1.   Introduction.   This program inputs an undirected graph and the names of two vertices in that graph (the "source" and "target" vertices). It outputs a not-necessarily-reduced binary decision diagram for the family of all simple paths from the source to the target.

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.

#**define** $maxn$ 255   /∗ maximum number of vertices; at most 255 ∗/
#**define** $maxm$ 2000   /∗ maximum number of edges ∗/
#**define** $logmemsize$ 27
#**define** $memsize$ $(1 \ll logmemsize)$
#**define** $loghtsize$ 25
#**define** $htsize$ $(1 \ll loghtsize)$

#**include** <stdio.h>
#**include** <stdlib.h>
#**include** <string.h>
#**include** "gb_graph.h"
#**include** "gb_save.h"

  unsigned 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 $arcto[maxm]$;   /∗ destination number of each arc ∗/
  int $firstarc[maxn + 2]$;   /∗ where arcs from a vertex start in $arcto$ ∗/
  unsigned char $mate[maxn + 3]$;   /∗ encoded state ∗/
  int $serial$, $newserial$;   /∗ state numbers ∗/

  ⟨Subroutines 13∗⟩

  $main(\textbf{int}\ argc, \textbf{char}\ *argv[\,])$
  {
    register int i, j, jj, jm, k, km, l, ll, m, n, t, $hash$;
    register Graph ∗g;
    register Arc ∗a, ∗b;
    register Vertex ∗u, ∗v;
    Vertex ∗$source = \Lambda$, ∗$target = \Lambda$;

    ⟨Input the graph 2⟩;
    ⟨Renumber the vertices 3⟩;
    ⟨Reformat the edges 4⟩;
    ⟨Do the algorithm 5⟩;
  }

2. ⟨ Input the graph 2 ⟩ ≡

```
  if (argc ≠ 4) {
    fprintf (stderr, "Usage:␣%s␣foo.gb␣source␣target\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 > 2 ∗ maxm) {
    fprintf (stderr, "Sorry,␣that␣graph␣has␣%ld␣edges;␣", (g→m + 1)/2);
    fprintf (stderr, "I␣can't␣handle␣more␣than␣%d!\n", maxm);
    exit(−3);
  }
  for (v = g→vertices; v < g→vertices + n; v++) {
    if (strcmp(argv[2], v→name) ≡ 0)  source = v;
    if (strcmp(argv[3], v→name) ≡ 0)  target = v;
    for (a = v→arcs; a; a = a→next) {
      u = a→tip;
      if (u ≡ v) {
        fprintf (stderr, "Sorry,␣the␣graph␣contains␣a␣loop␣%s--%s!\n", v→name, v→name);
        exit(−4);
      }
      b = (v < u ? a + 1 : a − 1);
      if (b→tip ≠ v) {
        fprintf (stderr, "Sorry,␣the␣graph␣isn't␣undirected!\n");
        fprintf (stderr, "(%s->%s␣has␣mate␣pointing␣to␣%s)\n", v→name, u→name, b→tip→name);
        exit(−5);
      }
    }
  }
  if (¬source) {
    fprintf (stderr, "I␣can't␣find␣source␣vertex␣%s␣in␣the␣graph!\n", argv[2]);
    exit(−6);
  }
  if (¬target) {
    fprintf (stderr, "I␣can't␣find␣target␣vertex␣%s␣in␣the␣graph!\n", argv[3]);
    exit(−7);
  }
```

This code is used in section 1.

3.   If the source vertex is the first vertex in the graph, I'll process vertices according to the graph's own ordering.

Otherwise, I use a simple breadth-first strategy to number the vertices: The source is vertex 1. Then, for each $j \geq 1$, I run through the arcs from vertex j and assign the first unused number to any of its neighbors that haven't already got one.

#**define** num $z.I$

⟨ Renumber the vertices 3 ⟩ ≡
  if $(source \equiv g\text{-}vertices)$ {
    for $(k = 0; \ k < n; \ k\text{++})$ $(g\text{-}vertices + k)\text{-}num = k + 1, vert[k+1] = g\text{-}vertices + k;$
  } **else** {
    for $(k = 0; \ k < n; \ k\text{++})$ $(g\text{-}vertices + k)\text{-}num = 0;$
    $vert[1] = source, source\text{-}num = 1;$
    for $(j = 0, k = 1; \ j < k; \ j\text{++})$ {
      $v = vert[j+1];$
      for $(a = v\text{-}arcs; \ a; \ a = a\text{-}next)$ {
        $u = a\text{-}tip;$
        if $(u\text{-}num \equiv 0)$ $u\text{-}num = \text{++}k, vert[k] = u;$
      }
    }
    if $(target\text{-}num \equiv 0)$ {
      $fprintf(stderr, \texttt{"Sorry,\_there's\_no\_path\_from\_\%s\_to\_\%s\_in\_the\_graph!\textbackslash n"}, argv[2], argv[3]);$
      $exit(-8);$
    }
    if $(k < n)$ {
      $fprintf(stderr, \texttt{"The\_graph\_isn't\_connected\_(\%d<\%d)!\textbackslash n"}, k, n);$
      $fprintf(stderr, \texttt{"But\_that's\_OK;\_I'll\_work\_with\_the\_component\_of\_\%s.\textbackslash n"}, argv[2]);$
      $n = k;$
    }
  }

This code is used in section 1.

