


GNU Linear Programming Kit

Graph and Network Routines

for GLPK Version 4.59

(DRAFT, March 2016)


Contents

Chapter 1

Basic Graph API Routines

1.1 Graph program object

In GLPK the base program object used to represent graphs and networks is a directed graph (digraph).

Formally, *digraph* (or simply, *graph*) is a pair $G = (V, A)$, where V is a set of *vertices*, and A is a set *arcs*.[1] Each arc $a \in A$ is an ordered pair of vertices $a = (x, y)$, where $x \in V$ is called *tail vertex* of arc a, and $y \in V$ is called its *head vertex*.

Representation of a graph in the program includes three structs defined by typedef in the header `glpk.h`:

— `glp_graph`, which represents the graph in a whole,

— `glp_vertex`, which represents a vertex of the graph, and

— `glp_arc`, which represents an arc of the graph.

All these three structs are "semi-opaque", i.e. the application program can directly access their fields through pointers, however, changing the fields directly is not allowed — all changes should be performed only with appropriate GLPK API routines.

1.1.1 Structure glp_graph

The struct `glp_graph` has the following fields available to the application program.

```
char *name;
```

Symbolic name assigned to the graph. It is a pointer to a null terminated character string of length from 1 to 255 characters. If no name is assigned to the graph, this field contains `NULL`.

```
int nv;
```

The number of vertices in the graph, $nv \geq 0$.

```
int na;
```

The number of arcs in the graph, $na \geq 0$.

[1] A may be a multiset.

`glp_vertex **v;`

Pointer to an array containing the list of vertices. Element $v[0]$ is not used. Element $v[i]$, $1 \leq i \leq nv$, is a pointer to i-th vertex of the graph. Note that on adding new vertices to the graph the field v may be altered due to reallocation. However, pointers $v[i]$ are not changed while corresponding vertices exist in the graph.

`int v_size;`

Size of vertex data blocks, in bytes, $0 \leq v\_size \leq 256$. (See also the field `data` in the struct `glp_vertex`.)

`int a_size;`

Size of arc data blocks, in bytes, $0 \leq v\_size \leq 256$. (See also the field `data` in the struct `glp_arc`.)

1.1.2 Structure glp_vertex

The struct `glp_vertex` has the following fields available to the application program.

`int i;`

Ordinal number of the vertex, $1 \leq i \leq nv$. Note that element $v[i]$ in the struct `glp_graph` points to the vertex, whose ordinal number is i.

`char *name;`

Symbolic name assigned to the vertex. It is a pointer to a null terminated character string of length from 1 to 255 characters. If no name is assigned to the vertex, this field contains `NULL`.

`void *data;`

Pointer to a data block associated with the vertex. This data block is automatically allocated on creating a new vertex and freed on deleting the vertex. If $v\_size = 0$, the block is not allocated, and this field contains `NULL`.

`void *temp;`

Working pointer, which may be used freely for any purposes. The application program can change this field directly.

`glp_arc *in;`

Pointer to the (unordered) list of incoming arcs. If the vertex has no incoming arcs, this field contains `NULL`.

`glp_arc *out;`

Pointer to the (unordered) list of outgoing arcs. If the vertex has no outgoing arcs, this field contains `NULL`.

1.1.3 Structure glp_arc

The struct `glp_arc` has the following fields available to the application program.

`glp_vertex *tail;`

Pointer to a vertex, which is tail endpoint of the arc.

`glp_vertex *head;`

Pointer to a vertex, which is head endpoint of the arc.

`void *data;`

Pointer to a data block associated with the arc. This data block is automatically allocated on creating a new arc and freed on deleting the arc. If $v\_size = 0$, the block is not allocated, and this field contains `NULL`.

`void *temp;`

Working pointer, which may be used freely for any purposes. The application program can change this field directly.

`glp_arc *t_next;`

Pointer to another arc, which has the same tail endpoint as this one. `NULL` in this field indicates the end of the list of outgoing arcs.

`glp_arc *h_next;`

Pointer to another arc, which has the same head endpoint as this one. `NULL` in this field indicates the end of the list of incoming arcs.

1.2 Graph creating and modifying routines

1.2.1 glp_create_graph — create graph

Synopsis

```
glp_graph *glp_create_graph(int v_size, int a_size);
```

Description

The routine `glp_create_graph` creates a new graph, which initially is empty, i.e. has no vertices and arcs.

The parameter `v_size` specifies the size of vertex data blocks, in bytes, $0 \leq v\_size \leq 256$.

The parameter `a_size` specifies the size of arc data blocks, in bytes, $0 \leq a\_size \leq 256$.

Returns

The routine returns a pointer to the graph object created.

1.2.2 glp_set_graph_name — assign (change) graph name

Synopsis

```
void glp_set_graph_name(glp_graph *G, const char *name);
```

Description

The routine `glp_set_graph_name` assigns a symbolic name specified by the character string `name` (1 to 255 chars) to the graph.

If the parameter `name` is `NULL` or an empty string, the routine erases the existing symbolic name of the graph.

1.2.3 glp_add_vertices — add new vertices to graph

Synopsis

```
int glp_add_vertices(glp_graph *G, int nadd);
```

Description

The routine `glp_add_vertices` adds `nadd` vertices to the specified graph. New vertices are always added to the end of the vertex list, so ordinal numbers of existing vertices remain unchanged. Note that this operation may change the field `v` in the struct `glp_graph` (pointer to the vertex array) due to reallocation.

Being added each new vertex is isolated, i.e. has no incident arcs.

If the size of vertex data blocks specified on creating the graph is non-zero, the routine also allocates a memory block of that size for each new vertex added, fills it by binary zeros, and stores a pointer to it in the field `data` of the struct `glp_vertex`. Otherwise, if the block size is zero, the field `data` is set to `NULL`.

Returns

The routine `glp_add_vertices` returns the ordinal number of the first new vertex added to the graph.

1.2.4 glp_set_vertex_name — assign (change) vertex name

Synopsis

```
void glp_set_vertex_name(glp_graph *G, int i, const char *name);
```

Description

The routine `glp_set_vertex_name` assigns a given symbolic name (1 up to 255 characters) to `i`-th vertex of the specified graph.

If the parameter `name` is `NULL` or empty string, the routine erases an existing name of `i`-th vertex.

1.2.5 glp_add_arc — add new arc to graph

Synopsis

```
glp_arc *glp_add_arc(glp_graph *G, int i, int j);
```

Description

The routine `glp_add_arc` adds one new arc to the specified graph.

The parameters `i` and `j` specify the ordinal numbers of, resp., tail and head endpoints (vertices) of the arc. Note that self-loops and multiple arcs are allowed.

If the size of arc data blocks specified on creating the graph is non-zero, the routine also allocates a memory block of that size, fills it by binary zeros, and stores a pointer to it in the field `data` of the struct `glp_arc`. Otherwise, if the block size is zero, the field `data` is set to `NULL`.

1.2.6 glp_del_vertices — delete vertices from graph

Synopsis

```
void glp_del_vertices(glp_graph *G, int ndel, const int num[]);
```

Description

The routine `glp_del_vertices` deletes vertices along with all incident arcs from the specified graph. Ordinal numbers of vertices to be deleted should be placed in locations `num[1]`, ..., `num[ndel]`, `ndel` > 0.

Note that deleting vertices involves changing ordinal numbers of other vertices remaining in the graph. New ordinal numbers of the remaining vertices are assigned under the assumption that the original order of vertices is not changed.

1.2.7 glp_del_arc — delete arc from graph

Synopsis

```
void glp_del_arc(glp_graph *G, glp_arc *a);
```

Description

The routine `glp_del_arc` deletes an arc from the specified graph. The arc to be deleted must exist.

1.2.8 glp_erase_graph — erase graph content

Synopsis

```
void glp_erase_graph(glp_graph *G, int v_size, int a_size);
```

Description

The routine `glp_erase_graph` erases the content of the specified graph. The effect of this operation is the same as if the graph would be deleted with the routine `glp_delete_graph` and then created anew with the routine `glp_create_graph`, with exception that the pointer to the graph remains valid.

The parameters `v_size` and `a_size` have the same meaning as for `glp_create_graph`.

1.2.9 glp_delete_graph — delete graph

Synopsis

```
void glp_delete_graph(glp_graph *G);
```

Description

The routine `glp_delete_graph` deletes the specified graph and frees all the memory allocated to this program object.

1.3 Graph searching routines

1.3.1 glp_create_v_index — create vertex name index

Synopsis

```
void glp_create_v_index(glp_graph *G);
```

Description

The routine `glp_create_v_index` creates the name index for the specified graph. The name index is an auxiliary data structure, which is intended to quickly (i.e. for logarithmic time) find vertices by their names.

This routine can be called at any time. If the name index already exists, the routine does nothing.

