May 19, 2018 at 02:30

1. Intro. This program generates clauses that yield solutions (if any exist) of the following problem: Given a graph and t disjoint subsets S_j of its vertices, find disjoint connected subsets $T_j \supseteq S_j$.

(If $t = 1$ and we try to minimize T_1, this is essentially the Steiner tree problem. I'm not necessarily trying to minimize $\bigcup T_j$ in the clauses generated here, but additional cardinality constraints could be added.)

Notice that if each S_j is a pair of elements, we get interesting routing problems, including some well-known puzzles created by Loyd, Dudeney, and Dawson in the days of Queen Victoria. "Connect A to A, B to B, ..., H to H, via disjoint paths." Martin Garner reprinted one of these classics in his first column on graph theory [*Scientific American*, April 1964; *Martin Gardner's Sixthe Book of Mathematical Games*, Chapter 10], calling it a "printed-circuit problem."

The command line should specify the graph. The subsets are specified in t lines of *stdin*, by listing the vertex names (separated by spaces).

I introduce Boolean variables by appending a character to each vertex name. Therefore all vertex names should have length 7 or less.

Each S_j of size s leads to $s - 1$ sets of variables, one per vertex; every such set constrains the variables of color j to contain at least one path between the first vertex, w, of S_j, and some other vertex, z.

When these clauses are satisfied, the Boolean variables of set k that are true will be a subset of vertices whose induced graph is a path between w and z, together with zero or more cycles. Equivalently, it will be a subset in which w and z have degree 1, while all other vertices have degree 0 or 2. This subset must be disjoint from all subsets for $S_1, \ldots, S_{j-1}$. Then T_j will be the union of these subsets, over all $s - 1$ choices of z in S_j.

Since I assume that t is rather small, I don't do anything fancy to reduce the number of clauses that enforce disjointness.

```
#define bufsize 80 /* maximum length of each line of input */
#include <stdio.h>
#include <stdlib.h>
#include "gb_graph.h"
#include "gb_save.h"
  char buf[bufsize];
  char namew[bufsize], namez[bufsize];
  char code[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
  main(int argc, char *argv[])
  {
    register int j, t, m, mm;
    register Graph *g;
    register Vertex *v, *w, *z;
    register Arc *a, *b, *c;
    register char *p, *q;
    ⟨Process the command line 2⟩;
    ⟨Mark all vertices unseen 3⟩;
    for (m = 0, t = 1; ; t++) {
      if (¬fgets(buf, bufsize, stdin)) break;
      ⟨Generate clauses for a new set of vertices 4⟩;
    }
    ⟨Disable singleton vertices 9⟩;
  }
```

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);
}
hash_setup(g);
printf("~␣sat-connection␣%s\n", argv[1]);
```

This code is used in section 1.

3. **#define** *seen* *z.I*

⟨ Mark all vertices unseen 3 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + g→n; v++) v→seen = 0;
```

This code is used in section 1.

