The Stanford GraphBase

A Platform for Combinatorial Computing

DONALD E. KNUTH

The Stanford GraphBase

A Platform for Combinatorial Computing

DONALD E. KNUTH

Stanford University

ACM PRESS
New York, New York

ADDISON–WESLEY
Boston · San Francisco · New York · Toronto · Montréal
London · Munich · Paris · Madrid
Capetown · Sydney · Tokyo · Singapore · Mexico City

94. Graph products. Three ways have traditionally been used to define the product of two graphs. In all three cases the vertices of the product graph are ordered pairs (v, v'), where v and v' are vertices of the original graphs; the difference occurs in the definition of arcs. Suppose g has m arcs and n vertices, while g' has m' arcs and n' vertices. The *cartesian product* of g and g' has $mn' + m'n$ arcs, namely from (u, u') to (v, u') whenever there's an arc from u to v in g, and from (u, u') to (u, v') whenever there's an arc from u' to v' in g'. The *direct product* has mm' arcs, namely from (u, u') to (v, v') in the same circumstances. The *strong product* has both the arcs of the cartesian product and the direct product.

Notice that an undirected graph with m edges has $2m$ arcs. Thus the number of edges in the direct product of two undirected graphs is twice the product of the number of edges in the individual graphs. A self-loop in g will combine with an edge in g' to make two parallel edges in the direct product.

The subroutine call $product(g, gg, type, directed)$ produces the product graph of one of these three types, where $type = 0$ for cartesian product, $type = 1$ for direct product, and $type = 2$ for strong product. The length of an arc in the cartesian product is copied from the length of the original arc that it replicates; the length of an arc in the direct product is the minimum of the two arc lengths that induce it. If $directed = 0$, the product graph will be an undirected graph with edges consisting of consecutive arc pairs according to the standard GraphBase conventions, and the input graphs should adhere to the same conventions.

⟨ gb_basic.h 1 ⟩ +≡

```
#define cartesian 0
#define direct 1
#define strong 2
```

Arc = **struct**, GB_GRAPH §10.
arcs: **Arc** ∗, GB_GRAPH §9.
directed: **long**, §95.
g: **Graph** ∗, §87.
g: **Graph** ∗, §95.
gb_new_arc: **void** (), GB_GRAPH §30.
gb_new_edge: **void** (), GB_GRAPH §31.
gg: **Graph** ∗, §95.
m: **register long**, §87.
map = $z.V$, §88.
n: **long**, GB_GRAPH §20.
new_graph: **Graph** ∗, §9.
next: **Arc** ∗, GB_GRAPH §10.
product: **Graph** ∗(), §95.
tip: **Vertex** ∗, GB_GRAPH §10.
type: **long**, §95.
u: **register Vertex** ∗, §87.
u: **util**, GB_GRAPH §9.
v: **register Vertex** ∗, §9.
v: **util**, GB_GRAPH §9.
V: **Vertex** ∗, GB_GRAPH §8.
Vertex = **struct**, GB_GRAPH §9.
vertices: **Vertex** ∗, GB_GRAPH §20.

95. ⟨ Basic subroutines 8 ⟩ +≡

```
Graph *product(g, gg, type, directed)
      Graph *g, *gg;    /* graphs to be multiplied */
      long type;    /* cartesian, direct, or strong */
      long directed;    /* should the graph be directed? */
{ ⟨ Vanilla local variables 9 ⟩
  register Vertex *u, *vv;
  register long n;    /* the number of vertices in the product graph */
  if (g ≡ Λ ∨ gg ≡ Λ) panic(missing_operand);    /* where are g and gg? */
  ⟨ Set up a graph with ordered pairs of vertices 96 ⟩;
  if ((type & 1) ≡ 0) ⟨ Insert arcs or edges for cartesian product 97 ⟩;
  if (type) ⟨ Insert arcs or edges for direct product 99 ⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);    /* @¿*#!, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

96. We must be constantly on guard against running out of memory, especially when multiplying information.

The vertex names in the product are pairs of original vertex names separated by commas. Thus, for example, if you cross an *econ* graph with a *roget* graph, you can get vertices like `"Financial␣services,␣mediocrity"`.

⟨ Set up a graph with ordered pairs of vertices 96 ⟩ ≡

