(-15)	11100(-15)	11100(-15)	11100(-15)
0000(-18)	1111(-18)	001(-15)	001(-15)	001(-15)
110	0000(-18)	1111(-18)	1111(-18)	1111(-18)
10(-24)	110(-21)	0000(-18)	0000(-18)	0000(-18)
	10(-24)	110(-21)	110(-21)	110(-21)
		10(-24)	10(-24)	10(-24)

prijatej sekvencie v príklade 2.4. Táto výpočtová výhoda sekvenčného dekódovania cez Viterbiho algoritmus je typická, keď doručená sekvencia nieje príliš hlučná, to je, keď to obsahuje zlomok chýb nie príliš väčší než je kanálová pravdepodobnosť priechodu p. Všimnime si, že $\tilde{v}$ v príklade 2.4 nesúhlasí s r v len dvoch pozíciách a súhlasí v ďalších 19 pozíciách. Dajme tomu že $\tilde{v}$ bolo skutočne prenesené, zlomok chýb v r je 2/21 = 0,095 zhruba rovný kanálovej pravdepodobnosti p = 0,10 v tomto prípade.

Situácia je značne rôzna keď prijatá sekvencia je velmi hlučná. Toto je ilustrované v nasledujúcom príklade:

Príklad 2.5

Pre rovnaký kód, kanál a metrickú tabulku ako v príklade 2.4, vezmime že sekvencia:

$\mathbf{r}$ = (1 1 0, 1 1 0, 1 1 0, 1 1 1, 0 1 0, 1 0 1, 1 0 1).

je doručená. Obsah skladiska po každom kroku algoritmu ja ukázaný v tabuľke 2.3. Algoritmus sa ukončí po 20 dekódujúcich krokoch a konečná dekódovaná cesta je:

$\tilde{\mathbf{v}}$ = (1 1 1, 0 1 0, 1 1 0, 0 1 1, 1 1 1, 1 0 1, 0 1 1).

Odpovedajúce informačnej sekvencii u = (1 1 0 0 1).

V tomto príklade sekvenčný dekodér vykoná 20 krokov, zatiaľčo Viterbiho algoritmus by znovu požadoval len 15 krokov. Toto upozorňuje na jeden z najdôležitejších rozdielov medzi sekvenčným dekódovaním a Viterbiho dekódovaním, že množstvo počítania v podaní sekvenčného dekodéru je náhodná hodnota, zatiaľčo výpočtové množstvo Viterbiho algoritmu je pevné. Veľmi hlučné prijaté sekvencie požadujú veľké množstvo počítania so sekvenčným dekodérom, niekedy viac ako ako pevné množstvo počítaní požadovaných Viterbiho algoritmom, ako je vo vyššie uvedenom príklade. Avšak odkedy sa veľmi hlučné doručené sekvencie nevyskytujú príliš často, priemerné množstvo počítania v sekvenčnom dekodéri je normálne omnoho menšie ako pevné číslo v podaní Viterbiho algoritmu.

Prijatá sekvencia v príklade 2.5 bola tiež dekódovaná Viterbiho algoritmom. Konečná dekódovaná cesta bola rovnaká v oboch prípadoch. Toto naznačuje dôležitý fakt, že sekvenčný dekodér skoro vždy produkuje maximálnu cestu pravdepodobnosti, dokonca aj keď je prijatá sekvencia veľmi hlučná. Avšak, pre daný kód, chybová pravdepodobnosť sekvenčného dekódovania je v podstate rovnaká ako pri Viterbiho dekódovaní.

Tabuľka 2.3: Obsah skladiska v príklade 2.5

Krok 1	*Krok 2*	*Krok 3*	*Krok 4*	*Krok 5*	*Krok 6*	*Krok 7*
1(-3)	11(-6)	110(-3)	1100(-6)	11000(-9)	0(-9)	1101(-12)
0(-9)	0(-9)	0(-9)	0(-9)	0(-9)	1101(-12)	01(-12)
	10(-12)	10(-12)	1101(-12)	1101(-12)	10(-12)	10(-12)
		111(-21)	10(-12)	10(-12)	11001(-15)	11001(-15)
			111(-21)	11001(-15)	110000(-18)	110000(-18)
				111(-21)	111(-21)	00(-18)
						111(-21)

Krok 8	*Krok 9*	*Krok 10*	*Krok 11*	*Krok 12*	*Krok 13*	*Krok 14*
11011(-9)	01(-12)	10(-12)	11001(-15)	110010(-12)	101 (-15)	011(-15)
01(-12)	10(-12)	11001(-15)	101(-15)	101(-15)	011(-15)	110110(-18)
10(-12)	11001(-15)	011(-15)	011(-15)	011(-15)	110110(-18)	110000(-18)
11001(-15)	110110(-18)	110110(-18)	110110(-18)	110110(-18)	110000(-18)	1010(-18)
110000(-18)	110000(-18)	110000(-18)	110000(-18)	110000(-18)	00(-18)	00(-18)
00(-18)	00(-18)	00(-18)	00 (-18)	00 (-18)	1100100	110100
111(-21)	111(-21)	010(-21)	100(-21)	100(-21)	100(-21)	100(-21)
11010(-27)	11010(-27)	111(-21)	010(-21)	010(-21)	010(-21)	010(-21)
		11010(-27)	111(-21)	111(-21)	111(-21)	111(-21)
			11010(-27)	11010(-27)	11010(-27)	1011(-24)
						11010(-27)

Krok 15	*Krok 16*	*Krok 17*	*Krok 18*	*Krok 19*	*Krok 20*
110110(-18)					1100100(-21)
110000(-18)	110000(-18)	0110 (-18)	1010(-18)	00(-18)	10100(-21)
	0110(-18)	1010(18)	00(18)	1100100(21)	
0110(-18)					01100(-21)
	1010 (-18)	00(-18)	1100100(-21)	10100(-21)	001(21)
1010(-18)	00(18)	1100100(21)	01100(-21)	01100(-21)	
					100(-21)
00(-18)	1100100(-21)	100(-21)	100(-21)	100(-21)	010(21)
1100100(21)	100(-21)	010(21)	010(21)	010(-1)	
					111(-21)
100(-21)	010(-21)	111(-21)	111(-21)	111(-21)	0111(24)
010(21)	111(21)	0111(-24)	0111(24)	0111(24)	
					1011(-24)
111(-21)	0111(-24)	1011(-24)	1011(-24)	1011(-24)	1100000(-27)
0111(-24)	1011(24)	1100000(-27)	1100000(-27)	1100000(-27)	
					1101100(-27)
10111(-24)	1101100(-27)	1101100(-27)	1101100(-27)	1101100(-27)	10101(-27)
010(27)	11010(-27)	11010(-27)	01101(-27)	10101(-27)	
					01101(-27)
			11010(-27)	01101(-27)	11010(-27)
				11010(-27	
					000(-27)

Avšak, existuje niekoľko problémov pridružených s implementáciou stack algoritmu sekvenčného dekódovania, ktoré obmedzia jeho celkový výkon. Prvý, od kedy dekodér stopuje pomerne náhodnú cestu sem a tam cez kódový strom, skáče z uzlu na uzol, dekodér musí mať vstupný buffer na skladovanie prichádzajúcich blokov prijatej sekvencie kým čakajú na spracovanie. V závislosti na rýchlostnom faktore dekodéru, to je pomer rýchlosti akej sú vykonávané výpočty a rýchlosti vstupujúcich dát (vo vetvách prijatých za sekundu), dlhé hľadania môžu spôsobiť, že vstupný buffer pretečie, čo vy malo za následok stratu dát alebo vymazanie. Buffer prijíma dáta v pevnom pomere $1/nT$ vetví za sekundu, kde T je časový interval vymedzený pre každý prenesený bit. To vystupujú k dekodéru asynchrónne, ako je požadované algoritmom. Normálne je informácia rozdelená do rámcov z L vetví, každý je ukončený sekvenciou $m << L$ všetky nulové vstupy pre návrat kodéru do nulového stavu. Aj keď vstupný buffer je o jedno alebo dve miesta väčší ako L, je istá pravdepodobnosť, že sa naplní behom dekódovania daného rámu a to spôsobí že ďalšia vetva prijatá z kanálu bude nedekódovaná vetva zo spracovávaného rámca a bude vysunutá von z bufferu. Tieto bity sú potom stratené a my povieme, že sú to zmazané výsledky. Pravdepodobnosť zmazania 10^{-3} je bežná v sekvenčnom dekódovaní, kde to znamená, že špecifický rámec má pravdepodobnosť 10^{-3} z nedekódovaných kvôli pretečeniu vstupného bufferu.

Priemerné množstvo počítaní v podaní konvolučného dekodéra a taktiež jeho pravdepodobnosť vymazania sú v podstate nezávislé na pamäti kodéru K. Preto, kódy s veľkými hodnotami K, a z toho dôvodu veľkými voľnými vzdialenosťami, môžu byť vybrané pre dekódovanie v sekvenčnom dekodéri. Nezistené chyby (vyplnené nesprávnym dekódovaním) sú potom extrémne nepravdepodobné a významnejšie ohraničenie kódového výkonu je pravdepodobnosť vymazania kvôli pretečeniu bufferu.

I keď pravdepodobnosť vymazania 10^{-3} môže byť dosť obtiažna v niektorých systémoch, môže byť v skutku prospešná v iných. Vymazania sa obvykle vyskytujú, keď doručená sekvencia je veľmi hlučná, ak dekódovanie bolo dokončené a dokonca, keď bola získaná maximálna cesta pravdepodobnosti, je dosť vysoká pravdepodobnosť, že tento odhad bude nesprávny. Vo väčšine systémoch by bolo prijateľnejšie vymazať takýto rámec, ako dekódovať ho nesprávne. Inými slovami, kompletný dekodér, ako Viterbiho algoritmus, by vždy dekódoval takýto rámec, aj keď asi bude dekódovaný nesprávne, zatiaľ čo sekvenčný dekodér by triedil pre vymazanie zmyslom hlučné rámce a proste ich vymazal. Táto vlastnosť sekvenčného dekódovania môže byť použitá v ARQ schéme opätovného prenosu ako indikátor, kedy rámec by mal byť opätovne prenesený. Druhý problém v praktickej implementácii stack algoritmu je, že veľkosť skladiska musí byť konečná. Inými slovami, je tam vždy nejaká pravdepodobnosť, že sa skladisko naplní predtým, ako je dekódovanie dokončené (alebo pretečie buffer) na danom ráme. Najbežnejší spôsob riešenia tohoto problému je jednoducho dovoliť ceste na spodku skladiska opustiť skladisko v ďalšom