1.3.2 glp_find_vertex — find vertex by its name

Synopsis

```
int glp_find_vertex(glp_graph *G, const char *name);
```

Returns

The routine `glp_find_vertex` returns the ordinal number of a vertex, which is assigned (by the routine `glp_set_vertex_name`) the specified symbolic `name`. If no such vertex exists, the routine returns 0.

1.3.3 glp_delete_v_index — delete vertex name index

Synopsis

```
void glp_delete_v_index(glp_graph *G);
```

Description

The routine `glp_delete_v_index` deletes the name index previously created by the routine `glp_create_v_index` and frees the memory allocated to this auxiliary data structure.

This routine can be called at any time. If the name index does not exist, the routine does nothing.

1.4 Graph reading/writing routines

1.4.1 glp_read_graph — read graph from text file

Synopsis

```
int glp_read_graph(glp_graph *G, const char *fname);
```

Description

The routine `glp_read_graph` reads a graph from a text file, whose name is specified by the parameter `fname`. It is equivalent to

```
glp_read_ccdata(G, -1, fname);
```

Note that before reading data the current content of the graph object is completely erased with the routine `glp_erase_graph`.

Returns

If the operation was successful, the routine returns zero. Otherwise it prints an error message and returns non-zero.

1.4.2 glp_write_graph — write graph to text file

Synopsis

```
int glp_write_graph(glp_graph *G, const char *fname);
```

Description

The routine `glp_write_graph` writes the graph to a text file, whose name is specified by the parameter `fname`. It is equivalent to

```
glp_write_ccdata(G, -1, fname);
```

Returns

If the operation was successful, the routine returns zero. Otherwise it prints an error message and returns non-zero.

1.4.3 glp_read_ccdata — read graph from text file in DIMACS clique/coloring format

Synopsis

```
int glp_read_ccdata(glp_graph *G, int v_wgt, const char *fname);
```

Description

The routine `glp_read_ccdata` reads a graph from a text file in DIMACS clique/coloring format. (Though this format is originally designed to represent data for the minimal vertex coloring and maximal clique problems, it may be used to represent general undirected and directed graphs, because the routine allows reading self-loops and multiple edges/arcs keeping the order of vertices specified for each edge/arc of the graph.)

The parameter `G` specifies the graph object to be read in. Note that before reading data the current content of the graph object is completely erased with the routine `glp_erase_graph`.

The parameter `v_wgt` specifies an offset of the field of type `double` in the vertex data block, to which the routine stores the vertex weight. If `v_wgt` < 0, the vertex weights are not stored.

The character string `fname` specifies the name of a text file to be read in. (If the file name ends with the suffix '`.gz`', the file is assumed to be compressed, in which case the routine decompresses it "on the fly".)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

DIMACS clique/coloring format[2]

The DIMACS input file is a plain ASCII text file. It contains *lines* of several types described below. A line is terminated with an end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

Note that DIMACS requires all numerical quantities to be integers in the range $[-2^{31}, 2^{31} - 1]$ while GLPK allows the quantities to be floating-point numbers.

Comment lines. Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character `c`.

```
c This is a comment line
```

Problem line. There is one problem line per data file. The problem line must appear before any node or edge descriptor lines. It has the following format:

```
p edge NODES EDGES
```

The lower-case letter `p` signifies that this is a problem line. The four-character problem designator `edge` identifies the file as containing data for the minimal vertex coloring or maximal clique problem. The `NODES` field contains an integer value specifying the number of vertices in the graph. The `EDGES` field contains an integer value specifying the number of edges (arcs) in the graph.

Vertex descriptors. These lines give the weight assigned to a vertex of the graph. There is one vertex descriptor line for each vertex, with the following format. Vertices without a descriptor take on a default value of 1.

```
n ID VALUE
```

The lower-case character `n` signifies that this is a vertex descriptor line. The `ID` field gives a vertex identification number, an integer between 1 and n, where n is the number of vertices in the graph. The `VALUE` field gives a vertex weight, which can either positive or negative (or zero).

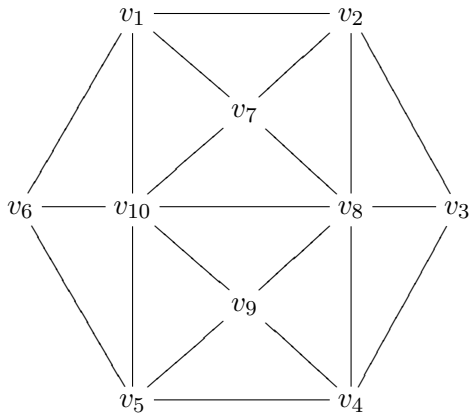
Edge descriptors. There is one edge descriptor line for each edge (arc) of the graph, each with the following format:

```
e I J
```

The lower-case character `e` signifies that this is an edge descriptor line. For an edge (arc) (i, j) the fields `I` and `J` specify its endpoints.

[2]This material is based on the paper "Clique and Coloring Problems Graph Format", which is publicly available at http://dimacs.rutgers.edu/Challenges.

Example. The following undirected graph



might be coded in DIMACS clique/coloring format as follows.

```
c sample.col
c
c This is an example of the vertex coloring problem data
c in DIMACS format.
c
p edge 10 21
c
e 1 2
e 1 6
e 1 7
e 1 10
e 2 3
e 2 7
e 2 8
e 3 4
e 3 8
e 4 5
e 4 8
e 4 9
e 5 6
e 5 9
e 5 10
e 6 10
e 7 8
e 7 10
e 8 9
e 8 10
e 9 10
c
c eof
```

1.4.4 glp_write_ccdata — write graph to text file in DIMACS clique/coloring format

Synopsis

```
int glp_write_ccdata(glp_graph *G, int v_wgt, const char *fname);
```

Description

The routine `glp_write_ccdata` writes the graph object specified by the parameter `G` to a text file in DIMACS clique/coloring format. (Though this format is originally designed to represent data for the minimal vertex coloring and maximal clique problems, it may be used to represent general undirected and directed graphs, because the routine allows writing self-loops and multiple edges/arcs keeping the order of vertices specified for each edge/arc of the graph.)

The parameter `v_wgt` specifies an offset of the field of type `double` in the vertex data block, which contains the vertex weight. If `v_wgt` < 0, it is assumed that the weight of each vertex is 1.

The character string `fname` specifies a name of the text file to be written out. (If the file name ends with suffix '`.gz`', the file is assumed to be compressed, in which case the routine performs automatic compression on writing it.)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

1.5 Graph analysis routines

1.5.1 glp_weak_comp — find all weakly connected components of graph

Synopsis

```
int glp_weak_comp(glp_graph *G, int v_num);
```

Description

The routine `glp_weak_comp` finds all weakly connected components of the specified graph.

The parameter `v_num` specifies an offset of the field of type `int` in the vertex data block, to which the routine stores the number of a weakly connected component containing that vertex. If `v_num` < 0, no component numbers are stored.

The components are numbered in arbitrary order from 1 to `nc`, where `nc` is the total number of components found, $0 \leq$ `nc` $\leq |V|$.

Returns

The routine returns `nc`, the total number of components found.

1.5.2 glp_strong_comp — find all strongly connected components of graph

Synopsis

```
int glp_strong_comp(glp_graph *G, int v_num);
```

Description

The routine `glp_strong_comp` finds all strongly connected components of the specified graph.

The parameter `v_num` specifies an offset of the field of type `int` in the vertex data block, to which the routine stores the number of a strongly connected component containing that vertex. If `v_num` < 0, no component numbers are stored.

The components are numbered in arbitrary order from 1 to `nc`, where `nc` is the total number of components found, $0 \leq$ `nc` $\leq |V|$. However, the component numbering has the property that for every arc $(i \rightarrow j)$ in the graph the condition $num(i) \geq num(j)$ holds.

Returns

The routine returns `nc`, the total number of components found.

References

I. S. Duff, J. K. Reid, Algorithm 529: Permutations to block triangular form, ACM Trans. on Math. Softw. 4 (1978), 189-92.

Example

The following program reads a graph from a plain text file '`graph.txt`' and finds all its strongly connected components.

