May 19, 2018 at 02:31

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 $v_k \mathrel{—} v_{k+1}$ 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 $v_k \mathrel{—} v_{k+3}$ 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 $v_i \mathrel{—} v_j$ 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 $n \le 256$.

Additional clauses that enforce left-to-right order for commutative moves are included.

This version of the program implements "late binding solitaire," a game that I made up long ago (and used as a warmup problem when I taught Stanford's CS problem seminar in 1989). We're given a sequence of playing cards, chosen at random. The cards have two-letter names; for example, `Ah` is the ace of hearts, `6s` is the six of spades, `Td` is the ten of diamonds. Two cards are adjacent in the graph if and only if they agree in suit or rank.

```
#define nmax 256
#include <stdio.h>
#include <stdlib.h>
#include "gb_graph.h"
#include "gb_save.h"
#include <string.h>
#include "gb_flip.h"
  char *cardname[52] = {"Ac","2c","3c","4c","5c","6c","7c","8c","9c","Tc","Jc","Qc","Kc",
      "Ad","2d","3d","4d","5d","6d","7d","8d","9d","Td","Jd","Qd","Kd","Ah","2h","3h","4h",
      "5h","6h","7h","8h","9h","Th","Jh","Qh","Kh","As","2s","3s","4s","5s","6s","7s","8s",
      "9s","Ts","Js","Qs","Ks"};
  int seed;
  main(int argc, char *argv[])
  {
    register char *tt;
    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 noncommutativity clauses 7*⟩;
    ⟨Generate the transition clauses 6⟩;
  }
```

2*. ⟨ Process the command line 2* ⟩ ≡

```
if (argc ≠ 2 ∨ sscanf(argv[1], "%d", &seed) ≠ 1) {
  fprintf(stderr, "Usage: %s seed\n", argv[0]);
  exit(−1);
}
gb_init_rand(seed);
n = 18;
for (j = 0; j < n; j++) {
  i = j + gb_unif_rand(52 − j);
  tt = cardname[i], cardname[i] = cardname[j], cardname[j] = tt;
}
printf("~ sat-graph-quench-noncomm-latebinding-random %d\n", seed);
printf("~");
for (j = 0; j < n; j++) printf(" %s", cardname[j]);
printf("\n");
```

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 00ij would be pure literals and might as well be true.) But when vertices v_i and v_j are *not* adjacent, we must make 00ij false.

#define *stamp* *u.I*

⟨ Specify the initial nonadjacencies 3* ⟩ ≡

```
for (i = 0; i < n; i++)
  for (j = i + 1; j < n; j++) {
    if (cardname[i][0] ≡ cardname[j][0]) continue;
    if (cardname[i][1] ≡ cardname[j][1]) continue;
    printf("~00%02x%02x\n", i, j);
  }
```

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* The commutativity relations, when $t' = t+1$ and $j' = j+1$, are: $t\mathtt{Q}i \wedge t'\mathtt{Q}j = t\mathtt{Q}j' \wedge t'\mathtt{Q}i$, if $i < j$; $t\mathtt{S}i \wedge t'\mathtt{S}j = t\mathtt{S}j' \wedge t'\mathtt{S}i$, if $i+2 < j$; $t\mathtt{Q}i \wedge t'\mathtt{S}j = t\mathtt{S}j' \wedge t'\mathtt{Q}i$, if $i < j$; $t\mathtt{S}i \wedge t'\mathtt{Q}j = t\mathtt{Q}j' \wedge t'\mathtt{S}i$, if $i+2 < j$; and (surprise!) $t\mathtt{S}i \wedge t'\mathtt{S}i = t\mathtt{Q}(i+3) \wedge t'\mathtt{S}i$. Furthermore, there also is commutativity in the cases $t\mathtt{Q}i \wedge t'\mathtt{Q}i = t\mathtt{Q}(i+1) \wedge t'\mathtt{Q}i$, $t\mathtt{S}i \wedge t'\mathtt{Q}(i-1) = t\mathtt{Q}i \wedge t'\mathtt{S}(i-1)$, `but` *only* `when both sides are applicable. If only one of the two sides can be`

⟨ Generate the noncommutativity clauses 7* ⟩ ≡

```
for (t = 0; t <= n - 3; t++) {
  for (i = 0; i <= n - t - 2; i++)
    for (j = i + 2; j <= n - t - 2; j++) printf("~%02xQ%02x ~%02xQ%02x\n", t, j, t + 1, i);
  for (i = 0; i <= n - t - 2; i++)
    for (j = i + 2; j <= n - t - 4; j++) printf("~%02xS%02x ~%02xQ%02x\n", t, j, t + 1, i);
  for (i = 0; i <= n - t - 4; i++)
    for (j = i + 4; j <= n - t - 4; j++) printf("~%02xS%02x ~%02xS%02x\n", t, j, t + 1, i);
  for (i = 0; i <= n - t - 4; i++)
    for (j = i + 3; j <= n - t - 2; j++) printf("~%02xQ%02x ~%02xS%02x\n", t, j, t + 1, i);
  for (j = 1; j <= n - t - 2; j++)
    printf("~%02xQ%02x ~%02xQ%02x ~%02x%02x%02x\n", t, j, t + 1, j - 1, t, j - 1, j);
  for (j = 1; j <= n - t - 4; j++)
    printf("~%02xQ%02x ~%02xS%02x ~%02x%02x%02x\n", t, j, t + 1, j - 1, t, j, j + 3);
}
```

This code is used in section 1*.

8* Index.

The following sections were changed by the change file: 1, 2, 3, 7, 8.

a: 1*
Arc: 1*
argc: 1*, 2*
argv: 1*, 2*
cardname: 1*, 2*, 3*
exit: 2*
fprintf: 2*
g: 1*
gb_init_rand: 2*
gb_unif_rand: 2*
Graph: 1*
i: 1*
iprev: 6.
j: 1*
jprev: 6.
k: 1*
main: 1*
n: 1*
nmax: 1*
printf: 2*, 3*, 4, 5, 6, 7*
seed: 1*, 2*
sscanf: 2*
stamp: 3*
stderr: 2*
t: 1*
tt: 1*, 2*
v: 1*
Vertex: 1*
w: 1*

$\langle$ Generate the enabling clauses 5 $\rangle$ Used in section 1*.
$\langle$ Generate the noncommutativity clauses 7* $\rangle$ Used in section 1*.
$\langle$ Generate the possible-move clauses 4 $\rangle$ Used in section 1*.
$\langle$ Generate the transition clauses 6 $\rangle$ Used in section 1*.
$\langle$ Process the command line 2* $\rangle$ Used in section 1*.
$\langle$ Specify the initial nonadjacencies 3* $\rangle$ Used in section 1*.

SAT-GRAPH-QUENCH-NONCOMM-LATEBINDING-RANDOM

	Section	Page
Intro	1	1
Index	8	4