November 24, 2020 at 13:23

1. Intro. I'm hurriedly experimenting with a new(?) way to explore the complexity of 4-variable Boolean functions. Namely, I calculate the "footprint" of each function, the set of all first steps by which I know how to evaluate the function in k steps. Then, if the footprints of f and g overlap, I can compute $f \circ g$ in $\mathrm{cost}(f) + \mathrm{cost}(g)$ steps.

I can restrict consideration to the 2^{15} functions that take $(0,0,0,0) \mapsto 0$.

This program extends FCHAINS4 by allowing several additional functions to be precomputed. Those functions appear on the command line, in hexadecimal form.

```
#define footsize 100
#include <stdio.h>
#include <stdlib.h>
  typedef struct node_struct {
    unsigned int footprint[footsize];
    int parent;
    int cost;
    struct node_struct *prev, *next;
  } node;
  node func[1 ≪ 15];
  node head[9];
  int x[100];
  char buf[100];    /* lines of input */
  char name[32 * footsize][16];
  unsigned int ttt;    /* truth table found in input line */
  main(int argc, char *argv[])
  {
    register int c, j, k, r, t, m, mm, s;
    register unsigned int u;
    register node *p, *q, *pp;
    ⟨Read the initial functions 2⟩;
    ⟨Initialize the tables 8⟩;
    for (r = 2; c; r++)
      for (k = (r − 1) ≫ 1; k ≥ 0; k−−) ⟨Combine all functions of costs k and r − 1 − k 3⟩;
    ⟨Answer queries 12⟩;
  }
```

2. ⟨Read the initial functions 2⟩ ≡

```
  m = argc + 3;
  for (k = 1; k ≤ m; k++) {
    if (k ≤ 4) x[k] = #ffff/((1 ≪ (1 ≪ (4 − k))) + 1);
    else if (sscanf(argv[k − 4], "%x", &x[k]) ≠ 1) {
      fprintf(stderr, "Parameter␣%s␣should␣have␣been␣hexadecimal!\n", argv[k − 4]);
      exit(−1);
    }
    if (x[k] > #ffff) {
      fprintf(stderr, "Parameter␣%s␣is␣too␣big!\n", argv[k − 4]);
      exit(−1);
    }
    if (x[k] ≥ #8000) x[k] ⊕= #ffff;
  }
```

This code is used in section 1.

3. ⟨Combine all functions of costs k and $r-1-k$ 3⟩ ≡

```
for (p = head[k].next; p→parent ≥ 0; p = p→next)
  for (q = head[r − 1 − k].next; q→parent ≥ 0; q = q→next) {
    for (j = 0; j < mm; j++)
      if (p→footprint[j] & q→footprint[j]) ⟨Try for breakthru and goto pqdone 6⟩
    ⟨Try for new function 4⟩;
  pqdone: continue;
  }
```

This code is used in section 1.

4. `#define fun(p) ((p) − func)`

⟨Try for new function 4⟩ ≡

```
{
  t = fun(p) & fun(q);
  if (func[t].cost ≥ r) ⟨Update the table for cost r 5⟩;
  t = fun(p) & (∼fun(q));
  if (func[t].cost ≥ r) ⟨Update the table for cost r 5⟩;
  t = (∼fun(p)) & fun(q);
  if (func[t].cost ≥ r) ⟨Update the table for cost r 5⟩;
  t = fun(p) | fun(q);
  if (func[t].cost ≥ r) ⟨Update the table for cost r 5⟩;
  t = fun(p) ⊕ fun(q);
  if (func[t].cost ≥ r) ⟨Update the table for cost r 5⟩;
}
```

This code is used in section 3.

5. ⟨Update the table for cost r 5⟩ ≡

```
{
  pp = &func[t];
  if (pp→cost > r) {
    if (pp→cost ≡ 8) c−−;
    pp→next→prev = pp→prev, pp→prev→next = pp→next;
    pp→cost = r, pp→parent = (fun(p) ≪ 16) + fun(q);
    for (j = 0; j < mm; j++) pp→footprint[j] = 0;
    pp→next = head[r].next, pp→prev = &head[r];
    pp→next→prev = pp, pp→prev→next = pp;
  }
  for (j = 0; j < mm; j++) pp→footprint[j] |= p→footprint[j] | q→footprint[j];
}
```

This code is used in section 4.

6. ⟨ Try for breakthru and **goto** *pqdone* 6 ⟩ ≡

```
{
  t = fun(p) & fun(q);
  if (func[t].cost ≥ r − 1) ⟨ Update the table for cost r − 1 7 ⟩;
  t = fun(p) & (∼fun(q));
  if (func[t].cost ≥ r − 1) ⟨ Update the table for cost r − 1 7 ⟩;
  t = (∼fun(p)) & fun(q);
  if (func[t].cost ≥ r − 1) ⟨ Update the table for cost r − 1 7 ⟩;
  t = fun(p) | fun(q);
  if (func[t].cost ≥ r − 1) ⟨ Update the table for cost r − 1 7 ⟩;
  t = fun(p) ⊕ fun(q);
  if (func[t].cost ≥ r − 1) ⟨ Update the table for cost r − 1 7 ⟩;
  goto pqdone;
}
```

This code is used in section 3.