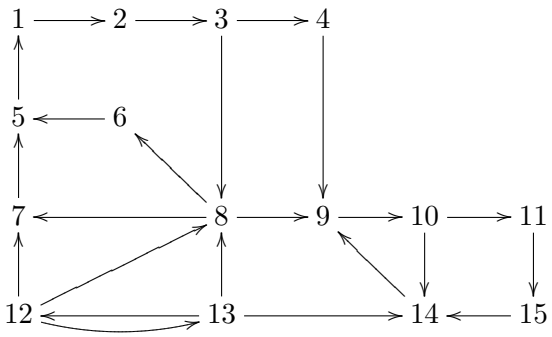
```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>

typedef struct { int num; } v_data;

#define vertex(v) ((v_data *)((v)->data))

int main(void)
{     glp_graph *G;
      int i, nc;
      G = glp_create_graph(sizeof(v_data), 0);
      glp_read_graph(G, "graph.txt");
      nc = glp_strong_comp(G, offsetof(v_data, num));
      printf("nc = %d\n", nc);
      for (i = 1; i <= G->nv; i++)
         printf("num[%d] = %d\n", i, vertex(G->v[i])->num);
      glp_delete_graph(G);
      return 0;
}
```

If the file '`graph.txt`' contains the following graph:



the program output may look like follows:

```
Reading graph from 'graph.txt'...
Graph has 15 vertices and 30 arcs
31 lines were read
nc = 4
num[1] = 3
num[2] = 3
num[3] = 3
num[4] = 2
num[5] = 3
num[6] = 3
num[7] = 3
num[8] = 3
num[9] = 1
num[10] = 1
num[11] = 1
num[12] = 4
```

```
num[13] = 4
num[14] = 1
num[15] = 1
```

1.5.3 glp_top_sort — topological sorting of acyclic digraph

Synopsis

```
int glp_top_sort(glp_graph *G, int v_num);
```

Description

The routine `glp_top_sort` performs topological sorting of vertices of the specified acyclic digraph.

The parameter `v_num` specifies an offset of the field of type `int` in the vertex data block, to which the routine stores the vertex number assigned. If `v_num` < 0, vertex numbers are not stored.

The vertices are numbered from 1 to n, where n is the total number of vertices in the graph. The vertex numbering has the property that for every arc $(i \rightarrow j)$ in the graph the condition $num(i) < num(j)$ holds. Special case $num(i) = 0$ means that vertex i is not assigned a number, because the graph is *not* acyclic.

Returns

If the graph is acyclic and therefore all the vertices have been assigned numbers, the routine `glp_top_sort` returns zero. Otherwise, if the graph is not acyclic, the routine returns the number of vertices which have not been numbered, i.e. for which $num(i) = 0$.

Example

The following program reads a digraph from a plain text file '`graph.txt`' and performs topological sorting of its vertices.

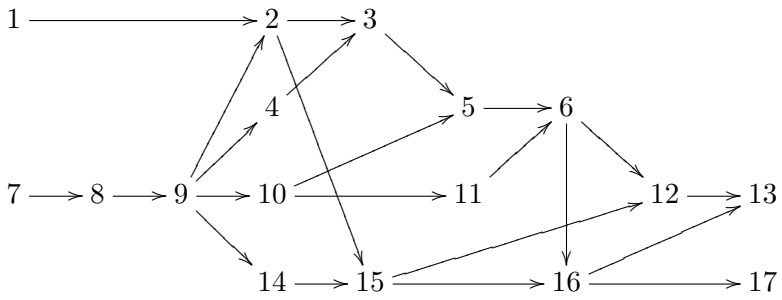
```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>

typedef struct { int num; } v_data;

#define vertex(v) ((v_data *)((v)->data))

int main(void)
{     glp_graph *G;
      int i, cnt;
      G = glp_create_graph(sizeof(v_data), 0);
      glp_read_graph(G, "graph.txt");
      cnt = glp_top_sort(G, offsetof(v_data, num));
      printf("cnt = %d\n", cnt);
      for (i = 1; i <= G->nv; i++)
         printf("num[%d] = %d\n", i, vertex(G->v[i])->num);
      glp_delete_graph(G);
      return 0;
}
```

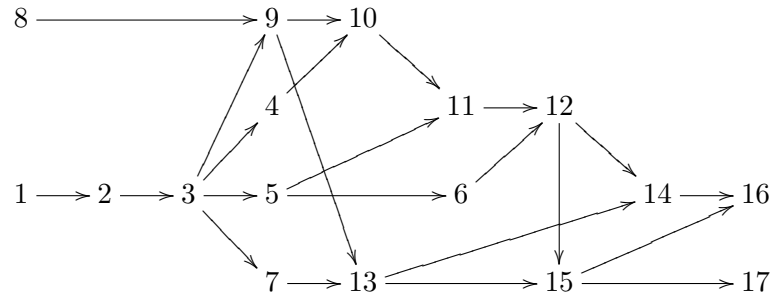
If the file '`graph.txt`' contains the following graph:



the program output may look like follows:

```
Reading graph from 'graph.txt'...
Graph has 17 vertices and 23 arcs
24 lines were read
cnt = 0
num[1] = 8
num[2] = 9
num[3] = 10
num[4] = 4
num[5] = 11
num[6] = 12
num[7] = 1
num[8] = 2
num[9] = 3
num[10] = 5
num[11] = 6
num[12] = 14
num[13] = 16
num[14] = 7
num[15] = 13
num[16] = 15
num[17] = 17
```

The output corresponds to the following vertex numbering:



Chapter 2

Network optimization API routines

2.1 Minimum cost flow problem

2.1.1 Background

The *minimum cost flow problem* (MCFP) is stated as follows. Let there be given a directed graph (flow network) $G = (V, A)$, where V is a set of vertices (nodes), and $A \subseteq V \times V$ is a set of arcs. Let for each node $i \in V$ there be given a quantity b_i having the following meaning:

if $b_i > 0$, then $|b_i|$ is a *supply* at node i, which shows how many flow units are *generated* at node i (or, equivalently, entering the network through node i from outside);

if $b_i < 0$, then $|b_i|$ is a *demand* at node i, which shows how many flow units are *lost* at node i (or, equivalently, leaving the network through node i to outside);

if $b_i = 0$, then i is a *transshipment* node, at which the flow is conserved, i.e. neither generated nor lost.

Let also for each arc $a = (i, j) \in A$ there be given the following three quantities:

l_{ij}, a (non-negative) lower bound to the flow through arc (i, j);

u_{ij}, an upper bound to the flow through arc (i, j), which is the *arc capacity*;

c_{ij}, a per-unit cost of the flow through arc (i, j).

The problem is to find flows x_{ij} through every arc of the network, which satisfy the specified bounds and the conservation constraints at all nodes, and minimize the total flow cost. Here the conservation constraint at a node means that the total flow entering this node through its incoming arcs plus the supply at this node must be equal to the total flow leaving this node through its outgoing arcs plus the demand at this node.

An example of the minimum cost flow problem is shown on Fig. 1.

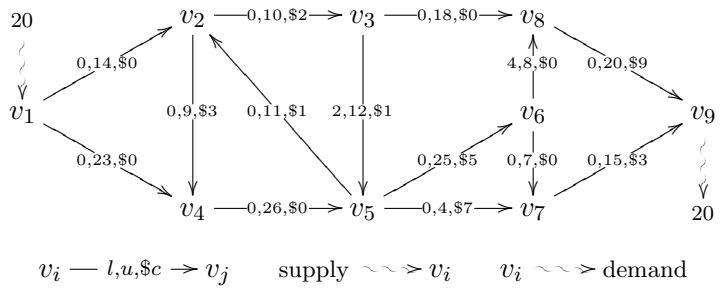


Fig. 1. An example of the minimum cost flow problem.

The minimum cost flow problem can be naturally formulated as the following LP problem:

minimize

$$z = \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1}$$

subject to

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b_i \quad \text{for all } i \in V \tag{2}$$

$$l_{ij} \le x_{ij} \le u_{ij} \quad \text{for all } (i,j) \in A \tag{3}$$

2.1.2 glp_read_mincost — read minimum cost flow problem data in DIMACS format

Synopsis

```
int glp_read_mincost(glp_graph *G, int v_rhs, int a_low, int a_cap,
                     int a_cost, const char *fname);
```

Description

The routine `glp_read_mincost` reads the minimum cost flow problem data from a text file in DIMACS format.

The parameter `G` specifies the graph object, to which the problem data have to be stored. Note that before reading data the current content of the graph object is completely erased with the routine `glp_erase_graph`.

The parameter `v_rhs` specifies an offset of the field of type `double` in the vertex data block, to which the routine stores b_i, the supply/demand value. If `v_rhs` < 0, the value is not stored.

The parameter `a_low` specifies an offset of the field of type `double` in the arc data block, to which the routine stores l_{ij}, the lower bound to the arc flow. If `a_low` < 0, the lower bound is not stored.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, to which the routine stores u_{ij}, the upper bound to the arc flow (the arc capacity). If `a_cap` < 0, the upper bound is not stored.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, to which the routine stores c_{ij}, the per-unit cost of the arc flow. If `a_cost` < 0, the cost is not stored.

