Important: Before reading GB_DIJK, please read or at least skim the program for GB_GRAPH.

1. Introduction.   The GraphBase demonstration routine $dijkstra(uu, vv, gg, hh)$ finds a shortest path from vertex uu to vertex vv in graph gg, with the aid of an optional heuristic function hh. This function implements a version of Dijkstra's algorithm, a general procedure for determining shortest paths in a directed graph that has nonnegative arc lengths [E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik* **1** (1959), 269–271].

If hh is null, the length of every arc in gg must be nonnegative. If hh is non-null, hh should be a function defined on the vertices of the graph such that the length d of an arc from u to v always satisfies the condition

$$d \geq hh(u) - hh(v).$$

In such a case, we can effectively replace each arc length d by $d - hh(u) + hh(v)$, obtaining a graph with nonnegative arc lengths. The shortest paths between vertices in this modified graph are the same as they were in the original graph.

The basic idea of Dijkstra's algorithm is to explore the vertices of the graph in order of their distance from the starting vertex uu, proceeding until vv is encountered. If the distances have been modified by a heuristic function hh such that $hh(u)$ happens to equal the true distance from u to vv, for all u, then all of the modified distances on shortest paths to vv will be zero. This means that the algorithm will explore all of the most useful arcs first, without wandering off in unfruitful directions. In practice we usually don't know the exact distances to vv in advance, but we can often compute an approximate value $hh(u)$ that will help focus the search.

If the external variable *verbose* is nonzero, *dijkstra* will record its activities on the standard output file by printing the distances from uu to all vertices it visits.

After *dijkstra* has found a shortest path, it returns the length of that path. If no path from uu to vv exists (in particular, if vv is Λ), it returns -1; in such a case, the shortest distances from uu to all vertices reachable from uu will have been computed and stored in the graph. An auxiliary function, $print\_dijkstra\_result(vv)$, can be used to display the actual path found, if one exists.

Examples of the use of *dijkstra* appear in the LADDERS demonstration module.

2.   This C module is meant to be loaded as part of another program. It has the following simple structure:

#**include** `"gb_graph.h"`   /∗ define the standard GraphBase data structures ∗/
  ⟨ Preprocessor definitions ⟩
  ⟨ Priority queue procedures 16 ⟩
  ⟨ Global declarations 8 ⟩
  ⟨ The *dijkstra* procedure 9 ⟩
  ⟨ The *print_dijkstra_result* procedure 14 ⟩

3.   Users of GB_DIJK should include the header file `gb_dijk.h`:

⟨ `gb_dijk.h` 3 ⟩ ≡
  extern long *dijkstra*( );   /∗ procedure to calculate shortest paths ∗/
#**define** *print_dijkstra_result p_dijkstra_result*   /∗ shorthand for linker ∗/
  extern void *print_dijkstra_result*( );   /∗ procedure to display the answer ∗/
See also sections 5, 6, 7, and 25.

4.   The main algorithm.   As Dijkstra's algorithm proceeds, it "knows" shortest paths from uu to more and more vertices; we will call these vertices "known." Initially only uu itself is known. The procedure terminates when vv becomes known, or when all vertices reachable from uu are known.

Dijkstra's algorithm looks at all vertices adjacent to known vertices. A vertex is said to have been "seen" if it is either known or adjacent to a vertex that's known.

The algorithm proceeds by learning to know all vertices in a greater and greater radius from the starting point. Thus, if v is a known vertex at distance d from uu, every vertex at distance less than d from uu will also be known. (Throughout this discussion the word "distance" actually means "distance modified by the heuristic function"; we omit mentioning the heuristic because we can assume that the algorithm is operating on a graph with modified distances.)

The algorithm maintains an auxiliary list of all vertices that have been seen but aren't yet known. For every such vertex v, it remembers the shortest distance d from uu to v by a path that passes entirely through known vertices except for the very last arc.

This auxiliary list is actually a priority queue, ordered by the d values. If v is a vertex of the priority queue having the smallest d, we can remove v from the queue and consider it known, because there cannot be a path of length less than d from uu to v. (This is where the assumption of nonnegative arc length is crucial to the algorithm's validity.)