4.   The edges will be considered as arcs $j \to k$ 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. The edges 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.

$\langle$ Reformat the edges 4 $\rangle \equiv$

```
for (m = 0, k = 1;  k ≤ n;  k++) {
    firstarc[k] = m;
    v = vert[k];
    printf("%ld(%s)\n", v→num, v→name);
    for (a = v→arcs;  a;  a = a→next) {
        u = a→tip;
        if (u→num > k) {
            arcto[m++] = u→num;
            if (a→len ≡ 1)  printf("␣->␣%ld(%s)␣#%d\n", u→num, u→name, m);
            else  printf("␣->␣%ld(%s,%ld)␣#%d\n", u→num, u→name, a→len, m);
        }
    }
}
firstarc[k] = m;
```

This code is used in section 1.

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 $k = arcto[i]$. Let l be the maximum vertex number in arcs less than i. (If the breadth-first ordering option was taken above, we'll always have $k \leq l + 1$, because of the way we did the numbering and reformatting; but that method is not always best.)

The state before we decide whether or not to include arc i is represented by a table of values $mate[t]$, for $j \leq t \leq l$, 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 connected t with u by a simple path. If $mate[t] = 0$, the previous arcs have "saturated" vertex t; we can't touch it again.

We also use a (slick?) trick: We imagine that an edge between the source and target has already been included. Then the final arc of a simple path will be an arc that completes a cycle, when no other incomplete paths are present. (Think about it.)

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 $l + 1 - j$ numbers.

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 $s = l + 1 - j$ bytes each, and the states after *boundary* are sequences of $ss = ll + 1 - jj$ bytes each, where ll and jj are the values of l and j for arc $i + 1$.

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 ⟩ ≡
  ⟨ Initialize the *mate* table 6 ⟩;
  ⟨ Initialize the queue 7 ⟩;
  for $(i = 0; \ i < m; \ i{+}{+})$ {
    $printf\,(\texttt{"\#\%d:\textbackslash n"}, i + 1)$;   /∗ announce that we're beginning a new arc ∗/
    $fprintf\,(stderr,\texttt{"Beginning\textvisiblespace arc\textvisiblespace\%d\textvisiblespace(serial=\%d,head-tail=\%lld)\textbackslash n"}, i + 1, serial, head - tail)$;
    $fflush\,(stderr)$;
    ⟨ Process arc i 8 ⟩;
  }
This code is used in section 1.

6.   ⟨ Initialize the *mate* table 6 ⟩ ≡
  for $(t = 2; \ t \leq n; \ t{+}{+}) \ mate[t] = t$;
  $mate[target{\rightarrow}num] = 1, mate[1] = target{\rightarrow}num$;
This code is used in section 5.

7.   ⟨ Initialize the queue 7 ⟩ ≡
  $jj = ll = 1$;
  $mem[0] = mate[1]$;
  $tail = 0, head = 1$;
  $serial = 2$;
This code is used in section 5.

8.   Each state for a particular arc gets a distinguishing number. 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.

   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 8 ⟩ ≡
    $boundary = head, htcount = 0, htid = (i + wrap) \ll logmemsize;$
    if $(htid \equiv 0)$ {
       for $(hash = 0; \ hash < htsize; \ hash \mathbin{+\!\!+})$ $htable[hash] = 0;$
       $wrap \mathbin{+\!\!+}, htid = 1 \ll logmemsize;$
    }
    $newserial = serial + ((head - tail)/(ll + 1 - jj));$
    $j = jj, k = arcto[i], l = ll;$
    while $(jj \leq n \wedge firstarc[jj + 1] \equiv i + 1)$ $jj \mathbin{+\!\!+};$
    $ll = (k > l \ ? \ k : l);$
    while $(tail < boundary)$ {
       $printf(\texttt{"\%x:"}, serial);$
       $serial \mathbin{+\!\!+};$
       ⟨ Unpack a state, and move *tail* up 9 ⟩;
       ⟨ Print the successor if arc i is not chosen 11 ⟩;
       $printf(\texttt{","});$
       ⟨ Print the successor if arc i is chosen 10 ⟩;
       $printf(\texttt{"\textbackslash n"});$
    }

This code is used in section 5.

9.   If the target vertex hasn't entered the action yet (that is, if it exceeds l), we must update its *mate* entry at this point.

