SVEUČILIŠTE U ZAGREBU
Fakultet elektrotehnike i računarstva

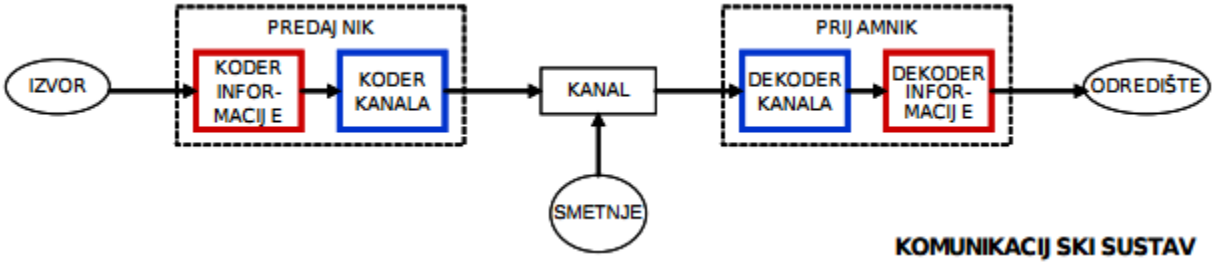
Predmet: Teorija informacije
Ak. godina: 2010./2011.

Laboratorijske vježbe:
IZVJEŠTAJ

Grupa {NSK19}:

1. {Leo Osvald, 0036443136}
2. {Filip Špoljar, 0036442950}
3. {Josip Bagarić, 0036443211}
4. {Karlo Zanki, 0036443227}

Zadatak



KOMUNIKACIJ SKI SUSTAV

Na sl. 1 zadan je komunikacijski sustav koji se sastoji od izvora informacije, kodera informacije, kodera kanala, kanala na koji utječu smetnje, dekodera kanala, dekodera informacije i odredišta. Izvorište generira simbole sa zadanim vjerojatnostima pojavljivanja. Te simbole potrebno je kodirati nekom od metoda entropijskog kodiranja. Nakon toga, novonastale slijedove simbola (tzv. kodirana poruka) potrebno je kodirati nekom od metoda zaštitnog kodiranja (dobivamo zaštitno kodiranu poruku). Zaštitno kodirana poruka prenosi se komunikacijskim kanalom u kojem:

i) u jednom slučaju nema smetnji, dok

ii) u drugom slučaju smetnje djeluju na poslane podatke.

Podatke koji su prošli kroz komunikacijski kanal potrebno je dekodirati na odredištu.

Izvorište:

p(a) = 0.4, p(b) = 0.1, p(c) = 0.1, p(d) = 0.2, p(e) = 0.2

Generirajte slučajni slijed od 10000 simbola (u ovisnoti o rednom broju izvorišta) u kojem će se simboli pojavljivati s zadanim vjerojatnostima.

Metoda entropijskog kodiranja:

Huffmanovo kodiranje

Metoda zaštitnog kodiranja:

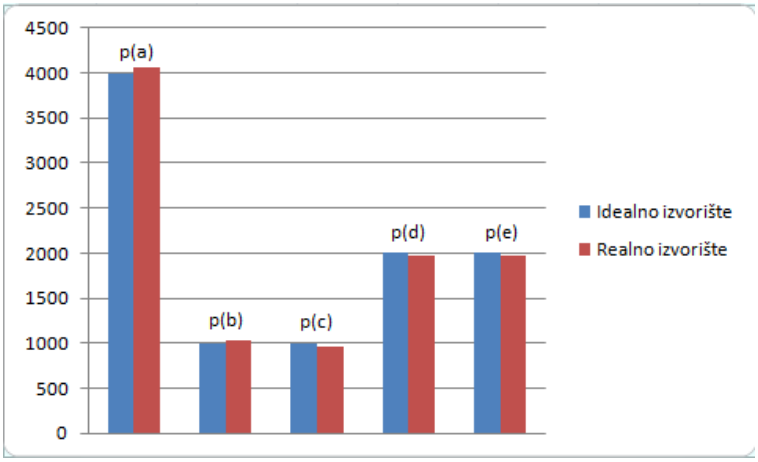
Hammingov kôd [15, 11, 3]

Kanal:

p = 1/20

Rješenje

Informacijska svojstva kanala



2. Odredite vjerojatnosti pojavljivanja simbola u generiranom slijedu:
   $p(a) = 0.404400$
   $p(b) = 0.098500$
   $p(c) = 0.102500$
   $p(d) = 0.198800$
   $p(e) = 0.195800$

