(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. This program implements and explains the classical HK algorithm for bipartite matching, due to John E. Hopcroft and Richard M. Karp ["An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing* **2** (1973), 225–231].

Input on *stdin* is an $m \times n$ matrix (a_{ij}) of 0s and 1s, one row per line. In this exposition the rows correspond to "boys" and the columns correspond to "girls." (Of course I do this in order to take advantage of expressive words in the English language, not to imply any distinction between male and female.) Boy i can be matched to girl j only if $a_{ij} = 1$. The object is to match as many pairs as possible. We assume that $m \le n$, because that makes the algorithm slightly faster.

Given a partial matching (possibly empty), an *augmenting path* is a simple path of the form

$$g_0 — b_0 = g_1 — b_1 = \cdots — b_{k-1} = g_k — b_k,$$

where g_0 is a free (unmatched) girl, each g_i is matchable to b_i, each b_i for $i < k$ is currently matched to g_{i+1}, and b_k is a free boy. It's called augmenting because it tells us how to improve the current matching, by breaking up k existing couples and forming $k + 1$ new ones, namely by pairing up each g_i and b_i.

The HK algorithm begins with the empty matching and proceeds in *rounds*, where each round finds a maximal set of vertex-disjoint augmenting paths, each of which has the smallest possible length. If round r finds k such paths, we've successly reduced the number of free boys (and free girls) by k.

Each round is performed in $O(t+n)$ steps, where $t = \sum a_{ij}$ is the total number of 1s in the matrix. If s is the size of a maximum matching, we shall prove that our partial matching after r rounds always has made at least $\frac{r}{r+1}s$ pairings. A clever argument (see below) shows that at most $2\sqrt{s}$ rounds are therefore needed. Consequently the worst-case running time is quite modest, only $O((t+n)\sqrt{s}\,) = O((t+n)\sqrt{m}\,)$. (Moreover, we usually don't experience a worst-case scenario. With luck we might even guess an optimum matching on the very first round.)

```
#define maxn 1000    /* n should be less than this */
#define maxt 10000    /* t should be less than this */
#define show_rounds #1    /* vbose bit for showing current round */
#define show_partials #2    /* vbose bit for showing partial matchings */
#define show_updates #4    /* vbose bit for showing each augmentation */
#define show_dags #8    /* vbose bit for showing the dag of SAPs */
#define show_answer #10    /* vbose bit for showing a maximum matching */

#include <stdio.h>
#include <stdlib.h>
  unsigned int vbose = show_rounds;    /* nonzero bits trigger optional output */
  ⟨Global variables 5⟩;    /* the main data structures are introduced in context */
  main()
  {
    register int b, f, g, i, j, k, l, m, n, p, q, qq, r, t, tt, marks, final_level;
    ⟨Input the matrix and convert it to a sparse data structure 2⟩;
    for (r = 1, f = n; f > n − m; r++) {
      if (vbose & show_rounds) fprintf(stderr, "Beginning␣round␣%d...\n", r);
      ⟨Build the dag of shortest augmenting paths (SAPs) 8⟩;
      ⟨If there are no SAPs, break 11⟩;
      ⟨Find a maximal set of disjoint SAPs, and incorporate them into the current matching 13⟩;
      if (vbose & show_rounds)
        fprintf(stderr, "␣...␣%d␣pairs␣now␣matched␣(rank␣%d).\n", n − f, final_level);
      if (vbose & show_partials) ⟨Print the current matching 7⟩;
    }
    fprintf (stderr, "Maximum␣matching␣of␣size␣%d␣found␣after␣%d␣round%s.\n", n − f,
        f ≡ n − m ? r − 1 : r, (f ≡ n − m ? r − 1 : r) ≡ 1 ? "" : "s");
    if (vbose & show_answer) ⟨Print the current matching 7⟩;
  }
```

2. The easy stuff (input).

This part of the program is trivial, but it usually takes longer than the HK algorithm itself. (Because a 0–1 matrix is not an efficient representation of a sparse matrix.)

