June 25, 2025 at 17:42

1. Introduction. This program inputs a directed graph. It outputs a not-necessarily-reduced binary decision diagram for the family of all simple oriented cycles in that graph.

The format of the output is described in another program, SIMPATH-REDUCE. Let me just say here that it is intended only for computational convenience, not for human readability.

I've tried to make this program simple, whenever I had to choose between simplicity and efficiency. But I haven't gone out of my way to be inefficient.

(Notes, 30 November 2015: My original version of this program, written in August 2008, was hacked from SIMPATH. I don't think I used it much at that time, if at all, because I made a change in February 2010 to make it compile without errors. Today I'm making two fundamental changes: (i) Each "frontier" in SIMPATH was required to be an interval of vertices, according to the vertex numbering. Now the elements of each frontier are listed explicitly; so I needn't waste space by including elements that don't really participate in frontier activities. (ii) I do *not* renumber the vertices. The main advantage of these two changes is that I can put a dummy vertex at the end, with arcs to and from every other vertex; then we get all the simple *paths* instead of all the simple *cycles*, while the frontiers stay the same size except for the dummy element. And we can modify this program to get all the oriented *Hamiltonian* paths as well.)

```
#define maxn 90   /* maximum number of vertices; at most 126 */
#define maxm 2000   /* maximum number of arcs */
#define logmemsize 27
#define memsize (1 << logmemsize)
#define loghtsize 24
#define htsize (1 << loghtsize)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_graph.h"
#include "gb_save.h"
  char mem[memsize];   /* the big workspace */
  unsigned long long tail, boundary, head;   /* queue pointers */
  unsigned int htable[htsize];   /* hash table */
  unsigned int htid;   /* "time stamp" for hash entries */
  int htcount;   /* number of entries in the hash table */
  int wrap = 1;   /* wraparound counter for hash table clearing */
  Vertex *vert[maxn + 1];
  int f[maxn + 2], ff[maxn + 2];   /* elements of the current and the next frontier */
  int s, ss;   /* the sizes of f and ff */
  int curfront[maxn + 1], nextfront[maxn + 1];   /* inverse frontier map */
  int arcto[maxm];   /* destination number of each arc */
  int firstarc[maxn + 2];   /* where arcs from a vertex start in arcto */
  char mate[maxn + 3];   /* encoded state */
  int serial, newserial;   /* state numbers */
  ⟨Subroutines 13⟩
  main(int argc, char *argv[])
  {
    register int i, j, jj, jm, k, km, l, ll, m, n, p, t, hash, sign;
    register Graph *g;
    register Arc *a, *b;
    register Vertex *u, *v;
    ⟨Input the graph 2⟩;
    ⟨Reformat the arcs 3⟩;
```

```
    ⟨ Do the algorithm 5 ⟩;
  }
```

2. ⟨ Input the graph 2 ⟩ ≡

```
  if (argc ≠ 2) {
    fprintf(stderr, "Usage:␣%s␣foo.gb\n", argv[0]);
    exit(−1);
  }
  g = restore_graph(argv[1]);
  if (¬g) {
    fprintf(stderr, "I␣can't␣input␣the␣graph␣%s␣(panic␣code␣%ld)!\n", argv[1], panic_code);
    exit(−2);
  }
  n = g→n;
  if (n > maxn) {
    fprintf(stderr, "Sorry,␣that␣graph␣has␣%d␣vertices;␣", n);
    fprintf(stderr, "I␣can't␣handle␣more␣than␣%d!\n", maxn);
    exit(−3);
  }
  if (g→m > maxm) {
    fprintf(stderr, "Sorry,␣that␣graph␣has␣%ld␣arcs;␣", (g→m + 1)/2);
    fprintf(stderr, "I␣can't␣handle␣more␣than␣%d!\n", maxm);
    exit(−3);
  }
```

This code is used in section 1.

3. The arcs will be either $j \to k$ or $j \leftarrow k$ between vertex number j and vertex number k, when $j < k$ and those vertices are adjacent in the graph. We process them in order of increasing j; but for fixed j, the values of k aren't necessarily increasing.

The k values appear in the *arcto* array, with $-k$ used for the arcs that emanate from k. The arcs for fixed j occur in positions *firstarc*[j] through *firstarc*[$j+1$] − 1 of that array.

