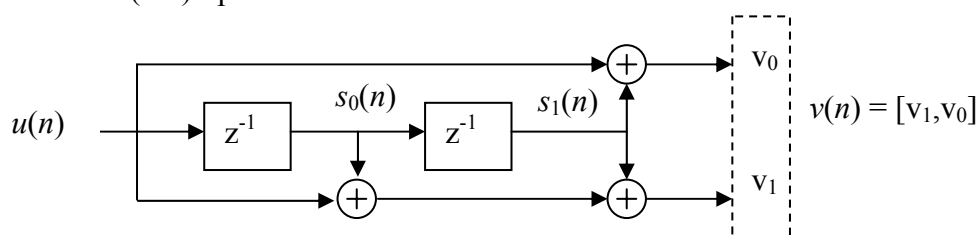


## 15. A konvolúciós kódolás és a trellis-kód moduláció

### 15.1. A konvolúciós kódolás

A digitális adatátvitelben gyakran alkalmaznak hibajavító kódolási eljárásokat (FEC, Forward Error Correction), annak érdekében, hogy a csatornában bekövetkező hibák ellenére az adatátvitel viszonylag kis hiba-valószínűség mellett valósuljon meg. A hibajavító kódolás egy gyakran használt fajtája a *konvolúciós kódolás*. Ebben az eljárásban a pillanatnyi kimenet a bemenet aktuális értékén kívül annak korábbi értékeitől is függ: általában a kimenet a kódoló súlyfüggvényének és a bemenetnek a konvolúciója-ként áll elő.

A 15.1. ábrán példaként egy viszonylag egyszerű konvolúciós kódoló felépítése látható. A bemenet  $u(n)$  1 bites, a kimenet 2 bites  $v_1, v_0$ , amit egyetlen kétbites változóba  $v(n)=[v_1, v_0]$  foghatunk össze. Így minden üzenet bithez egy redundáns bit adódik, szokásos jelöléssel ez a kódoló (1:2) típusú.



15.1. ábra Egy (1:2) típusú konvolúciós kódoló felépítése

Az összeadók átvitel nélküli 1 bites összeadók (XOR kapuk). A kódolóban lévő 1 bites tárolók a kódoló állapotát (**state**) írják le, amit szintén összefoghatunk egyetlen változóba:  $s(n)=[s_1, s_0]$ . A tárolók számához hozzáadva 1-et, a kódoló u.n. **kényszerhosszúság**-át kapjuk.

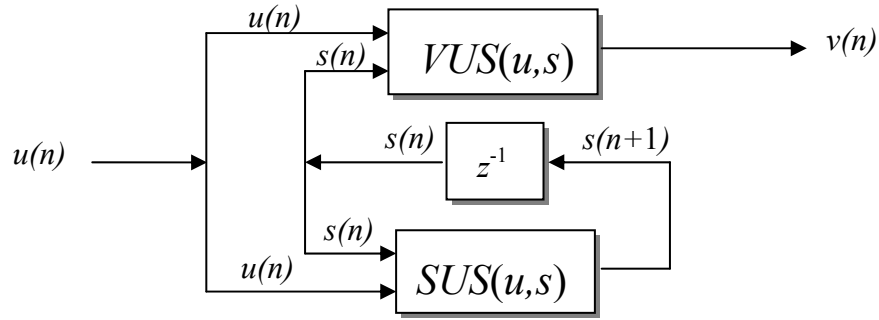
A kódolónak, mint egy kombinációs sorrendi hálózatnak az igazság táblázatát a 15.1. táblázatban foglaltuk össze.