The character string `fname` specifies the name of a text file to be read in. (If the file name name ends with the suffix '`.gz`', the file is assumed to be compressed, in which case the routine decompresses it "on the fly".)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

Example

```
typedef struct
{     /* vertex data block */
      ...
      double rhs;
      ...
} v_data;

typedef struct
{     /* arc data block */
      ...
      double low, cap, cost;
      ...
} a_data;

int main(void)
{     glp_graph *G;
      int ret;
      G = glp_create_graph(sizeof(v_data), sizeof(a_data));
      ret = glp_read_mincost(G, offsetof(v_data, rhs),
         offsetof(a_data, low), offsetof(a_data, cap),
         offsetof(a_data, cost), "sample.min");
      if (ret != 0) goto ...
      ...
}
```

DIMACS minimum cost flow problem format[1]

The DIMACS input file is a plain ASCII text file. It contains *lines* of several types described below. A line is terminated with an end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

Note that DIMACS requires all numerical quantities to be integers in the range $[-2^{31},\ 2^{31}-1]$ while GLPK allows the quantities to be floating-point numbers.

Comment lines. Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character `c`.

```
c This is a comment line
```

Problem line. There is one problem line per data file. The problem line must appear before any node or arc descriptor lines. It has the following format:

```
p min NODES ARCS
```

[1] This material is based on the paper "The First DIMACS International Algorithm Implementation Challenge: Problem Definitions and Specifications", which is publicly available at http://dimacs.rutgers.edu/Challenges.

The lower-case character `p` signifies that this is a problem line. The three-character problem designator `min` identifies the file as containing specification information for the minimum cost flow problem. The `NODES` field contains an integer value specifying the number of nodes in the network. The `ARCS` field contains an integer value specifying the number of arcs in the network.

Node descriptors. All node descriptor lines must appear before all arc descriptor lines. The node descriptor lines describe supply and demand nodes, but not transshipment nodes. That is, only nodes with non-zero node supply/demand values appear. There is one node descriptor line for each such node, with the following format:

```
n ID FLOW
```

The lower-case character `n` signifies that this is a node descriptor line. The `ID` field gives a node identification number, an integer between 1 and `NODES`. The `FLOW` field gives the amount of supply (if positive) or demand (if negative) at node `ID`.

Arc descriptors. There is one arc descriptor line for each arc in the network. Arc descriptor lines are of the following format:

```
a SRC DST LOW CAP COST
```

The lower-case character `a` signifies that this is an arc descriptor line. For a directed arc (i, j) the `SRC` field gives the identification number i for the tail endpoint, and the `DST` field gives the identification number j for the head endpoint. Identification numbers are integers between 1 and `NODES`. The `LOW` field specifies the minimum amount of flow that can be sent along arc (i, j), and the `CAP` field gives the maximum amount of flow that can be sent along arc (i, j) in a feasible flow. The `COST` field contains the per-unit cost of flow sent along arc (i, j).

Example. Below here is an example of the data file in DIMACS format corresponding to the minimum cost flow problem shown on Fig 1.

```
c sample.min
c
c This is an example of the minimum cost flow problem data
c in DIMACS format.
c
p min 9 14
c
n 1 20
n 9 -20
c
a 1 2 0 14 0
a 1 4 0 23 0
a 2 3 0 10 2
a 2 4 0  9 3
a 3 5 2 12 1
a 3 8 0 18 0
a 4 5 0 26 0
a 5 2 0 11 1
a 5 6 0 25 5
a 5 7 0  4 7
a 6 7 0  7 0
a 6 8 4  8 0
a 7 9 0 15 3
a 8 9 0 20 9
c
c eof
```

2.1.3 glp_write_mincost — write minimum cost flow problem data in DIMACS format

Synopsis

```
int glp_write_mincost(glp_graph *G, int v_rhs, int a_low, int a_cap,
                      int a_cost, const char *fname);
```

Description

The routine `glp_write_mincost` writes the minimum cost flow problem data to a text file in DIMACS format.

The parameter `G` is the graph (network) program object, which specifies the minimum cost flow problem instance.

The parameter `v_rhs` specifies an offset of the field of type `double` in the vertex data block, which contains b_i, the supply/demand value. If `v_rhs` < 0, it is assumed that $b_i = 0$ for all nodes.

The parameter `a_low` specifies an offset of the field of type `double` in the arc data block, which contains l_{ij}, the lower bound to the arc flow. If `a_low` < 0, it is assumed that $l_{ij} = 0$ for all arcs.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, which contains u_{ij}, the upper bound to the arc flow (the arc capacity). If the upper bound is specified as `DBL_MAX`, it is assumed that $u_{ij} = \infty$, i.e. the arc is uncapacitated. If `a_cap` < 0, it is assumed that $u_{ij} = 1$ for all arcs.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, which contains c_{ij}, the per-unit cost of the arc flow. If `a_cost` < 0, it is assumed that $c_{ij} = 0$ for all arcs.

The character string `fname` specifies a name of the text file to be written out. (If the file name ends with suffix '`.gz`', the file is assumed to be compressed, in which case the routine performs automatic compression on writing it.)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

2.1.4 glp_mincost_lp — convert minimum cost flow problem to LP

Synopsis

```
void glp_mincost_lp(glp_prob *P, glp_graph *G, int names, int v_rhs,
                    int a_low, int a_cap, int a_cost);
```

Description

The routine `glp_mincost_lp` builds LP problem (1)—(3), which corresponds to the specified minimum cost flow problem.

The parameter `P` is the resultant LP problem object to be built. Note that on entry its current content is erased with the routine `glp_erase_prob`.

The parameter `G` is the graph (network) program object, which specifies the minimum cost flow problem instance.

The parameter `names` is a flag. If it is `GLP_ON`, the routine uses symbolic names of the graph object components to assign symbolic names to the LP problem object components. If the flag is `GLP_OFF`, no symbolic names are assigned.

The parameter `v_rhs` specifies an offset of the field of type `double` in the vertex data block, which contains b_i, the supply/demand value. If `v_rhs` < 0, it is assumed that $b_i = 0$ for all nodes.

The parameter `a_low` specifies an offset of the field of type `double` in the arc data block, which contains l_{ij}, the lower bound to the arc flow. If `a_low` < 0, it is assumed that $l_{ij} = 0$ for all arcs.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, which contains u_{ij}, the upper bound to the arc flow (the arc capacity). If the upper bound is specified as `DBL_MAX`, it is assumed that $u_{ij} = \infty$, i.e. the arc is uncapacitated. If `a_cap` < 0, it is assumed that $u_{ij} = 1$ for all arcs.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, which contains c_{ij}, the per-unit cost of the arc flow. If `a_cost` < 0, it is assumed that $c_{ij} = 0$ for all arcs.

Example

The example program below reads the minimum cost problem instance in DIMACS format from file '`sample.min`', converts the instance to LP, and then writes the resultant LP in CPLEX format to file '`mincost.lp`'.

```
#include <stddef.h>
#include <glpk.h>

typedef struct { double rhs; } v_data;
typedef struct { double low, cap, cost; } a_data;

int main(void)
{     glp_graph *G;
      glp_prob *P;
      G = glp_create_graph(sizeof(v_data), sizeof(a_data));
      glp_read_mincost(G, offsetof(v_data, rhs),
         offsetof(a_data, low), offsetof(a_data, cap),
         offsetof(a_data, cost), "sample.min");
      P = glp_create_prob();
      glp_mincost_lp(P, G, GLP_ON, offsetof(v_data, rhs),
         offsetof(a_data, low), offsetof(a_data, cap),
         offsetof(a_data, cost));
      glp_delete_graph(G);
      glp_write_lp(P, NULL, "mincost.lp");
      glp_delete_prob(P);
      return 0;
}
```

If '`sample.min`' is the example data file from the subsection describing `glp_read_mincost`, file '`mincost.lp`' may look like follows:

```
Minimize
 obj: + 3 x(2,4) + 2 x(2,3) + x(3,5) + 7 x(5,7) + 5 x(5,6)
 + x(5,2) + 3 x(7,9) + 9 x(8,9)

Subject To
 r_1: + x(1,2) + x(1,4) = 20
 r_2: - x(5,2) + x(2,3) + x(2,4) - x(1,2) = 0
 r_3: + x(3,5) + x(3,8) - x(2,3) = 0
 r_4: + x(4,5) - x(2,4) - x(1,4) = 0
```

```
 r_5: + x(5,2) + x(5,6) + x(5,7) - x(4,5) - x(3,5) = 0
 r_6: + x(6,7) + x(6,8) - x(5,6) = 0
 r_7: + x(7,9) - x(6,7) - x(5,7) = 0
 r_8: + x(8,9) - x(6,8) - x(3,8) = 0
 r_9: - x(8,9) - x(7,9) = -20

Bounds
 0 <= x(1,4) <= 23
 0 <= x(1,2) <= 14
 0 <= x(2,4) <= 9
 0 <= x(2,3) <= 10
 0 <= x(3,8) <= 18
 2 <= x(3,5) <= 12
 0 <= x(4,5) <= 26
 0 <= x(5,7) <= 4
 0 <= x(5,6) <= 25
 0 <= x(5,2) <= 11
 4 <= x(6,8) <= 8
 0 <= x(6,7) <= 7
 0 <= x(7,9) <= 15
 0 <= x(8,9) <= 20

End
```

2.1.5 glp_mincost_okalg — solve minimum cost flow problem with out-of-kilter algorithm

Synopsis

```
int glp_mincost_okalg(glp_graph *G, int v_rhs, int a_low, int a_cap,
                      int a_cost, double *sol, int a_x, int v_pi);
```

Description

The routine `glp_mincost_okalg` finds optimal solution to the minimum cost flow problem with the out-of-kilter algorithm.[2] Note that this routine requires all the problem data to be integer-valued.