5.   To implement the ideas just sketched, we use several of the utility fields in vertex records. Each vertex v has a *dist* field $v\rightarrow dist$, which represents its true distance from uu if v is known; otherwise $v\rightarrow dist$ represents the shortest distance from uu discovered so far.

Each vertex v also has a *backlink* field $v\rightarrow backlink$, which is non-Λ if and only if v has been seen. In that case $v\rightarrow backlink$ is a vertex one step "closer" to uu, on a path from uu to v that achieves the current distance $v\rightarrow dist$. (Exception: Vertex uu has a backlink pointing to itself.) The backlink fields thereby allow us to construct shortest paths from uu to all the known vertices, if desired.

#**define** *dist*   *z.I*   /* distance from uu, modified by hh, appears in vertex utility field z */
#**define** *backlink*   *y.V*   /* pointer to previous vertex appears in utility field y */

$\langle$ gb_dijk.h 3 $\rangle$ +≡
#**define** *dist* *z.I*
#**define** *backlink* *y.V*

6.   The priority queue is implemented by four procedures:

  init_queue(d) makes the queue empty and prepares for subsequent keys $\geq d$.

  enqueue(v, d) puts vertex v in the queue and assigns it the key value $v\rightarrow dist = d$.

  requeue(v, d) takes vertex v out of the queue and enters it again with the smaller key value $v\rightarrow dist = d$.

  del_min( ) removes a vertex with minimum key from the queue and returns a pointer to that vertex. If the queue is empty, Λ is returned.

These procedures are accessed via external pointers, so that the user of GB_DIJK can supply alternate queueing methods if desired.

$\langle$ gb_dijk.h 3 $\rangle$ +≡
  extern void (*init_queue)( );   /* create an empty priority queue for *dijkstra* */
  extern void (*enqueue)( );   /* insert a new element in the priority queue */
  extern void (*requeue)( );   /* decrease the key of an element in the queue */
  extern Vertex *(*del_min)( );   /* remove an element with smallest key */

7.   The heuristic function might take a while to compute, so we avoid recomputation by storing $hh(v)$ in another utility field $v\rightarrow hh\_val$ once we've evaluated it.

#**define** *hh_val*   *x.I*   /* computed value of $hh(v)$ */

$\langle$ gb_dijk.h 3 $\rangle$ +≡
#**define** *hh_val* *x.I*

8. If no heuristic function is supplied by the user, we replace it by a dummy function that simply returns 0 in all cases.

⟨ Global declarations 8 ⟩ ≡
  static long *dummy*(*v*)
      Vertex ∗*v*;
  { **return** 0; }

See also section 15.

This code is used in section 2.

9. Here now is *dijkstra*:

⟨ The *dijkstra* procedure 9 ⟩ ≡
  long *dijkstra*(*uu*, *vv*, *gg*, *hh*)
      Vertex ∗*uu*;   /∗ the starting point ∗/
      Vertex ∗*vv*;   /∗ the ending point ∗/
      Graph ∗*gg*;   /∗ the graph they belong to ∗/
      long (∗*hh*)( );   /∗ heuristic function ∗/
  { **register Vertex** ∗*t*;   /∗ current vertex of interest ∗/

    if (¬*hh*) *hh* = *dummy*;   /∗ change to default heuristic ∗/
    ⟨ Make *uu* the only vertex seen; also make it known 10 ⟩;
    t = *uu*;
    if (*verbose*) ⟨ Print initial message 12 ⟩;
    while (*t* ≠ *vv*) {
      ⟨ Put all unseen vertices adjacent to *t* into the queue, and update the distances of other vertices
          adjacent to *t* 11 ⟩;
      t = (∗*del_min*)( );
      if (*t* ≡ Λ) **return** −1;   /∗ if the queue becomes empty, there's no way to get to *vv* ∗/
      if (*verbose*) ⟨ Print the distance to *t* 13 ⟩;
    }
    return *vv*→*dist* − *vv*→*hh_val* + *uu*→*hh_val*;   /∗ true distance from *uu* to *vv* ∗/
  }

This code is used in section 2.

10. As stated above, a vertex is considered seen only when its backlink isn't null, and known only when it is seen but not in the queue.

