Important: Before reading LADDERS, please read or at least skim the programs for GB_WORDS and GB_DIJK.

**1. Introduction.** This demonstration program uses graphs constructed by the GB_WORDS module to produce an interactive program called `ladders`, which finds shortest paths between two given five-letter words of English.

The program assumes that UNIX conventions are being used. Some code in sections listed under 'UNIX dependencies' in the index might need to change if this program is ported to other operating systems.

To run the program under UNIX, say '`ladders` ⟨options⟩', where ⟨options⟩ consists of zero or more of the following specifications in any order:

- `-v` Verbosely print all words encountered during the shortest-path computation, showing also their distances from the goal word.
- `-a` Use alphabetic distance instead of considering adjacent words to be one unit apart; for example, the alphabetic distance from '`words`' to '`woods`' is 3, because '`r`' is three places from '`o`' in the alphabet.
- `-f` Use distance based on frequency (see below), instead of considering adjacent words to be one unit apart. This option is ignored if either `-a` or `-r` has been specified.
- `-h` Use a lower-bound heuristic to shorten the search (see below). This option is ignored if option `-f` has been selected.
- `-e` Echo the input to the output (useful if input comes from a file instead of from the terminal).
- `-n`⟨number⟩ Limit the graph to the $n$ most common English words, where $n$ is the given ⟨number⟩.
- `-r`⟨number⟩ Limit the graph to ⟨number⟩ randomly selected words. This option is incompatible with `-n`.
- `-s`⟨number⟩ Use ⟨number⟩ instead of 0 as the seed for random numbers, to get different random samples or to explore words of equal frequency in a different order.

Option `-f` assigns a cost of 0 to the most common words and a cost of 16 to the least common words; a cost between 0 and 16 is assigned to words of intermediate frequency. The word ladders that are found will then have minimum total cost by this criterion. Experience shows that the `-f` option tends to give the "friendliest," most intuitively appealing ladders.

Option `-h` attempts to focus the search by giving priority to words that are near the goal. (More precisely, it modifies distances between adjacent words by using a heuristic function $hh(v)$, which would be the shortest possible distance between $v$ and the goal if every five-letter combination happened to be an English word.) The GB_DIJK module explains more about such heuristics; this option is most interesting to watch when used in conjunction with `-v`.

**2.** The program will prompt you for a starting word. If you simply type ⟨return⟩, it exits; otherwise you should enter a five-letter word (with no uppercase letters) before typing ⟨return⟩.

Then the program will prompt you for a goal word. If you simply type ⟨return⟩ at this point, it will go back and ask for a new starting word; otherwise you should specify another five-letter word.

Then the program will find and display an optimal word ladder from the start to the goal, if there is a path from one to the other that changes only one letter at a time.

And then you have a chance to start all over again, with another starting word.

The start and goal words need not be present in the program's graph of "known" words. They are temporarily added to that graph, but removed again whenever new start and goal words are given. (Thus you can go from `sturm` to `drang` even though those words aren't English.) If the `-f` option is being used, the cost of the goal word will be 20 when it is not in the program's dictionary.

**3.** Here is the general layout of this program, as seen by the C compiler:

```
#include <ctype.h>      /* system file for character types */
#include "gb_graph.h"   /* the standard GraphBase data structures */
#include "gb_words.h"   /* routines for five-letter word graphs */
#include "gb_dijk.h"    /* routines for shortest paths */
  ⟨ Preprocessor definitions ⟩
  ⟨ Global variables 4 ⟩
  ⟨ Subroutines 11 ⟩
  main(argc, argv)
      int argc;      /* the number of command-line arguments */
      char *argv[];      /* an array of strings containing those arguments */
  {
    ⟨ Scan the command-line options 5 ⟩;
    ⟨ Set up the graph of words 6 ⟩;
    while (1) {
      ⟨ Prompt for starting word and goal word; break if none given 26 ⟩;
      ⟨ Find a minimal ladder from start to goal, if one exists, and print it 13 ⟩;
    }
    return 0;      /* normal exit */
  }
```

**4. Parsing the options.** Let's get the UNIX command-line junk out of the way first, so that we can concentrate on meatier stuff. Our job in this part of the program is to see if the default value zero of external variable *verbose* should change, and/or if the default values of any of the following internal variables should change:

⟨ Global variables 4 ⟩ ≡
```
char alph = 0;   /* nonzero if the alphabetic distance option is selected */
char freq = 0;   /* nonzero if the frequency-based distance option is selected */
char heur = 0;   /* nonzero if the heuristic search option is selected */
char echo = 0;   /* nonzero if the input-echo option is selected */
unsigned long n = 0;   /* maximum number of words in the graph (0 means infinity) */
char randm = 0;   /* nonzero if we will ignore the weight of words */
long seed = 0;   /* seed for random number generator */
```
See also sections 7, 12, and 23.