The parameter `G` is a graph (network) program object which specifies the minimum cost flow problem instance to be solved.

The parameter `v_rhs` specifies an offset of the field of type `double` in the vertex data block, which contains b_i, the supply/demand value. This value must be integer in the range [$-$`INT_MAX`, $+$`INT_MAX`]. If `v_rhs` < 0, it is assumed that $b_i = 0$ for all nodes.

The parameter `a_low` specifies an offset of the field of type `double` in the arc data block, which contains l_{ij}, the lower bound to the arc flow. This bound must be integer in the range [0, `INT_MAX`]. If `a_low` < 0, it is assumed that $l_{ij} = 0$ for all arcs.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, which contains u_{ij}, the upper bound to the arc flow (the arc capacity). This bound must be integer in the range [l_{ij}, `INT_MAX`]. If `a_cap` < 0, it is assumed that $u_{ij} = 1$ for all arcs.

[2] GLPK implementation of the out-of-kilter algorithm is based on the following book: L. R. Ford, Jr., and D. R. Fulkerson, "Flows in Networks," The RAND Corp., Report R-375-PR (August 1962), Chap. III "Minimal Cost Flow Problems," pp. 113-26.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, which contains c_{ij}, the per-unit cost of the arc flow. This value must be integer in the range [$-$`INT_MAX`, $+$`INT_MAX`]. If `a_cost` < 0, it is assumed that $c_{ij} = 0$ for all arcs.

The parameter `sol` specifies a location, to which the routine stores the objective value (that is, the total cost) found. If `sol` is NULL, the objective value is not stored.

The parameter `a_x` specifies an offset of the field of type `double` in the arc data block, to which the routine stores x_{ij}, the arc flow found. If `a_x` < 0, the arc flow value is not stored.

The parameter `v_pi` specifies an offset of the field of type `double` in the vertex data block, to which the routine stores π_i, the node potential, which is the Lagrange multiplier for the corresponding flow conservation equality constraint (see (2) in Subsection "Background"). If necessary, the application program may use the node potentials to compute λ_{ij}, reduced costs of the arc flows x_{ij}, which are the Lagrange multipliers for the arc flow bound constraints (see (3) ibid.), using the following formula:

$$\lambda_{ij} = c_{ij} - (\pi_i - \pi_j),$$

where c_{ij} is the per-unit cost for arc (i, j).

Note that all solution components (the objective value, arc flows, and node potentials) computed by the routine are always integer-valued.

Returns

0	Optimal solution found.
`GLP_ENOPFS`	No (primal) feasible solution exists.
`GLP_EDATA`	Unable to start the search, because some problem data are either not integer-valued or out of range. This code is also returned if the total supply, which is the sum of b_i over all source nodes (nodes with $b_i > 0$), exceeds `INT_MAX`.
`GLP_ERANGE`	The search was prematurely terminated because of integer overflow.
`GLP_EFAIL`	An error has been detected in the program logic. (If this code is returned for your problem instance, please report to `<bug-glpk@gnu.org>`.)

Comments

By design the out-of-kilter algorithm is applicable only to networks, where $b_i = 0$ for *all* nodes, i.e. actually this algorithm finds a minimal cost *circulation*. Due to this requirement the routine `glp_mincost_okalg` converts the original network to a network suitable for the out-of-kilter algorithm in the following way:[3]

1) it adds two auxiliary nodes s and t;

2) for each original node i with $b_i > 0$ the routine adds auxiliary supply arc $(s \rightarrow i)$, flow x_{si} through which is costless ($c_{si} = 0$) and fixed to $+b_i$ ($l_{si} = u_{si} = +b_i$);

3) for each original node i with $b_i < 0$ the routine adds auxiliary demand arc $(i \rightarrow t)$, flow x_{it} through which is costless ($c_{it} = 0$) and fixed to $-b_i$ ($l_{it} = u_{it} = -b_i$);

4) finally, the routine adds auxiliary feedback arc $(t \rightarrow s)$, flow x_{ts} through which is also costless ($c_{ts} = 0$) and fixed to F ($l_{ts} = u_{ts} = F$), where $F = \sum_{b_i > 0} b_i$ is the total supply.

[3] The conversion is performed internally and does not change the original network program object passed to the routine.

Example

The example program below reads the minimum cost problem instance in DIMACS format from file '`sample.min`', solves it by using the routine `glp_mincost_okalg`, and writes the solution found on the standard output.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>

typedef struct { double rhs, pi; } v_data;
typedef struct { double low, cap, cost, x; } a_data;

#define node(v) ((v_data *)((v)->data))
#define arc(a)  ((a_data *)((a)->data))

int main(void)
{     glp_graph *G;
      glp_vertex *v, *w;
      glp_arc *a;
      int i, ret;
      double sol;
      G = glp_create_graph(sizeof(v_data), sizeof(a_data));
      glp_read_mincost(G, offsetof(v_data, rhs),
         offsetof(a_data, low), offsetof(a_data, cap),
         offsetof(a_data, cost), "sample.min");
      ret = glp_mincost_okalg(G, offsetof(v_data, rhs),
         offsetof(a_data, low), offsetof(a_data, cap),
         offsetof(a_data, cost), &sol, offsetof(a_data, x),
         offsetof(v_data, pi));
      printf("ret = %d; sol = %5g\n", ret, sol);
      for (i = 1; i <= G->nv; i++)
      {  v = G->v[i];
         printf("node %d:    pi = %5g\n", i, node(v)->pi);
         for (a = v->out; a != NULL; a = a->t_next)
         {  w = a->head;
            printf("arc  %d->%d: x  = %5g; lambda = %5g\n",
               v->i, w->i, arc(a)->x,
               arc(a)->cost - (node(v)->pi - node(w)->pi));
         }
      }
      glp_delete_graph(G);
      return 0;
}
```

If '`sample.min`' is the example data file from the subsection describing `glp_read_mincost`, the output may look like follows:

```
Reading min-cost flow problem data from 'sample.min'...
Flow network has 9 nodes and 14 arcs
24 lines were read
ret = 0; sol =   213
node 1:    pi =   -12
arc  1->4: x  =    13; lambda =     0
arc  1->2: x  =     7; lambda =     0
node 2:    pi =   -12
arc  2->4: x  =     0; lambda =     3
arc  2->3: x  =     7; lambda =     0
node 3:    pi =   -14
```

```
arc  3->8: x  =     5; lambda =     0
arc  3->5: x  =     2; lambda =     3
node 4:    pi =   -12
arc  4->5: x  =    13; lambda =     0
node 5:    pi =   -12
arc  5->7: x  =     4; lambda =    -1
arc  5->6: x  =    11; lambda =     0
arc  5->2: x  =     0; lambda =     1
node 6:    pi =   -17
arc  6->8: x  =     4; lambda =     3
arc  6->7: x  =     7; lambda =    -3
node 7:    pi =   -20
arc  7->9: x  =    11; lambda =     0
node 8:    pi =   -14
arc  8->9: x  =     9; lambda =     0
node 9:    pi =   -23
```

2.1.6 glp_mincost_relax4 — solve minimum cost flow problem with relaxation method of Bertsekas and Tseng (RELAX-IV)

Synopsis