After this step, we forget the GraphBase data structures and just work with our homegrown integer-only representation.

```
⟨ Reformat the arcs 3 ⟩ ≡
  ⟨ Make the inverse-arc lists 4 ⟩;
  for (m = 0, k = 1; k ≤ n; k++) {
    firstarc[k] = m;
    v = vert[k];
    printf("%d(%s)\n", k, v→name);
    for (a = v→arcs; a; a = a→next) {
      u = a→tip;
      if (u > v) {
        arcto[m++] = u − g→vertices + 1;
        if (a→len ≡ 1) printf("␣->␣%ld(%s)␣#%d\n", u − g→vertices + 1, u→name, m);
        else printf("␣->␣%ld(%s,%ld)␣#%d\n", u − g→vertices + 1, u→name, a→len, m);
      }
    }
    for (a = v→invarcs; a; a = a→next) {
      u = a→tip;
      if (u > v) {
        arcto[m++] = −(u − g→vertices + 1);
        if (a→len ≡ 1) printf("␣<-␣%ld(%s)␣#%d\n", u − g→vertices + 1, u→name, m);
        else printf("␣<-␣%ld(%s,%ld)␣#%d\n", u − g→vertices + 1, u→name, a→len, m);
      }
    }
  }
  firstarc[k] = m;
```

This code is used in section 1.

4. To aid in the desired sorting, we first create an inverse-arc list for each vertex v, namely a list of vertices that point to v.

```
#define invarcs y.A
```

```
⟨ Make the inverse-arc lists 4 ⟩ ≡
  for (v = g→vertices; v < g→vertices + n; v++) v→invarcs = Λ;
  for (v = g→vertices; v < g→vertices + n; v++) {
    vert[v − g→vertices + 1] = v;
    for (a = v→arcs; a; a = a→next) {
      register Arc *b = gb_virgin_arc();

      u = a→tip;
      b→tip = v;
      b→len = a→len;
      b→next = u→invarcs;
      u→invarcs = b;
    }
  }
```

This code is used in section 3.

5. The algorithm. Now comes the fun part. We systematically construct a binary decision diagram for all simple paths by working top-down, considering the arcs in *arcto*, one by one.

When we're dealing with arc i, we've already constructed a table of all possible states that might arise when each of the previous arcs has been chosen-or-not, except for states that obviously cannot be part of a simple path.

Arc i runs from vertex j to vertex $k = arcto[i]$, or from $k = -arcto[i]$ to j.

Let $F_i = \{v_1, \ldots, v_s\}$ be the *frontier* at arc i, namely the set of vertex numbers $\geq j$ that appear in arcs $<$ i.

The state before we decide whether or not to include arc i is represented by a table of values $mate[t]$, for $t \in F_i \cup \{j,k\}$, with the following significance: If $mate[t] = t$, the previous arcs haven't touched vertex t. If $mate[t] = u$ and $u \neq t$, the previous arcs have made a simple directed path from t to u. If $mate[t] = -u$, the previous arcs have made a simple directed path from u to t. If $mate[t] = 0$, the previous arcs have "saturated" vertex t; we can't touch it again.

The *mate* information is all that we need to know about the behavior of previous arcs. And it's easily updated when we add the ith arc (or not). So each "state" is equivalent to a *mate* table, consisting of s numbers, where s is the size of F_i.

The states are stored in a queue, indexed by 64-bit numbers *tail*, *boundary*, and *head*, where $tail \leq boundary \leq head$. Between *tail* and *boundary* are the pre-arc-i states that haven't yet been processed; between *boundary* and *head* are the post-arc-i states that will be considered later. The states before *boundary* are sequences of s bytes each, and the states after *boundary* are sequences of ss bytes each, where ss is the size of F_{i+1}.

(Exception: If $s = 0$, we use one byte to represent the state, although we ignore it when reading from the queue later. In this way we know how many states are present.)

Bytes of the queue are stored in *mem*, which wraps around modulo *memsize*. We ensure that $head - tail$ never exceeds *memsize*.

```
⟨ Do the algorithm 5 ⟩ ≡
  for (t = 1; t ≤ n; t++) mate[t] = t;
  ⟨ Initialize the queue 6 ⟩;
  for (i = 0; i < m; i++) {
    printf("#%d:\n", i + 1);    /* announce that we're beginning a new arc */
    fprintf(stderr, "Beginning␣arc␣%d␣(serial=%d,head-tail=%lld)\n", i + 1, serial, head − tail);
    fflush(stderr);
    ⟨ Process arc i 7 ⟩;
  }
  printf("%x:0,0\n", serial);
```

This code is used in section 1.