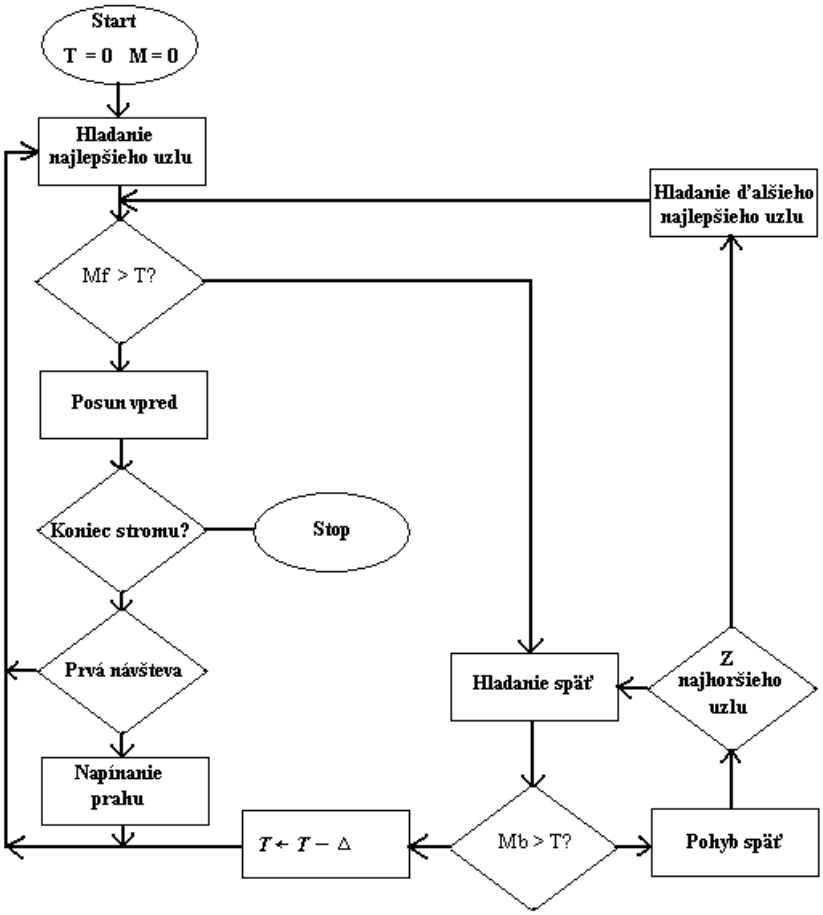
dekódujúcom kroku. Táto cesta je potom stratená a nemôže sa nikdy vrátiť do skladiska. Pre typické veľkosti skladiska ako je 1000 položiek, pravdepodobnosť že cesta na spodku skladiska sa môže obnoviť a dosiahnuť vrcholu skladiska tak malá, že strata výkonu je zanedbateľná.

Súvisiaci problém je, čo robiť s preskupením skladiska po každom dekódujúcom kroku. Toto sa môže stať celkom zdĺhavé jak počet položiek v skladišti narastá a uplatňujú sa rôzne ohraničenia dekódujúcej rýchlosti, ktoré môžu byť dosiahnuté základným algoritmom. Jelínek navrhoval upravený algoritmus, nazývaný stack-bucket algoritmus, v ktorom obsah skladiska nemusí byť preskupovaný po každom dekódovacom kroku. V stack-bucket algoritme rozsah možných metrických hodnôt (od +21 do +110 v predchádzajúcich príkladoch) je kvantovaný do pevného množstva intervalov. Každému metrickému intervalu je priradený určitý počet lokalít na sklade, nazvaných vedro. Keď je cesta predĺžená, je to vymazané zo skladu a každá nová cesta je vložená ako vrchný vstup do vedra, ktoré zahrnuje jeho metrickú hodnotu. Žiadne preskupenie ciest vo vnútri vedier sa nepožaduje. Vrchná cesta vo vrchnom neprázdnom vedre je vybraná na predĺženie. Počítanie teraz zahrňuje len nájdenie správneho vedra, v ktorom umiestniť každú novú cestu, ktorá je nezávislá na počte predtým predĺžených ciest, a je preto rýchlejšie než preskupovanie stále väčšieho a väčšieho skladiska. Nevýhoda tohoto algoritmu je, že to nieje vždy najlepšia cesta, ktorá je predĺžená ale len veľmi dobrá cesta (t.j. Cesta na vrchole neprázdneho vedra alebo najlepšie vedro). Typicky, ak kvantovanie vedra nieje príliš veľké a prijatá sekvencia nieje príliš hlučná, najlepšie vedro obsahuje len najlepšiu cestu, a klesnutie výkonu zo základného algoritmu je veľmi nepatrné. Rýchlostné úspory algoritmu vedra sú značné, i keď, veškeré praktické implementácie stack algoritmu používali tento prístup.

2.2 Fanov algoritmus

Iný prístup k sekvenčnému dekódovaniu, Fanov algoritmus, obetuje nejakú rýchlosť v porovnaní s algoritmom skladisko, ale v podstate nevyžaduje žiadny sklad. Rýchlostná nevýhoda Fanovho algoritmu je daná tým, že v skutočnosti rozširuje väčšie uzly než algoritmus skladisko, aj keď tam nieje žiadne skladisko na uskutočnenie preskupenia. Vo Fanovom algoritme dekodér skúma sekvencie uzlov v strome začínajúci pôvodným uzlom a končiaci jedným s koncových uzlov. Dekodér nikdy neskáče z uzlu na uzol ako v algoritme skladisko, ale vždy pokračuje k susednému uzlu, buď vpred k jednému z 2^k uzlov opúšťajúcich prítomný uzol, alebo spätný k uzlu vediacemu do prítomného uzlu. Metrika ďalšieho preskúmaného uzlu potom môže byť vypočítaná pripočítaním (alebo odpočítaním) metriky spojovacej vetvy k metrike prítomného uzlu. Toto odstraňuje potrebu uskladnenia metriky predtým skúmaných uzlov ako požaduje algoritmus skladisko. Každopádne niektoré uzly sú navštívené viac ako raz a v takomto prípade ich metrické hodnoty musia byť prepočítané. Dekodér bude pokračovať cez strom tak dlho, pokiaľ sa hodnota metriky doposiaľ vypočítanej cesty bude zvyšovať. Pokiaľ hodnota metriky spadne pod prahovú hodnotu, dekodér sa vráti na začiatok a skúma iné cesty. Ak sa nenájde žiadna cesta, ktorej hodnota metriky zostane nad prahom, prah je znížený a dekodér sa pokúsi k tomu ,aby sa pohol dopredu s nižším prahom. Vždy, keď je daný uzol navštívený v smere vpred, prah je menší než na predchádzajúcej návšteve tohoto uzlu. Toto predchádza tzv. retiazkovaniu v algoritme a dekodér bude musieť nakoniec dosiahnuť koniec stromu, kde algoritmus končí.

Kompletný postupový diagram Fanovho algoritmu je ukázaný na obrázku 2.3. Dekodér začína v pôvodnom uzle s prahom $\boldsymbol{T} = 0$ a metrickou hodnotou $\boldsymbol{M} = 0$. To potom pozerá vpred na



Obrázok 2.3: Postupový diagram Fanovho algoritmu

jeden z 2^k nasledujúcich uzlov (t.j. Jeden s najväčšou metrikou). Pokiaľ M_F je metrika predchádzajúceho preskúmaného uzlu a ak $M_F \geq T$, dekodér sa presunie do tohoto uzlu. Po skontrolovaní, či bol dosiahnutý koniec stromu, je vykonané "napínanie prahu" ak je tento uzol preskúmaný po prvý krát. Toto zahŕňa zvýšenie T s najväčším možným násobením prahového zvýšenia Δ tak, že nová prahová hodnota presahuje aktuálnu metriku. Pokiaľ tento uzol už bol preskúmaný predtým, žiadne napínanie prahu sa nevykonáva. Potom dekodér znovu postupuje vpred do najlepšieho nasledujúceho uzlu. Ak $M_F < T$, dekodér sa vracia späť do predchádzajúceho uzlu. Pokiaľ M_B je preskúmaná metrika spätného uzlu, a $M_B < T$, potom T je znížené o Δ a krok hľadania najlepšieho uzlu sa opakuje. Ak $M_B \geq T$ dešifrant sa vracia späť do predchádzajúceho uzlu. Volajme tento uzol P. Pokiaľ tento spätný pohyb bol najhorším z 2^k uzlov nasledujúcich za uzlom P, dekodér sa znovu pozerá na uzol predchádzajúci pred uzlom P. Pokiaľ nie, dekodér hľadá ďalší najlepší z 2^k uzlov nasledujúcich po uzle P a kontroluje či $M_F \geq T$. Pokiaľ dekodér niekedy pozerá späť z pôvodného uzlu, vezmeme, že hodnota metriky pôvodného uzlu je $-\infty$, takže prah je vždy znížený o Δ v tomto prípade. Väzby v metrických hodnotách môžu byť vyhodnotené

ľubovolne bez ovplyvnenia priemerného výkonu dekodéra.

Teraz zopakujeme príklad spracovaný skôr pre algoritmus skladisko, tentokrát použitím Fanovho algoritmu.

Príklad 2.6:

Pre rovnakú prijatú sekvenciu dekódovanú algoritmom skladisko v príklade 2.4, sú kroky Fanovho algoritmu zobrazené v tabuľke 2.4 pre hodnotu $\mathbf{\Delta} = 1$. Algoritmus sa ukončí po 40 dekódujúcich krokoch a konečná dekódovaná cesta je rovnaká ako cesta dekódovaná algoritmom skladisko. V tabuľke 2.4, *LFB* znamená hľadanie najlepšieho uzla, *LFNB* znamená hľadanie ďalšieho najlepšieho uzla a *X* znamená pôvodný uzol.

Počítanie vo Fanovom algoritme je obvykle urobené vždy, keď bol vykonaný krok hľadania. Z toho dôvodu pre tento príklad, Fanov algoritmus vyžaduje 40 počítaní, v porovnaní s len 10 pre algoritmus skladisko. Treba si všimnúť, že niektoré uzly sú navštívené niekoľkokrát. V skutočnosti, pôvodný uzol je navštívený 8 krát, cestovný 0 uzol je navštívený 11 krát, cestovný 00 uzol je navštívený 5 krát a uzly predstavujúce cestu 000, 1, 11, 111, 1110, 11101, 111010 a 1110100 každý jedenkrát, pri celkovo 32 navštívených uzloch. Toto je menej ako 40 počítaní, pretože nie každý pohľad vpred má za následok pohyb k inému uzlu ale niekedy len k prahu znižovania.

Množstvo počítania v podaní Fanovho algoritmu závisí ako je prahové zvýšenie $\mathbf{\Delta}$ nastavené. Obecne, ak $\mathbf{\Delta}$ je príliš malé, vyplýva z toho veľké množstvo počítania, ako v príklade 2.6. Zvolením väčšieho $\mathbf{\Delta}$ zredukujeme počet počítaní.