$u(n)$ bin.	$s(n)$		$s(n)$ dec.	$v(n)$		$v(n)$ dec.	$s(n+1)$		$s(n+1)$ dec.
	$s_1$	$s_0$		$v_1$	$v_0$		$s_1$	$s_0$	
0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	2	1	0	2
0	1	0	2	1	1	3	0	0	0
0	1	1	3	0	1	1	1	0	2
1	0	0	0	1	1	3	0	1	1
1	0	1	1	0	1	1	1	1	3
1	1	0	2	0	0	0	0	1	1
1	1	1	3	1	0	2	1	1	3

15.1. Táblázat A kódoló igazságtáblázata

A példánkat rövid időre félretéve, általánoságban a konvolúciós kódolókat felfoghatjuk úgy is, mint egy véges állapotú állapotgépet, melynek a következő állapota  $s(n+1)$  csak az előző (az aktuális) állapottól  $s(n)$  és a pillanatnyi bemenettől  $u(n)$ , míg a  $v(n)$

kimenet ugyancsak az előző állapottól és a bemenettől függ. Az általános felépítés tehát az alábbi ábra szerinti:



15.2. ábra A konvolúciós kódoló általános felépítése

A kimenet kétféle változós függvénye a bemenetnek és a kódoló pillanatnyi állapotának:

$$v(n) = \mathbf{VUS}[u(n), s(n)] \quad (15.1.)$$

A kódoló következő időréseben érvényes állapota (s) szintén kétféle változós függvénye a bemenetnek és a kódoló pillanatnyi állapotának.

$$s(n+1) = \mathbf{SUS}[u(n), s(n)] \quad (15.2.)$$

Valós idejű alkalmazásokban ezeket a függvényeket célszerű előre kiszámítani az igazságtáblázat alapján, és táblázatok formájában tárolni.

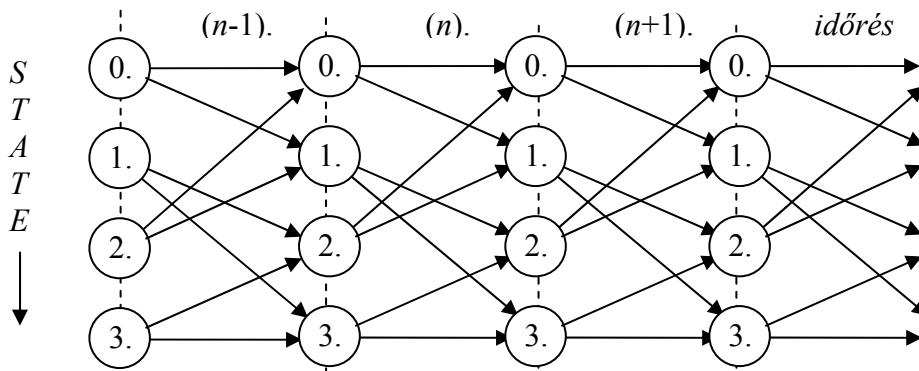
Példánkhoz visszatérve a két függvény két darab kétdimenziós tömbként adható meg:

$\mathbf{VUS}$		$s(n)$			
		0	1	2	3
$u(n)$	0	0	2	3	1
	1	3	1	0	2

$\mathbf{SUS}$		$s(n)$			
		0	1	2	3
$u(n)$	0	0	2	0	2
	1	1	3	1	3

15.2. és 15.3. Táblázat A  $v(n)=\mathbf{VUS}[u(n), s(n)]$  és az  $s(n+1)=\mathbf{SUS}[u(n), s(n)]$  függvény

Az állapotátmenetek lehetséges módjait egy gráfban is ábrázolhatjuk, amit a kódoló **treillis-ének** nevezünk:

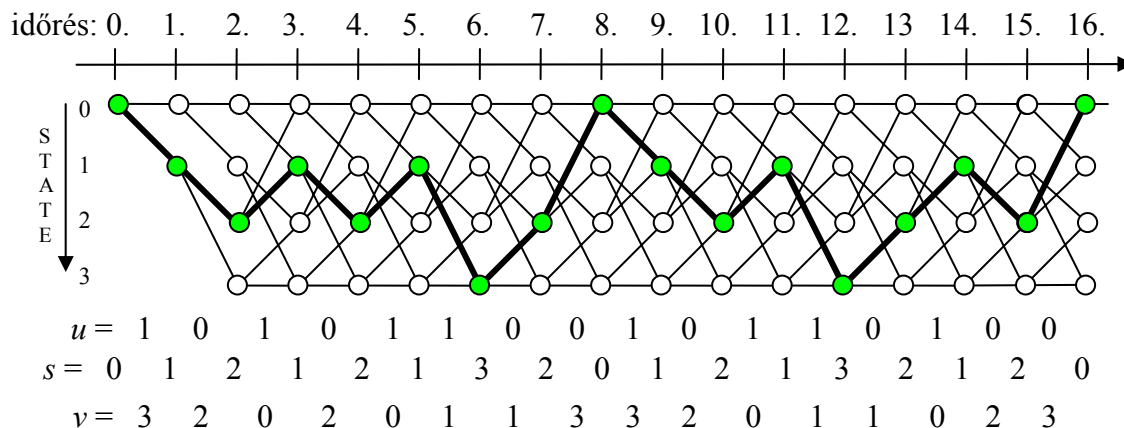


15.3 ábra A konvolúciós kódolónk trellis diagramja

Esetünkben minden csomópontból két ág indul és minden csomópontba két ág érkezik. (Más kódolónál ez természetesen más lehet). Esetünkben az elágazási szabály

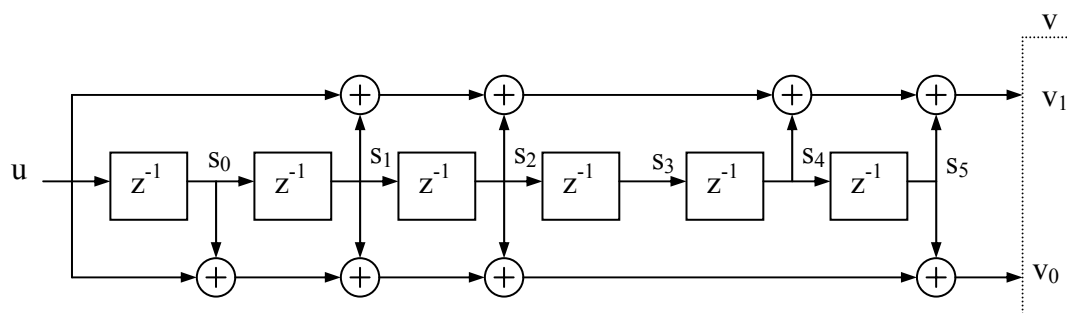
olyan, hogy egy csomópontból kiindulva a felső ághoz az  $u=0$ , az alsó ághoz az  $u=1$  üzenet tartozik.

Az alábbi ábrán a kódolónk állapot-átmeneteit követhetjük 16 időrésen keresztül. Az ábrában feltüntettük a bemeneti ( $u$ ) és a kimeneti ( $v$ ) sorozatot is. A kivastagított vonal a bemenet által meghatározott ténylegesen létrejövő állapot-átmeneteket ( $s$ ) mutatja.



15.4. ábra Az állapot-átmenetek alakulása az első 16 ütemben

A konvolúciós kódolók hibajavító képessége a kényszer-hosszúság növelésével javul. Az előző példánkban a könnyebb követhetőség kedvéért egy nagyon rövid konvolúciós kódolót választottunk. A ténylegesen használatos kódolók hosszabbak. Az alábbi ábrán egy szabványos, az INMARSAT-C műholdas adatátviteli rendszerben használatos, 7 kényszer-hosszúságú (1:2) konvolúciós kódoló felépítését mutatjuk meg. Természetesen ez a kódoló is leírható az előzőekben elmondottak szerint.



15.5. ábra Az INMARSAT-C konvolúciós kódolója

Létezik a konvolúciós kódoló struktúra leírásának egy széles körben elterjedt, rövid formája is, mely a következő: írjunk le annyi, a kényszerhosszúsággal egyező hosszúságú bináris vektort, amennyi a kimenetek száma. Az egyes vektorokban az egyes helyértékeken a szerint írunk 0-at vagy 1-et, hogy a kérdéses kimenethez a shift-regiszter bemenete illetve kimenetei milyen súllyal járulnak hozzá. A legelső bit (a bal szélső) a bemenethez, a legutolsó (a jobb szélső) az utolsó tároló kimenethez tartozik. A 15.5. ábra szerinti kódolóra ez a két vektor : [1011011] ill. [1111001]. Bontsuk a vektorok biteit hármask csoportokra jobbról kezdve: [1 011 011] ill. [1 111 001], majd a hármask csoportok bináris számait írjuk át oktális formátumra. Így két számmal leírhatjuk a kimenetet: [133], [171].