⟨ Unpack a state, and move *tail* up 9 ⟩ ≡
    for $(t = j; \ t \leq l; \ t\mathbin{+\!\!+}, tail \mathbin{+\!\!+})$ {
       $mate[t] = mem[trunc(tail)];$
       if $(mate[t] > l)$ $mate[mate[t]] = t;$
    }

This code is used in section 8.

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 ⟩ ≡
    $jm = mate[j], km = mate[k];$
    if $(jm \equiv 0 \vee km \equiv 0)$ $printf(\texttt{"0"});$ /* we mustn't touch a saturated vertex */
    else if $(jm \equiv k)$ ⟨ Print 1 or 0, depending on whether this arc wins or loses 12* ⟩
    else {
       $mate[j] = 0, mate[k] = 0;$
       $mate[jm] = km, mate[km] = jm;$
       $printstate(j, jj, ll);$
       $mate[jm] = j, mate[km] = k, mate[j] = jm, mate[k] = km;$ /* restore original state */
    }
    $done:$

This code is used in section 8.

11.   ⟨ Print the successor if arc i is not chosen 11 ⟩ ≡
  $printstate(j, jj, ll);$

This code is used in section 8.

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 $(t = j + 1; \ t \leq ll; \ t{+}{+})$
      if $(t \neq k)$ {
        if $(mate[t])$ **break**;
      }
    if $(t > n)$ $printf("1");$ /∗ 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 j through $jj - 1$, which are about to disappear; and it puts the contents of $mate[jj]$ through $mate[ll]$ into the queue, checking to see if it was already there.

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

⟨ Subroutines 13* ⟩ ≡
  void $printstate(\textbf{int}\ j, \textbf{int}\ jj, \textbf{int}\ ll)$
  {
    register int $h, \ hh, \ ss, \ t, \ tt, \ hash;$
    for $(t = j; \ t < jj; \ t{+}{+})$
      if $(mate[t])$ **break**;
    if $(t < jj)$ $printf("0");$ /∗ incomplete junk mustn't be left hanging ∗/
    else if $(ll < jj)$ $printf("0");$ /∗ nothing is viable ∗/
    else {
      $ss = ll + 1 - jj;$
      if $(head + ss - tail > memsize)$ {
        $fprintf(stderr, \texttt{"Oops,\_I'm\_out\_of\_memory\_(memsize=\%d,\_serial=\%d)!\textbackslash n"}, memsize, serial);$
        $fflush(stdout);$
        $exit(-69);$
      }
      ⟨ 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 $(t = jj, h = trunc(head), hash = 0; \ t \leq ll; \ t{+}{+}, h = trunc(h + 1))$ {
    $mem[h] = mate[t];$
    $hash = hash * 31415926525 + mate[t];$
  }

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 $(hash = hash \mathbin{\&} (htsize - 1); \; ; \; hash = (hash + 1) \mathbin{\&} (htsize - 1))$ {
    $hh = htable[hash]$;
    if $((hh \oplus htid) \geq memsize)$ ⟨ 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 $(mem[t] \neq mem[h])$ **break**;
      if $(t \equiv tt)$ **goto** *found*;
    }
  }
  found:

This code is used in section 13*.

16.   ⟨ Insert new entry and **goto** *found* 16 ⟩ ≡
  {
    if $(\mathbin{++}htcount > (htsize \gg 1))$ {
      $fprintf(stderr, \texttt{"Sorry,\_the\_hash\_table\_is\_full\_(htsize=\%d,\_serial=\%d)!\textbackslash n"}, htsize, serial)$;
      $exit(-96)$;
    }
    $hh = trunc(head)$;
    $htable[hash] = htid + hh$;
    $head \mathbin{+}= ss$;
    goto *found*;
  }

This code is used in section 15.

17* Index.

The following sections were changed by the change file: 12, 13, 17.

⟨ Do the algorithm 5 ⟩   Used in section 1.
⟨ Find the first match, *hh*, for the current state after *boundary* 15 ⟩   Used in section 13*.
⟨ Initialize the queue 7 ⟩   Used in section 5.
⟨ Initialize the *mate* table 6 ⟩   Used in section 5.
⟨ Input the graph 2 ⟩   Used in section 1.
⟨ Insert new entry and **goto** *found* 16 ⟩   Used in section 15.
⟨ Move the current state into position after *head*, and compute *hash* 14 ⟩   Used in section 13*.
⟨ Print 1 or 0, depending on whether this arc wins or loses 12* ⟩   Used in section 10.
⟨ Print the successor if arc *i* is chosen 10 ⟩   Used in section 8.
⟨ Print the successor if arc *i* is not chosen 11 ⟩   Used in section 8.
⟨ Process arc *i* 8 ⟩   Used in section 5.
⟨ Reformat the edges 4 ⟩   Used in section 1.
⟨ Renumber the vertices 3 ⟩   Used in section 1.
⟨ Subroutines 13* ⟩   Used in section 1.
⟨ Unpack a state, and move *tail* up 9 ⟩   Used in section 8.

SIMPATH-HAM