```
{ float test_product = ((float)(g→n)) * ((float)(gg→n));
  if (test_product > MAX_NNN) panic(very_bad_specs);    /* way too many vertices */
}
n = (g→n) * (gg→n);
new_graph = gb_new_graph(n);
if (new_graph ≡ Λ) panic(no_room);    /* out of memory before we're even started */
for (u = new_graph→vertices, v = g→vertices, vv = gg→vertices;
        u < new_graph→vertices + n; u++) {
  sprintf(buffer, "%.*s,%.*s", BUF_SIZE/2 − 1, v→name, (BUF_SIZE − 1)/2, vv→name);
  u→name = gb_save_string(buffer);
  if (++vv ≡ gg→vertices + gg→n) vv = gg→vertices, v++;    /* "carry" */
}
sprintf(buffer, ",%d,%d)", (type ? 2 : 0) − (int)(type & 1), directed ? 1 : 0);
make_double_compound_id(new_graph, "product(", g, ",", gg, buffer);
```

This code is used in section 95.

97. ⟨ Insert arcs or edges for cartesian product 97 ⟩ ≡

```
{ register Vertex *uu, *uuu;
  register Arc *a;
  register siz_t delta;    /* difference in memory addresses */
  delta = ((siz_t)(new_graph→vertices)) − ((siz_t)(gg→vertices));
  for (u = gg→vertices; u < gg→vertices + gg→n; u++)
```

```
      for (a = u→arcs; a; a = a→next) {
        v = a→tip;
        if (¬directed) {
          if (u > v) continue;
          if (u ≡ v ∧ a→next ≡ a + 1) a++;      /* skip second half of self-loop */
        }
        for (uu = vert_offset(u, delta), vv = vert_offset(v, delta);
                uu < new_graph→vertices + n; uu += gg→n, vv += gg→n)
          if (directed) gb_new_arc(uu, vv, a→len);
          else gb_new_edge(uu, vv, a→len);
      }
    ⟨ Insert arcs or edges for first component of cartesian product 98 ⟩;
  }
```

This code is used in section 95.

98. ⟨ Insert arcs or edges for first component of cartesian product 98 ⟩ ≡

```
  for (u = g→vertices, uu = new_graph→vertices; uu < new_graph→vertices + n; u++, uu += gg→n)
    for (a = u→arcs; a; a = a→next) {
      v = a→tip;
      if (¬directed) {
        if (u > v) continue;
        if (u ≡ v ∧ a→next ≡ a + 1) a++;      /* skip second half of self-loop */
      }
      vv = new_graph→vertices + ((gg→n) * (v − g→vertices));
      for (uuu = uu; uuu < uu + gg→n; uuu++, vv++)
        if (directed) gb_new_arc(uuu, vv, a→len);
        else gb_new_edge(uuu, vv, a→len);
    }
```

This code is used in section 97.

---

alloc_fault = −1, GB_GRAPH §7.
Arc = **struct**, GB_GRAPH §10.
arcs: **Arc** ∗, GB_GRAPH §9.
`BUF_SIZE` = 4096, §5.
buffer: **static char** [ ], §5.
cartesian = 0, §94.
direct = 1, §94.
econ: **Graph** ∗( ), GB_ECON §7.
gb_new_arc: **void** ( ), GB_GRAPH §30.
gb_new_edge: **void** ( ), GB_GRAPH §31.
gb_new_graph: **Graph** ∗( ), GB_GRAPH §23.
gb_recycle: **void** ( ), GB_GRAPH §40.
gb_save_string: **char** ∗( ), GB_GRAPH §35.
gb_trouble_code: **long**, GB_GRAPH §14.
Graph = **struct**, GB_GRAPH §20.
len: **long**, GB_GRAPH §10.
make_double_compound_id: **void** ( ), GB_GRAPH §27.
`MAX_NNN` = 1000000000.0, §13.
missing_operand = 50, GB_GRAPH §7.
n: **long**, GB_GRAPH §20.
name: **char** ∗, GB_GRAPH §9.
new_graph: **Graph** ∗, §9.
next: **Arc** ∗, GB_GRAPH §10.
no_room = 1, GB_GRAPH §7.
panic = macro ( ), §4.
roget: **Graph** ∗( ), GB_ROGET §4.
siz_t = **unsigned long**, GB_GRAPH §34.
sprintf: **int** ( ), `<stdio.h>`.
strong = 2, §94.
tip: **Vertex** ∗, GB_GRAPH §10.
v: **register Vertex** ∗, §9.
vert_offset = macro ( ), §75.
Vertex = **struct**, GB_GRAPH §9.
vertices: **Vertex** ∗, GB_GRAPH §20.
very_bad_specs = 40, GB_GRAPH §7.

99. ⟨ Insert arcs or edges for direct product 99 ⟩ ≡