### 15.3. A dekódolás ( a Viterbi-dekóder)

A konvolúciós kódolás inverzét a dekódolást a Viterbi-dekóderrel valósítjuk meg. A Viterbi-dekóder egy algoritmus, amelyben sorozatos feltételes döntések után hozzuk meg az üzenetre vonatkozó végső döntésünket (nevét felfedezőjéről kapta).

Mielőtt az algoritmusra rátérnénk definiáljuk két azonos dimenziójú bináris vektor Hamming távolságát. Két azonos dimenziójú, bináris vektor Hamming távolságán ( $d$ ) azon koordináták számát értjük, melyekben a két vektor különbözik.

$$d = HDST[v_1, v_2] = HDST[v_2, v_1] \quad (15.3.)$$

<b>HDST</b>		$v_2$ (bin.)			
		[0,0]	[0,1]	[1,0]	[1,1]
$v_1$ (bin.)	[0,0]	0	1	1	2
	[0,1]	1	0	2	1
	[1,0]	1	2	0	1
	[1,1]	2	1	1	0

15.4. Táblázat A két darab kétdimenziós bináris vektor ( $v_1, v_2$ ) Hamming távolsága

A dekódolás filozófiájának megértése céljából válasszunk először blokkos adatátvitelt! Legyen a felhasznált időrések száma  $N$ , (az előző példánál maradván  $N = 16$ ).

Rajzoljuk most le a kódoló trellis-ét  $N$ -szer (mint ahogy azt a 15.4 ábrában megtettük)! Ezen a gráfon a 0-ik időrés kezdetétől az  $N$ -ik időrés végéig haladva  $2^N$  számú különböző utat jelölhetünk ki. A kérdés az, hogy a dekódolóban ezen sokaságból hogyan válasszuk ki azt az egyetlen utat ami a kódolóban megvalósult?

Ha a gráfban kiválasztunk egy  $i$ -ik utat, az út  $n$ -ik trellisének két csomópontja közötti átmenethez mindig megmondható, hogy milyen kimeneti  $v_i(n)$  érték tartozna. Ugyanakkor a dekódolóban rendelkezésre áll a vett  $v_{be}(n)$  sorozat is. Így minden időrésben ki tudjuk számolni az éppen feltételezett úthoz tartozó kód és a vett kód közötti Hamming távolságot.

$$d_i(n) = HDST[v_i(n), v_{be}(n)] \quad (15.4.)$$

A teljes  $i$ -ik utat tekintve, az erre az útra jellemző mértéket definiálhatunk, amit akkumulált távolságnak (*accumulated distance*) nevezünk:

$$D_i = \sum_{n=1}^N d_i(n) \quad (15.5.)$$

Tételezzük fel, hogy a kódolt  $2N$  számú bitet egy memóriamentes szimmetrikus csatornán (BSC) továbbítottuk, mely csatornában a bithiba valószínűsége  $p_{berr}$ . A gyakorlatban a csatornák bithiba valószínűsége kicsi (tipikusan  $p_{berr} < 10^{-3}$ ), amiből az következik, hogy az átvitel majdnem mindig hiba mentes, és ezért a kevés hiba valószínűsége jóval nagyobb, mint a több hibát is tartalmazó eseményeké.

Ha az összes lehetséges út közül azt az utat választjuk, amelyiknek az akkumulált távolsága a minimális, akkor a lehető legvalószínűbb utat választottuk. A helyes út kiválasztási valószínűségének a maximalizálását tehát úgy végezzük, hogy a “legjobban hasonlító” utat fogjuk megkeresni (*maximum likelihood method*).