⟨ Make *uu* the only vertex seen; also make it known 10 ⟩ ≡
  for (*t* = *gg*→*vertices* + *gg*→*n* − 1; *t* ≥ *gg*→*vertices*; *t*−−) *t*→*backlink* = Λ;
  uu→*backlink* = *uu*;
  uu→*dist* = 0;
  uu→*hh_val* = (∗*hh*)(*uu*);
  (∗*init_queue*)(0_L);   /∗ make the priority queue empty ∗/

This code is used in section 9.

11.   Here we help the C compiler in case it hasn't got a great optimizer.

⟨ Put all unseen vertices adjacent to t into the queue, and update the distances of other vertices adjacent
     to t 11 ⟩ ≡
  { **register Arc** $*a$;   /∗ an arc leading from t ∗/
    register long $d = t{\rightarrow}dist - t{\rightarrow}hh\_val$;

    for $(a = t{\rightarrow}arcs; \; a; \; a = a{\rightarrow}next)$ {
      register Vertex $*v = a{\rightarrow}tip$;   /∗ a vertex adjacent to t ∗/

      if $(v{\rightarrow}backlink)$ {   /∗ v has already been seen ∗/
        register long $dd = d + a{\rightarrow}len + v{\rightarrow}hh\_val$;

        if $(dd < v{\rightarrow}dist)$ {
          $v{\rightarrow}backlink = t$;
          $(*requeue)(v, dd)$;   /∗ we found a better way to get there ∗/
        }
      } **else** {   /∗ v hasn't been seen before ∗/
        $v{\rightarrow}hh\_val = (*hh)(v)$;
        $v{\rightarrow}backlink = t$;
        $(*enqueue)(v, d + a{\rightarrow}len + v{\rightarrow}hh\_val)$;
      }
    }
  }

This code is used in section 9.

12.   The *dist* fields don't contain true distances in the graph; they represent distances modified by the
heuristic function. The true distance from uu to vertex v is $v{\rightarrow}dist - v{\rightarrow}hh\_val + uu{\rightarrow}hh\_val$.

When printing the results, we show true distances. Also, if a nontrivial heuristic is being used, we give the
hh value in brackets; the user can then observe that vertices are becoming known in order of true distance
plus *hh* value.

⟨ Print initial message 12 ⟩ ≡
  { $printf(\texttt{"Distances}_\sqcup\texttt{from}_\sqcup\texttt{\%s"}, uu{\rightarrow}name)$;
    if $(hh \neq dummy)$ $printf(\texttt{"}_\sqcup\texttt{[\%ld]"}, uu{\rightarrow}hh\_val)$;
    $printf(\texttt{":\textbackslash n"})$;
  }

This code is used in section 9.

13.   ⟨ Print the distance to t 13 ⟩ ≡
  { $printf(\texttt{"}_\sqcup\texttt{\%ld}_\sqcup\texttt{to}_\sqcup\texttt{\%s"}, t{\rightarrow}dist - t{\rightarrow}hh\_val + uu{\rightarrow}hh\_val, t{\rightarrow}name)$;
    if $(hh \neq dummy)$ $printf(\texttt{"}_\sqcup\texttt{[\%ld]"}, t{\rightarrow}hh\_val)$;
    $printf(\texttt{"}_\sqcup\texttt{via}_\sqcup\texttt{\%s\textbackslash n"}, t{\rightarrow}backlink{\rightarrow}name)$;
  }

This code is used in section 9.

14.   After *dijkstra* has found a shortest path, the backlinks from *vv* specify the steps of that path. We want to print the path in the forward direction, so we reverse the links.

    We also unreverse them again, just in case the user didn't want the backlinks to be trashed. Indeed, this procedure can be used for any vertex *vv* whose backlink is non-null, not only the *vv* that was a parameter to *dijkstra*.

    List reversal is conveniently regarded as a process of popping off one stack and pushing onto another.