7. This code is not executed when $k = 0$, because q's footprint is zero in that case.

⟨ Update the table for cost $r - 1$ 7 ⟩ ≡

```
{
  pp = &func[t];
  if (pp→cost > r − 1) {
    if (pp→cost ≡ 8) c−−;
    pp→next→prev = pp→prev, pp→prev→next = pp→next;
    pp→cost = r − 1, pp→parent = (fun(p) ≪ 16) + fun(q);
    for (j = 0; j < mm; j++) pp→footprint[j] = 0;
    pp→next = head[r − 1].next, pp→prev = &head[r − 1];
    pp→next→prev = pp, pp→prev→next = pp;
  }
  for (j = 0; j < mm; j++) pp→footprint[j] |= p→footprint[j] & q→footprint[j];
}
```

This code is used in section 6.

8. ⟨ Initialize the tables 8 ⟩ ≡

```
for (p = &func[2]; p < &func[#8000]; p++) (p − 1)→next = p, p→prev = p − 1, p→cost = 8;
func[1].cost = 8;
for (k = 0; k ≤ 8; k++) head[k].parent = −1, head[k].next = head[k].prev = &head[k];
head[0].next = head[0].prev = &func[0];
func[0].next = func[0].prev = &head[0];
head[8].next = &func[1], func[1].prev = &head[8];
head[8].prev = &func[#7fff], func[#7fff].next = &head[8];
⟨ Initialize the functions of cost 0 9 ⟩;
⟨ Initialize the functions of cost 1 10 ⟩;
```

This code is used in section 1.

9. ⟨ Initialize the functions of cost 0 9 ⟩ ≡

```
for (k = 1; k ≤ m; k++) {
  p = &func[x[k]];
  if (p→cost ≡ 0) continue;
  p→next→prev = p→prev, p→prev→next = p→next;
  p→cost = 0;
  p→next = head[0].next, p→prev = &head[0];
  p→next→prev = p, p→prev→next = p;
}
c = (1 ≪ 15) − 1 − m;
```

This code is used in section 8.

10. ⟨ Initialize the functions of cost 1 10 ⟩ ≡

```
s = 0;
for (r = 2; r ≤ m; r++)
  for (k = 1; k < r; k++) {
    t = x[k] & x[r], sprintf(name[s], "%d&%d(%04x)", k, r, t);
    ⟨ Update for cost 1 11 ⟩;
    t = x[k] & (∼x[r]), sprintf(name[s], "%d>%d(%04x)", k, r, t);
    ⟨ Update for cost 1 11 ⟩;
    t = (∼x[k]) & x[r], sprintf(name[s], "%d<%d(%04x)", k, r, t);
    ⟨ Update for cost 1 11 ⟩;
    t = x[k] | x[r], sprintf(name[s], "%d|%d(%04x)", k, r, t);
    ⟨ Update for cost 1 11 ⟩;
    t = x[k] ⊕ x[r], sprintf(name[s], "%d^%d(%04x)", k, r, t);
    ⟨ Update for cost 1 11 ⟩;
  }
mm = (s + 31)/32;
```

This code is used in section 8.

11. ⟨ Update for cost 1 11 ⟩ ≡

```
p = &func[t];
if (p→cost > 1) {
  if (s ≥ 32 ∗ footsize) {
    fprintf(stderr, "Too␣many␣special␣functions␣(footsize=%d)!\n", footsize);
    exit(−3);
  }
  p→next→prev = p→prev, p→prev→next = p→next;
  p→cost = 1, p→parent = (x[k] ≪ 16) + x[r];
  p→footprint[s ≫ 5] = 1 ≪ (s & #1f);
  p→next = head[1].next, p→prev = &head[1];
  p→next→prev = p, p→prev→next = p;
  s++;
  c−−;
}
```

This code is used in section 10.

12. ⟨ Answer queries 12 ⟩ ≡

```
while (1) {
  printf("Truth␣table␣(hex):␣");
  fflush(stdout);
  if (¬fgets(buf, 100, stdin)) break;
  if (sscanf(buf, "%x", &ttt) ≠ 1) break;
  printf("%04x␣has␣cost␣", ttt);
  if (ttt & #8000) ttt ⊕= #ffff;
  printf("%d,␣parents␣(%04x,%04x),␣and␣footprint", func[ttt].cost, func[ttt].parent ≫ 16,
      func[ttt].parent & #ffff);
  for (j = 0; j < mm; j++)
    if (func[ttt].footprint[j]) {
      s = 32 ∗ j;
      for (u = func[ttt].footprint[j]; u; u ≫= 1, s++)
        if (u & 1) printf("␣%s", name[s]);
    }
  printf("\n");
}
```

This code is used in section 1.

13. Index.

⟨ Answer queries 12 ⟩ Used in section 1.
⟨ Combine all functions of costs k and $r - 1 - k$ 3 ⟩ Used in section 1.
⟨ Initialize the functions of cost 0 9 ⟩ Used in section 8.
⟨ Initialize the functions of cost 1 10 ⟩ Used in section 8.
⟨ Initialize the tables 8 ⟩ Used in section 1.
⟨ Read the initial functions 2 ⟩ Used in section 1.
⟨ Try for breakthru and **goto** *pqdone* 6 ⟩ Used in section 3.
⟨ Try for new function 4 ⟩ Used in section 3.
⟨ Update for cost 1 11 ⟩ Used in section 10.
⟨ Update the table for cost $r - 1$ 7 ⟩ Used in section 6.
⟨ Update the table for cost r 5 ⟩ Used in section 4.

FCHAINS4X