This code is used in section 3.

**5.** ⟨ Scan the command-line options 5 ⟩ ≡
```
while (--argc) {
  if (strcmp(argv[argc], "-v") ≡ 0) verbose = 1;
  else if (strcmp(argv[argc], "-a") ≡ 0) alph = 1;
  else if (strcmp(argv[argc], "-f") ≡ 0) freq = 1;
  else if (strcmp(argv[argc], "-h") ≡ 0) heur = 1;
  else if (strcmp(argv[argc], "-e") ≡ 0) echo = 1;
  else if (sscanf(argv[argc], "-n%lu", &n) ≡ 1) randm = 0;
  else if (sscanf(argv[argc], "-r%lu", &n) ≡ 1) randm = 1;
  else if (sscanf(argv[argc], "-s%ld", &seed) ≡ 1) ;
  else {
    fprintf(stderr, "Usage:␣%s␣[-v][-a][-f][-h][-e][-nN][-rN][-sN]\n", argv[0]);
    return -2;
  }
}
if (alph ∨ randm) freq = 0;
if (freq) heur = 0;
```
This code is used in section 3.

**6. Creating the graph.** The GraphBase *words* procedure will produce the five-letter words we want, organized in a graph structure.

```
#define quit_if(x, c)
        if (x) {
          fprintf(stderr, "Sorry, I couldn't build a dictionary (trouble code %ld)!\n", c);
          return c;
        }
```

```
⟨Set up the graph of words 6⟩ ≡
  g = words(n, (randm ? zero_vector : Λ), 0L, seed);
  quit_if(g ≡ Λ, panic_code);
  ⟨Confirm the options selected 8⟩;
  ⟨Modify the edge lengths, if the alph or freq option was selected 9⟩;
  ⟨Modify the priority queue algorithm, if unequal edge lengths are possible 10⟩;
```

This code is used in section 3.

**7.**
```
⟨Global variables 4⟩ +≡
  Graph *g;    /* graph created by words */
  long zero_vector[9];    /* weights to use when ignoring all frequency information */
```

**8.** The actual number of words might be decreased to the size of the GraphBase dictionary, so we wait until the graph is generated before confirming the user-selected options.

```
⟨Confirm the options selected 8⟩ ≡
  if (verbose) {
    if (alph) printf("(alphabetic distance selected)\n");
    if (freq) printf("(frequency-based distances selected)\n");
    if (heur) printf("(lowerbound heuristic will be used to focus the search)\n");
    if (randm) printf("(random selection of %ld words with seed %ld)\n", g->n, seed);
    else printf("(the graph has %ld words)\n", g->n);
  }
```

This code is used in section 6.

**9.** The edges in a *words* graph normally have length 1, so we must change them if the user has selected *alph* or *freq*. The character position in which adjacent words differ is recorded in the *loc* field of each arc. The frequency of a word is stored in the *weight* field of its vertex.

```
#define a_dist(k) (*(p + k) < *(q + k) ? *(q + k) - *(p + k) : *(p + k) - *(q + k))
```

```
⟨Modify the edge lengths, if the alph or freq option was selected 9⟩ ≡
  if (alph) { register Vertex *u;
    for (u = g->vertices + g->n - 1; u ≥ g->vertices; u--) { register Arc *a;
      register char *p = u->name;
      for (a = u->arcs; a; a = a->next) { register char *q = a->tip->name;
        a->len = a_dist(a->loc);
      }
    }
  } else if (freq) { register Vertex *u;
    for (u = g->vertices + g->n - 1; u ≥ g->vertices; u--) { register Arc *a;
      for (a = u->arcs; a; a = a->next) a->len = freq_cost(a->tip);
    }
  }
```

This code is used in section 6.

**10.** The default priority queue algorithm of *dijkstra* is quite efficient when all edge lengths are 1. Otherwise we change it to the alternative method that works best for edge lengths less than 128.

⟨ Modify the priority queue algorithm, if unequal edge lengths are possible 10 ⟩ ≡

```
if (alph ∨ freq ∨ heur) {
  init_queue = init_128;
  del_min = del_128;
  enqueue = enq_128;
  requeue = req_128;
}
```

This code is used in section 6.