#**define** *print_dijkstra_result*   *p_dijkstra_result*   /∗ shorthand for linker ∗/

⟨ The *print_dijkstra_result* procedure 14 ⟩ ≡
  void *print_dijkstra_result*(*vv*)
      Vertex ∗*vv*;   /∗ ending vertex ∗/
  { **register Vertex** ∗*t*, ∗*p*, ∗*q*;   /∗ registers for reversing links ∗/

    t = Λ, *p* = *vv*;
    if (¬*p*→*backlink*) {
      printf("Sorry,␣%s␣is␣unreachable.\n", *p*→*name*);
      return;
    }
    do {   /∗ pop an item from *p* to *t* ∗/
      q = *p*→*backlink*;
      p→*backlink* = *t*;
      t = *p*;
      p = *q*;
    } **while** (*t* ≠ *p*);   /∗ the loop stops with $t \equiv p \equiv uu$ ∗/
    do {
      printf("%10ld␣%s\n", *t*→*dist* − *t*→*hh_val* + *p*→*hh_val*, *t*→*name*);
      t = *t*→*backlink*;
    } **while** (*t*);
    t = *p*;
    do {   /∗ pop an item from *t* to *p* ∗/
      q = *t*→*backlink*;
      t→*backlink* = *p*;
      p = *t*;
      t = *q*;
    } **while** (*p* ≠ *vv*);
  }

This code is used in section 2.

15.   Priority queues.   Here we provide a simple doubly linked list for queueing; this is a convenient default, good enough for applications that aren't too large. (See MILES_SPAN for implementations of other schemes that are more efficient when the queue gets large.)

The two queue links occupy two of a vertex's remaining utility fields.

#**define** *llink*   *v.V*   /* *llink* is stored in utility field *v* of a vertex */
#**define** *rlink*   *w.V*   /* *rlink* is stored in utility field *w* of a vertex */

⟨ Global declarations 8 ⟩ +≡
   void (*init_queue)( ) = *init_dlist*;   /* create an empty dlist */
   void (*enqueue)( ) = *enlist*;   /* insert a new element in dlist */
   void (*requeue)( ) = *reenlist*;   /* decrease the key of an element in dlist */
   Vertex *(*del_min)( ) = *del_first*;   /* remove element with smallest key */

16.   There's a special list head, from which we get to everything else in the queue in decreasing order of keys by following *llink* fields.

The following declaration actually provides for 128 list heads. Only the first of these is used here, but we'll find something to do with the other 127 later.

⟨ Priority queue procedures 16 ⟩ ≡
   static Vertex *head*[128];   /* list-head elements that are always present */

   void *init_dlist*(*d*)
      long *d*;
  {
    head→*llink* = *head*→*rlink* = *head*;
    head→*dist* = *d* − 1;   /* a value guaranteed to be smaller than any actual key */
  }
See also sections 17, 18, 19, 21, 22, 23, and 24.

This code is used in section 2.

17.   It seems reasonable to assume that an element entering the queue for the first time will tend to have a larger key than the other elements.

Indeed, in the special case that all arcs in the graph have the same length, this strategy turns out to be quite fast. For in that case, every vertex is added to the end of the queue and deleted from the front, without any requeueing; the algorithm produces a strict first-in-first-out queueing discipline and performs a breadth-first search.

⟨ Priority queue procedures 16 ⟩ +≡
   void *enlist*(*v*, *d*)
      Vertex *v;
      long *d*;
  { **register Vertex** *t = *head*→*llink*;
    v→*dist* = *d*;
    while (*d* < *t*→*dist*) *t* = *t*→*llink*;
    v→*llink* = *t*;
    (*v*→*rlink* = *t*→*rlink*)→*llink* = *v*;
    t→*rlink* = *v*;
  }

18. ⟨ Priority queue procedures 16 ⟩ +≡
   void *reenlist*(*v*, *d*)
      Vertex *∗v*;
      long *d*;
  { **register Vertex** *∗t* = *v⃗llink*;

    (*t⃗rlink* = *v⃗rlink*)*⃗llink* = *v⃗llink*;   /∗ remove *v* ∗/
    v⃗dist = *d*;   /∗ we assume that the new *dist* is smaller than it was before ∗/
    while (*d* < *t⃗dist*) *t* = *t⃗llink*;
    v⃗llink = *t*;
    (*v⃗rlink* = *t⃗rlink*)*⃗llink* = *v*;
    t⃗rlink = *v*;
  }

19. ⟨ Priority queue procedures 16 ⟩ +≡
  Vertex *∗del_first*( )
  { **Vertex** *∗t*;

    t = *head⃗rlink*;
    if (*t* ≡ *head*) **return** Λ;
    (*head⃗rlink* = *t⃗rlink*)*⃗llink* = *head*;
    return *t*;
  }

20.   A special case.   When the arc lengths in the graph are all fairly small, we can substitute another queueing discipline that does each operation quickly. Suppose the only lengths are 0, 1, ..., $k-1$; then we can prove easily that the priority queue will never contain more than k different values at once. Moreover, we can implement it by maintaining k doubly linked lists, one for each key value mod k.