Príklad 2.7:

Opakujeme Fanov algoritmus pre rovnakú prijatú sekvenciu ako v príklade 2.6 a pre hodnotu $\mathbf{\Delta} = 3$. Výsledky sú zobrazené v tabuľke 2.5. Je potrebných 22 počítaní, navštívených bolo 20 uzlov a je dekódovaná rovnaká cesta. Takže, v tomto prípade zvýšením $\mathbf{\Delta}$ na 3, zredukovali sme počet počítaní takmer na polovicu.

Avšak, zvyšovanie prahu $\mathbf{\Delta}$ nemôže pokračovať donekonečna, bez ovplyvnenia pravdepodobnosti chýb. Aby algoritmus našiel maximálnu cestu pravdepodobnosti, $\boldsymbol{T}$ musí byť v určitých bodoch znížené pod minimálnu metriku pozdĺž maximálnej cesty pravdepodobnosti.

Tabuľka 2.4: Kroky dekódovania pre Fanov algoritmus s $\Delta = 1$

KROK	HLADANIE	M_F	M_B	UZOL	METRIKA	*T*
0				X	0	0
1	LFB	- 3	- ∞	X	0	- 1
2	LFB	- 3	- ∞	X	0	- 2
3	LFB	- 3	- ∞	X	0	- 3
4	LFB	- 3		0	- 3	- 3
5	LFB	- 6	0	X	0	- 3
6	LFNB	- 9	- ∞	X	0	- 4
7	LFB	- 3		0	- 3	- 4
8	LFB	- 6	0	X	0	- 4
9	LFNB	- 9	- ∞	X	0	- 5
10	LFB	- 3		0	- 3	- 5
11	LFB	- 6	0	X	0	- 5
12	LFNB	- 9	- ∞	X	0	- 6
13	LFB	- 3		0	- 3	- 6
14	LFB	- 6		00	- 6	- 6
15	LFB	- 9	- 3	0	- 3	- 6
16	LFNB	- 12	0	X	0	- 6
17	LFNB	- 9	- ∞	X	0	- 7
18	LFB	- 3		0	- 3	- 7
19	LFB	- 6		00	- 6	- 7
20	LFB	- 9	- 3	0	- 3	- 7
21	LFNB	- 12	0	X	0	- 7
22	LFNB	- 9	- ∞	X	0	- 8
23	LFB	- 3		0	- 3	- 8
24	LFB	- 6		00	- 6	- 8
25	LFB	- 9	- 3	0	- 3	- 8
26	LFNB	- 12	0	X	0	- 8
27	LFNB	- 9	- ∞	X	0	- 9
28	LFB	- 3		0	- 3	- 9
29	LFB	- 6		00	- 6	- 9
30	LFB	- 9		000	- 9	- 9
31	LFB	- 12	- 6	00	- 6	- 9
32	LFNB	- 15	- 3	0	- 3	- 9
33	LFNB	- 12	0	X	0	- 9
34	LFNB	- 9		1	- 9	- 9
35	LFB	- 6		11	- 6	- 6
36	LFB	- 3		111	- 3	- 3
37	LFB	0		1110	0	0
38	LFB	+ 3		11101	+ 3	+ 3
39	LFB	+ 6		111010	+ 6	+ 6
40	LFB	+9		1110100	+9	Stop

Ak **Δ** je príliš veľké, keď je znížené pod minimálnu metriku maximálnej cesty pravdepodobnosti, môže byť znížené pod minimálnu metriku niekoľkých ďalších ciest, to spôsobuje možnosť ktorejkoľvek z týchto ciest byť dekódovanou pred maximálnou cestou pravdepodobnosti. Nastavenie **Δ** príliš veľké môže tiež spôsobiť, že sa množstvo počítania zvýši opäť a viacej zlých ciest môže nasledovať po strome pokiaľ ***T*** je znížené príliš. Skúsenosť ukázala, že ak sa používa metrika neopatrená merítkom, **Δ** by mala byť vybraná medzi 2 a 8. Pokiaľ je metrika opatrená merítkom, **Δ** by mala byť nastavená rovnaký spôsobom. V príklade 2.7 bola mierka 1/0,52 ,ukazujúca, že **Δ** by mala byť vybraná medzi 3.85 a 15,38. A voľba **Δ** medzi 6 a 10 by bola dobrý kompromis v tomto prípade.

Tabuľka 2.5: Kroky dekódovania pre Fanov algoritmus s $\Delta = 3$

KROK	HLADANIE	M_F	M_B	UZOL	METRIKA	*T*
0				X	0	0
1	LFB	- 3	- ∞	X	0	- 3
2	LFB	- 3		0	- 3	- 3
3	LFB	- 6	0	X	0	- 3
4	LFNB	- 9	- ∞	X	0	- 6
5	LFB	- 3		0	- 3	- 6
6	LFB	- 6		00	- 6	- 6
7	LFB	- 9	- 3	0	- 3	- 6
8	LFNB	- 12	0	X	0	- 6
9	LFNB	- 9	- ∞	X	0	- 9
10	LFB	- 3		0	- 3	- 9
11	LFB	- 6		00	- 6	- 9
12	LFB	- 9		000	- 9	- 9
13	LFB	- 12	- 6	00	- 6	- 9
14	LFNB	- 15	- 3	0	- 3	- 9
15	LFNB	- 12	0	X	0	- 9
16	LFNB	- 9		1	- 9	- 9
17	LFB	- 6		11	- 6	- 6
18	LFB	- 3		111	- 3	- 3
19	LFB	0		1110	0	0
20	LFB	+ 3		11101	+ 3	+ 3
21	LFB	+ 6		111010	+ 6	+ 6
22	LFB	+9		1110100	+9	Stop

V príkladoch použitia Fanovho algoritmu, bola v oboch prípadoch dekódovaná rovnaká cesta ako v algoritme skladisko. Fanov algoritmus skoro vždy nachádza rovnakú cestu ako stack – bucket algoritmus, pokiaľ Δ je vybraná z rovnakého intervalu ako je interval kvantovanie metrických hodnôt v skladisku. Fanov algoritmus obecne dokóduje rýchlejšie ako stack – bucket algoritmus pre mierne rozsahy[3]. Toto je pretože Fanov algoritmus nieje spomalený kontrolnými problémami v skladisku ako stack – bucket algoritmus. Pre vyššie sadzby stack – bucket algoritmus je trošku rýchlejší kvôli dodatočnému výpočtovému množstvu Fanovho algoritmu. Pretože nevyžaduje žiadny sklad, je Fanov algoritmus obvykle vybraný v praktických implementáciách sekvenčného dekódovania.

3 Dekódovanie konvolučných kódov s väčšinovou logikou

Do dekódovania konvolučných kódov môže byť vzatý aj algebraický prístup. Väčšinová logika alebo prah dekódovania bol ukázaný Masseyom v roku 1963 ako použiteľný pre konvolučné kódy. To sa líši od Viterbiho dekódovania a sekvenčného dekódovania v konečnom rozhodnutí urobenom na danom informačnom bloku založenom len na jednej nútenej dĺžke prijatých blokov skôr ako na celej prijatej sekvencii. Toto má za následok nižší výkon v porovnaní s Viterbiho alebo sekvenčným dekódovaním, ale implementácia dekodéru je moc jednoduchá. Toto viedlo k používaniu väčšinovej logiky dekódovania v aplikáciách ako telefónia a VF rádio, kde je požadované mierne množstvo kódovacieho zisku v relatívne nízkej cene.

3.1 Feedback dekódovanie

Začneme svoju diskusiu o väčšinovej logike dekódovania uvažovaním $R = ½$ systematický kód s generátorom sekvencií $\mathbf{g}^{(1)} = (1\ 0\ 0\ \ldots)$ a $\mathbf{g}^{(2)} = (\mathbf{g}_1^{(2)}\ \mathbf{g}_2^{(2)}\ \mathbf{g}_3^{(2)})$ a generátorom matice:

$$\mathbf{G} = \begin{bmatrix} 1 & g_0^{(2)} & 0 & g_1^{(2)} & 0 & g_2^{(2)} & \cdots & 0 & g_m^{(2)} & & & & \\ & & 1 & g_0^{(2)} & 0 & g_1^{(2)} & \cdots & 0 & g_{m-1}^{(2)} & 0 & g_m^{(2)} & & \\ & & & & 1 & g_0^{(2)} & \cdots & 0 & g_{m-2}^{(2)} & 0 & g_{m-1}^{(2)} & 0 & g_m^{(2)} \\ & & & & & \cdot & & & & & & & \cdot \\ & & & & & & \cdot & & & & & \cdot & \\ & & & & & & & \cdot & & & & & \end{bmatrix}. \quad (3.1)$$

Pre informačnú sekvenciu *u*, kódujúce rovnosti dostanú:

$$\mathbf{v}^{(1)} = \mathbf{u}*\mathbf{g}^{(1)} = \mathbf{u}$$

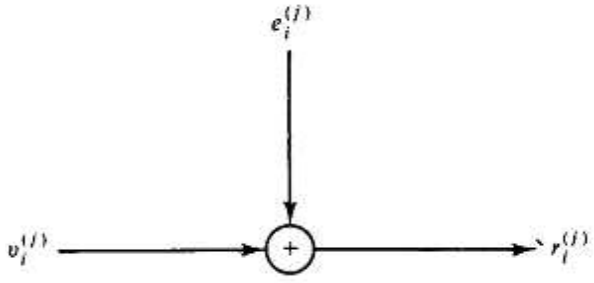
$$\mathbf{v}^{(2)} = \mathbf{u}*\mathbf{g}^{(2)}\ , \quad (3.2)$$

a prenesené kódové slovo je $v = uG$. Pokiaľ je v poslané cez BSC, binárna doručená sekvencia r môže byť zapísaná ako:

$$\mathbf{r} = (\mathbf{r}_0^{(1)}\ \mathbf{r}_0^{(2)})\ ,\ \mathbf{r}_1^{(1)}\ \mathbf{r}_1^{(2)},\ \mathbf{r}_2^{(1)}\ \mathbf{r}_2^{(2)},\ ....) = \mathbf{v} + \mathbf{e} \quad (3.3)$$

kde binárna sekvencia $e = (e_0^{(1)}\ e_0^{(2)})\ ,\ e_1^{(1)}\ e_1^{(2)},\ e_2^{(1)}\ e_2^{(2)},\ ....)$ je nazývaná kanálovou chybovou sekvenciou[1]. Chybný bit $e_i^{(J)} = 1$ práve vtedy keď $r_i^{(J)} \neq v_i^{(J)}$ (t.j. Chyba nastala behom prenosu). Tento model pre BSC je na obrázku 3.1. Prijatá sekvencia môže byť rozdelená na *prijatá sekvencia informácií*

$$\mathbf{r}^{(1)} = (r_0^{(1)}, r_1^{(1)}, r_2^{(1)}, \ldots) = \mathbf{v}^{(1)} + \mathbf{e}^{(1)} = \mathbf{u} + \mathbf{e}^{(1)} \quad (3.4a)$$



Obrázok 3.1: Zjednodušený model pre BSC

prijatá sekvencia rovnosti

$$\mathbf{r}^{(2)} = (\, r_0^{(2)}, r_1^{(2)}, r_2^{(2)}, \ldots) = \mathbf{v}^{(2)} + \mathbf{e}^{(2)} = \mathbf{u}*\mathbf{g}^{(2)} + \mathbf{e}^{(2)} \qquad (3.4b)$$

kde $e^{(1)} = (\, e_0^{(1)}, e_1^{(1)}, e_2^{(1)}, \ldots.)$ je chybová sekvencia informácií a $e^{(2)} = (\, e_0^{(2)}), \, e_1^{(2)}, \, e_2^{(2)}, \ldots.)$ je chybová sekvencia rovnosti.