3. Izračunajte entropiju izvorišnog skupa simbola.
   H(izv.)= 2.130677 [bita/simbolu]

4. Izračunajte entropiju odredišnog skupa simbola u slučaju da se koristi zaštitno kodiranje.
   H(odr.)= 2.111224 [bita/simbolu]

5. Izračunajte transinformaciju u slučaju da se koristi zaštitno kodiranje:
   $I(X;Y)$ = 4.242358 [bita/simbolu]

6. Izračunajte entropiju odredišnog skupa simbola u slučaju da se NE koristi zaštitno kodiranje.
   H(odr.) = 2.082651 [bita/simbolu]

7. Izračunajte transinformaciju u slučaju da se NE koristi zaštitno kodiranje:
   $I(X;Y)$ = 3.059561 [bita/simbolu]

Zaštitno kodiranje

1. Statistički dobivena udaljenost za zadani zaštitni kod iznosi: $d(K) = 3$

2. Računski dobivena udaljenost za zadani zaštitni kod iznosi $d(K) = 3$

3. Kod može ispraviti 1 pogrešku.

4. Kod može detektirati 2 pogreške.

5. Primjer u kojemu se pristigla poruka s greškom ne bi otkrila, a time ni dekodirala ispravno:

10011001101 se kodira u 1110001010011101 i to se pošalje kanalom. Ako dođe do pogreške na na 3., 5. i 6. bitu poruke. Dekoder to ne može ispraviti nit otkriti. Poruka s greškom na 3., 5. i 6. bitu bi izgledala ovako : 1100111010011101

Vjerojatnost ispravnog dekodiranja poruke iznosi (statistički dobiven podatak): 0.2584000

6. Kodna brzina zaštitnog kôda iznosi: $R = k/n = 11/15 = 0.7333333$
7. Generirajuća matrica **G**:

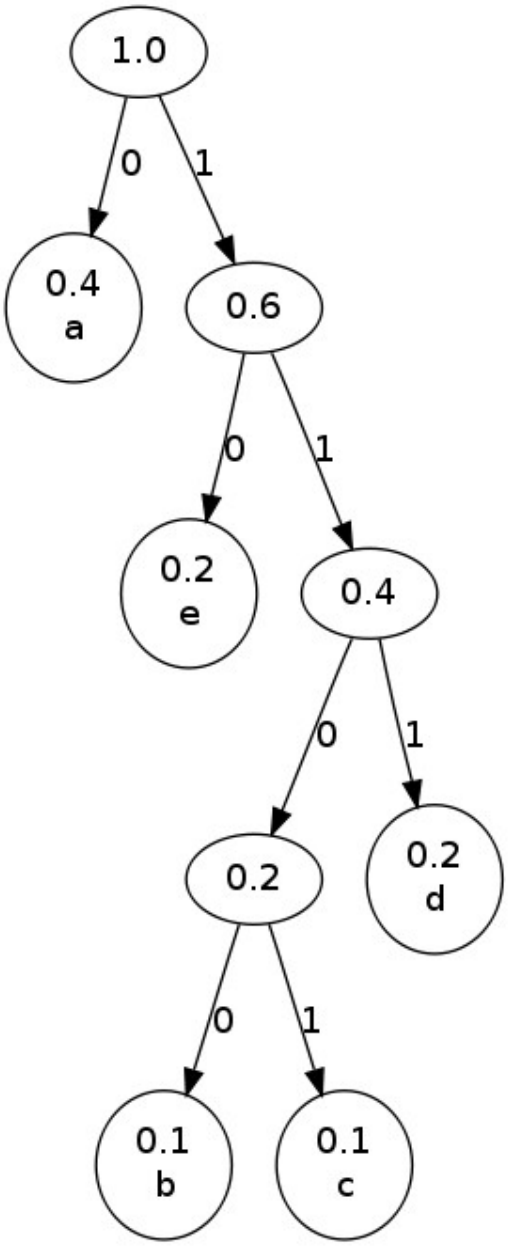
```
111000000000000
100110000000000
010101000000000
110100100000000
100000011000000
010000010100000
110000010010000
000100010001000
100100010000100
010100010000010
110100010000001
```

8. Matrica provjere pariteta **H** za zadani kod:

```
101010101010101
011001100110011
000111100001111
000000011111111
```

Entropijsko kodiranje

Prosječna duljina kodne riječi: *L = 2.3777 [bita/simbolu]*



Huffman(a) = 0
Huffman(e) = 10
Huffman(b) = 1100
Huffman(c) = 1101
Huffman(d) = 111