```
int glp_mincost_relax4(glp_graph *G, int v_rhs, int a_low, int a_cap,
              int a_cost, int crash, double *sol, int a_x, int a_rc);
```

Description

The routine `glp_mincost_relax4` finds optimal solution to the minimum cost flow problem with the relaxation method RELAX-IV developed by Bertsekas and Tseng.[4] This method is one of most efficient methods for network optimization.

Note that this routine requires all the problem data to be integer-valued.

The parameter `G` is a graph (network) program object which specifies the minimum cost flow problem instance to be solved.

The parameter `v_rhs` specifies an offset of the field of type `double` in the vertex data block, which contains b_i, the supply/demand value. This value must be integer in the range [$-$`INT_MAX`/4, $+$`INT_MAX`/4]. If `v_rhs` < 0, it is assumed that $b_i = 0$ for all nodes.

The parameter `a_low` specifies an offset of the field of type `double` in the arc data block, which contains l_{ij}, the lower bound to the arc flow. This bound must be integer in the range [0, `INT_MAX`/4]. If `a_low` < 0, it is assumed that $l_{ij} = 0$ for all arcs.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, which contains u_{ij}, the upper bound to the arc flow (the arc capacity). This bound must be integer in the range [l_{ij}, `INT_MAX`/4]. If `a_cap` < 0, it is assumed that $u_{ij} = 1$ for all arcs.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, which contains c_{ij}, the per-unit cost of the arc flow. This value must be integer in the range [$-$`INT_MAX`/4, $+$`INT_MAX`/4]. If `a_cost` < 0, it is assumed that $c_{ij} = 0$ for all arcs.

[4] GLPK implementation of this method is based on a C translation of the original Fortran code `RELAX4` written by Dimitri P. Bertsekas and Paul Tseng, with a contribution by Jonathan Eckstein in the phase II initialization.

The parameter `crash` is an option that specifies initialization method:

0 — default initialization is used;

1 — auction initialization is used.

If `crash` = 1, initialization is performed with a special crash procedure based on an auction/shorest path method. This option is recommended for difficult problems where the default initialization results in long running times.

The parameter `sol` specifies a location, to which the routine stores the objective value (that is, the total cost) found. If `sol` is NULL, the objective value is not stored.

The parameter `a_x` specifies an offset of the field of type `double` in the arc data block, to which the routine stores x_{ij}, the arc flow found. If `a_x` < 0, the arc flow value is not stored.

The parameter `a_rc` specifies an offset of the field of type `double` in the arc data block, to which the routine stores the reduced cost for corresponding arc flow (see (3) in Subsection "Background"). If `a_rc` < 0, the reduced cost is not stored.

Note that all solution components (the objective value, arc flows, and node potentials) computed by the routine are always integer-valued.

Returns

0	Optimal solution found.
`GLP_ENOPFS`	No (primal) feasible solution exists.
`GLP_EDATA`	Unable to start the search, because some problem data are either not integer-valued or out of range.
`GLP_ERANGE`	Unable to start the search because of integer overflow.

Example

The example program below reads the minimum cost problem instance in DIMACS format from file '`sample.min`', solves it by using the routine `glp_mincost_relax4`, and writes the solution found on the standard output.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>

typedef struct { double rhs; } v_data;
typedef struct { double low, cap, cost, x, rc; } a_data;

#define node(v) ((v_data *)((v)->data))
#define arc(a)  ((a_data *)((a)->data))

int main(void)
{     glp_graph *G;
      glp_vertex *v, *w;
      glp_arc *a;
      int i, ret;
      double sol;
      G = glp_create_graph(sizeof(v_data), sizeof(a_data));
      glp_read_mincost(G, offsetof(v_data, rhs),
         offsetof(a_data, low), offsetof(a_data, cap),
```

```
         offsetof(a_data, cost), "sample.min");
      ret = glp_mincost_relax4(G, offsetof(v_data, rhs),
         offsetof(a_data, low), offsetof(a_data, cap),
         offsetof(a_data, cost), 0, &sol, offsetof(a_data, x),
         offsetof(a_data, rc));
      printf("ret = %d; sol = %5g\n", ret, sol);
      for (i = 1; i <= G->nv; i++)
      {  v = G->v[i];
         for (a = v->out; a != NULL; a = a->t_next)
         {  w = a->head;
            printf("arc %d->%d: x = %5g; rc = %5g\n",
               v->i, w->i, arc(a)->x, arc(a)->rc);
         }
      }
      glp_delete_graph(G);
      return 0;
}
```

If '`sample.min`' is the example data file from the subsection describing `glp_read_mincost`, the output may look like follows:

```
Reading min-cost flow problem data from 'sample.min'...
Flow network has 9 nodes and 14 arcs
24 lines were read
ret = 0; sol =   213
arc 1->4: x =    13; rc =     0
arc 1->2: x =     7; rc =     0
arc 2->4: x =     0; rc =     3
arc 2->3: x =     7; rc =     0
arc 3->8: x =     5; rc =     0
arc 3->5: x =     2; rc =     3
arc 4->5: x =    13; rc =     0
arc 5->7: x =     4; rc =    -1
arc 5->6: x =    11; rc =     0
arc 5->2: x =     0; rc =     1
arc 6->8: x =     4; rc =     3
arc 6->7: x =     7; rc =    -3
arc 7->9: x =    11; rc =     0
arc 8->9: x =     9; rc =     0
```

2.1.7 glp_netgen — Klingman's network problem generator

Synopsis

```
int glp_netgen(glp_graph *G, int v_rhs, int a_cap, int a_cost,
               const int parm[1+15]);
```

Description

The routine `glp_netgen` is a GLPK version of the network problem generator developed by Dr. Darwin Klingman.[5] It can create capacitated and uncapacitated minimum cost flow (or transshipment), transportation, and assignment problems.

The parameter `G` specifies the graph object, to which the generated problem data have to be stored. Note that on entry the graph object is erased with the routine `glp_erase_graph`.

[5] D. Klingman, A. Napier, and J. Stutz. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow networks. Management Science 20 (1974), 814-20.

The parameter `v_rhs` specifies an offset of the field of type `double` in the vertex data block, to which the routine stores the supply or demand value. If `v_rhs` < 0, the value is not stored.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, to which the routine stores the arc capacity. If `a_cap` < 0, the capacity is not stored.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, to which the routine stores the per-unit cost if the arc flow. If `a_cost` < 0, the cost is not stored.

The array `parm` contains description of the network to be generated:

`parm[0]`		not used
`parm[1]`	`iseed`	8-digit positive random number seed
`parm[2]`	`nprob`	8-digit problem id number
`parm[3]`	`nodes`	total number of nodes
`parm[4]`	`nsorc`	total number of source nodes (including transshipment nodes)
`parm[5]`	`nsink`	total number of sink nodes (including transshipment nodes)
`parm[6]`	`iarcs`	number of arc
`parm[7]`	`mincst`	minimum cost for arcs
`parm[8]`	`maxcst`	maximum cost for arcs
`parm[9]`	`itsup`	total supply
`parm[10]`	`ntsorc`	number of transshipment source nodes
`parm[11]`	`ntsink`	number of transshipment sink nodes
`parm[12]`	`iphic`	percentage of skeleton arcs to be given the maximum cost
`parm[13]`	`ipcap`	percentage of arcs to be capacitated
`parm[14]`	`mincap`	minimum upper bound for capacitated arcs
`parm[15]`	`maxcap`	maximum upper bound for capacitated arcs

Returns

If the instance was successfully generated, the routine `glp_netgen` returns zero; otherwise, if specified parameters are inconsistent, the routine returns a non-zero error code.

Notes

1. The routine generates a transportation problem if:

$$\texttt{nsorc} + \texttt{nsink} = \texttt{nodes},\ \texttt{ntsorc} = 0,\ \text{and}\ \texttt{ntsink} = 0.$$

2. The routine generates an assignment problem if the requirements for a transportation problem are met and:

$$\texttt{nsorc} = \texttt{nsink}\ \text{and}\ \texttt{itsup} = \texttt{nsorc}.$$

3. The routine always generates connected graphs. So, if the number of requested arcs has been reached and the generated instance is not fully connected, the routine generates a few remaining arcs to ensure connectedness. Thus, the actual number of arcs generated by the routine may be greater than the requested number of arcs.

2.1.8 glp_netgen_prob — Klingman's standard network problem instance

Synopsis

```
void glp_netgen_prob(int nprob, int parm[1+15]);
```

Description

The routine `glp_netgen_prob` provides the set of parameters for Klingman's network problem generator (see the routine `glp_netgen`), which describe a standard network problem instance.

The parameter `nprob` ($101 \leq$ `nprob` ≤ 150) specifies the problem instance number.

The array `parm` contains description of the network, provided by the routine. (For detailed description of these parameters see comments to the routine `glp_netgen`.)

Problem characteristics

The table below shows characteristics of Klingman's standard network problem instances.