Sekvencia syndrómu $s = (s_0, s_1, s_2, \ldots)$ je definovaná ako

Kde matica kontroly parity je daná: $\mathbf{s} \triangleq \mathbf{r}\mathbf{H}^T,$ (3.5)

$$\mathbf{H} = \begin{bmatrix}
g_0^{(2)} & 1 & & & & & & & & & & \\
g_1^{(2)} & 0 & g_0^{(2)} & 1 & & & & & & & & \\
g_2^{(2)} & 0 & g_1^{(2)} & 0 & g_0^{(2)} & 1 & & & & & & \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & & & & \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & \cdot & & & & \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & & \cdot & & & \\
g_m^{(2)} & 0 & g_{m-1}^{(2)} & 0 & g_{m-2}^{(2)} & 0 & \cdots & g_0^{(2)} & 1 & & & \\
 & & g_m^{(2)} & 0 & g_{m-1}^{(2)} & 0 & \cdots & g_1^{(2)} & 0 & g_0^{(2)} & 1 & \\
 & & & & g_m^{(2)} & 0 & \cdots & g_2^{(2)} & 0 & g_1^{(2)} & 0 & g_0^{(2)} \\
 & & & & & \cdot & & & & & & \\
 & & & & & & \cdot & & & & & \\
 & & & & & & & \cdot & & & & \\
 & & & & & & & & \cdot & & & \\
 & & & & & & & & & \cdot & & \\
 & & & & & & & & & & \cdot &
\end{bmatrix}. \qquad (13.6)$$

Ako v prípade s blokovými kódmi, $GH^T = 0$, a v je kódové slovo vtedy a len vtedy ak $vH^T = 0$. Každopádne, nie ako blokové kódy, G a H sú nekonečné jedným smerom matice, čo odráža skutočnosť, že informačná sekvencia a kódové slovo sú ľubovolnej dĺžky. Od $r = v + e$, sekvencia syndrómu môže byť zapísaná ako:

$$\mathbf{s} = (\mathbf{v} + \mathbf{e})\mathbf{H}^T = \mathbf{v}\mathbf{H}^T + \mathbf{e}\mathbf{H}^T = \mathbf{e}\mathbf{H}^t, \qquad (3.7)$$

Tu vidíme, že *s* závisí len na chybovej sekvencii kanálu a nie na prenesenom špecifickom kódovom slove. Pre dekódovanie sa zamýšľa, nech známe *s* je ekvivalentné známemu *r,* a preto dekodér môže byť navrhnutý k tomu, aby pracoval radšej s *s* ako s *r.* Taký dekodér sa potom nazýva dekodér syndrómu. Použitím polynomického zápisu my môžeme písať:

$$\mathbf{v}^{(1)}(\mathbf{D}) = \mathbf{u}(\mathbf{D})\mathbf{g}^{(1)}(\mathbf{D}) = \mathbf{u}(\mathbf{D}) \qquad (3.8a)$$

$$\mathbf{v}^{(2)}(\mathbf{D}) = \mathbf{u}(\mathbf{D})\mathbf{g}^{(2)}(\mathbf{D}) \qquad (3.8b)$$

$$\mathbf{r}^{(1)}(\mathbf{D}) = \mathbf{v}^{(1)} + \mathbf{e}^{(1)}(\mathbf{D}) - \mathbf{u}(\mathbf{D}) + \mathbf{e}^{(1)}(\mathbf{D}) \qquad (3.9a)$$

$$\mathbf{r}^{(2)}(\mathbf{D}) = \mathbf{v}^{(2)} + \mathbf{e}^{(2)}(\mathbf{D}) = \mathbf{u}(\mathbf{D})\mathbf{g}^{(2)}(\mathbf{D}) + \mathbf{e}^{(2)}(\mathbf{D}) \qquad (3.9b)$$

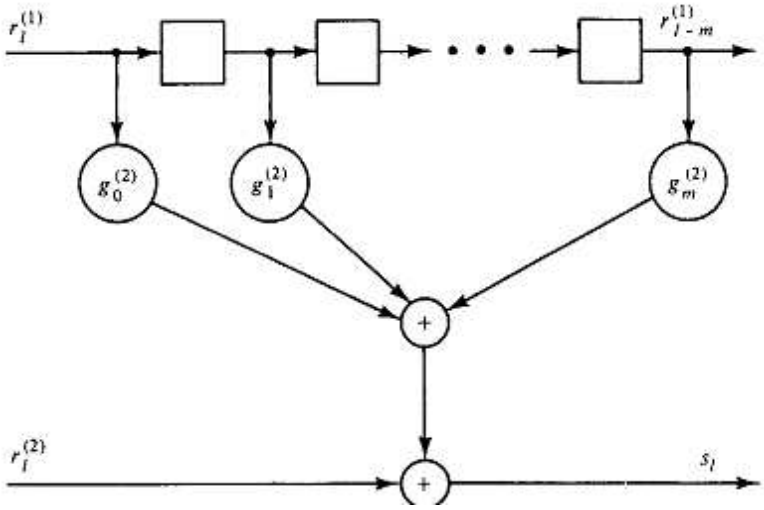
Prijímač, sekvencia syndrómu je vytvorená ako

$$\mathbf{s(D) = r^{(1)}(D)g^{(2)}(D) + r^{(2)}(D)} \qquad (3.10)$$

Od $v^{(2)}(D) = u(D)g^{(2)}(D) = v^{(1)}(D)g^{(2)}(D)$ Je formovanie syndrómu rovnocenné zakódovaniu $r^{(1)}(D)$ a potom pridaniu ho k $r^{(2)}(D)$. Blokový diagram formovača syndrómu pre $R = ½$ systematický kód je ukázaná na obrázku 3.2. Používanie 3.9 vynesené 3.10

$$\mathbf{s(D) = [u(D) + e^{(1)}(D)]g^{(2)}(D) + u(D)g^{(2)}(D) + e^{(2)}(D) = e^{(1)}(D)g^{(2)}(D) + e^{(2)}(D),} \qquad (3.11)$$

A znovu vidíme, že *s(D)* závisí len na kanálovej chybovej sekvencii a nie na prenesenom kódovom slove. Väčšinové dekódovacie logiky konvolučných kódov sú založené na koncepcii pravouhlej kontroly parity časti. Z 3.7 a 3.11 vidíme, že niaky bit syndrómu, alebo niaka suma bitov syndrómu, predstavuje známu sumu kanálových chybových bitov, a je nazvaná ako suma kontroly parít. Pokiaľ prijatá sekvencia je kódové slovo, kanálová chybová sekvencia je tiež kódové slovo a všetky syndrómové bity, a preto všetky kontrolné súčty musia byť nula. Každopádne, pokiaľ prijatá sekvencia nieje kódové slovo, niektoré kontrolné súčty nebudú nula. Bitová chyba e_j je kontrolovaná kontrolným súčtom ak e_j je obsiahnutá v súčte. Sada J kontrolných súčtov je ortogonálna na e_j pokiaľ každý kontrolný súčet kontroluje e_j , ale žiadny iný kontrolný bit nie je kontrolovaný viac než jedným kontrolným súčtom. Daná sada J pozostávajúca z ortogonálnych kontrolných súčtov na chybový bit e_j, pravidlo väčšinovej logiky dekódovania môže byť použité na odhadovanie hodnoty e.



Obrázok 3.2: Obvod formovania syndrómu pre $R = ½$ systematický kód

Pravidlo väčšinovej logiky dekódovania: Definujme $t_{ML} = J/2$. Vyberieme odhad $e_j = 1$ ak a len vtedy ak viac ako t_{ML} z J kontrolných súm ortogonálnych na e_j má hodnotu 1.

Teória 3.1: Ak chybové bity kontrolované J ortogonálnymi chybovými sumami obsahujú t_{ML} alebo menej kanálových chýb, pravidlo väčšinovej logiky dekódovania správne odhaduje e_j.

Dôkaz: Pokiaľ $e_j = 0$, najviac t_{ML} chybových bitov môže zapríčiniť, že najviac t_{ML} z J kontrolných súm má hodnotu 1. Z toho dôvodu, $e_j = 0$, keď je správne. Na druhú stranu, keď $e_j = 1$, najviac $t_{ML} - 1$ iných chýb môže zapríčiniť, že nie viac ako $t_{ML} - 1$ z J kontrolných súm má hodnotu 0, takže najmenej $t_{ML} + 1$ musí mať hodnotu 1, Z toho dôvodu, $e_j = 1$, ktorý je znovu správne.

V dôsledku teórie 3.1, t_{ML} je nazývané ako spôsobilosť opravy chýb väčšinovej logiky kódu.