Malformed input is rudely rejected.

```
⟨Input the matrix and convert it to a sparse data structure 2⟩ ≡
  ⟨Read the first line, and set the value of n 3⟩;
  if (n ≡ 0) {
    fprintf(stderr, "There␣must␣be␣at␣least␣one␣girl!\n");
    exit(−1);
  }
  else if (n ≡ maxn) {
    fprintf(stderr, "Recompile␣me:␣I␣can't␣deal␣with␣%d␣or␣more␣girls!\n", maxn);
    exit(−2);
  }
  for (t = 0, m = 1; ; m++) {
    if (m > n) {
      fprintf(stderr, "There␣are␣%d␣girls,␣so␣there␣can't␣be␣more␣than␣%d␣boys!\n", n, n);
      exit(−3);
    }
    ⟨Record the potential matches for boy m 4⟩;
    if (¬fgets(buf, maxn + 1, stdin)) break;
    if (buf[n] ≠ '\n') {
      fprintf(stderr, "The␣input␣for␣boy␣%d␣doesn't␣specify␣%d␣girls!\n", m + 1, n);
      exit(−4);
    }
  }
  fprintf(stderr, "OK,␣I've␣read␣the␣matrix␣for␣%d␣boys,␣%d␣girls,␣%d␣potential␣matchups.\n",
      m, n, t);
  ⟨Initialize the other data structures 16⟩;
```

This code is used in section 1.

3. ⟨Read the first line, and set the value of n 3⟩ ≡

```
  n = 0;
  if (fgets(buf, maxn + 1, stdin))
    for ( ; buf[n] ≠ '\n' ∧ n < maxn; n++) ;
```

This code is used in section 2.

4. We use a simple data structure to record the input: The potential partners for girl j are in a linked list beginning at *glink*[j], linked in *next*, and terminated by a zero link. The partner at link l is stored in *tip*[l].

⟨ Record the potential matches for boy m 4 ⟩ ≡

```
for (j = n; j; j--) {
  p = buf[j - 1] - '0';
  if (p < 0 ∨ p > 1) {
    fprintf(stderr, "Improper␣character␣`%c'␣in␣the␣input␣for␣boy␣%d!\n", buf[j - 1], m);
    exit(-5);
  }
  if (p) {
    if (++t ≥ maxt) {
      fprintf(stderr, "Recompile␣me:␣I␣can't␣handle␣%d␣or␣more␣potential␣matchups!\n", maxt);
      exit(-6);
    }
    tip[t] = m, next[t] = glink[j], glink[j] = t;
  }
}
```

This code is used in section 2.

5. Later there will be a similar data structure for some of the boys' partners.

The *next* and *tip* arrays must be able to accommodate $2t + m$ entries: t for the original graph, t for the edges at round 0, and m for the edges from ⊤.

⟨ Global variables 5 ⟩ ≡

```
char buf[maxn + 1];   /* buffer for input from stdin */
int blink[maxn], glink[maxn];   /* list heads for potential partners */
int next[maxt + maxt + maxn], tip[maxt + maxt + maxn];   /* links and suitable partners */
```

See also sections 6, 9, and 15.

This code is used in section 1.

6. Even easier stuff (the output). There's a *mate* table, showing the current partner (if any) of every boy. Also an *imate* table, showing the current partner (if any) of every girl.

⟨ Global variables 5 ⟩ +≡
```
  int mate[maxn], imate[maxn];
```

7. To report the matches-so-far, we simply show every boy's mate.

⟨ Print the current matching 7 ⟩ ≡
```
  {
    for (p = 1; p ≤ m; p++) fprintf(stderr, "␣%d", mate[p]);
    fprintf(stderr, "\n");
  }
```
This code is used in section 1.

8. The cool stuff (a dag of SAPs). With a breadth-first search, we can create a directed acyclic graph in which the paths from a dummy node called $\top$ to a dummy node called $\bot$ correspond one-to-one with the augmenting paths of minimum length. Each of those paths will contain *final_level* existing matches.

This dag has a representation something like our representation of the girls' choices, but even sparser: The first arc from boy i to a suitable girl is in *blink*[i], with *tip* and *next* as before. Each girl, however, has exactly one outgoing arc in the dag, namely her *imate*. An *imate* of 0 is a link to $\bot$. The other dummy node, $\top$, has a list of free boys, beginning at *dlink*.

An array called *mark* keeps track of the level (plus 1) at which a boy has entered the dag. All marks must be zero when we begin.

```
⟨Build the dag of shortest augmenting paths (SAPs) 8⟩ ≡
  final_level = −1, tt = t;
  for (marks = l = i = 0, q = f; ; l++) {
    for (qq = q; i < qq; i++) {
      g = queue[i];
      for (k = glink[g]; k; k = next[k]) {
        b = tip[k], p = mark[b];
        if (p ≡ 0) ⟨Enter b into the dag 10⟩
        else if (p ≤ l) continue;
        if (vbose & show_dags) fprintf(stderr, "␣%d->%d=>%d\n", b, g, imate[g]);
              /* arc from b to g, then to her mate */
        tip[++tt] = g, next[tt] = blink[b], blink[b] = tt;
      }
    }
    if (q ≡ qq) break;    /* nothing new on the queue for the next level */
  }
```

This code is used in section 1.