Algoritam prvo sortira simbole po njihovoj vjerojatnosti i onda uzima ona dva simbola sa najmanjim vjerojatnostima. Ako ih je više sa jednakom najmanjom vjerojatnošću, uzimaju se oni koji su prvi po abecedi. Nakon toga, odabranim simbolima dodijelimo simbole 1 i 0, njihove vjerojatnosti se zbroje i kombiniramo ih u jedan nadsimbol. Algoritam se ponavlja dok ne dođemo do zadnja dva simbola kojima je zbroj vjerojatnosti jednak 1.00 kada se algoritam zaustavlja.

Opisi korištenih algoritama

Sve komponente radili smo na algoritamski efikasan način, s malom vremenskom i memorijskom složenošću. Slijedi analiza vremenske složenosti pojedinih komponenti:

- S - veličina skupa simbola (broj različitih simbola)
- L - broj bitova/znakova na ulazu/izlazu u komponentu (engl. ***L**ength*)
- N - veličina bloka kod Hamminga
- K - broj podatkovnih bitova koji se kodira u blok kod Hammingovog kodiranja = N - $\log_2(N)$

Komponenta	Vremenska složenost (big-O)	Memorijska složenost (big-O)
Izvor	(L + S) log S	S
Huffmanov koder	L + S log S	S
Hammingov koder	L log N	N
Generator vektora pogreške	L	1
Kanal	L	1
Hammingov dekoder	L log N	N
Huffmanov dekoder	L + S log S	S

Izvor

Vremenska složenost komponente izvora prije svega ovisi o učinkovitosti generacije simbola. Potrebno je ostvariti na efikasan način simulator slučajne varijable. Mi smo ga ostvarili na način da smo za svaki od S simbola stvoril po jedan pretinac čija je veličina proporcionalna njegovoj vjerojatnosti pojavljivanja. Po dodavanju ishoda u slučajnu ishoda odmah se osvježuje polje koje pamti sumu veličina pretinaca do svakog indeksa. Budući da dodajemo na kraj, samo trošimo jednu operaciju za to pa je složenost dodavanja konstanta.
Kad dodamo sve simbole, sada možemo simulirati ishod slučajne varijable, tako da odaberemo slučajan broj u intervalu [0, zbroj_veličine_svih_pretinaca). Kada odredimo taj slučajan broj, preostaje binarnim pretraživanjem naći indeks pretinca koji sadrži taj broj (pretinci predstavljaju intervale). Ovime dobivamo vremensku složenost simulacije jednog ishoda O(log S).

```
T get() const {
	if (sum_.empty())
		return not_found_val_;
	int totalSum = sum_.at(sum_.size() - 1);
	if (totalSum <= 0)
		return not_found_val_;
	int target = random<int>(seed_) % totalSum;
	int lo = 0, hi = sum_.size() - 1;
```

```
        while (lo < hi) {
              int mid = lo + (hi - lo - 1) / 2;
              if (target < sum_.at(mid))
                    hi = mid;
              else
                    lo = mid + 1;
        }
        return outcomes_.at(lo);
  }
```

Huffmanov koder i dekoder

Huffmanov koder i dekoder baziraju se na gradnji Huffmanovog stabla. Huffmanovo stablo implementirano je na način da se napravi čvor koji sadrži znak koji predstavlja simbol (koji je jedino bitan ako je taj čvor list u stablu), te vjerojatnost podstabla:

```
  class Node {
        const char symbol_;
        const double probability_;
        const Node* const left_;
        const Node* const right_;

      public:

        Node(char symbol, double probability)
        : symbol_(symbol), probability_(probability),
          left_(NULL), right_(NULL) {
        }

        Node(const Node* child1, const Node* child2)
        : symbol_(0), probability_(child1->probability_ +
             child2->probability_),
          left_(child1), right_(child2) {
        }
       ...
  };
```