Príklad 3.1:

Uvažujme nájdutie sady pozostávajúcej z ortogonálnych kontrolných súčtov na $e_0^{(1)}$, prvý informačný chybový bit, pre $R = ½$ systematický kód s $g^{(2)}(D) = 1 + D + D^4 + D^6$. Prvýkrát ukázané v 3.7 a 3.11 že $e_0^{(1)}$ môže ovplyvniť len syndróm bitov od s_0 do s_6 (t.j. Prvá nútená dĺžka syndrómu bitu). Ponechaním $[s]_6 = (s_0, s_1, \ldots, s_6)$ a požitím záznamu pre skrátenú sekvenciu a maticu:

$$[\mathbf{s}]_6 = [\mathbf{e}]_6[\mathbf{H}^T]_6$$

$$= [\mathbf{e}]_6 \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 1 & 1 & 0 & 0 & 1 & 0 \\ & 1 & 0 & 0 & 0 & 0 & 0 \\ & & 1 & 1 & 0 & 0 & 1 \\ & & 1 & 0 & 0 & 0 & 0 \\ & & & 1 & 1 & 0 & 0 \\ & & & 1 & 0 & 0 & 0 \\ & & & & 1 & 1 & 0 \\ & & & & 1 & 0 & 0 \\ & & & & & 1 & 1 \\ & & & & & 1 & 0 \\ & & & & & & 1 \\ & & & & & & 1 \end{bmatrix}. \qquad (3.12)$$

Spravenie transpozície z oboch strán má za následok

$$[\mathbf{s}]_6^T = \begin{bmatrix} 1 & 1 & & & & & & & & & & & & \\ 1 & 0 & 1 & 1 & & & & & & & & & & \\ 0 & 0 & 1 & 0 & 1 & 1 & & & & & & & & \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & & & & \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & & \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} [\mathbf{e}]_6^T. \quad (3.13)$$

Všimnime si, že párne stĺpce $\mathbf{H}^T$ (t.j. Tie stĺpce odpovedajúce chybovej sekvencii parity) tvoria maticu identity. Z toho dôvodu môžme 3.13 prepísať ako

$$[\mathbf{s}]_6^T = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} = \begin{bmatrix} 1 & & & & & & \\ 1 & 1 & & & & & \\ 0 & 1 & 1 & & & & \\ 0 & 0 & 1 & 1 & & & \\ 1 & 0 & 0 & 1 & 1 & & \\ 0 & 1 & 0 & 0 & 1 & 1 & \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} e_0^{(1)} \\ e_1^{(1)} \\ e_2^{(1)} \\ e_3^{(1)} \\ e_4^{(1)} \\ e_5^{(1)} \\ e_6^{(1)} \end{bmatrix} + \begin{bmatrix} e_0^{(2)} \\ e_1^{(2)} \\ e_2^{(2)} \\ e_3^{(2)} \\ e_4^{(2)} \\ e_5^{(2)} \\ e_6^{(2)} \end{bmatrix}. \quad (3.14)$$

Matica 3.14, ktorá násobí chybovú sekvenciu informácií je nazývaná trojuholník rovnosti kódu. Všimnime si, že prvý stĺpec trojuholníku rovnosti je sekvencia generátoru $g^{(2)}$, ktorá je posúvaná dolu o jedno a skrátená v každom nasledujúcom stĺpci.

Trojuholník rovnosti kódu môže byť použitý na výber sady ortogonálnych kontrolných súčtov na $e_0^{(1)}$. Ako prvé si všimnime, že žiadny bit syndrómu nemôže byť použitý vo viac ako jednom ortogonálnom kontrolnom súčte, pretože chybový paritný bit by bol kontrolovaný viac ako raz. Pretože tam sú len štyri syndrómové bity ktoré kontrolujú $e_0^{(1)}$, nieje možné získať viac ako štyri ortogonálne kontrolné súčty na e0(1) v tomto príklade. Vybrané ortogonálne kontrolné súčty môžu byť ilustrované použitím trojuholníka parity nasledovne:

$$
\begin{array}{lcccccc}
\longrightarrow & 1 & & & & & \\
\longrightarrow & 1 & \boxed{1} & & & & \\
 & 0 & 1 & 1 & & & \\
 & 0 & 0 & 1 & 1 & & \\
\longrightarrow & 1 & 0 & 0 & \boxed{1} & \boxed{1} & \\
 & 0 & 1 & 0 & 0 & 1 & 1 \\
\longrightarrow & 1 & 0 & \boxed{1} & 0 & 0 & \boxed{1}\ \boxed{1}.
\end{array}
$$

Šípky signalizujú syndrómové bity alebo súčty syndrómových bitov, ktoré sú vybrané ako ortogonálne kontrolné súčty na $e_0^{(1)}$ a štvorce označujú, ktoré informačné chybové bity iné ako $e_0^{(1)}$ sú kontrolované. Zreteľne, najviac jedna šípka môže ukazovať na každý riadok a jeden štvorec sa môže objaviť v každom stĺpci. Rovnice pre ortogonálne kontrolné súčty sú

$$
\begin{aligned}
s_0 &= e_0^{(1)} + e_0^{(2)} \\
s_1 &= e_0^{(1)} + e_1^{(1)} + e_1^{(2)} \\
s_4 &= e_0^{(1)} + e_3^{(1)} + e_4^{(1)} + e_4^{(2)} \\
s_6 &= e_0^{(1)} + e_2^{(1)} + e_5^{(1)} + e_6^{(1)} + e_6^{(2)}.
\end{aligned}
\tag{3.15}
$$

Všimnime si, že $e_0^{(1)}$ sa objaví v každom kontrolnom súčte, ale žiadny iný chybový bit sa neobjaví viac ako raz. Z každého kontrolného súčtu je jeden syndrómový bit a nie suma syndrómových bitov, toto je nazvané self-ortogonálny kód. Jedenásť odlišných kanálových bitov bude kontrolovaných štyrmi ortogonálnymi kontrolnými súčtami 3.15, pravidlo väčšinovej logiky dekódovania bude správne odhadovať $e_0^{(1)}$ kedykoľvek $t_{ML} = 2$ alebo menej z týchto 11 chybových bitov sú kanálové chyby. Celkový počet kanálových chybových bitov kontrolovaných rovnicami ortogonálnych kontrolných súčtov sa nazýva efektívna nútená dĺžka n_e kódu. Z toho dôvodu $n_e = 11$ pre tento kód.

Kompletný kodér/dekodér blokový diagram self-ortogonálneho kódu z príkladu 3.1 s dekódovaním väčšinovou logikou je zobrazený na obrázku 3.3. Práca dekodéra môže byť popísaná nasledovne:

Krok 1. Prvá nútená dĺžka syndrómových bitov $s_0, s_1, \ldots, s_6$ je vypočítaná.

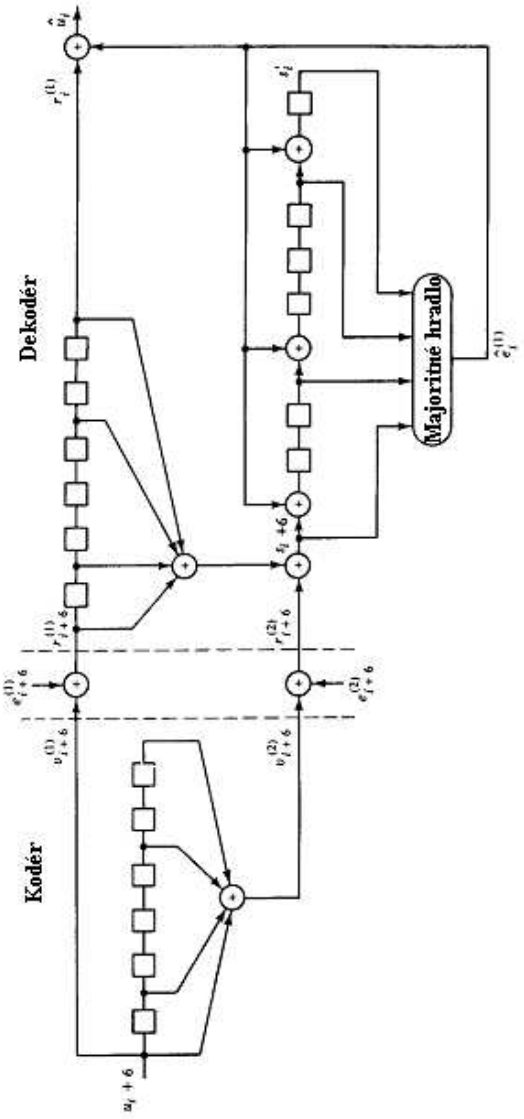
Krok 2. Sada štyroch ortogonálnych kontrolných súčtov na $e_0^{(1)}$ je formovaná zo syndrómových bitov počítaných v prvom kroku.

Krok 3. Štyri kontrolné sumy sú vložené do člena väčšinovej logiky, ktorá produkuje výstup 1 ak a len ak tri zo štyroch (viac ako polovica) z týchto vstupov je 1. Ak tieto výstupy sú 1 [$e_0^{(1)} = 1$], $r_0^{(1)}$ je predpokladané ako nesprávne, a preto musí byť opravené. Ak tieto výstupy sú 0 [$e_0^{(1)} = 0$], $r_0^{(1)}$ je predpokladané ako správne. Oprava spočíva pridaním výstupu prahového hradla [$e_0^{(1)}$] do $r_0^{(1)}$. Výstup prahového hradla [$e_0^{(1)}$] je tiež vrátený späť a odčítaný z každého syndrómového bitu, ktorý to ovplyvňuje.[Nieje nutné aby sa odčítalo $e_0^{(1)}$) z s_0 pretože syndrómové kúsky niesu použité v žiadnych budúcich odhadoch.]

Krok 4. Odhadovaný informačný bit $\mathbf{u}_0 = r_0^{(1)} + e_0^{(1)}$ je posunutý von z dekodéru. Zoznam syndrómu sa posunie o jedno doprava, ďalší blok prijatých bitov [$r_7^{(1)}$ a $r_7^{(2)}$] je posunutý do dekodéru a ďalší syndrómový bit s_7 je vypočítaný a posunutý do lavého krajného stupňa zoznamu syndrómu.

Krok 5. Zoznam syndrómu teraz obsahuje upravené bity syndrómu $s'_0, s'_1, \ldots, s'_6$ spoločne *s* s_7. Dekodér opakuje kroky 2, 3 a 4 a odhaduje $e_1^{(1)}$. Všetky nasledovné chybné informačné bity sú potom odhadnuté rovnakým spôsobom.

Pretože každý odhad sa musí vrátiť (fed back) späť do modifikovania zoznamu syndrómu pred tým ako je možné urobiť ďalší odhad, nazývame ho feedback dekodér. Je to podobné ako odozva každého odhadu v Meggittovom dekodéri pre cyklické kódy. Všimnime si, že každý odhad urobený feedback majority logic dekodérom závisí len na jednej nútenej dĺžke chybových bitov, efekt z predchádzajúcich chybových bitov je odstránený odozvou. Toto vedie k približne optimálnemu výkonu v porovnaní s dekódovaním s maximálnou pravdepodobnosťou. Jednoduchosť dekodéra robí z väčšinovej logiky dekódovania atraktívnu alternatívu k Viterbiho algoritmu alebo sekvenčnému dekódovaniu v niektorých prípadoch.



Obrázok 3.3: Kompletný systém blokového diagramu pre (2, 1, 6) self-ortogonálny kód s dekódovaním väčšinovej logiky

4 Porovnanie algoritmov

V tejto kapitole porovnáme náročnosť výpočtu na prijatú sekvenciu pre rôzne algoritmy a určíme, ktorý z týchto rozoberaných algoritmov je najviac vyhovujúci pre dekódovanú sekvenciu.

Sekvencia informácií u = (1 1 0 0 1) bola zakódovaná konvolučným kodérom s vytváracími mnohočlenmi: $g_1 = (1 + D)$, $g_2 = (1 + D^2)$, $g_3 = (1 + D + D^2)$. Toto zodpovedá zakódovanej sekvencii informácií:

v = (1 1 1, 0 1 0, 1 1 0, 0 1 1, 1 1 1, 1 0 1, 0 1 1),

Viterbiho dekodér potreboval k dekódovaniu prijatej sekvencie 15 dekódovacích krokov jak pri málo, tak aj pri veľmi hlučnom kanáli. Stačilo mu v oboch prípadoch 15 počítaní a konečný survivor bol zhodný so zakódovanou sekvenciou.