6. Each state for a particular arc gets a distinguishing number, where its ZDD instructions begin. Two states are special: 0 means the losing state, when a simple path is impossible; 1 means the winning state, when a simple path has been completed. The other states are 2 or more.

Initially i will be zero, and the queue is empty. We'll want jj to be the the j vertex of arc $i+1$, and ss to be the size of F_{i+1}. Also *serial* is the identifying number for arc $i+1$.

```
⟨ Initialize the queue 6 ⟩ ≡
  jj = 1, ss = 0;
  while (firstarc[jj + 1] ≡ 0) jj++;    /* unnecessary unless vertex 1 is isolated */
  tail = head = 0;
  serial = 2;
```

This code is used in section 5.

7. The output format on *stdout* simply shows the identifying numbers of a state and its two successors, in hexadecimal.

```
#define trunc(addr) ((addr) & (memsize − 1))

⟨ Process arc i 7 ⟩ ≡
  if (ss ≡ 0) head++;     /* put a dummy byte into the queue */
  boundary = head, htcount = 0, htid = (i + wrap) ≪ logmemsize;
  if (htid ≡ 0) {
    for (hash = 0; hash < htsize; hash++) htable[hash] = 0;
    wrap++, htid = 1 ≪ logmemsize;
  }
  newserial = serial + (head − tail)/(ss ? ss : 1);
  j = jj, sign = arcto[i], k = (sign > 0 ? sign : −sign), s = ss;
  for (p = 0; p < s; p++) f[p] = ff[p];
  ⟨ Compute jj and F_{i+1} 8 ⟩;
  while (tail < boundary) {
    printf("%x:", serial);
    serial++;
    ⟨ Unpack a state, and move tail up 9 ⟩;
    ⟨ Print the successor if arc i is not chosen 11 ⟩;
    printf(",");
    ⟨ Print the successor if arc i is chosen 10 ⟩;
    printf("\n");
  }
```

This code is used in section 5.

8. Here we set *nextfront*$[t]$ to $i+1$ whenever $t \in F_{i+1}$. And we also set *curfront*$[t]$ to $i+1$ wheneer $t \in F_i$; I use $i+1$, not i, because the *curfront* array is initially zero.

```
⟨ Compute jj and F_{i+1} 8 ⟩ ≡
  while (jj ≤ n ∧ firstarc[jj + 1] ≡ i + 1) jj++;
  for (p = ss = 0; p < s; p++) {
    t = f[p];
    curfront[t] = i + 1;
    if (t ≥ jj) {
      nextfront[t] = i + 1;
      ff[ss++] = t;
    }
  }
  if (j ≡ jj ∧ nextfront[j] ≠ i + 1) nextfront[j] = i + 1, ff[ss++] = j;
  if (k ≥ jj ∧ nextfront[k] ≠ i + 1) nextfront[k] = i + 1, ff[ss++] = k;
```

This code is used in section 7.

9. This step sets $mate[t]$ for all $t \in F_i \cup \{j,k\}$, based on a queued state, while taking s bytes out of the queue.

⟨ Unpack a state, and move *tail* up 9 ⟩ ≡

```
if (s ≡ 0) tail++;
else {
  for (p = 0; p < s; p++, tail++) {
    t = f[p];
    mate[t] = mem[trunc(tail)];
  }
}
if (curfront[j] ≠ i + 1) mate[j] = j;
if (curfront[k] ≠ i + 1) mate[k] = k;
```

This code is used in section 7.

10. Here's where we update the mates. The order of doing this is carefully chosen so that it works fine when $mate[j] = j$ and/or $mate[k] = k$.

⟨ Print the successor if arc i is chosen 10 ⟩ ≡

```
if (sign > 0) {
  jm = mate[j], km = mate[k];
  if (jm ≡ j) jm = −j;
  if (jm ≥ 0 ∨ km ≤ 0) printf("0");      /* we mustn't touch a saturated vertex */
  else if (jm ≡ −k) ⟨ Print 1 or 0, depending on whether this arc wins or loses 12 ⟩
  else {
    mate[j] = 0, mate[k] = 0;
    mate[−jm] = km, mate[km] = jm;
    printstate(j, jj, i, k);
  }
} else {
  jm = mate[j], km = mate[k];
  if (km ≡ k) km = −k;
  if (jm ≤ 0 ∨ km ≥ 0) printf("0");      /* we mustn't touch a saturated vertex */
  else if (km ≡ −j) ⟨ Print 1 or 0, depending on whether this arc wins or loses 12 ⟩
  else {
    mate[j] = 0, mate[k] = 0;
    mate[jm] = km, mate[−km] = jm;
    printstate(j, jj, i, k);
  }
}
```

This code is used in section 7.