Vegyük észre, hogy a dekóder kimeneti értékéről ezen módszer szerint nem minden időrésben döntünk (*hard decision* = kemény döntés), hanem a teljes út ismeretében (*soft*

*decision = lágy döntés*). Így ha egy időrésben átviteli hiba lép fel, az nem feltétlenül jelenik meg a demodulátor kimenetében.

Az előzőekben leírt módszerrel kapcsolatosan azonban lényeges kifogások merülnek fel. Az összes lehetséges útra kiszámítani az akkumulált távolságot, és azok közül kiválasztani a megfelelőt, túlságosan sok számítási kapacitást és memóriát igényel. A blokkos adatátvitel meg jelentős információ átviteli késleltetést okoz (különösen hosszabb blokkok esetében).

## 15.4. A Viterbi algoritmus

A Viterbi algoritmus a fentebb leírt problémákra megoldást ad. Folytonos adatátvitel esetén is működik, azaz nem kell a blokkos szervezés így a késleltetése is kicsi marad. Az algoritmust két fő eljárásra bonthatjuk fel:

### 1. A túlélő utak megkeresése

Az akkumulált távolságot a dekódolás kezdetétől fogva számítani lehet az éppen futó időrésig. Azt is észrevehetjük, hogy nem kell az összes lehetséges utat számolni, elég csak a legesélyesebbeket (a túlélőket) feljegyezni (*a túlélési út = survival path*). Azokat az utakat, melyeknél a távolságmérték nagy, egyszerűen eldobjuk, mert ezek nem túlélők.

A túlélő utakat egy véges méretű, kétdimenziós (**State\_From**) tömbben fogjuk nyilvántartani. A tömb sorainak a száma a lehetséges állapotok számával egyezzen, az oszlopok száma **egy vonuló időablak**  $L$  hosszúsága legyen. Az  $L$  legyen a kényszerhosszúság 3-4 -szerese. (Mint látni fogjuk a döntés késleltetése ezzel az  $L$ -lel fog megegyezni.)

Hogyan lehet egy ilyen tömbben az utakat nyilvántartani? Minden időrésben a tömb 0. oszlopát fogjuk kitölteni úgy, hogy az állapotokon (a trellis csomópontjain) feltétel nélkül végigmegyünk és meghatározzuk azt, hogy ebbe a csomópontba **nagyobb valószínűséggel** melyik csomópontból érkezhettünk. A tömbbe tehát egy pointert helyezünk el, ami a tömb előző időrésben kitöltött oszlopának valamelyik sorára mutat.

A nagyobb valószínűségű előző állapotot az akkumulált távolságokat tartalmazó (**Acc\_Dist**) egydimenziós tömb alkalmazásával fogjuk kijelölni. A tömb mérete az állapotok számával egyezzen meg.

A Viterbi-dekóderben rekonstruálni próbáljuk a kódoló állapotgépét. Az  $n$ -edik időrésben algoritmus első lépéseként feltétel nélkül végigmegyünk minden állapoton (példánkban:  $s = 0,1,2,3$ ), megvizsgálva azt, hogy az éppen vizsgált állapotba melyik állapotból érkezhettünk a legnagyobb valószínűséggel.

A trellist nézve láthatjuk, hogy minden csomópontba (esetünkben most) két él fut be. Rendeljük az egyes befutó élekhez egy-egy paraméter értéket ( $p = 0$  és  $1$ ).

A kódolót leíró igazságtábla (15.1. Táblázat) segítségével meghatározhatjuk azt a függvényt, amelyik megadja, hogy egy tetszőleges  $s(n)$  állapotba (a  $p$  paraméter különböző értékeit sorba véve) mely állapotokból érkezhettünk:

$$s_p(n-1) = \text{SPS}[p, s(n)] \quad (15.6.)$$

Szükségünk lesz még arra a függvényre is, amelyik megadja, hogy a fenti állapotátmenethez milyen kimeneti adat tartozna. Ez a függvény is a 15.1. Táblázatból felírható.

$$v_p(n-1) = \text{VPS}[p, s(n)] \quad (15.7.)$$

A kérdéses függvények a példánkban:

<i>SPS</i>		<i>s(n)</i>			
		0	1	2	3
<i>p</i>	0	0	0	1	1
	1	2	2	3	3

<i>VPS</i>		<i>s(n)</i>			
		0	1	2	3
<i>p</i>	0	0	3	2	1
	1	3	0	2	1

15.5.és 15.6. Táblázat Az  $s(n-1)=SPS[p,s(n)]$  és az  $v(n-1)=VPS[p,s(n)]$  függvény

A fentebb bevezetett függvényekkel most már formalizálhatjuk az algoritmust. Első lépésként egy egymásba ágyazott kettős ciklust fogalmaztunk meg. A külső ciklusban végig megyünk a lehetséges állapotokon ( $s = 0,1,2,3$ ), belsőben a paraméter ( $p = 0,1$ ) értékeken.

A belső ciklust a könnyebb követhetőség érdekében itt most kibontjuk. Így a külső ciklus maga az alábbi számításokat tartalmazza:

Az  $s$ -edik állapotba  $s_0$  és  $s_1$ -ből érkezhettünk ( $p = 0,1$ ).

$$s_0 = SPS[0, s] \quad s_1 = SPS[1, s] \quad (15.6.)$$

A fenti állapot-átmenetekhez a  $v_0$  és  $v_1$  kimenetek tartoznának:

$$v_0 = VPS[0, s] \quad v_1 = VPS[1, s] \quad (15.7.)$$

A vett  $v_{in}$  üzenet és a fenti két kimenet Hamming távolságai a ( $d_0$  és  $d_1$ ):

$$d_0 = HDST[v_0, v_{ibe}] \quad d_1 = HDST[v_1, v_{be}] \quad (15.8.)$$

Határozzuk meg az  $s_0$  és  $s_1$ -állapotokhoz tartozó akkumulált hiba-távolság nagyságát:

$$Ad_0 = Acc\_Dist[s_0] + d_0; \quad Ad_1 = Acc\_Dist[s_1] + d_1 \quad (15.9.)$$

A nagyobb valószínűségű eseményre olyan módon döntünk, hogy a kisebb akkumulált hiba-távolságot választjuk. (A kevesebb hibának nagyobb a valószínűsége.)

Ez a döntés két eredményt szolgáltat:

- Az  $s$ -edik állapot új akkumulált hibája a kisebb ágon érkező hiba lesz,
- A  $State\_From(s,0)$  helyen rögzítjük, hogy ebbe az állapotba melyikből érkezhettünk.

$$\begin{aligned} \text{If } Ad_0 < Ad_1 \text{ Then } \{ & DST[s] := Ad_0; State\_From[s, 0] := s_0 \} \\ \text{Else } \{ & DST[s] := Ad_1; State\_From[s, 0] := s_1 \} \end{aligned} \quad (15.10.)$$

Az új akkumulált hibát először a  $DIST[s]$  segédváltozóba írjuk, hogy a ciklus lejártá előtt ne rontsuk el  $Acc\_Dist[s]$  értékét a további  $s$ -ek számításához. A külső ciklus lejártá után persze átmásoljuk ezeket az értékeket az  $Acc\_Dist$  tömbbe.

*Procedure Survival\_Path*(vbe);

```
{ for s:= 0 to 3 do
    {
        s0 := SPS[ 0, s ];
        v0 := VPS[ 0, s ];
        d0 := HDST(vbe,v0);
        s1 := SPS[ 1, s ];
        v1 := VPS[ 1, s ];
        d1 := HDST(vbe,v1);
```

```

Ad0:= Acc_Dist[s0]+d0;           Ad1:= Acc_Dist[s1]+d1;
If Ad0 < Ad1
    Then { Dist[s] := Ad0; State_From[s,0] := s0; }
    Else { Dist[s] := Ad1; State_From[s,0] := s1; };
};

```

for s := 0 to 3 do { **Acc\_Dist**[s] := **Dist**[s] };

};

Az előzőekben elkezdett példánkat folytatva, a fenti algoritmus a dekóderben az állapotváltozók értékeire a 15.6. ábra szerinti eredményt adja. ( Az **Acc\_Dist** = [0,7,7,7] kezdő értékkel indítva, mivel a kódoló az s=0-ból indul.)