V Stack algoritme sa ukázalo, že potrebný počet dekódovacích krokov je závislý na hlučnosti kanálu. Zatiaľčo čo pri málo hlučnom kanáli potreboval len 10 dekódujúcich krokov, čo je lepšie ako u Viterbiho algoritme, pri veľmi hlučnom kanále bolo nutných až 20 počítaní aby dosial rovnakú dekódovanú správu.

Počet dekódovacích krokov u Fanovho algoritmu bol závislý na nastavení prahového zvýšenia Δ. Pri zvolení prahového zvýšenia $\Delta = 1$, potreboval algoritmus 40 dekódovacích krokov a pri $\Delta = 3$ bolo potrebných 22 počítaní, kým algoritmus skončil.

Tabuľka 4.1: Porovnanie algoritmov pre konvolučné dekódovanie

Algoritmus	**Množstvo počítaní / kanál**	
	Málo hlučný	**Hlučný**
Viterbi	15	15
Stack	10	20
Fano	$\Delta = 1$	$\Delta = 3$
	40	22

5 Záver

Pri vypracovaní projektu, teda porovnaní jednotlivých algoritmov boli brané do úvahy Viterbiho algoritmus, Stack algoritmus, Stack-bucket algoritmus, Fano algoritmus a Feedback dekódovania. Ako prvým som sa zaoberal Viterbiho algoritmom, ktorý patrí pod dekódovanie konvolučných kódov s najväčšou pravdepodobnosťou, ktoré pracuje s metrikou pridruženou s cestou. Maximálna cesta pravdepodobnosti je tuto cesta s najväčšou metrikou. Používanie bitovej metriky u Viterbiho algoritmu sa dá nahradiť celočíselnou metrikou bez straty výkonu. Pri aplikovaní Viterbiho algoritmu na BSC, $d(r_i, v_i)$ sa stáva metrikou vetve a algoritmus musí nájsť cestu skrz laťkovú mrežu s najnižšou metrikou. Detaily algoritmu sú v skutku rovnaké, ibaže Hammingova vzdialenosť nahrádza logickú pravdepodobnostnú funkciu ako metriku a survivor v každom stave je cesta s najmenšou metrikou.

Ďalej som sa zaoberal sekvenčným dekódovaním, z ktorého som sa zaoberal takzvaným stack algoritmom (algoritmus skladište) a Fano algoritmom. Účel algoritmu sekvenčného dekódovania je hľadať cestu skrz uzly kódového stromu účinným spôsobom v pokuse nájsť maximálnu cestu pravdepodobnosti. Či špecifická cesta bude časť maximálnej cesty pravdepodobnosti závisí na metrickej hodnote pridruženej k tejto ceste. Metrika je miera blízkosti cesty k prijatej sekvencii. Fanov algoritmus obecne dokóduje rýchlejšie ako stack – bucket algoritmus pre mierne rozsahy. Toto je pretože Fanov algoritmus nieje spomalený kontrolnými problémami v skladisku ako stack – bucket algoritmus. Pre vyššie sadzby stack – bucket algoritmus je trošku rýchlejší kvôli dodatočnému výpočtovému množstvu Fanovho algoritmu. Pretože nevyžaduje žiadny sklad, je Fanov algoritmus obvykle vybraný v praktických implementáciách sekvenčného dekódovania.

Nakoniec som rozoberal feedback dekódovanie, ktoré patrí do skupiny dekódovania konvolučných kódov s najväčšou logikou. Toto sa líši od predchádzajúcich metód v konečnom rozhodnutí urobenom na danom informačnom bloku založenom len na jednej nútenej dĺžke prijatých blokov skôr ako na celej prijatej sekvencii.

Bol zostavený návrh konvolučného dekodéra a dekodéra, ktorý pri svojej činnosti využíva Viterbiho algoritmus a Stack algoritmus, čo slúži pre porovnanie oboch metód. Ukázalo sa, že Viterbiho algoritmus je vhodný hlavne pre kódy s menšou nútenou dĺžkou, pretože má pevné množstvo počítaní. Pri Stack algoritme je množstvo počítaní závislé na dĺžke prijatej sekvencie, čo môže mať za následok oveľa menší počet počítaní, avšak pri veľmi hlučnom kanáli sa môže stať, že počítaní bude omnoho viac ako u Viterbiho algoritmu. Nič menej je Stack algoritmus alebo jeho variácia Stack-bucket algoritmus používaný hlavne na kódy s väčšou nútenou dĺžkou. Feedback dekódovanie je zase pre jednoduchú konštrukciu dekodéra vhodnou alternatívou k dekódovaniu s najväčšou pravdepodobnosťou a k sekvenčnému dekódovaniu.

Zoznam použitej literatúry

[1] MICHAEL PURSER: Introduction to error correcting codes, *ISBN 0-89006-784-8, British cataloguing library in publication data.*

[2] SHU LIN, DANIEL J. COSTELLO, Jr: Error control coding: Fundamentals and aplications, *Prentice Hall, Jew Jersey 07632*

[3] ROBERT H. MORELOS-ZARAGOZA: Error correcting coding, *SONY Computer science laboratories, Inc, 2002*

[4] Prof.Ing. ANTON ČIŽMÁR, Csc.: Kódovanie a modulácia: Konvolučné kódy

[5] J.M.WOZENCRAFT and I.M.JACOBS: Principles of Communications Engineering, *Wiley, New York, 1965*

[6] L. L. JOINER and J. J. Komo: Sequential decoding of reed-solomon codes in an incremental redundancy system, *in Milcom 1997 Proceedings, pp. 1-4, Oct. 1997.*

[7] R. E. BLAHUT: Theory and Practice of Error Control Codes. *Reading, MA:Addison-Wesley,*

[8] R. GALLEGAR: Information Theory and Reliable Communication, *Wiley, New York, 1968*

[9] H. EL. GAMAL and M.O. DAMEN. Universal space-time coding, *IEEE Transaction info. Theory, 49:1097-1119, May 2003*

[10] ARUL D. MURUGAN, H. EL. GAMAL, M. O. DAMEN and G. CAIRE: A unified framework for tre search decoding rdiscovering the sequencial decoder. *IEEE Transaction info. Theory, 52(3):933-953, March 2006*

[11] J. PROAKIS: Digital Communications, *McGraw-Hill, 4^{th} ed edition, 2000*

[12] E. VITERBO and J. BOUTROS: A universal lattice code decoder for fading channels, *IEEE Transaction info. Theory, 45(5):1633-1642, July 1999*

PRÍLOHY

A ZDROJOVÉ TEXTY

A.1 Pomocné funkcie

```
#include <iostream>

#include <string>
#include <vector>
#include <algorithm>
#include <bitset>

#include <cstring>

#define LINESIZE 1024

using namespace std;

// štruktúra použitá v Stack algoritme reprezentujúca jeden uzol
// metrika pre daný uzol
// pozícia vyjadruje pozíciu v načítaných bitoch
// výstup predstavuje dekódované data
struct t_uzel
{
	int metrika;
	int stav;
	size_t pozice;
	vector<bitset<1> > vystup;
};
// funkcia pre radenie v Stack algoritme od najnizšej po najvyššiu
// kterú využíva algoritmus sort z knihovne STL
bool porovnej (t_uzel i, t_uzel j)
{
	return (i.metrika < j.metrika);
}
// pomocná funkcia prekopíruje obsah jedného vektoru do druhého
void copyVector(vector<bitset<1> > *vstup, vector<bitset<1> > *vystup)
{
	vystup->clear();
	size_t size = vstup->size();
	for (size_t i = 0; i < size; i++)
		vystup->push_back(vstup->at(i));
	return;
}
// fce prevedie vstupný reťazec znakov na reťazec 0 a 1 ale v obrátenom poradí
string StrToBin(char* AsciiData)
{
	size_t length = strlen(AsciiData);
	string pole("");
	for (size_t i = 0; i < length; i++)
	{
		char c = AsciiData[i];
		for (int j = 0; j < 8; j++)
		{
			pole += ('0' + (c % 2));
			c = c/2;
		}
	}

	return pole;
```

```
}
// prevedie postupnosť reťazcových 0 a 1 na vektor binárnych 0 a 1
void StrToBinvector(vector<bitset<1> > *vystup, const char *str)
{
	// pomocné premenné reprezentujúce binárne hodnoty
	bitset <1> nula;
	bitset <1> jedna;
	jedna.set();

	size_t str_length = strlen(str);
	string pom("");
	// prechádza sekvenčne celý reťazec a porovnáva
	// ukladaá cez ukazatel do vektoru
	for (size_t i = 0; i < str_length; i++)
	{
		pom = str[i];
		if (pom.compare("0") == 0)
			vystup->push_back(nula);
		else
			vystup->push_back(jedna);
	}

	return;
}

// fce prevedie vektor binárnych hodnôt späť na reťazec znakov
// vracia výsledný reťazec
string BinvecToStr(vector<bitset<1> > *vstup)
{
	string vystup("");
	size_t size = vstup->size()/8;
	bitset <1> pom;
	bitset <1> nula;
	size_t kroky(0);
	int i(0);
	while (kroky < size)
	{
		char znak = 0;
		pom = vstup->at(i++);
		if (pom != nula)
			znak = static_cast<char>(128);

		pom = vstup->at(i++);
		if (pom != nula)
			znak += static_cast<char>(64);

		pom = vstup->at(i++);
		if (pom != nula)
			znak += static_cast<char>(32);

		pom = vstup->at(i++);
		if (pom != nula)
			znak += static_cast<char>(16);

		pom = vstup->at(i++);
		if (pom != nula)
			znak += static_cast<char>(8);

		pom = vstup->at(i++);
		if (pom != nula)
			znak += static_cast<char>(4);

		pom = vstup->at(i++);
		if (pom != nula)
			znak += static_cast<char>(2);
```

```
			pom = vstup->at(i++);
			if (pom != nula)
				znak += static_cast<char>(1);
			vystup += znak;
			kroky++;
		}
		return vystup;

}

// pomocná fce pre výpis vektoru bitov na obrazovku
void BinToStr(vector<bitset<1> > *vstup)
{
//	string vystup("");
		bitset <1> pom;
		string pomoc("");
		size_t vector_size = vstup->size();
		for (size_t i = 0; i < vector_size; i++)
		{
			pom = vstup->at(i);
			cout << pom.to_string<char,char_traits<char>,allocator<char> >();
			if (((i+1)%3) == 0)
				cout << " ";
		}

		cout << endl;
		return;
}
// pomocná fce pre prevrátenie reťazca
string Reverse(string word)
{
		string vystup("");
		string::reverse_iterator rit;
		for ( rit = word.rbegin() ; rit < word.rend(); rit++ )
			vystup += *rit;

		return vystup;
}
// funkcia pre zakódovanie vstupného vektoru
void Encode(vector<bitset<1> > *vstup, vector<bitset<1> > *vystup, size_t size)
{
		// pomocne promenne pro vysledky operaci xor a nacteni hodnot z vektoru
		bitset <1> xor1;
		bitset <1> xor2;
		bitset <1> xor3;
		bitset <1> pom1;
		bitset <1> pom2;
		bitset <1> pom3;
		// prevedie tri operácie xor s patričnými premennými
		// D0 xor D1, D0 xor D2, D0 xor D1 xor D2
		for (size_t i = 0; i < size; i++)
		{
			pom1 = vstup->at(i);
			pom2 = vstup->at(i + 1);
			pom3 = vstup->at(i + 2);
			xor1 = (pom3 ^ pom2);
			xor2 = (pom1 ^ pom3);
			xor3 = (pom1 ^ pom2 ^ pom3);

			// uloží výsledky do výstupného vektoru
			vystup->push_back(xor1);
			vystup->push_back(xor2);
			vystup->push_back(xor3);
		}
```

```
	return;
}
// pomocná funkcia pri dekódovaní Viterbiho algoritmom
// zistí metriku (v kolkých bitoch sa líšii od vstupnej trojice) podla daného
stavu
// vracia ohodnotenie pre daný stav
int Viterbi_metrika(vector<bitset<1> > *vstup, size_t pos, int stav, int stav2)
{
	// reprezentuje binárne hodnoty
	bitset <1> nula;
	bitset <1> jedna;
	jedna.set();
	bitset <1> pom;
	int metrika(0);
	// prepínač pre 4 stavy každý obsahuje ešte ďalsie dva podstavy
	switch (stav)
	{
		case 0:
			// stav s0 porovná v kolkých bitoch sa líši od 000
			if (stav2 == 0)
			{
				if ((vstup->at(pos)) != nula)
					metrika++;

				if ((vstup->at(pos + 1)) != nula)
					metrika++;

				if ((vstup->at(pos + 2)) != nula)
					metrika++;
			}
			// stav s0 porovná v kolkých bitoch sa líši od 111
			else
			{
				if ((vstup->at(pos)) != jedna)
					metrika++;

				if ((vstup->at(pos + 1)) != jedna)
					metrika++;

				if ((vstup->at(pos + 2)) != jedna)
					metrika++;
			}
		break;

		case 1:
			// stav s1 porovná v kolkých bitoch sa líši od 101
			if (stav2 == 0)
			{
				if ((vstup->at(pos)) != jedna)
					metrika++;

				if ((vstup->at(pos + 1)) != nula)
					metrika++;

				if ((vstup->at(pos + 2)) != jedna)
					metrika++;
			}
			// stav s1 porovná v kolkých bitoch sa líši od 010
			else
			{
				if ((vstup->at(pos)) != nula)
					metrika++;

				if ((vstup->at(pos + 1)) != jedna)
```

```
                        metrika++;

                    if ((vstup->at(pos + 2)) != nula)
                        metrika++;
                }
            break;

            case 2:
                // stav s2 porovná v kolkých bitoch sa líši od 011
                if (stav2 == 0)
                {
                    if ((vstup->at(pos)) != nula)
                        metrika++;

                    if ((vstup->at(pos + 1)) != jedna)
                        metrika++;

                    if ((vstup->at(pos + 2)) != jedna)
                        metrika++;
                }
                // stav s2 porovná v kolkých bitoch sa líši od 100
                else
                {
                    if ((vstup->at(pos)) != jedna)
                        metrika++;

                    if ((vstup->at(pos + 1)) != nula)
                        metrika++;

                    if ((vstup->at(pos + 2)) != nula)
                        metrika++;
                }
            break;

            case 3:
                // stav s3 porovná v kolkých bitoch sa líši od 110
                if (stav2 == 0)
                {
                    if ((vstup->at(pos)) != jedna)
                        metrika++;

                    if ((vstup->at(pos + 1)) != jedna)
                        metrika++;

                    if ((vstup->at(pos + 2)) != nula)
                        metrika++;
                }
                // stav s3 porovná v kolkých bitoch sa líši od 001
                else
                {
                    if ((vstup->at(pos)) != nula)
                        metrika++;

                    if ((vstup->at(pos + 1)) != nula)
                        metrika++;

                    if ((vstup->at(pos + 2)) != jedna)
                        metrika++;
                }
            break;
        }

        return metrika;
}
```