9. ⟨Global variables 5⟩ +≡

```
  int queue[maxn];   /* girls seen during the breadth-first search */
  int iqueue[maxn];   /* inverse permutation, for the first f entries */
  int mark[maxn];   /* where boys appear in the dag */
  int marked[maxn];   /* which boys have been marked */
  int dlink;   /* head of the list of free boys in the dag */
```

10. Once we know we've reached the final level, we don't allow any more boys at that level unless they're free. We also reset q to qq, so that the dag will not reach a greater level.

```
⟨Enter b into the dag 10⟩ ≡
  {
    if (final_level ≥ 0 ∧ mate[b]) continue;
    else if (final_level < 0 ∧ mate[b] ≡ 0) final_level = l, dlink = 0, q = qq;
    mark[b] = l + 1, marked[marks++] = b, blink[b] = 0;
    if (mate[b]) queue[q++] = mate[b];
    else {
      if (vbose & show_dags) fprintf(stderr, "␣top->%d\n", b);    /* arc from ⊤ to the free boy b */
      tip[++tt] = b, next[tt] = dlink, dlink = tt;
    }
  }
```

This code is used in section 8.

11. We have no SAPs if and only no free boys were found.

⟨ If there are no SAPs, **break** 11 ⟩ ≡

```
if (final_level < 0) break;
```

This code is used in section 1.

12. ⟨ Reset all marks to zero 12 ⟩ ≡

```
while (marks) mark[marked[--marks]] = 0;
```

This code is used in section 13.

13. Makin' progress. We've just built the dag of shortest augmenting paths, by starting from dummy node $\perp$ at the bottom and proceeding breadth-first until discovering *final_level* and essentially reaching the dummy node $\top$. Now we more or less reverse the process: We start at $\top$ and proceed *depth*-first, harvesting a maximal set of vertex-disjoint augmenting paths as we go. (Any maximal set will be fine; we needn't bother to look for an especially large one.)

The dag is gradually dismantled as SAPs are removed, so that their boys and girls won't be reused. A subtle point arises here when we look at a girl g who was part of a previous SAP: In that case her mate will have been changed to a boy whose *mark* is negative. This is true even if $l = 0$ and g was previously free.

```
⟨Find a maximal set of disjoint SAPs, and incorporate them into the current matching 13⟩ ≡
  while (dlink) {
    b = tip[dlink], dlink = next[dlink];
    l = final_level;
  enter_level: boy[l] = b;
  advance: if (blink[b]) {
      g = tip[blink[b]], blink[b] = next[blink[b]];
      if (imate[g] ≡ 0) ⟨Augment the current matching and continue 14⟩;
      if (mark[imate[g]] < 0) goto advance;
      b = imate[g], l−−;
      goto enter_level;
    }
    if (++l > final_level) continue;
    b = boy[l];
    goto advance;
  }
  ⟨Reset all marks to zero 12⟩;
```

This code is used in section 1.

14. At this point $g = g_0$ and $b = boy[0] = b_0$ in an augmenting path. The other boys are $boy[1]$, $boy[2]$, etc.

```
⟨Augment the current matching and continue 14⟩ ≡
  {
    if (l) fprintf(stderr, "I'm␣confused!\n");      /* a free girl should occur only at level 0 */
    ⟨Remove g from the list of free girls 17⟩;
    while (1) {
      if (vbose & show_updates) fprintf(stderr, "%s␣%d-%d", l ? "," : "␣match", b, g);
      mark[b] = −1;
      j = mate[b], mate[b] = g, imate[g] = b;
      if (j ≡ 0) break;      /* b was free */
      g = j, b = boy[++l];
    }
    if (vbose & show_updates) fprintf(stderr, "\n");
    continue;
  }
```

This code is used in section 13.

15. ⟨Global variables 5⟩ +≡

```
  int boy[maxn];      /* the boys being explored during the depth-first search */
```

16. When the matrix was input, we tacitly assumed that *glink*[*g*] was initially 0 for all *g*. When each dag was built, we explicitly initialized each *blink*[*b*] to 0. The *queue* and *iqueue* structures need to be initialized to nonzero values, so we do that here.