```
{ Vertex *uu; Arc *a;
  siz_t delta0 = ((siz_t)(new_graph→vertices)) − ((siz_t)(gg→vertices));
  siz_t del = (gg→n) * sizeof(Vertex);
  register siz_t delta, ddelta;
  for (uu = g→vertices, delta = delta0; uu < g→vertices + g→n; uu++, delta += del)
    for (a = uu→arcs; a; a = a→next) {
      vv = a→tip;
      if (¬directed) {
        if (uu > vv) continue;
        if (uu ≡ vv ∧ a→next ≡ a + 1) a++;     /* skip second half of self-loop */
      }
      ddelta = delta0 + del * (vv − g→vertices);
      for (u = gg→vertices; u < gg→vertices + gg→n; u++) { register Arc *aa;
        for (aa = u→arcs; aa; aa = aa→next) { long length = a→len;
          if (length > aa→len) length = aa→len;
          v = aa→tip;
          if (directed) gb_new_arc(vert_offset(u, delta), vert_offset(v, ddelta), length);
          else gb_new_edge(vert_offset(u, delta), vert_offset(v, ddelta), length);
        }
      }
    }
}
```

This code is used in section 95.

100. Induced graphs. Another important way to transform a graph is to remove, identify, or split some of its vertices. All of these operations are performed by the *induced* routine, which users can invoke by calling '*induced*(*g*, *description*, *self*, *multi*, *directed*)'.

Each vertex v of g should first be assigned an "induction code" in its field $v\to ind$, which is actually utility field z. The induction code is 0 if v is to be eliminated; it is 1 if v is to be retained; it is $k > 1$ if v is to be split into k nonadjacent vertices having the same neighbors as v did; and it is $k < 0$ if v is to be identified with all other vertices having the same value of k.

For example, suppose g is a circuit with vertices $\{0, 1, \ldots, 9\}$, where j is adjacent to k if and only if $k = (j \pm 1) \bmod 10$. If we set

$$0\to ind = 0, \quad 1\to ind = 5\to ind = 9\to ind = -1, \quad 2\to ind = 3\to ind = -2,$$
$$4\to ind = 6\to ind = 8\to ind = 1, \quad \text{and } 7\to ind = 3,$$

the induced graph will have vertices $\{-1, -2, 4, 6, 7, 7', 7'', 8\}$. The vertices adjacent to 6, say, will be -1 (formerly 5), 7, $7'$, and $7''$. The vertices adjacent to -1 will be those formerly adjacent to 1, 5, or 9, namely -2 (formerly 2), 4, 6, and 8. The vertices adjacent to -2 will be those formerly adjacent to 2 or 3, namely -1 (formerly 1), -2 (formerly 3), -2 (formerly 2), and 4. Duplicate edges will be discarded if *multi* $\equiv 0$, and self-loops will be discarded if *self* $\equiv 0$.

The total number of vertices in the induced graph will be the sum of the positive *ind* fields plus the absolute value of the most negative *ind* field. This rule implies, for example, that if at least one vertex has *ind* $= -5$, the induced graph will always have a vertex -4, even though no *ind* field has been set to -4.

The *description* parameter is a string that will appear as part of the name of the induced graph; if *description* $= 0$, this string will be empty. In the latter case, users are encouraged to assign a suitable name to the *id* field of the induced graph themselves, characterizing the method by which the *ind* codes were set.

If the *directed* parameter is zero, the input graph will be assumed to be undirected, and the output graph will be undirected.

When *multi* $= 0$, the length of an arc that represents multiple arcs will be the minimum of the multiple arc lengths.

#define *ind* *z.I*

⟨ gb_basic.h 1 ⟩ +≡
#define *ind* *z.I* /* utility field *z* when used to induce a graph */

Arc = **struct**, GB_GRAPH §10.
arcs: **Arc** ∗, GB_GRAPH §9.
description: **char** ∗, §105.
directed: **long**, §95.
directed: **long**, §105.
g: **Graph** ∗, §95.
g: **Graph** ∗, §105.
gb_new_arc: **void** (), GB_GRAPH §30.
gb_new_edge: **void** (), GB_GRAPH §31.
gg: **Graph** ∗, §95.
I: **long**, GB_GRAPH §8.
id: **char** [], GB_GRAPH §20.
induced: **Graph** ∗(), §105.
len: **long**, GB_GRAPH §10.
multi: **long**, §105.
n: **long**, GB_GRAPH §20.
new_graph: **Graph** ∗, §9.
next: **Arc** ∗, GB_GRAPH §10.
self: **long**, §105.
siz_t = **unsigned long**, GB_GRAPH §34.
tip: **Vertex** ∗, GB_GRAPH §10.
u: **register Vertex** ∗, §95.
v: **register Vertex** ∗, §9.
vert_offset = macro (), §75.
Vertex = **struct**, GB_GRAPH §9.
vertices: **Vertex** ∗, GB_GRAPH §20.
vv: **register Vertex** ∗, §95.
z: **util**, GB_GRAPH §9.

101. Here's a simple example: To get a complete bipartite graph with parts of sizes *n1* and *n2*, we can start with a trivial two-point graph and split its vertices into *n1* and *n2* parts.

⟨ Applications of basic subroutines 101 ⟩ ≡