A.2 Verbiho dekodér

```
// funkcia pre dekódovanie postupnosti Viterbiho algoritmom
void Viterbi_decode(vector<bitset<1> > *vstup, vector<bitset<1> > *vystup)
{
	// reprezentuje binárne hodnoty
	bitset <1> nula;
	bitset <1> jedna;
	jedna.set();
	size_t vstup_size = vstup->size();

	// pomocné premenné pre uloženie metrík pri prechode mezi stavmi
	int met_s0_s0(0);
	int met_s0_s1(0);
	int met_s1_s2(0);
	int met_s1_s3(0);
	int met_s2_s1(0);
	int met_s2_s0(0);
	int met_s3_s3(0);
	int met_s3_s2(0);

	// promenné pre uloženie metriky daného stavu
	int s0(0);
	int s1(0);
	int s2(0);
	int s3(0);

	// promenné pre uloženie metriky daného stavu v predchádzajúcom kroku
	int puv_s0(0);
	int puv_s1(0);
	int puv_s2(0);
	int puv_s3(0);

	// vektory pre uloženie dekodóvaných postupností pre dané stavy
	vector<bitset<1> > v0;
	vector<bitset<1> > v1;
	vector<bitset<1> > v2;
	vector<bitset<1> > v3;
	vector<bitset<1> > puv_v0;
	vector<bitset<1> > puv_v1;
	vector<bitset<1> > puv_v2;
	vector<bitset<1> > puv_v3;

	// je v počiatočnom stave s0 s hodnotou 0
	// ručne vygeneruje metriky pre prvý krok pre stav s0
	met_s0_s0 = Viterbi_metrika(vstup, 0, 0, 0);
	met_s0_s1 = Viterbi_metrika(vstup, 0, 0, 1);
	s0 += met_s0_s0;
	s1 += met_s0_s1;
	v0.push_back(nula);
	v1.push_back(jedna);

	// má stavy s0 a s1 ručne vygeneruje ďalšie stavy a aktualizuje pôvodné
	// vypočíta metriky pre všetky cesty
	puv_s0 = s0;
	puv_s1 = s1;
	puv_v0 = v0;
	puv_v1 = v1;
	met_s0_s0 = Viterbi_metrika(vstup, 3, 0, 0);
	met_s0_s1 = Viterbi_metrika(vstup, 3, 0, 1);
	met_s1_s2 = Viterbi_metrika(vstup, 3, 1, 0);
	met_s1_s3 = Viterbi_metrika(vstup, 3, 1, 1);
	s0 = puv_s0 + met_s0_s0;
	s1 = puv_s0 + met_s0_s1;
```

```
s2 = puv_s1 + met_s1_s2;
s3 = puv_s1 + met_s1_s3;
v0.push_back(nula);
v1 = puv_v0;
v1.push_back(jedna);
v2 = puv_v1;
v2.push_back(nula);
v3 = puv_v1;
v3.push_back(jedna);

// v cykle prechádza stavy rozgeneruje nové a vybere tie s lepšou metrikou
for (size_t i = 6; i < vstup_size - 6; i += 3)
{
	// uloží pôvodné hodnoty stavu a vektoru
	puv_s0 = s0;
	puv_s1 = s1;
	puv_s2 = s2;
	puv_s3 = s3;
	puv_v0 = v0;
	puv_v1 = v1;
	puv_v2 = v2;
	puv_v3 = v3;

	// zistí metriky do ďalších stavov
	met_s0_s0 = Viterbi_metrika(vstup, i, 0, 0);
	met_s0_s1 = Viterbi_metrika(vstup, i, 0, 1);
	met_s1_s2 = Viterbi_metrika(vstup, i, 1, 0);
	met_s1_s3 = Viterbi_metrika(vstup, i, 1, 1);
	met_s2_s0 = Viterbi_metrika(vstup, i, 2, 0);
	met_s2_s1 = Viterbi_metrika(vstup, i, 2, 1);
	met_s3_s2 = Viterbi_metrika(vstup, i, 3, 0);
	met_s3_s3 = Viterbi_metrika(vstup, i, 3, 1);

	// vybere výhodnejšiu cestu do nového stavu s0
	if (puv_s0 + met_s0_s0 < puv_s2 + met_s2_s0)
	{
		// ide zo stavu s0 do s0
		s0 = puv_s0 + met_s0_s0;
		v0 = puv_v0;
		v0.push_back(nula);
	}
	else
	{
		// ide zo stavu s2 do s0
		s0 = puv_s2 + met_s2_s0;
		v0 = puv_v2;
		v0.push_back(nula);
	}

	// vybere výhodnejšiu cestu do nového stavu s1
	if (puv_s0 + met_s0_s1 < puv_s2 + met_s2_s1)
	{
		// ide zo stavu s0 do s1
		s1 = puv_s0 + met_s0_s1;
		v1 = puv_v0;
		v1.push_back(jedna);
	}
	else
	{
		// ide zo stavu s2 do s1
		s1 = puv_s2 + met_s2_s1;
		v1 = puv_v2;
		v1.push_back(jedna);
	}
```

```
			// vybere výhodnejšiu cestu do nového stavu s2
			if (puv_s1 + met_s1_s2 < puv_s3 + met_s3_s2)
			{
				// ide zo stavu s1 do s2
				s2 = puv_s1 + met_s1_s2;
				v2 = puv_v1;
				v2.push_back(nula);
			}
			else
			{
				// ide zo stavu s3 do s2
				s2 = puv_s3 + met_s3_s2;
				v2 = puv_v3;
				v2.push_back(nula);
			}

			// vybere výhodnejšiu cestu do nového stavu s3
			if (puv_s1 + met_s1_s3 < puv_s3 + met_s3_s3)
			{
				// ide zo stavu s1 do s3
				s3 = puv_s1 + met_s1_s3;
				v3 = puv_v1;
				v3.push_back(jedna);
			}
			else
			{
				// ide zo stavu s3 do s3
				s3 = puv_s3 + met_s3_s3;
				v3 = puv_v3;
				v3.push_back(jedna);
			}
		}

		int pole [] = {s0, s1, s2, s3};

		// vybere stav s najmenšou metrikou a vráti vektor jeho binárnych hodnôt
		int minimum = *min_element(pole,pole+4);
		if (minimum == s0)
			copyVector(&v0, vystup);
		else if (minimum == s1)
			copyVector(&v1, vystup);
		else if (minimum == s2)
			copyVector(&v2, vystup);
		else if (minimum == s3)
			copyVector(&v3, vystup);

		return;
}

// funkcia pre výpočet metriky Stack algoritmu pre daný vstup
// +1 pokial bit súhlasí -5 pokial sa líši
int Stack_metrika(vector<bitset<1> > *vstup, size_t pos, int stav)
{
		// reprezentuje binárne hodnoty
		bitset <1> nula;
		bitset <1> jedna;
		jedna.set();

		// premenná pre uloženie výslednej metriky
		int metrika(0);

		switch (stav)
		{
			case 000:
				if ((vstup->at(pos)) == nula)
```

```
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 1)) == nula)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 2)) == nula)
            metrika++;
      else
            metrika -= 5;
break;

case 001:
      if ((vstup->at(pos)) == nula)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 1)) == nula)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 2)) == jedna)
            metrika++;
      else
            metrika -= 5;
break;

case 010:
      if ((vstup->at(pos)) == nula)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 1)) == jedna)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 2)) == nula)
            metrika++;
      else
            metrika -= 5;
break;

case 011:
      if ((vstup->at(pos)) == nula)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 1)) == jedna)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 2)) == jedna)
            metrika++;
      else
            metrika -= 5;
break;
```

```
case 100:
      if ((vstup->at(pos)) == jedna)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 1)) == nula)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 2)) == nula)
            metrika++;
      else
            metrika -= 5;
break;

case 101:
      if ((vstup->at(pos)) == jedna)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 1)) == nula)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 2)) == jedna)
            metrika++;
      else
            metrika -= 5;
break;

case 110:
      if ((vstup->at(pos)) == jedna)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 1)) == jedna)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 2)) == nula)
            metrika++;
      else
            metrika -= 5;
break;

case 111:
      if ((vstup->at(pos)) == jedna)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 1)) == jedna)
            metrika++;
      else
            metrika -= 5;

      if ((vstup->at(pos + 2)) == jedna)
            metrika++;
```

```
                else
                        metrika -= 5;
                break;
        }

        return metrika;
}

// funkcia pre generovanie nových uzlov Stack algoritmu
void Stack_generate(t_uzel *U, vector <t_uzel> *sklad, vector<bitset<1> >
*vstup)
{
        // reprezentuje binárne hodnoty
        bitset <1> nula;                        // reprezentuje binárnu nulu
        bitset <1> jedna;                       // binárnu jedničku
        jedna.set();

        // pomocné premenné pre vstupné a generované uzly
        t_uzel uzel, uzel0, uzel1;

        // počiatočná inicializácia vstupnými dátami
        uzel.metrika = U->metrika;
        uzel.stav = U->stav;
        uzel.pozice = U->pozice;
        uzel.vystup = U->vystup;
        uzel0 = uzel1 = uzel;

        int pomoc0;
        int pomoc1;

        // pre uzol daného stavu vygeneruje dvoch možných následníkov
        // aktualizuje metriky kódu, stav, a pozíciu vo vstupných dátach
        // všetko uloží do skladu
        switch (uzel.stav)
        {
                case 000:
                        pomoc0 = Stack_metrika(vstup, uzel.pozice, 000);
                        pomoc1 = Stack_metrika(vstup, uzel.pozice, 111);
                        uzel0.metrika += pomoc0;
                        uzel1.metrika += pomoc1;
                        uzel0.stav=000;
                        uzel1.stav = 111;
                        uzel0.pozice += 3;
                        uzel1.pozice += 3;
                        (uzel0.vystup).push_back(nula);
                        (uzel1.vystup).push_back(jedna);
                        sklad->push_back(uzel0);
                        sklad->push_back(uzel1);
                break;

                case 001:
                        pomoc0 = Stack_metrika(vstup, uzel.pozice, 110);
                        pomoc1 = Stack_metrika(vstup, uzel.pozice, 001);
                        uzel0.metrika += pomoc0;
                        uzel1.metrika += pomoc1;
                        uzel0.stav=110;
                        uzel1.stav = 001;
                        uzel0.pozice += 3;
                        uzel1.pozice += 3;
                        (uzel0.vystup).push_back(nula);
                        (uzel1.vystup).push_back(jedna);
                        sklad->push_back(uzel0);
                        sklad->push_back(uzel1);
                break;
```

```
case 010:
      pomoc0 = Stack_metrika(vstup, uzel.pozice, 110);
      pomoc1 = Stack_metrika(vstup, uzel.pozice, 001);
      uzel0.metrika += pomoc0;
      uzel1.metrika += pomoc1;
      uzel0.stav=110;
      uzel1.stav = 001;
      uzel0.pozice += 3;
      uzel1.pozice += 3;
      (uzel0.vystup).push_back(nula);
      (uzel1.vystup).push_back(jedna);
      sklad->push_back(uzel0);
      sklad->push_back(uzel1);
break;

case 011:
      pomoc0 = Stack_metrika(vstup, uzel.pozice, 000);
      pomoc1 = Stack_metrika(vstup, uzel.pozice, 111);
      uzel0.metrika += pomoc0;
      uzel1.metrika += pomoc1;
      uzel0.stav=000;
      uzel1.stav = 111;
      uzel0.pozice += 3;
      uzel1.pozice += 3;
      (uzel0.vystup).push_back(nula);
      (uzel1.vystup).push_back(jedna);
      sklad->push_back(uzel0);
      sklad->push_back(uzel1);
break;

case 100:
      pomoc0 = Stack_metrika(vstup, uzel.pozice, 101);
      pomoc1 = Stack_metrika(vstup, uzel.pozice, 010);
      uzel0.metrika += pomoc0;
      uzel1.metrika += pomoc1;
      uzel0.stav=101;
      uzel1.stav = 010;
      uzel0.pozice += 3;
      uzel1.pozice += 3;
      (uzel0.vystup).push_back(nula);
      (uzel1.vystup).push_back(jedna);
      sklad->push_back(uzel0);
      sklad->push_back(uzel1);
break;

case 101:
      pomoc0 = Stack_metrika(vstup, uzel.pozice, 011);
      pomoc1 = Stack_metrika(vstup, uzel.pozice, 100);
      uzel0.metrika += pomoc0;
      uzel1.metrika += pomoc1;
      uzel0.stav=011;
      uzel1.stav = 100;
      uzel0.pozice += 3;
      uzel1.pozice += 3;
      (uzel0.vystup).push_back(nula);
      (uzel1.vystup).push_back(jedna);
      sklad->push_back(uzel0);
      sklad->push_back(uzel1);
break;

case 110:
      pomoc0 = Stack_metrika(vstup, uzel.pozice, 011);
      pomoc1 = Stack_metrika(vstup, uzel.pozice, 100);
      uzel0.metrika += pomoc0;
      uzel1.metrika += pomoc1;
```

```
                  uzel0.stav=011;
                  uzel1.stav = 100;
                  uzel0.pozice += 3;
                  uzel1.pozice += 3;
                  (uzel0.vystup).push_back(nula);
                  (uzel1.vystup).push_back(jedna);
                  sklad->push_back(uzel0);
                  sklad->push_back(uzel1);
            break;

            case 111:
                  pomoc0 = Stack_metrika(vstup, uzel.pozice, 101);
                  pomoc1 = Stack_metrika(vstup, uzel.pozice, 010);
                  uzel0.metrika += pomoc0;
                  uzel1.metrika += pomoc1;
                  uzel0.stav=101;
                  uzel1.stav = 010;
                  uzel0.pozice += 3;
                  uzel1.pozice += 3;
                  (uzel0.vystup).push_back(nula);
                  (uzel1.vystup).push_back(jedna);
                  sklad->push_back(uzel0);
                  sklad->push_back(uzel1);
            break;
      }

      return;
}
```