<i>n</i>	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
<i>u</i>		1	0	1	0	1	1	0	0	1	0	1	1	0	1	0	0
<i>v<sub>be</sub></i>		3	2	0	2	0	1	1	3	3	2	0	1	1	0	2	3

**Acc\_Dist**[*s*]

<i>n</i>	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
<i>s</i> =0	0	2	3	2	3	2	3	3	0	2	3	2	3	3	2	3	0
<i>s</i> =1	7	0	3	0	3	0	3	3	2	0	3	0	3	3	0	3	2
<i>s</i> =2	7	8	0	3	0	3	2	0	3	3	0	3	2	0	3	0	3
<i>s</i> =3	7	8	2	3	2	3	0	2	3	3	2	3	0	2	3	2	3

**State\_From**[*s*,*k*]

$s_{\min} = 0$

<i>n</i>	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
<i>s</i> =0		0	0	2	0	2	0	2	2	0	0	2	0	2	2	0	2
<i>s</i> =1		0	0	2	0	2	0	2	2	0	0	2	0	2	2	0	2
<i>s</i> =2		3	1	3	1	3	1	3	3	1	1	3	1	3	3	1	3
<i>s</i> =3		3	1	3	1	3	1	3	3	1	1	3	1	3	3	1	3
<i>k</i>											6	5	4	3	2	1	0

15.6. ábra A dekóder állapotváltozóinak alakulása az első 16 időrésben

Az ábrában a vastagon keretezett részek mutatják a két változónak a 16. időrésben érvényes tartalmát.

## 2. A nyomkövetés

Az algoritmus második fő eljárása az időben visszafelé történő nyomkövetés (*Trace-Back*).

Első lépésként minden időrésben keressük meg az **Acc\_Dist** tömb elemei közül a minimális értékűt, melynek sorszáma legyen  $s_{\min}$ . Az aktuális időrésben ez a legvalószínűbb állapotot jelöli ki. Ezzel a mutatóval mutassunk rá a másik tömb **State\_From**[  $s = s_{\min}$ ,  $k=0$  ] elemére. Innen egy újabb mutatót olvashatunk ki, ami a táblázat eggyel korábbi, ( $k=1$ ) oszlopának valamelyik sorára ( $s=2$ ) mutat. Ezt az utolsó lépést  $L = 7$ -szer ismételve, az

időben visszafelé nyomon tudjuk követni (*Trace\_Back*) a legvalószínűbb állapotok alakulását.

A nyomkövetés lépéseit belerajzolva a 15.6. ábrába, felismerhetjük a korábbi példánkban szereplő kódoló állapot-átmeneteit (lásd 15.4. ábra). A fenti algoritmus láthatóan alkalmas arra, hogy rekonstruálja a dekóderben a kódoló viselkedését.

A  $k = 6$ -ik lépés után a kiolvasott mutató az 1-es. Az  $s = 1$  állapot páratlan sorszámu. Páratlan sorszámu állapotba a kódoló csak akkor érkezhets, ha a küldött üzenet a  $(16-L)$ -ik (azaz a 9-ik) időrésben az  $u = 1$  érték volt.

*Prucedure Trace\_Back;*

```
{
    Dmin := 32767;
    for s := 0 to 3 do
        { if Acc_Dist[s] ≤ Dmin then { Dmin := Acc_Dist[s]; s_min := s };
        };
    s_old:=s_min;
    for k:= 0 to (L-1) do
        { s_new := State_From( s_old, k ); s_old:=s_new ; }
        };
    u = (s_new) AND (#1);      { s_new páros vagy páratlan ? }
};
```

A dekóderben az  $n$ -ik időrésben végrehajtva a fenti eljárást, az  $u$  változóban az  $(n-L)$ -ik időréshez tartozó üzenetbitet kapjuk.