⟨Initialize the other data structures 16⟩ ≡

```
  for ($g = 1$; $g \le n$; $g$++) $queue[g-1] = g$, $iqueue[g] = g-1$;
```

This code is used in section 2.

17. ⟨Remove *g* from the list of free girls 17⟩ ≡

```
  $f$−−;    /* $f$ is the number of free girls */
  $j = iqueue[g]$;    /* where is $g$ in $queue$? */
  $i = queue[f]$, $queue[j] = i$, $iqueue[i] = j$;    /* OK to clobber $queue[f]$ */
```

This code is used in section 14.

18. Why it works. The HK algorithm relies on three simple lemmas about matchings in graphs. Recall that a *matching* in G is a subgraph of G in which every vertex has degree ≤ 1.

Suppose A and B are matchings in the same graph G, and let $A \oplus B$ be the set of edges that are in one but not both. This is a subgraph in which every vertex has degree 2 or less. Consequently every connected component of $A \oplus B$ is either a cycle or a path.

A cycle in $A \oplus B$ must have even length, because its A edges alternate with its B edges. For example, the shortest possible cycle is a 4-cycle such as

$$v_0 \mathrel{\text{—}} v_1 = v_2 \mathrel{\text{—}} v_3 = v_4,$$

where '—' denotes an edge of A and '$=$' denotes an edge of B.

A path in $A \oplus B$ can have either odd or even length. If it involves $2k+1$ vertices, so that its length is $2k$, it must be "balanced," with k edges from A and k edges from B. But if it involves $2k$ vertices, and has length $2k-1$, it must be either an A-path (with k edges from A and $k-1$ from B) or a B-path (with k edges from B and $k-1$ from A). A path in $A \oplus B$ of length $2k-1$ is said to have *rank* k. Therefore the A-paths of ranks 1, 2, 3, ..., look like this:

$$v_0 \mathrel{\text{—}} v_1, \qquad v_0 \mathrel{\text{—}} v_1 = v_2 \mathrel{\text{—}} v_3, \qquad v_0 \mathrel{\text{—}} v_1 = v_2 \mathrel{\text{—}} v_3 = v_4 \mathrel{\text{—}} v_5, \qquad \ldots;$$

and the B-paths are similar but with '—' and '$=$' reversed.

Suppose c_A components are A-paths, c_B components are B-paths, and c_0 components are either cycles or balanced paths. Then $c_A - c_B = |A| - |B|$, where $|A|$ stands for the number of edges in A. For example, if $|A| = 15$ and $|B| = 12$, we might have five A-paths and two B-paths in $A \oplus B$; or we might have four and one, etc. But in any case $A \oplus B$ must contain at least three A-paths.

Notice that every A-path is an augmenting path for B, and every B-path is an augmenting path. Therefore we have

Lemma 1. *If A and B are matchings in G with $|A| > |B|$, then B has at least $|A| - |B|$ augmenting paths that are vertex-disjoint.*

Proof. The A-paths of $A \oplus B$ are vertex-disjoint. ▌

Thus, if s is the size of a maximum matching, and if B is any matching with $|B| = r$, there must be $s - r$ disjoint augmenting paths. We don't know what they are; but we do know that they exist. That's why the HK algorithm knows that it has found a maximum matching when it is discovered that there are no augmenting paths.

19. Now let's consider the effect of an augmentation. Suppose a new matching C is created by changing, say,

$$v_0 \mathrel{\text{—}} v_1 = v_2 \mathrel{\text{—}} v_3 = v_4 \mathrel{\text{—}} v_5 \qquad \text{to} \qquad v_0 \equiv v_1 = v_2 \equiv v_3 = v_4 \equiv v_5$$

and retaining the other edges of B. What are the components of $A \oplus C$? Answer: Every vertex of $\{v_0, v_1, \ldots, v_5\}$ is isolated (has degree 0) in $A \oplus C$; so one former A-component of $A \oplus B$ has essentially vanished. But every *other* component of $A \oplus B$ remains unchanged in $A \oplus C$, except that each '$=$' becomes '$\equiv$', because it involves none of $\{v_0, v_1, \ldots, v_5\}$. (In particular, balanced components remain balanced, A-paths remain A-paths, and B-paths become C-paths.)

Algorithm HK actually works only with the *shortest* augmenting paths for B, namely the A-paths of minimum rank k. It finds q such paths, all disjoint, and uses them to find a larger matching C with $|C| = q + r$. Furthermore, the algorithm knows that every *other* augmenting path of rank k has at least one vertex v whose mate in C is different from its mate (if any) in B. (Those other SAPs appeared in the dag, when the dag was created, but they were eventually removed.)