```
Graph *bi_complete(n1, n2, directed)
    unsigned long n1;    /* size of first part */
    unsigned long n2;    /* size of second part */
    long directed;    /* should all arcs go from first part to second? */
{ Graph *new_graph = board(2L, 0L, 0L, 0L, 1L, 0L, directed);
  if (new_graph) {
    new_graph->vertices->ind = n1;
    (new_graph->vertices + 1)->ind = n2;
    new_graph = induced(new_graph, Λ, 0L, 0L, directed);
    if (new_graph) {
      sprintf(new_graph->id, "bi_complete(%lu,%lu,%d)",
          n1, n2, directed ? 1 : 0);
      mark_bipartite(new_graph, n1);
    }
  }
  return new_graph;
}
```

See also section 103.

This code is used in section 2.

102. The *induced* routine also provides a special feature not mentioned above: If the *ind* field of any vertex v is `IND_GRAPH` or greater (where `IND_GRAPH` is a large constant, much larger than the number of vertices that would fit in computer memory), then utility field v→*subst* should point to a graph. A copy of the vertices of that graph will then be substituted for v in the induced graph.

This feature extends the ordinary case when v→*ind* > 0, which essentially substitutes an empty graph for v.

If substitution is being used to replace all of g's vertices by disjoint copies of some other graph g', the induced graph will be somewhat similar to a product graph. But it will not be the same as any of the three types of output produced by *product*, because the relation between g and g' is not symmetrical. Assuming that no self-loops are present, and that graphs (g, g') have respectively (m, m') arcs and (n, n') vertices, the result of substituting g' for all vertices of g has $m'n + mn'^2$ arcs.

```
#define IND_GRAPH 1000000000    /* when ind is a billion or more, */
#define subst y.G    /* we'll look at the subst field */
```

⟨ `gb_basic.h` 1 ⟩ +≡

```
#define IND_GRAPH 1000000000
#define subst y.G
```

103. For example, we can use the `IND_GRAPH` feature to create a "wheel" of n vertices arranged cyclically, all connected to one or more center points. In the directed case, the arcs will run from the center(s) to a cycle; in the undirected case, the edges will join the center(s) to a circuit.

⟨ Applications of basic subroutines 101 ⟩ +≡