June 25, 2025 at 17:35

1. Intro. This program takes the output of SIMPATH (on *stdin*) and converts it to a ZDD (on *stdout*).

The output is in the same format as might be output by BDD15, except that the branches are in bottom-up order rather than top-down.

The input begins with lines that specify the names of the vertices and arcs. A copy of those lines is written to the file `/tmp/simpath-names`.

Then come the lines we want to reduce, which might begin like this:

```
#1:
2:3,4
#2:
3:5,6
4:7,0
```

meaning that node 2 of the unreduced dag has branches to nodes 3 and 4, etc. Nodes 0 and 1 are the sinks.

```
#define memsize (1 ≪ 25)
#define varsize 1000
#include <stdio.h>
#include <stdlib.h>
  int lo[memsize], hi[memsize];
  int firstnode[varsize];
  int head;
  int nodesout;
  char buf[100];
  int nbuf, lbuf, hbuf;
  FILE *tempfile;
  main()
  {
    register int j, k, p, q, r, s, t;
    ⟨Store all the input in lo and hi 2⟩;
    ⟨Reduce and output 3⟩;
    fprintf(stderr, "%d␣branch␣nodes␣output.\n", nodesout);
  }
```

2. ⟨ Store all the input in *lo* and *hi* 2 ⟩ ≡

```
  tempfile = fopen("/tmp/simpath-names", "w");
  if (¬tempfile) {
    fprintf(stderr, "I␣can't␣open␣/tmp/simpath-names␣for␣writing!\n");
    exit(−1);
  }
  while (1) {
    if (¬fgets(buf, 100, stdin)) {
      fprintf(stderr, "The␣input␣line␣ended␣unexpectedly!\n");
      exit(−2);
    }
    if (buf[0] ≡ '#') break;
    fprintf(tempfile, buf);
  }
  fclose(tempfile);
  for (t = 1, s = 2; ; t++) {      /* t is arc number, s is node number */
    if (t + 1 ≥ varsize) {
      fprintf(stderr, "Memory␣overflow␣(varsize=%d)!\n", varsize);
      exit(−3);
    }
    firstnode[t] = s;
    if (sscanf(buf + 1, "%d", &nbuf) ≠ 1 ∨ nbuf ≠ t) {
      fprintf(stderr, "Bad␣input␣line␣for␣arc␣%d:␣%s", t, buf);
      exit(−4);
    }
    for ( ; ; s++) {
      if (s ≥ memsize) {
        fprintf(stderr, "Memory␣overflow␣(memsize=%d)!\n", memsize);
        exit(−5);
      }
      if (¬fgets(buf, 100, stdin)) goto done_reading;
      if (buf[0] ≡ '#') break;
      if (sscanf(buf, "%x:%x,%x", &nbuf, &lbuf, &hbuf) ≠ 3 ∨ nbuf ≠ s) {
        fprintf(stderr, "Bad␣input␣line␣for␣node␣%x:␣%s", s, buf);
        exit(−6);
      }
      lo[s] = lbuf, hi[s] = hbuf;
    }
  }
done_reading: fprintf(stderr, "%d␣arcs␣and␣%d␣branch␣nodes␣successfully␣read.\n", t, s − 2);
  firstnode[t + 1] = s;
```

This code is used in section 1.

3. Here I use an algorithm something like that of Sieling and Wegener, and something like the ones I used in BDD9 and CONNECTED and other programs. But I've changed it again, for fun and variety.

All nodes below the current level have already been output. If node p on such a level has been reduced away in favor of node q, we've set $lo[p] = q$. But if that node has been output, we set $lo[p] < 0$. We also keep $hi[p] \ge 0$ in such nodes, except temporarily when using $hi[p]$ as a pointer to a stack.

We go through all nodes on the current level and link together the ones with a common *hi* field p. The most recent such node is $q = -hi[p]$; the next most recent is $hi[q]$, if that is positive; then $hi[hi[q]]$ and so on. But if $hi[q] \le 0$, it specifies another p value, in a list of lists.

```
⟨ Reduce and output 3 ⟩ ≡
  lo[0] = lo[1] = −1;    /∗ sinks are implicitly present ∗/
  for ( ; t; t−−) {
    head = 0;
    for (k = firstnode[t]; k < firstnode[t + 1]; k++) {
      q = lo[k];
      if (lo[q] ≥ 0) lo[k] = lo[q];    /∗ replace lo[k] by its clone ∗/
      q = hi[k];
      if (lo[q] ≥ 0) hi[k] = q = lo[q];    /∗ likewise hi[k] ∗/
      if (q) ⟨ Put k onto the list for q 4 ⟩;
    }
    ⟨ Go through the list of lists 5 ⟩;
  }
```

This code is used in section 1.

4. ⟨ Put k onto the list for q 4 ⟩ ≡

```
  {
    if (hi[q] ≥ 0) hi[k] = −head, head = q;    /∗ start a new list ∗/
    else hi[k] = −hi[q];    /∗ point to previous in list ∗/
    hi[q] = −k;
  }
```

This code is used in section 3.

5. We go through each list twice, once to output instructions and once to clean up our tracks.

```
⟨ Go through the list of lists 5 ⟩ ≡
  for (p = head; p; p = −q) {
    for (q = −hi[p]; q > 0; q = hi[q]) {
      r = lo[q];
      if (lo[r] ≤ 0) {
        printf("%x:␣(~%d?%x:%x)\n", q, t, r, p);
        nodesout++;
        lo[r] = q, lo[q] = −r − 1;
      } else lo[q] = lo[r];    /∗ make q point to its previously output clone ∗/
    }
    for (q = −hi[p], hi[p] = 0; q > 0; r = q, q = hi[r]) {
      r = lo[q];
      if (r < 0) lo[−r − 1] = −1;
    }
    hi[r] = 0;
  }
```

This code is used in section 3.

6. Index.

SIMPATH-REDUCE