Lemma 2. *In that scenario, every augmenting path of the matching C has rank greater than k.*

Proof. We know that an augmentation obliterates components but makes no shorter ones. The only question is whether any A-paths of rank k still exist with respect to C. But every such path was wiped out when one of its vertices was first assigned a new mate. ▌

20. An example will be helpful at this point. Consider the bipartite graph consisting of n girls $\{x_1,\ldots,x_n\}$ and n boys $\{y_1,\ldots,y_n\}$, with x_i matchable to y_j if and only if $i \le j$. This graph has a unique matching, because x_n can only be matched to y_n, and then x_{n-1} can only be matched to y_{n-1}, etc.

If A is that perfect matching and B is the empty matching, every edge $x_i \mathbin{—} y_j$ with $i \le j$ is an A-path of rank 0. One of the sets of *disjoint* A-paths is

$$x_1 \mathbin{—} y_2, \qquad x_2 \mathbin{—} y_3, \qquad \cdots, \qquad x_{n-1} \mathbin{—} y_n;$$

and this set is *maximal*, because it leaves out only x_n and y_1 (an unmatchable pair).

If we augment B with all those A-paths, we get the matching C whose $n-1$ edges are

$$x_1 \equiv y_2, \qquad x_2 \equiv y_3, \qquad \cdots, \qquad x_{n-1} \equiv y_n.$$

And Lemma 2 proves that C has no augmenting path of rank 0. In fact, C has only one augmenting path, namely

$$y_1 \mathbin{—} x_1 \equiv y_2 \mathbin{—} x_2 \equiv y_3 \mathbin{—} x_3 \cdots \mathbin{—} x_{n-1} \equiv y_n = x_n;$$

and its rank is $n-1$.

(Suppose we permute the girls of this example in any of $n!$ ways, then independently permute the boys in any of $n!$ ways, and feed the result to this program as a matrix of 0s and 1s. Surprise: The dag that is constructed in round 1 will always have its lists ordered in such a way that the optimum maximum matching is found immediately, with no need for round 2! Thus it's not as easy to find good test data as we might think, if we want to exercise the more subtle parts of this algorithm.)

21. That was an extreme example. The SAPs found in round 1 always have rank 0, and the SAPs found in round 2 usually have rank 1 (not $n-1$). And we do know, in general, that the only augmenting paths that remain after round r will have rank r or more.

Suppose a maximum matching A has s edges, and we've found a matching B of size q in round r. If $q < \frac{r}{r+1}s$, Lemmas 1 and 2 tell us that there exist $s-q$ disjoint augmenting paths, each of rank r or more. So each of them contains at least $r+1$ different edges of A, a total of at least $(r+1)(s-q)$. But that's greater than $(r+1)(s-\frac{r}{r+1}s) = s$; a contradiction! We've proved

Lemma 3. *The current matching after round r has at least $\frac{r}{r+1}s$ as many edges as the final matching.* ▌

Hopcroft and Karp therefore said, "Aha! Look at round $r = \lceil\sqrt{s}-1\rceil$. It gives us a partial match of at least $\frac{r}{r+1}s \ge s-\sqrt{s}$ edges. Therefore we'll finish in at most $\sqrt{s}$ more rounds." ▌

22. By the way, our proofs of Lemmas 1 and 2 show that they are true in *any* graph, bipartite or not. We used bipartiteness only because it gave us a way to construct a dag of all SAPs.

Hopcroft and Karp said they were hoping to extend their method from the bipartite case to the general case, presumably by taking advantage of the structure found by Jack Edmonds in his pioneering work on blossoms ["Paths, trees and flowers," *Canadian Journal on Mathematics* **17** (1965), 449–467]. That dream eventually came true when Harold N. Gabow and Robert E. Tarjan published "Faster scaling algorithms for general graph-matching problems," [*Journal of the ACM* **38** (1991), 815–853], a paper that emphasized the solution to considerably more general problems. See Harold N. Gabow, "The weighted matching approach to maximum cardinality matching," *Fundamenta Informaticae* **154** (2017), 109–130, for a simplified exposition.

23. An idea for further investigation: We've seen that at least half of the matches are initiated already in the first round. The first round is a lot simpler than subsequent rounds, because it begins with everybody free. Therefore it might be a good idea to do that round faster, with a custom-tuned implementation just to get off to a quicker start.

24. Index.

HOPCROFT-KARP