4. ⟨ Generate clauses for a new set of vertices 4 ⟩ ≡

```
mm = m;      /* remember the number of clauses sets for previous colors */
for (p = buf; *p ≡ '␣'; p++) ;      /* skip blanks */
if (*p ≡ '\n') fprintf(stderr, "Warning:␣An␣empty␣line␣of␣input␣is␣being␣ignored!\n");
else {
  for (namew[0] = *p, q = p + 1; *q ≠ '␣' ∧ *q ≠ '\n'; q++) namew[q − p] = *q;
  namew[q − p] = '\0';
  if (q − p > 7) {
    fprintf(stderr, "Sorry,␣the␣name␣of␣vertex␣%s␣is␣too␣long!\n", namew);
    exit(−3);
  }
  w = hash_out(namew);
  if (¬w) {
    fprintf(stderr, "Vertex␣%s␣isn't␣in␣that␣graph!\n", namew);
    exit(−33);
  }
  if (w→seen) {
    fprintf(stderr, "Vertex␣%s␣has␣already␣occurred!\n", namew);
    exit(−6);
  }
  w→seen = 1;
  while (1) {
    for (p = q; *p ≡ '␣'; p++) ;      /* skip blanks */
    if (*p ≡ '\n') break;
    for (namez[0] = *p, q = p + 1; *q ≠ '␣' ∧ *q ≠ '\n'; q++) namez[q − p] = *q;
    namez[q − p] = '\0';
    if (q − p > 7) {
      fprintf(stderr, "Sorry,␣the␣name␣of␣vertex␣%s␣is␣too␣long!\n", namez);
      exit(−4);
    }
    z = hash_out(namez);
    if (¬z) {
      fprintf(stderr, "Vertex␣%s␣isn't␣in␣that␣graph!\n", namez);
      exit(−44);
    }
    if (z→seen) {
      fprintf(stderr, "Vertex␣%s␣has␣already␣occurred!\n", namez);
      exit(−66);
    }
    z→seen = 1;
    if (¬code[m]) {
      fprintf(stderr, "Sorry,␣I␣can't␣handle␣this␣many␣cases!\n");
      fprintf(stderr, "Recompile␣me␣with␣a␣longer␣code␣string.\n");
      exit(−5);
    }
    printf("~␣step␣%c,␣connecting␣%s␣to␣%s\n", code[m], namew, namez);
    ⟨ Generate clauses to connect w with z 5 ⟩;
    m++;
  }
  if (mm ≡ m) {
    w→seen = −1;
    printf("~␣singleton␣vertex␣%s\n", namew);
```

```
    }
  }
```

This code is used in section 1.

5. ⟨ Generate clauses to connect w with z 5 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + g→n; v++)
  for (j = 0; j < mm; j++) printf("~%s%c␣~%s%c\n", v→name, code[m], v→name, code[j]);
for (v = g→vertices; v < g→vertices + g→n; v++) {
  if (v ≡ w ∨ v ≡ z) ⟨ Generate clauses for an endpoint 6 ⟩
  else {
    ⟨ Generate clauses to forbid v of degree < 2 7 ⟩;
    ⟨ Generate clauses to forbid v of degree > 2 8 ⟩;
  }
}
```

This code is used in section 4.

6. ⟨ Generate clauses for an endpoint 6 ⟩ ≡

```
{
  printf("%s%c\n", v→name, code[m]);    /* the endpoint is present */
  for (a = v→arcs; a; a = a→next) printf("␣%s%c", a→tip→name, code[m]);
  printf("\n");    /* at least one neighbor is present */
  for (a = v→arcs; a; a = a→next)
    for (b = a→next; b; b = b→next)
      printf("~%s%c␣~%s%c\n", a→tip→name, code[m], b→tip→name, code[m]);
        /* at most one neighbor is present */
}
```

This code is used in section 5.

7. ⟨ Generate clauses to forbid v of degree < 2 7 ⟩ ≡

```
for (a = v→arcs; a; a = a→next) {
  printf("~%s%c", v→name, code[m]);
  for (b = v→arcs; b; b = b→next)
    if (a ≠ b) printf("␣%s%c", b→tip→name, code[m]);
  printf("\n");
}
```

This code is used in section 5.

8. ⟨ Generate clauses to forbid v of degree > 2 8 ⟩ ≡

```
for (a = v→arcs; a; a = a→next)
  for (b = a→next; b; b = b→next)
    for (c = b→next; c; c = c→next) printf("~%s%c␣~%s%c␣~%s%c␣~%s%c\n", v→name, code[m],
          a→tip→name, code[m], b→tip→name, code[m], c→tip→name, code[m]);
```

This code is used in section 5.

9. The logic is a little tricky for cases when S_j contains just a single vertex, u. We want $T_j = S_j$ in such cases, but no clauses are generated. The only record of past singletons is the fact that *u→seen* is -1; so we use that fact to disallow u in $T_{j'}$ for all $j' \neq j$.

⟨ Disable singleton vertices 9 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + g→n; v++)
  if (v→seen ≡ −1)
    for (j = 0; j < m; j++) printf("~%s%c\n", v→name, code[j]);
```

This code is used in section 1.

10. Index.

SAT-CONNECTION