Problem	Nodes	Arcs	Optimum
101	5000	25336	6191726
102	5000	25387	72337144
103	5000	25355	218947553
104	5000	25344	−19100371
105	5000	25332	31192578
106	5000	12870	4314276
107	5000	37832	7393769
108	5000	50309	8405738
109	5000	75299	9190300
110	5000	12825	8975048
111	5000	37828	4747532
112	5000	50325	4012671
113	5000	75318	2979725
114	5000	26514	5821181
115	5000	25962	6353310
116	5000	25304	5915426
117	5000	12816	4420560
118	5000	37797	7045842
119	5000	50301	7724179
120	5000	75330	8455200
121	5000	25000	66366360
122	5000	25000	30997529
123	5000	25000	23388777
124	5000	25000	17803443
125	5000	25000	14119622
126	5000	12500	18802218
127	5000	37500	27674647
128	5000	50000	30906194
129	5000	75000	40905209
130	5000	12500	38939608
131	5000	37500	16752978
132	5000	50000	13302951
133	5000	75000	9830268
134	1000	25000	3804874
135	2500	25000	11729616
136	7500	25000	33318101
137	10000	25000	46426030
138	5000	25000	60710879
139	5000	25000	32729682
140	5000	25000	27183831
141	5000	25000	19963286
142	5000	25000	20243457
143	5000	25000	18586777
144	5000	25000	2504591
145	5000	25000	215956138
146	5000	25000	2253113811
147	5000	25000	−427908373
148	5000	25000	−92965318
149	5000	25000	86051224
150	5000	25000	619314919

2.1.9 glp_gridgen — grid-like network problem generator

Synopsis

```
int glp_gridgen(glp_graph *G, int v_rhs, int a_cap, int a_cost,
                const int parm[1+14]);
```

Description

The routine `glp_gridgen` is a GLPK version of the grid-like network problem generator developed by Yusin Lee and Jim Orlin.[6]

The parameter `G` specifies the graph object, to which the generated problem data have to be stored. Note that on entry the graph object is erased with the routine `glp_erase_graph`.

The parameter `v_rhs` specifies an offset of the field of type `double` in the vertex data block, to which the routine stores the supply or demand value. If `v_rhs` < 0, the value is not stored.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, to which the routine stores the arc capacity. If `a_cap` < 0, the capacity is not stored.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, to which the routine stores the per-unit cost if the arc flow. If `a_cost` < 0, the cost is not stored.

The array `parm` contains parameters of the network to be generated:

`parm[0]`	not used
`parm[1]`	two-ways arcs indicator: 1 — if links in both direction should be generated 0 — otherwise
`parm[2]`	random number seed (a positive integer)
`parm[3]`	number of nodes (the number of nodes generated might be slightly different to make the network a grid)
`parm[4]`	grid width
`parm[5]`	number of sources
`parm[6]`	number of sinks
`parm[7]`	average degree
`parm[8]`	total flow
`parm[9]`	distribution of arc costs: 1 — uniform, 2 — exponential
`parm[10]`	lower bound for arc cost (uniform), 100λ (exponential)
`parm[11]`	upper bound for arc cost (uniform), not used (exponential)
`parm[12]`	distribution of arc capacities: 1 — uniform, 2 — exponential
`parm[13]`	lower bound for arc capacity (uniform), 100λ (exponential)
`parm[14]`	upper bound for arc capacity (uniform), not used (exponential)

Returns

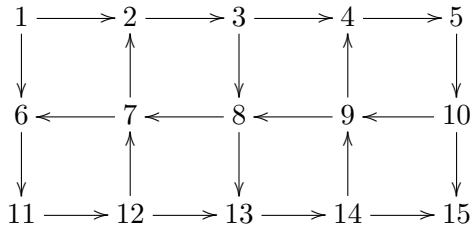
If the instance was successfully generated, the routine `glp_gridgen` returns zero; otherwise, if specified parameters are inconsistent, the routine returns a non-zero error code.

[6] Y. Lee and J. Orlin. GRIDGEN generator., 1991. The original code is publicly available from ftp://dimacs.rutgers.edu/pub/netflow/generators/network/gridgen.

Comments[7]

This network generator generates a grid-like network plus a super node. In additional to the arcs connecting the nodes in the grid, there is an arc from each supply node to the super node and from the super node to each demand node to guarantee feasiblity. These arcs have very high costs and very big capacities.

The idea of this network generator is as follows: First, a grid of $n_1 \times n_2$ is generated. For example, 5×3. The nodes are numbered as 1 to 15, and the supernode is numbered as $n_1 \times n_2 + 1$. Then arcs between adjacent nodes are generated. For these arcs, the user is allowed to specify either to generate two-way arcs or one-way arcs. If two-way arcs are to be generated, two arcs, one in each direction, will be generated between each adjacent node pairs. Otherwise, only one arc will be generated. If this is the case, the arcs will be generated in alterntive directions as shown below.



Then the arcs between the super node and the source/sink nodes are added as mentioned before. If the number of arcs still doesn't reach the requirement, additional arcs will be added by uniformly picking random node pairs. There is no checking to prevent multiple arcs between any pair of nodes. However, there will be no self-arcs (arcs that poins back to its tail node) in the network.

The source and sink nodes are selected uniformly in the network, and the imbalances of each source/sink node are also assigned by uniform distribution.

[7]This material is based on comments to the original version of GRIDGEN.

2.2 Maximum flow problem

2.2.1 Background

The *maximum flow problem* (MAXFLOW) is stated as follows. Let there be given a directed graph (flow network) $G = (V, A)$, where V is a set of vertices (nodes), and $A \subseteq V \times V$ is a set of arcs. Let also for each arc $a = (i, j) \in A$ there be given its capacity u_{ij}. The problem is, for given *source* node $s \in V$ and *sink* node $t \in V$, to find flows x_{ij} through every arc of the network, which satisfy the specified arc capacities and the conservation constraints at all nodes, and maximize the total flow F through the network from s to t. Here the conservation constraint at a node means that the total flow entering this node through its incoming arcs (plus F, if it is the source node) must be equal to the total flow leaving this node through its outgoing arcs (plus F, if it is the sink node). An example of the maximum flow problem, where $s = v_1$ and $t = v_9$, is shown on Fig. 2.

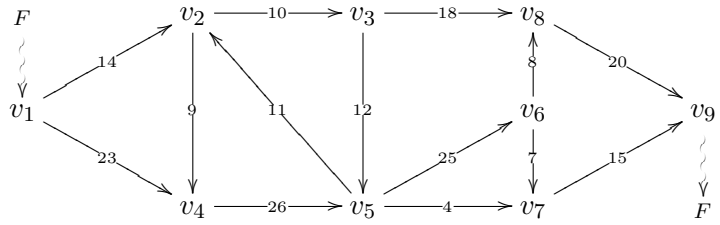


Fig. 2. An example of the maximum flow problem.

The maximum flow problem can be naturally formulated as the following LP problem:

maximize

$$F \tag{4}$$

subject to

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = \begin{cases} +F, & \text{for } i = s \\ 0, & \text{for all } i \in V\backslash\{s,t\} \\ -F, & \text{for } i = t \end{cases} \tag{5}$$

$$0 \le x_{ij} \le u_{ij} \quad \text{for all } (i,j) \in A \tag{6}$$

where $F \ge 0$ is an additional variable playing the role of the objective.

Another LP formulation of the maximum flow problem, which does not include the variable F, is the following:

maximize

$$z = \sum_{(s,j)\in A} x_{sj} - \sum_{(j,s)\in A} x_{js} \; (= F) \tag{7}$$

subject to

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} \begin{cases} \ge 0, & \text{for } i = s \\ = 0, & \text{for all } i \in V\backslash\{s,t\} \\ \le 0, & \text{for } i = t \end{cases} \tag{8}$$

$$0 \le x_{ij} \le u_{ij} \quad \text{for all } (i,j) \in A \tag{9}$$

2.2.2 glp_read_maxflow — read maximum flow problem data in DIMACS format

Synopsis

```
int glp_read_maxflow(glp_graph *G, int *s, int *t, int a_cap,
                     const char *fname);
```

Description

The routine `glp_read_maxflow` reads the maximum flow problem data from a text file in DIMACS format.

The parameter `G` specifies the graph object, to which the problem data have to be stored. Note that before reading data the current content of the graph object is completely erased with the routine `glp_erase_graph`.

The pointer `s` specifies a location, to which the routine stores the ordinal number of the source node. If `s` is `NULL`, the source node number is not stored.

The pointer `t` specifies a location, to which the routine stores the ordinal number of the sink node. If `t` is `NULL`, the sink node number is not stored.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, to which the routine stores u_{ij}, the arc capacity. If `a_cap` < 0, the arc capacity is not stored.

The character string `fname` specifies the name of a text file to be read in. (If the file name name ends with the suffix '`.gz`', the file is assumed to be compressed, in which case the routine decompresses it "on the fly".)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

Example

```
typedef struct
{     /* arc data block */
      ...
      double cap;
      ...
} a_data;

int main(void)
{     glp_graph *G;
      int s, t, ret;
      G = glp_create_graph(..., sizeof(a_data));
      ret = glp_read_maxflow(G, &s, &t, offsetof(a_data, cap),
         "sample.max");
      if (ret != 0) goto ...
      ...
}
```

DIMACS maximum flow problem format[8]

The DIMACS input file is a plain ASCII text file. It contains *lines* of several types described below. A line is terminated with an end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

Note that DIMACS requires all numerical quantities to be integers in the range $[-2^{31},\ 2^{31}-1]$ while GLPK allows the quantities to be floating-point numbers.