For example, let $k = 128$. Here is an alternate set of queue commands, to be used when the arc lengths are known to be less than 128.

21.   ⟨ Priority queue procedures 16 ⟩ +≡
   static long *master_key*;   /∗ smallest key that may be present in the priority queue ∗/

   void *init_128*(*d*)
      long *d*;
  { **register Vertex** ∗*u*;

    master_key = *d*;
    for (*u* = *head*; *u* < *head* + 128; *u*++) *u*→*llink* = *u*→*rlink* = *u*;
  }

22.   If the number of lists were not a power of 2, we would calculate a remainder by division instead of by bitwise-anding.
⟨ Priority queue procedures 16 ⟩ +≡
   Vertex ∗*del_128*( )
  { **long** *d*;
   register Vertex ∗*u*, ∗*t*;

   for (*d* = *master_key*; *d* < *master_key* + 128; *d*++) {
    u = *head* + (*d* & #7f);   /∗ that's *d* % 128 ∗/
    t = *u*→*rlink*;
    if (*t* ≠ *u*) {   /∗ we found a nonempty list with minimum key ∗/
      master_key = *d*;
      (*u*→*rlink* = *t*→*rlink*)→*llink* = *u*;
      return *t*;   /∗ incidentally, *t*→*dist* = *d* ∗/
    }
   }
   return Λ;   /∗ all 128 lists are empty ∗/
  }

23.   ⟨ Priority queue procedures 16 ⟩ +≡
   void *enq_128*(*v*, *d*)
      Vertex ∗*v*;   /∗ new vertex for the queue ∗/
      long *d*;   /∗ its *dist* ∗/
  { **register Vertex** ∗*u* = *head* + (*d* & #7f);

   v→*dist* = *d*;
   (*v*→*llink* = *u*→*llink*)→*rlink* = *v*;
   v→*rlink* = *u*;
   u→*llink* = *v*;
  }

24.   All of these operations have been so simple, one wonders why the lists should be doubly linked. Single linking would indeed be plenty—if we didn't have to support the *requeue* operation.

But requeueing involves deleting an arbitrary element from the middle of its list. And we do seem to need two links for that.

In the application to Dijkstra's algorithm, the new d will always be *master_key* or more. But we want to implement requeueing in general, so that this procedure can be used also for other algorithms such as the calculation of minimum spanning trees (see MILES_SPAN).

⟨ Priority queue procedures 16 ⟩ +≡
    void *req_128* (*v, d*)
        Vertex *∗v*;   /∗ vertex to be moved to another list ∗/
        long *d*;   /∗ its new *dist* ∗/
  { **register Vertex** *∗u* = *head* + (*d* & #**7f**);

    (*v*→*llink*→*rlink* = *v*→*rlink*)→*llink* = *v*→*llink*;   /∗ remove *v* ∗/
    v→*dist* = *d*;   /∗ the new *dist* is smaller than it was before ∗/
    (*v*→*llink* = *u*→*llink*)→*rlink* = *v*;
    v→*rlink* = *u*;
    u→*llink* = *v*;
    if (*d* < *master_key*) *master_key* = *d*;   /∗ not needed for Dijkstra's algorithm ∗/
  }

25.   The user of GB_DIJK needs to know the names of these queueing procedures if changes to the defaults are made, so we'd better put the necessary info into the header file.