A.3 Stack dekodér

```
// vlastný Stack algoritmus pre dekódovanie zakódovanej postupnosti
void Stack_decode(vector<bitset<1> > *vstup, vector<bitset<1> > *vystup)
{
	// reprezentuje binárne hodnoty
	bitset <1> nula;
	bitset <1> jedna;
	jedna.set();

	// vektor kde se ukladajú novo generované uzly
	vector <t_uzel> sklad;
	vector<bitset<1> > decode;

	// počiatočný uzol
	t_uzel uzel;
	uzel.metrika = 0;
	uzel.stav = 000;
	uzel.pozice = 0;
	uzel.vystup = decode;

	// vygeneruje nové uzly
	// zoradí podla metriky
	// načíta uzel s najvyššou metrikou a zmaže ho z vektoru
	size_t size = vstup->size() - 6;
	while (uzel.pozice != size)
	{
		Stack_generate(&uzel, &sklad, vstup);
		sort(sklad.begin(), sklad.end(), porovnej);
		uzel = sklad.back();
		sklad.pop_back();
	}
	// uloženie výsledného vektoru binárnych dát
	copyVector (&(uzel.vystup), vystup);
	return;
}
```

A.4 Hlavný program

```
int main ()
{
      char word [LINESIZE];                    // pre dáta zadané zo vstupu
      bitset <1> nula;                  // reprezentuje binárnu nulu
      bitset <1> jedna;                 // binárnu jednotku
      jedna.set();
      vector <bitset<1> > vstup;               // vektor (pole) bitov
      vector <bitset<1> > encoded;             // pre zakódované pole bitov
      vector <bitset<1> > Vit_decoded;         // dekódované Viterbiho alg.
      vector <bitset<1> > St_decoded;               // dekódovane Stack alg
////////////////////////////////////////////////////////////////////////////////

      cout << "Zadajte vetu lebo slovo, ktore sa budu kodovat: ";

      cin.getline(word, LINESIZE);             // načítanie vstupu
      string bin_word = StrToBin(word);        // získa postup. retaz 0 a 1
      bin_word = Reverse(bin_word);
      size_t bword_length = bin_word.length(); // zistím dĺžku postupnosti

      cout << "Prevracena vstupni posloupnost delky ";
      cout << bword_length << ": " << bin_word << endl;
////////////////////////////////////////////////////////////////////////////////

      vstup.push_back(nula);
      vstup.push_back(nula);

      StrToBinvector(&vstup, bin_word.c_str());// vytvorí vektor bin. 0 a 1

      vstup.push_back(nula);
      vstup.push_back(nula);
      cout << "Pripravena na kodovani: ";
      BinToStr(&vstup);
////////////////////////////////////////////////////////////////////////////////

      Encode (&vstup, &encoded, bword_length + 2);   // zakóduje postupnosť
      cout << "Zakodovana posloupnost delky " << encoded.size() << ": ";
      BinToStr(&encoded);
////////////////////////////////////////////////////////////////////////////////

      Viterbi_decode (&encoded, &Vit_decoded); // Viterbiho algoritmus
      cout << "Viterbiho algoritmem dekodovana posloupnost: ";
      BinToStr(&Vit_decoded);
      cout << "Dekodovane znaky: " << Reverse(BinvecToStr(&Vit_decoded)) <<
endl;
////////////////////////////////////////////////////////////////////////////////

      Stack_decode(&encoded, &St_decoded);          // Stack algoritmus
      cout << "Stack algoritmem dekodovana posloupnost: ";
      BinToStr(&St_decoded);
      cout << "Dekodovane znaky: " << Reverse(BinvecToStr(&St_decoded)) << endl;

      return 0;
}
```