**11.** The frequency has been computed with the default weights explained in the documentation of *words*; it is usually less than $2^{16}$. A word whose frequency is 0 costs 16; a word whose frequency is 1 costs 15; a word whose frequency is 2 or 3 costs 14; and the costs keep decreasing by 1 as the frequency doubles, until we reach a cost of 0.

⟨ Subroutines 11 ⟩ ≡

```
long freq_cost(v)
    Vertex *v;
{ register long acc = v→weight;    /* the frequency, to be shifted right */
  register long k = 16;
  while (acc) k--, acc >>= 1;
  return (k < 0 ? 0 : k);
}
```

See also sections 17, 18, 20, 22, and 27.

This code is used in section 3.

**12. Minimal ladders.** The guts of this program is a routine to compute shortest paths between two given words, *start* and *goal*.

The *dijkstra* procedure does this, in any graph with nonnegative arc lengths. The only complication we need to deal with here is that *start* and *goal* might not themselves be present in the graph. In that case we want to insert them, albeit temporarily.

The conventions of GB_GRAPH allow us to do the desired augmentation by creating a new graph *gg* whose vertices are borrowed from *g*. The graph *g* has space for two more vertices (actually for four), and any new memory blocks allocated for the additional arcs present in *gg* will be freed later by the operation *gb_recycle*(*gg*) without confusion.

⟨ Global variables 4 ⟩ +≡
```
Graph *gg;    /* clone of g with possible additional words */
char start[6], goal[6];    /* words dear to the user's heart, plus '\0' */
Vertex *uu, *vv;    /* start and goal vertices in gg */
```

**13.** ⟨ Find a minimal ladder from *start* to *goal*, if one exists, and print it 13 ⟩ ≡
```
⟨ Build the amplified graph gg 14 ⟩;
⟨ Let dijkstra do the hard work 21 ⟩;
⟨ Print the answer 24 ⟩;
⟨ Remove all traces of gg 25 ⟩;
```
This code is used in section 3.

**14.** ⟨ Build the amplified graph *gg* 14 ⟩ ≡
```
gg = gb_new_graph(0L);
quit_if(gg ≡ Λ, no_room + 5);    /* out of memory */
gg→vertices = g→vertices;
gg→n = g→n;
⟨ Put the start word into gg, and let uu point to it 15 ⟩;
⟨ Put the goal word into gg, and let vv point to it 16 ⟩;
if (gg→n ≡ g→n + 2) ⟨ Check if start is adjacent to goal 19 ⟩;
quit_if(gb_trouble_code, no_room + 6);    /* out of memory */
```
This code is used in section 13.

**15.** The *find_word* procedure returns Λ if it can't find the given word in the graph just constructed by *words*. In that case it has applied its second argument to every adjacent word. Hence the program logic here does everything needed to add a new vertex to *gg* when necessary.

⟨ Put the *start* word into *gg*, and let *uu* point to it 15 ⟩ ≡
```
(gg→vertices + gg→n)→name = start;    /* a tentative new vertex */
uu = find_word(start, plant_new_edge);
if (¬uu) uu = gg→vertices + gg→n++;    /* recognize the new vertex and refer to it */
```
This code is used in section 14.

**16.** ⟨ Put the *goal* word into *gg*, and let *vv* point to it 16 ⟩ ≡
```
if (strncmp(start, goal, 5) ≡ 0) vv = uu;    /* avoid inserting a word twice */
else {
  (gg→vertices + gg→n)→name = goal;    /* a tentative new vertex */
  vv = find_word(goal, plant_new_edge);
  if (¬vv) vv = gg→vertices + gg→n++;    /* recognize the new vertex and refer to it */
}
```
This code is used in section 14.

**17.** The *alph_dist* subroutine calculates the alphabetic distance between arbitrary five-letter words, whether they are adjacent or not.

⟨Subroutines 11⟩ +≡

```
long alph_dist(p, q)
    register char *p, *q;
{
  return a_dist(0) + a_dist(1) + a_dist(2) + a_dist(3) + a_dist(4);
}
```

**18.** ⟨Subroutines 11⟩ +≡

```
void plant_new_edge(v)
    Vertex *v;
{ Vertex *u = gg→vertices + gg→n;    /* the new edge runs from u to v */
  gb_new_edge(u, v, 1L);    /* we have u > v, hence v→arcs = u→arcs − 1 */
  if (alph) u→arcs→len = (u→arcs − 1)→len = alph_dist(u→name, v→name);
  else if (freq) {
    u→arcs→len = freq_cost(v);    /* adjust the arc length from u to v */
    (u→arcs − 1)→len = 20;    /* adjust the arc length from v to u */
  }
}
```

**19.** There's a bug in the above logic that could be embarrassing, although it will come up only when a user is trying to be clever: The *find_word* routine knows only the words of $g$, so it will fail to make any direct connection between *start* and *goal* if they happen to be adjacent to each other yet not in the original graph. We had better fix this, otherwise the computer will look stupid.