A fenti eljárásokat úgy írtuk le, hogy minden időrésben a **State\_From** tömb 0-ik (aktuális) oszlopát töltöttük fel. A dekódolás végén ezért szükség van még egy, a késleltetést megvalósító eljárás alkalmazására is:

*Procedure Delay;*

```
{ for s := 0 to 3 do
    { for k := (L-1) downto 1 do { State_From[s,k] := State_From[s, (k-1)] } }
};
```

Valós idejű alkalmazásokban ezen utóbbi eljárás elhagyható, ha cirkuláris címzést használunk.

Nézzük meg végül, hogy a választott példánkban **hogyan javít hibát** a *Viterbi-dekóder*!

Tételezzük fel, hogy a csatornában az átvitel során a 13. időrésben hiba történt és a  $v_{in}(13) = 1_{dec} = [0,1]_{bin}$  érték módosult a  $v_{in}(13) = 3_{dec} = [1,1]_{bin}$  értékre (1 bit hiba).

A numerikus viszonyokat a 15.7. ábrán követhetjük. A 12. időrésig a viszonyok természetesen ugyan azok, mint a hiba nélküli esetben. A 13. időrésben jelentkezik a hiba, amikor is az **Acc\_Dist** tömbben az utolsó legkisebb érték az (1), az  $s_{min} = 3$  állapotot jelöli ki.

A nyomkövetést az időben visszafelé végrehajtva a **State\_From** tömbben (szaggatott vonal) láthatjuk, hogy a nyom az  $n=12$ -ik időrésben csatlakozik a hiba nélküli eset nyomához tartozó túlélési úthoz (folytonos vonal).

A késleltetett döntés ezek után már helyes lesz, mert a kijelölt útnak csak az eleje hibás, a vége viszont már helyes, ahol a küldött üzenetre vonatkozó döntést végezzük.



Ugyanez érvényes a 14. a 15. és a 16. időrésre is. (A 16. időrés nyomkövetését rajzoltuk be az ábrába folytonos vonallal.) Az  $L$  késleltetést azért választottuk példánkban a kényszerhosszúság 3.5-szeresére, hogy a nyomkövető algoritmusnak legyen elég tere rátalálni a helyes nyomra. Az  $L$  értékének növelése elvileg javítja a dekóder hibajavító képességét, de az 5-6-szoros méret felett a járulékos számítási és memória igény már nem áll arányban az eredményezett javulással.

A példánkban észrevehetjük, hogy az **Acc\_Dist** tömbben a legkisebb érték a bekövetkezett hibák számát adja. Folytonos működést feltételezve, ahogy a hibák száma szaporodik, úgy fenn áll a túlszordulás veszélye (fixpontos változó esetén). A túlszordulás elkerülhető, ha a minimum megkeresése után a tömb minden eleméből kivonjuk a minimumot és ezt a lecsökkentett értéket tároljuk el a következő időrésre.

$n$	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
$u$		1	0	1	0	1	1	0	0	1	0	1	1	0	1	0	0
$v_{be}$		3	2	0	2	0	1	1	3	3	2	0	1	3	0	2	3

**Acc\_Dist[s]**

$n$	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
$s=0$	0	2	3	2	3	2	3	3	0	2	3	2	3	2	2	3	1
$s=1$	7	0	3	0	3	0	3	3	2	0	3	0	3	3	1	3	3
$s=2$	7	8	0	3	0	3	2	0	3	3	0	3	2	1	2	1	3
$s=3$	7	8	2	3	2	3	0	2	3	3	2	3	0	1	2	2	3

**State\_From[s,k]**

$n$	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
$s=0$		0	0	2	0	2	0	2	2	0	0	2	0	2	0	2	2
$s=1$		0	0	2	0	2	0	2	2	0	0	2	0	0	2	2	2
$s=2$		3	1	3	1	3	1	3	3	1	1	3	1	3	3	1	3
$s=3$		3	1	3	1	3	1	3	3	1	1	3	1	3	3	3	3
$k$											6	5	4	3	2	1	0

15.7. ábra A dekóder állapotváltozóinak alakulása átviteli hiba esetén