11. ⟨ Print the successor if arc i is not chosen 11 ⟩ ≡

```
printstate(j, jj, i, k);
```

This code is used in section 7.

12. See the note below regarding a change that will restrict consideration to Hamiltonian paths. A similar change is needed here.

⟨ Print 1 or 0, depending on whether this arc wins or loses 12 ⟩ ≡

```
{
  for (p = 0; p < s; p++) {
    t = f[p];
    if (t ≠ j ∧ t ≠ k ∧ mate[t] ∧ mate[t] ≠ t) break;
  }
  if (p ≡ s) printf("1");   /* we win: this cycle is all by itself */
  else printf("0");   /* we lose: there's junk outside this cycle */
}
```

This code is used in section 10.

13. The *printstate* subroutine does the rest of the work. It makes sure that no incomplete paths linger in positions that are about to disappear from the current frontier; and it puts the *mate* entries of the next frontier into the queue, checking to see if that state was already there.

If '$mate[t] \neq t$' is removed from the condition below, we get Hamiltonian cycles only (I mean, simple cycles that include every vertex).

⟨ Subroutines 13 ⟩ ≡

```
void printstate(int j, int jj, int i, int k)
{
  register int h, hh, p, t, tt, hash;
  for (p = 0; p < s; p++) {
    t = f[p];
    if (nextfront[t] ≠ i + 1 ∧ mate[t] ∧ mate[t] ≠ t) break;
  }
  if (p < s) printf("0");   /* incomplete junk mustn't be left hanging */
  else if (nextfront[j] ≠ i + 1 ∧ mate[j] ∧ mate[j] ≠ j) printf("0");
  else if (nextfront[k] ≠ i + 1 ∧ mate[k] ∧ mate[k] ≠ k) printf("0");
  else if (ss ≡ 0) printf("%x", newserial);
  else {
    if (head + ss − tail > memsize) {
      fprintf(stderr, "Oops,␣I'm␣out␣of␣memory:␣memsize=%d,␣serial=%d!\n", memsize, serial);
      exit(−69);
    }
    ⟨ Move the current state into position after head, and compute hash 14 ⟩;
    ⟨ Find the first match, hh, for the current state after boundary 15 ⟩;
    h = trunc(hh − boundary)/ss;
    printf("%x", newserial + h);
  }
}
```

This code is used in section 1.

14. ⟨ Move the current state into position after *head*, and compute *hash* 14 ⟩ ≡

```
for (p = 0, h = trunc(head), hash = 0; p < ss; p++, h = trunc(h + 1)) {
  t = ff[p];
  mem[h] = mate[t];
  hash = hash * 31415926525 + mate[t];
}
```

This code is used in section 13.

15. The hash table is automatically cleared whenever *htid* is increased, because we store *htid* with each relevant table entry.

⟨ Find the first match, *hh*, for the current state after *boundary* 15 ⟩ ≡
```
for (hash = hash & (htsize − 1); ; hash = (hash + 1) & (htsize − 1)) {
  hh = htable[hash];
  if ((hh ⊕ htid) ≥ memsize) ⟨ Insert new entry and goto found 16 ⟩;
  hh = trunc(hh);
  for (t = hh, h = trunc(head), tt = trunc(t + ss − 1); ; t = trunc(t + 1), h = trunc(h + 1)) {
    if (mem[t] ≠ mem[h]) break;
    if (t ≡ tt) goto found;
  }
}
found:
```
This code is used in section 13.

16. ⟨ Insert new entry and **goto** *found* 16 ⟩ ≡
```
{
  if (++htcount > (htsize ≫ 1)) {
    fprintf(stderr, "Sorry, the hash table is full (htsize=%d, serial=%d)!\n", htsize, serial);
    exit(−96);
  }
  hh = trunc(head);
  htable[hash] = htid + hh;
  head += ss;
  goto found;
}
```
This code is used in section 15.

17. Index.

SIMPATH-DIRECTED-CYCLES