Izgradnja Huffmanovog stabla započinje tako da se u prioritetni red ubaci po jedan čvor za svaki simbol koji ima odgovarajuće postavljenu vjerojatnost. Zatim se ponavlja iterativni postupak u kojem se izbacuju 2 čvora iz prioritetnog reda s najmanjom vjerojatnošću, te se radi novi čvor koji je roditelj ovih dvaju čvorova a ima vjerojatnost jednaku njihovom zbroju (vidi drugi konstruktor u kodu gore). Ako se za prioritetni red uzme npr. binarna gomila ili crveno-crno stablo, onda izbacivanje čvora s najmanjom vjerojatnošću ima vrem. složenost O (log S), pa je složenost algoritma izgradnje Huffmanovog stabla jednaka O (S log S).

```
HuffmanTree(const ProbabilityMap& symbol_probabilities) {
       using namespace std;
```

```
    // prioritetni red ostvaren preko crveno-crnog stabla
    // prioritet je definiran komparatorom (vidi komparator)
    multiset<Node*, Comparator> priority_queue;
    FOREACH (it, symbol_probabilities) {
          priority_queue.insert(new Node(it->first, it->second));
    }

    // OPREZ: poseban slucaj ako je 1 simbol - tad stvaramo 2 ista cvora
    if (priority_queue.size() == 1) {
          Node* node = *priority_queue.begin();
          priority_queue.insert(new Node(node->symbol(), 0));
    }

    root_ = NULL;
    while (priority_queue.size() > 1U) {
          // uzmi dva cvora iz prioritetnog reda s najmanjom vjerojatnoscu
          Node* min1 = extractMin(priority_queue);
          Node* min2 = extractMin(priority_queue);
          // i dodijeli im zajednickog roditelja (koji je korijen stabla)
          // koji ima vjerojatnost jednak zbroju vjerojatnosti djece
          root_ = new Node(min1, min2);
          priority_queue.insert(root_);
    }
}
```

Huffmanov koder radi tako da prvo izgradi Huffmanovo stablo i zatim prođe kroz sve čvorove i za mapira sve znakove u binarne nizove. Binarni niz se dobiva tako da se krene od korijena stabla i nadodaje se nula odnosno jedinica, ovisno o tome da li se prelazi na lijevo dijete ili desno dijete. Za ostvarenje mapiranja se može kod velikog broja simbola koristiti hash-tablica, a mi smo koristili običan niz (što je u suštini isto, samo troši malo više memorije). Kodiranje se ostvaruje tako da se čita znak po znak na ulazu i pogleda se u mapu što treba ispisati na izlaz. Budući da je pogled u mapu vrem. složenosti O(1), cjelokupno kodiranje je složenosti O(L). Tome valja pridodati vrijeme izgradnje Huffmanovog stabla pa dobivamo ukupnu vrem. složenost O(L + S log S).

Huffmanov dekoder je još jednostavniji. On niti ne treba nikakvu mapu. Kako bitovi redom dolaze na ulazu, on se samo šeće po već izgrađenom Huffmanovom stablu i kad dođe do lista stabla na izlaz pošalje odgovarajući simbol (koji je zapamćen u tom čvoru). Kad se jednom izgradi stablo dakle, kodiranje se vrši u linearnom vremenu (ovisno o broju bitova na ulazu). Stoga je algoritam dekodiranja ukupne vremenske složenost O(L + S log S).

Hammingov koder i dekoder

Rad komponenata Hammingovog koder i dekodera zasniva se na izgradnji Hammingovog koda za određeni binarni niz. Hammingov kod se može izgraditi na dva načina: iz drugog Hammingovog koda (tada se svi bitovi preslikavaju), ili na temelju podatkovnih bitova (tada se na određena mjesta dodaju zaštitni bitovi).

```
HammingCode(BinaryCode binary_code, bool hamming) {
       if (hamming) { // ako su ovo bitovi za hammingov kod, samo ih kopiraj
             setBits(binary_code);
       } else { // inace na potencije broja 2 dodaj nule, a ostale kopiraj
             int data_index = 0;
             for (int pos = 0; data_index < binary_code.size(); ++pos) {
                   if (isParityBit(pos))
                         append(false);
                   else {
                         bool bit = binary_code.isSet(data_index++);
                         append(bit);
                         updateParityBits(pos, bit);
                   }
             }
       }
    }
```

Kad zaštitno kodiramo niz podatkovnih bitova (vidi else u kodu iznad), ako umećemo bit na indeks na koji treba doći paritetni bit, stavljamo 0 jer znamo da prošli bitovi nisu mogli utjecati na njega (tj. on ih ne štiti). U protivnom, umećemo podatkovni bit te pogledamo koji to zaštitni bitovi štite njega i preokrenemo ih ako je potrebno. To radi metoda *updateParityBits*.

```
      void updateParityBits(int index, bool changed) {
             // ako se bit nije promijenio, onda se nije niti jedan paritetni
             if (!changed)
                   return ;

             ++index; // svedi na 1-bazirani indeks
             // prodji kroz sve bitove
             for (int low_bit; index > 0; index -= low_bit) {
                   low_bit = lowBit(index);
                   flip(low_bit - 1);
             }
          }
```