⟨Check if *start* is adjacent to *goal* 19⟩ ≡

```
if (hamm_dist(start, goal) ≡ 1) {
  gg→n−−;    /* temporarily pretend vv hasn't been added yet */
  plant_new_edge(uu);    /* make vv adjacent to uu */
  gg→n++;    /* and recognize it again */
}
```

This code is used in section 14.

**20.** The Hamming distance between words is the number of character positions in which they differ.

```
#define h_dist(k) (*(p + k) ≡ *(q + k) ? 0 : 1)
```

⟨Subroutines 11⟩ +≡

```
long hamm_dist(p, q)
    register char *p, *q;
{
  return h_dist(0) + h_dist(1) + h_dist(2) + h_dist(3) + h_dist(4);
}
```

**21.** OK, now we've got a graph in which *dijkstra* can operate.

⟨Let *dijkstra* do the hard work 21⟩ ≡

```
if (¬heur) min_dist = dijkstra(uu, vv, gg, Λ);
else if (alph) min_dist = dijkstra(uu, vv, gg, alph_heur);
else min_dist = dijkstra(uu, vv, gg, hamm_heur);
```

This code is used in section 13.

**22.** ⟨ Subroutines 11 ⟩ +≡

```
long alph_heur(v)
     Vertex *v;
{ return alph_dist(v→name, goal); }
long hamm_heur(v)
     Vertex *v;
{ return hamm_dist(v→name, goal); }
```

**23.** ⟨ Global variables 4 ⟩ +≡

```
long min_dist;    /* length of the shortest ladder */
```

**24.** ⟨ Print the answer 24 ⟩ ≡

```
if (min_dist < 0) printf("Sorry,␣there's␣no␣ladder␣from␣%s␣to␣%s.\n", start, goal);
else print_dijkstra_result(vv);
```

This code is used in section 13.

**25.** Finally, we have to clean up our tracks. It's easy to remove all arcs from the new vertices of *gg* to the old vertices of *g*; it's a bit trickier to remove the arcs from old to new. The loop here will also remove arcs properly between start and goal vertices, if they both belong to *gg* not *g*.

⟨ Remove all traces of *gg* 25 ⟩ ≡

```
for (uu = g→vertices + gg→n − 1; uu ≥ g→vertices + g→n; uu−−) { register Arc *a;
  for (a = uu→arcs; a; a = a→next) {
    vv = a→tip;    /* now vv→arcs ≡ a − 1, since arcs for edges come in pairs */
    vv→arcs = vv→arcs→next;
  }
  uu→arcs = Λ;    /* we needn't clear uu→name */
}
gb_recycle(gg);    /* the gg→data blocks disappear, but g→data remains */
```

This code is used in section 13.

**26. Terminal interaction.** We've finished doing all the interesting things. Only one minor part of the program still remains to be written.

```
⟨ Prompt for starting word and goal word; break if none given 26 ⟩ ≡
  putchar('\n');   /* make a blank line for visual punctuation */
restart:   /* if we try to avoid this label, the break command will be broken */
  if (prompt_for_five("Starting", start) ≠ 0) break;
  if (prompt_for_five("␣␣␣␣Goal", goal) ≠ 0) goto restart;
```

This code is used in section 3.

**27.** ⟨ Subroutines 11 ⟩ +≡

```
  long prompt_for_five(s, p)
      char *s;   /* string used in prompt message */
      register char *p;   /* where to put a string typed by the user */
  { register char *q;   /* current position to store characters */
    register long c;   /* current character of input */
    while (1) {
      printf("%s␣word:␣", s);
      fflush(stdout);   /* make sure the user sees the prompt */
      q = p;
      while (1) {
        c = getchar();
        if (c ≡ EOF) return −1;   /* end-of-file */
        if (echo) putchar(c);
        if (c ≡ '\n') break;
        if (¬islower(c)) q = p + 5;
        else if (q < p + 5) *q = c;
        q++;
      }
      if (q ≡ p + 5) return 0;   /* got a good five-letter word */
      if (q ≡ p) return 1;   /* got just ⟨return⟩ */
      printf("(Please␣type␣five␣lowercase␣letters␣and␣RETURN.)\n");
    }
  }
```

**28. Index.** Finally, here's a list that shows where the identifiers of this program are defined and used.

May 19, 2018 at 02:30

# LADDERS






Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.