May 19, 2018 at 02:30

1. Intro. This little program outputs clauses that are satisfiable if and only if the graph g can be "quenched."

Namely, a graph on one vertex can always be quenched. A graph on vertices $(v_1,\ldots,v_n)$ can also be quenched if there's a k with $1 \le k < n$ such that v_k — v_{k+1} and the graph on $(v_1,\ldots,v_{k-1},v_{k+1},\ldots,v_n)$ can be quenched; or if there's a k with $1 \le k < n-2$ such that v_k — v_{k+3} and the graph on $(v_1,\ldots,v_{k-1},v_{k+3},v_{k+1},v_{k+2},v_{k+4},$ can be quenched.

Thus the ordering of vertices is highly significant. Quenchability is a monotone property of the adjacency matrix. A quenchable graph is always connected. For each n there exists a set of I-know-not-how-many labeled spanning trees such that G is connected if and only if it contains one of these spanning trees. (Those spanning trees correspond to the prime implicants of the quenchability function. When $n = 4$ there are six of them: 1 — 2 — 3 — 4, 1 — 2 — 4 — 3, 1 — 4 — 2 — 3, 1 — 4 — 3 — 2, or the stars centered on 3 or 4.

The variables of the corresponding clauses are of several kinds: (i) `tij` means that v_i — v_j at time t, for $0 \le i < j < n-t$; (ii) t`Q`k means that a quenching move of the first kind is used to get to time $t+1$; (iii) t`S`k means that a quenching move of the second kind ("skip two") is used to get to time $t+1$. In each of these cases the number t, i, j, k are represented as two hexadecimal digits, because I assume that $n \le 256$.

```
#define nmax 256
#include <stdio.h>
#include <stdlib.h>
#include "gb_graph.h"
#include "gb_save.h"
  main(int argc, char *argv[])
  {
    register int i, j, k, t, n;
    register Arc *a;
    register Graph *g;
    register Vertex *v, *w;
    ⟨Process the command line 2⟩;
    ⟨Specify the initial nonadjacencies 3⟩;
    ⟨Generate the possible-move clauses 4⟩;
    ⟨Generate the enabling clauses 5⟩;
    ⟨Generate the transition clauses 6⟩;
  }
```

2. ⟨ Process the command line 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␣couldn't␣reconstruct␣graph␣%s!\n", argv[1]);
  exit(−2);
}
n = g→n;
if (n > nmax) {
  fprintf(stderr, "Sorry,␣that␣graph␣has␣too␣many␣vertices␣(%d>%d)!\n", n, nmax);
  exit(−3);
}
printf("~␣sat-graph-quench␣%s\n", argv[1]);
```

This code is used in section 1.

3. It's not necessary to assert anything at time 0 when vertices are adjacent, because of monotonicity. (Such variables `00`ij would be pure literals and might as well be true.) But when vertices v_i and v_j are *not* adjacent, we must make `00`ij false.

#define *stamp* *u.I*

⟨ Specify the initial nonadjacencies 3 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + n; v++) v→stamp = 0;
for (v = g→vertices, j = 1; v < g→vertices + n; v++, j++) {
  for (a = v→arcs; a; a = a→next)
    if (a→tip > v) a→tip→stamp = j;
  for (w = v + 1; w < g→vertices + n; w++)
    if (w→stamp ≠ j)
      printf("~00%02x%02x\n", (unsigned int)(v − g→vertices), (unsigned int)(w − g→vertices));
}
```

This code is used in section 1.

4. ⟨ Generate the possible-move clauses 4 ⟩ ≡

```
for (t = 0; t < n − 1; t++) {
  for (k = 1; k < n − t; k++) printf("␣%02xQ%02x", t, k − 1);
  for (k = 1; k < n − t − 2; k++) printf("␣%02xS%02x", t, k − 1);
  printf("\n");
}
```

This code is used in section 1.

5. ⟨ Generate the enabling clauses 5 ⟩ ≡

```
for (t = 0; t < n − 1; t++) {
  for (k = 1; k < n − t; k++) printf("~%02xQ%02x␣%02x%02x%02x\n", t, k − 1, t, k − 1, k);
  for (k = 1; k < n − t − 2; k++) printf("~%02xS%02x␣%02x%02x%02x\n", t, k − 1, t, k − 1, k + 2);
}
```

This code is used in section 1.

6. ⟨ Generate the transition clauses 6 ⟩ ≡

```
for (t = 0; t < n - 1; t++) {
  for (k = 1; k < n - t; k++)
    for (i = 1; i < n - t; i++)
      for (j = i + 1; j < n - t; j++) printf("~%02xQ%02x␣~%02x%02x%02x␣%02x%02x%02x\n", t, k - 1,
            t + 1, i - 1, j - 1, t, i < k ? i - 1 : i, j < k ? j - 1 : j);
  for (k = 1; k < n - t - 2; k++)
    for (i = 1; i < n - t; i++)
      for (j = i + 1; j < n - t; j++) {
        register iprev = (i ≡ k ? i + 2 : i < k + 3 ? i - 1 : i), jprev = (j ≡ k ? j + 2 : j < k + 3 ? j - 1 : j);
        printf("~%02xS%02x␣~%02x%02x%02x␣%02x%02x%02x\n", t, k - 1, t + 1, i - 1, j - 1, t,
            iprev < jprev ? iprev : jprev, iprev < jprev ? jprev : iprev);
      }
}
```

This code is used in section 1.

7. Index.

SAT-GRAPH-QUENCH