⟨ `gb_dijk.h` 3 ⟩ +≡
  extern void *init_dlist* ( );
  extern void *enlist* ( );
  extern void *reenlist* ( );
  extern Vertex *∗del_first* ( );
  extern void *init_128* ( );
  extern Vertex *∗del_128* ( );
  extern void *enq_128* ( );
  extern void *req_128* ( );

26. Index.   Here is a list that shows where the identifiers of this program are defined and used.

a: $\underline{11}$.
Arc: 11.
$arcs$:   11.
$backlink$: $\underline{5}$, 10, 11, 13, 14.
d: $\underline{11}$, $\underline{16}$, $\underline{17}$, $\underline{18}$, $\underline{21}$, $\underline{22}$, $\underline{23}$, $\underline{24}$.
dd: $\underline{11}$.
del_first: 15, $\underline{19}$, $\underline{25}$.
del_min: $\underline{6}$, 9, $\underline{15}$.
del_128: $\underline{22}$, $\underline{25}$.
$dijkstra$:   1, $\underline{3}$, 6, $\underline{9}$, 14.
Dijkstra, Edsger Wybe: 1.
$dist$: $\underline{5}$, 6, 9, 10, 11, 12, 13, 14, 16, 17, 18,
     22, 23, 24.
$dummy$: $\underline{8}$, 9, 12, 13.
$enlist$:   15, $\underline{17}$, $\underline{25}$.
enq_128: $\underline{23}$, $\underline{25}$.
$enqueue$: $\underline{6}$, 11, $\underline{15}$.
gg: 1, $\underline{9}$, 10.
Graph: 9.
$head$: $\underline{16}$, 17, 19, 21, 22, 23, 24.
hh: 1, 5, 7, $\underline{9}$, 10, 11, 12, 13.
hh_val: $\underline{7}$, 9, 10, 11, 12, 13, 14.
$init\_dlist$:   15, $\underline{16}$, $\underline{25}$.
$init\_queue$: $\underline{6}$, 10, $\underline{15}$.
$init\_128$: $\underline{21}$, $\underline{25}$.
len: 11.
$llink$: $\underline{15}$, 16, 17, 18, 19, 21, 22, 23, 24.
$master\_key$: $\underline{21}$, 22, 24.
$name$:   12, 13, 14.
$next$:   11.
p: $\underline{14}$.
$p\_dijkstra\_result$:   3, 14.
$print\_dijkstra\_result$:   1, $\underline{3}$, $\underline{14}$.
$printf$:   12, 13, 14.
q: $\underline{14}$.
$reenlist$:   15, $\underline{18}$, $\underline{25}$.
req_128: $\underline{24}$, $\underline{25}$.
$requeue$: $\underline{6}$, 11, $\underline{15}$, 24.
$rlink$: $\underline{15}$, 16, 17, 18, 19, 21, 22, 23, 24.
t: $\underline{9}$, $\underline{14}$, $\underline{17}$, $\underline{18}$, $\underline{19}$, $\underline{22}$.
tip: 11.
u: $\underline{21}$, $\underline{22}$, $\underline{23}$, $\underline{24}$.
uu: 1, 4, 5, $\underline{9}$, 10, 12, 13, 14.
v: $\underline{8}$, $\underline{11}$, $\underline{17}$, $\underline{18}$, $\underline{23}$, $\underline{24}$.
$verbose$:   1, 9.
Vertex: 6, 8, 9, 11, 14, 15, 16, 17, 18, 19, 21,
     22, 23, 24, 25.
$vertices$:   10.
vv: 1, 4, $\underline{9}$, $\underline{14}$.

⟨ Global declarations 8, 15 ⟩   Used in section 2.

⟨ Make *uu* the only vertex seen; also make it known 10 ⟩   Used in section 9.

⟨ Print initial message 12 ⟩   Used in section 9.

⟨ Print the distance to *t* 13 ⟩   Used in section 9.

⟨ Priority queue procedures 16, 17, 18, 19, 21, 22, 23, 24 ⟩   Used in section 2.

⟨ Put all unseen vertices adjacent to *t* into the queue, and update the distances of other vertices adjacent
        to *t* 11 ⟩   Used in section 9.

⟨ The *dijkstra* procedure 9 ⟩   Used in section 2.

⟨ The *print_dijkstra_result* procedure 14 ⟩   Used in section 2.

⟨ `gb_dijk.h` 3, 5, 6, 7, 25 ⟩

GB_DIJK