MAC323 Estruturas de Dados

Y. Kohayakawa (MAC/IME/USP)

2o. Semestre 2008

Aula bônus: papo aleatório

Aula bônus: papo aleatório

Discutiremos o algoritmo de geração de textos aleatórios, discutido e implementado por Kernighan e Pike, em *The practice of programming*.

Um exemplo de texto gerado

A fama de polêmico do cineasta Oliver Stone parece longe de terminar, e o alvo de seu próximo trabalho volta a ser o mesmo que é técnico, não dirigente. Mas nesta terça-feira abriu uma exceção: elogiou o meia Ibson, do Flamengo, e disse que existe chance real de o São Paulo também. Tem de entender o atleta. Ele está sendo correto e elegante, é uma prática do futebol. Não pode estar no clube e vem agradando ao técnico Muricy Ramalho, mas a única dúvida é se os ucranianos vão emprestá-lo só até o fim das férias escolares não aumentará os índices de congestionamento registrados na Cidade. "Acreditamos que aqueles 20% que ficam parados nas férias voltem, mas a única dúvida é se os ucranianos vão emprestá-lo só até o fim das férias escolares não aumentará os índices de congestionamento registrados na Cidade.

O algoritmo com cadeias de Markov

```
w_1, w_2 <- as 2 primeiras palavras do texto
imprima w_1 e w_2
repita:
  w_3 <- um dos sucessores do prefixo w_1 w_2 no texto
         (escolha aleatória)
  imprima w_3
  w_1 <- w_2; w_2 <- w_3
```

Uma implementação (K&P)

```
enum {
        NPREF   = 2,    /* number of prefix words */
        NHASH   = 4093, /* size of state hash table array */
        MAXGEN  = 10000 /* maximum words generated */
};
```

```
typedef struct State State;
typedef struct Suffix Suffix;

struct State {  /* prefix + suffix list */
        char    *pref[NPREF];   /* prefix words */
        Suffix  *suf;                   /* list of suffixes */
        State   *next;                  /* next in hash table */
};

struct Suffix { /* list of suffixes */
        char    *word;                  /* suffix */
        Suffix  *next;                  /* next in list of suffixes *
};
State   *statetab[NHASH];       /* hash table of states */
```

```
const int MULTIPLIER = 31;  /* for hash() */

/* hash: compute hash value for array of NPREF strings */
unsigned int hash(char *s[NPREF])
{
        unsigned int h;
        unsigned char *p;
        int i;

        h = 0;
        for (i = 0; i < NPREF; i++)
                for (p = (unsigned char *) s[i]; *p != '\0'; p++)
                        h = MULTIPLIER * h + *p;
        return h % NHASH;
}
```

```
/* lookup: search for prefix; create if requested. */
/*  returns pointer if present or created; NULL if not. */
/*  creation doesn't strdup so strings mustn't change later. */
State* lookup(char *prefix[NPREF], int create)
{
        int i, h;  State *sp;
        h = hash(prefix);
        for (sp = statetab[h]; sp != NULL; sp = sp->next) {
                for (i = 0; i < NPREF; i++)
                        if (strcmp(prefix[i], sp->pref[i]) != 0) brea
                if (i == NPREF)         /* found it */
                        return sp;
        }
        if (create) { ... }
        return sp;
}
```

```
        [...]
        if (create) {
                sp = (State *) emalloc(sizeof(State));
                for (i = 0; i < NPREF; i++)
                        sp->pref[i] = prefix[i];
                sp->suf = NULL;
                sp->next = statetab[h];
                statetab[h] = sp;
        }
        return sp;
}
```

```
/* build: read input, build prefix table */
void build(char *prefix[NPREF], FILE *f)
{
        char buf[100], fmt[10];

        /* create a format string; %s could overflow buf */
        sprintf(fmt, "%%%ds", sizeof(buf)-1);
        while (fscanf(f, fmt, buf) != EOF)
                add(prefix, estrdup(buf));
}
```

```
/* add: add word to suffix list, update prefix */
void add(char *prefix[NPREF], char *suffix)
{
        State *sp;

        sp = lookup(prefix, 1);  /* create if not found */
        addsuffix(sp, suffix);
        /* move the words down the prefix */
        memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
        prefix[NPREF-1] = suffix;
}
```

```
/* addsuffix: add to state. suffix must not change later */
void addsuffix(State *sp, char *suffix)
{
        Suffix *suf;

        suf = (Suffix *) emalloc(sizeof(Suffix));
        suf->word = suffix;
        suf->next = sp->suf;
        sp->suf = suf;
}
```

```
char NONWORD[] = "\n";  /* cannot appear as real word */

        [...]
        build(prefix, stdin);
        add(prefix, NONWORD);
        [...]
```

```
/* generate: produce output, one word per line */
void generate(int nwords)
{ State *sp; Suffix *suf; char *prefix[NPREF], *w; int i, nmatch;
  for (i = 0; i < NPREF; i++)     /* reset initial prefix */
     prefix[i] = NONWORD;
  for (i = 0; i < nwords; i++) {
    sp = lookup(prefix, 0);
    if (sp == NULL) eprintf("internal error: lookup failed");
    nmatch = 0;
    for (suf = sp->suf; suf != NULL; suf = suf->next)
      if (rand() % ++nmatch == 0) /* prob = 1/nmatch */
        w = suf->word;
    if (nmatch == 0)
      eprintf("internal error: no suffix %d %s", i, prefix[0]);
    [...]
  }
}
```

```
/* generate: produce output, one word per line */
void generate(int nwords)
{
    [...]
    if (strcmp(w, NONWORD) == 0) break;
    printf("%s\n", w);
    memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
    prefix[NPREF-1] = w;
  }
}
```

```
/* markov main: markov-chain random text generation */
int main(void)
{       int i, nwords = MAXGEN;
        char *prefix[NPREF];            /* current input prefix */
        int c; long seed;
        setprogname("markov");
        seed = time(NULL);
        srand(seed);
        for (i = 0; i < NPREF; i++)     /* set up initial prefix */
                prefix[i] = NONWORD;
        build(prefix, stdin);
        add(prefix, NONWORD);
        generate(nwords);
        return 0;
}
```

This completes our C implementation. We will return at the end of the chapter to a comparison of programs in different languages. The great strengths of C are that it gives the programmer complete control over implementation, and programs written in it tend to be fast. The cost, however, is that the C programmer must do more of the work, allocating and reclaiming memory, creating hash tables and linked lists, and the like. [. . .]

C is a razor-sharp tool, with which one can create an elegant and efficient program or a bloody mess.

Kernighan and Pike, *The practice of programming*