Algoritam radi tako da redom odbacuje najniži bit i na taj način prolazi kroz sve potencije broja 2 čija suma daje indeks tog podatkovnog bita, jer su upravo pozicije paritetnih bitova koji štite taj bit. Primijetimo, također, da ako smo upravo stavili 0 kao podatkovni bit (odnosno ako se nije promijenio) da se paritet ne mijenja, pa nema potrebe raditi ovaj postupak (vidi prvi if u metodi gore). Vremenska složenost ove metode je logaritamska, pa je vremenska složenost izgradnje bloka bitova zaštićenog Hammingovim kodom O(K log N).

Koder radi tako da samo izgradi Hammingov kod određene duljine na temelju dijela ulaznog bitovnog niza, te na izlaz pošalje blok duljine N bitova koji je zaštićen Hammingovim kodom. Budući da ako na ulazu imamo L bitova, a kodiramo blokove veličine K, izgradit ćemo N / K blokova. Vremenska složenost algoritma je stoga O(L log N). U memoriji trebamo pamtiti samo trenutni blok koji kodiramo, pa trošimo količinu memoriju proporcionalnu s N.

Dekoder radi tako da uzima sljedove od N bitova koji već jesu zakodirani Hammingovim kodom i samo iz njih izvuče podatkovne bitove. Prije toga, na temelju sindroma popravi bit ako je došlo do greške. Dovoljno je za svaki proći kroz sve paritetne bitove i za svaki provjeriti da li paritet odgovara.

```
int errorSyndrome() const {
	int ret = 0;
	for (int pb_index = parityBitCount() - 1; pb_index >= 0;
--pb_index) {
		bool parity = false;
		for (int i = 1; i <= size(); ++i)
			if ((1 << pb_index) & i)
				parity ^= isSet(i - 1);

		// ako paritet nije paran, onda dodajemo bit (1<<pb_index)
		// da idemo uzlazno bilo bi if(parity) ret |= 1<<pb_index;
		ret = (ret << 1) | parity;
	}

	// ako sindrom upucuje na gresku na indeksu izvan ovog bloka
	// onda je doslo do visestruke greske koju nije moguce ispraviti
	if (ret > size())
		return -1;

	return ret;
    }

    int fix() {
	int error_syndrome = errorSyndrome();

	// ako ima greske i ona se moze ispraviti, popravi je
	if (error_syndrome > 0)
		flip(error_syndrome - 1);

	return error_syndrome;
    }
```

Nakon toga, preostaje proći redom po svim bitovima i poslati na izlaz one koji su podatkovni. Ključno je, dakle, algoritam provjere da li je neki bit paritetan, koji je zaista trivijalan:

```
static inline int lowBit(int x) {
	return x & -x;
    }
static inline bool isPowerOfTwo(int x) {
	return lowBit(x) == x;
}
static inline bool isParityBit(int pos) {
	return isPowerOfTwo(pos + 1);
}
```

Analizator

Analizator izračunava entropiju izvora i odredišta, transinformaciju, vjerojatnosti i broj primljenih i poslanih simbola i sl. na sljedeći način. Kao argumenti u program predaju se nazivi dvaju datoteka koji predstavljaju niz poslanih i primljenih simbola (ili bitova), respektivno.

Vjerojatnosti simbola na izvoru i odredištu

Na temelju broja pojedinih poslanih odnosno primljenih simbola računa se njihova vjerojatnost statistički.

Entropija izvora i odredišta

Entropija izvora računa se na temelju prethodno izračunatih statističkih vrijednosti za vjerojatnost pojave pomoću formule:

$$H(X) = -\sum_{i} p(x_i) log_2(x_i)$$

$$H(Y) = -\sum_{i} p(y_i) log_2(y_i)$$

Transinformacija

Prvo se računaju združene vjerojatnosti tako da se za svaki par (primljeni_simbol$_i$, poslani_simbol$_i$) izračuna broj pojava. Zatim se uzme manji od broja primljenih i poslanih znakova (jer toliko smo puta povećali brojače parova) pa se ti brojači podijele s tom konstantom kako bi se dobila združena vjerojatnost p(x, y). Sad se pomoću formule:

$$I(X;Y) = -\sum_{i,j} p(x_i)p(y_i) log_2(xi,y_i)$$