Comment lines. Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character `c`.

```
c This is a comment line
```

Problem line. There is one problem line per data file. The problem line must appear before any node or arc descriptor lines. It has the following format:

```
p max NODES ARCS
```

The lower-case character `p` signifies that this is a problem line. The three-character problem designator `max` identifies the file as containing specification information for the maximum flow problem. The `NODES` field contains an integer value specifying the number of nodes in the network. The `ARCS` field contains an integer value specifying the number of arcs in the network.

Node descriptors. Two node descriptor lines for the source and sink nodes must appear before all arc descriptor lines. They may appear in either order, each with the following format:

```
n ID WHICH
```

The lower-case character `n` signifies that this a node descriptor line. The `ID` field gives a node identification number, an integer between 1 and `NODES`. The `WHICH` field gives either a lower-case `s` or `t`, designating the source and sink, respectively.

Arc descriptors. There is one arc descriptor line for each arc in the network. Arc descriptor lines are of the following format:

```
a SRC DST CAP
```

The lower-case character `a` signifies that this is an arc descriptor line. For a directed arc (i, j) the `SRC` field gives the identification number i for the tail endpoint, and the `DST` field gives the identification number j for the head endpoint. Identification numbers are integers between 1 and `NODES`. The `CAP` field gives the arc capacity, i.e. maximum amount of flow that can be sent along arc (i, j) in a feasible flow.

Example. Below here is an example of the data file in DIMACS format corresponding to the maximum flow problem shown on Fig 2.

```
c sample.max
c
c This is an example of the maximum flow problem data
c in DIMACS format.
c
p max 9 14
c
n 1 s
n 9 t
```

[8] This material is based on the paper "The First DIMACS International Algorithm Implementation Challenge: Problem Definitions and Specifications", which is publicly available at http://dimacs.rutgers.edu/Challenges/.

```
c
a 1 2 14
a 1 4 23
a 2 3 10
a 2 4  9
a 3 5 12
a 3 8 18
a 4 5 26
a 5 2 11
a 5 6 25
a 5 7  4
a 6 7  7
a 6 8  8
a 7 9 15
a 8 9 20
c
c eof
```

2.2.3 glp_write_maxflow — write maximum flow problem data in DIMACS format

Synopsis

```
int glp_write_maxflow(glp_graph *G, int s, int t, int a_cap,
                      const char *fname);
```

Description

The routine `glp_write_maxflow` writes the maximum flow problem data to a text file in DIMACS format.

The parameter `G` is the graph (network) program object, which specifies the maximum flow problem instance.

The parameter `s` specifies the ordinal number of the source node.

The parameter `t` specifies the ordinal number of the sink node.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, which contains u_{ij}, the upper bound to the arc flow (the arc capacity). If the upper bound is specified as `DBL_MAX`, it is assumed that $u_{ij} = \infty$, i.e. the arc is uncapacitated. If `a_cap` < 0, it is assumed that $u_{ij} = 1$ for all arcs.

The character string `fname` specifies a name of the text file to be written out. (If the file name ends with suffix '`.gz`', the file is assumed to be compressed, in which case the routine performs automatic compression on writing it.)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

2.2.4 glp_maxflow_lp — convert maximum flow problem to LP

Synopsis

```
void glp_maxflow_lp(glp_prob *P, glp_graph *G, int names, int s, int t,
                    int a_cap);
```

Description

The routine `glp_maxflow_lp` builds LP problem (7)—(9), which corresponds to the specified maximum flow problem.

The parameter `P` is the resultant LP problem object to be built. Note that on entry its current content is erased with the routine `glp_erase_prob`.

The parameter `G` is the graph (network) program object, which specifies the maximum flow problem instance.

The parameter `names` is a flag. If it is `GLP_ON`, the routine uses symbolic names of the graph object components to assign symbolic names to the LP problem object components. If the flag is `GLP_OFF`, no symbolic names are assigned.

The parameter `s` specifies the ordinal number of the source node.

The parameter `t` specifies the ordinal number of the sink node.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, which contains u_{ij}, the upper bound to the arc flow (the arc capacity). If the upper bound is specified as `DBL_MAX`, it is assumed that $u_{ij} = \infty$, i.e. the arc is uncapacitated. If `a_cap` < 0, it is assumed that $u_{ij} = 1$ for all arcs.

Example

The example program below reads the maximum flow problem in DIMACS format from file '`sample.max`', converts the instance to LP, and then writes the resultant LP in CPLEX format to file '`maxflow.lp`'.

```
#include <stddef.h>
#include <glpk.h>

int main(void)
{     glp_graph *G;
      glp_prob *P;
      int s, t;
      G = glp_create_graph(0, sizeof(double));
      glp_read_maxflow(G, &s, &t, 0, "sample.max");
      P = glp_create_prob();
      glp_maxflow_lp(P, G, GLP_ON, s, t, 0);
      glp_delete_graph(G);
      glp_write_lp(P, NULL, "maxflow.lp");
      glp_delete_prob(P);
      return 0;
}
```

If '`sample.max`' is the example data file from the previous subsection, the output '`maxflow.lp`' may look like follows:

```
Maximize
 obj: + x(1,4) + x(1,2)

Subject To
 r_1: + x(1,2) + x(1,4) >= 0
 r_2: - x(5,2) + x(2,3) + x(2,4) - x(1,2) = 0
 r_3: + x(3,5) + x(3,8) - x(2,3) = 0
 r_4: + x(4,5) - x(2,4) - x(1,4) = 0
 r_5: + x(5,2) + x(5,6) + x(5,7) - x(4,5) - x(3,5) = 0
 r_6: + x(6,7) + x(6,8) - x(5,6) = 0
 r_7: + x(7,9) - x(6,7) - x(5,7) = 0
 r_8: + x(8,9) - x(6,8) - x(3,8) = 0
 r_9: - x(8,9) - x(7,9) <= 0

Bounds
 0 <= x(1,4) <= 23
 0 <= x(1,2) <= 14
 0 <= x(2,4) <= 9
 0 <= x(2,3) <= 10
 0 <= x(3,8) <= 18
 0 <= x(3,5) <= 12
 0 <= x(4,5) <= 26
 0 <= x(5,7) <= 4
 0 <= x(5,6) <= 25
 0 <= x(5,2) <= 11
 0 <= x(6,8) <= 8
 0 <= x(6,7) <= 7
 0 <= x(7,9) <= 15
 0 <= x(8,9) <= 20

End
```

2.2.5 glp_maxflow_ffalg — solve maximum flow problem with Ford-Fulkerson algorithm

Synopsis

```
int glp_maxflow_ffalg(glp_graph *G, int s, int t, int a_cap, double *sol,
                      int a_x, int v_cut);
```

Description

The routine `glp_mincost_ffalg` finds optimal solution to the maximum flow problem with the Ford-Fulkerson algorithm.[9] Note that this routine requires all the problem data to be integer-valued.

The parameter `G` is a graph (network) program object which specifies the maximum flow problem instance to be solved.

The parameter s specifies the ordinal number of the source node.

The parameter t specifies the ordinal number of the sink node.

[9] GLPK implementation of the Ford-Fulkerson algorithm is based on the following book: L. R. Ford, Jr., and D. R. Fulkerson, "Flows in Networks," The RAND Corp., Report R-375-PR (August 1962), Chap. I "Static Maximal Flow," pp. 30-33.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, which contains u_{ij}, the upper bound to the arc flow (the arc capacity). This bound must be integer in the range [0, `INT_MAX`]. If `a_cap` < 0, it is assumed that $u_{ij} = 1$ for all arcs.

The parameter `sol` specifies a location, to which the routine stores the objective value (that is, the total flow from s to t) found. If `sol` is NULL, the objective value is not stored.

The parameter `a_x` specifies an offset of the field of type `double` in the arc data block, to which the routine stores x_{ij}, the arc flow found. If `a_x` < 0, the arc flow values are not stored.

The parameter `v_cut` specifies an offset of the field of type `int` in the vertex data block, to which the routine stores node flags corresponding to the optimal solution found: if the node flag is 1, the node is labelled, and if the node flag is 0, the node is unlabelled. The calling program may use these node flags to determine the *minimal cut*, which is a subset of arcs whose one endpoint is labelled and other is not. If `v_cut` < 0, the node flags are not stored.

Note that all solution components (the objective value and arc flows) computed by the routine are always integer-valued.

Returns

0	Optimal solution found.
`GLP_EDATA`	Unable to start the search, because some problem data are either not integer-valued or out of range.

Example

The example program shown below reads the maximum flow problem instance in DIMACS format from file '`sample.max`', solves it using the routine `glp_maxflow_ffalg`, and write the solution found to the standard output.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>

typedef struct { int cut; } v_data;
typedef struct { double cap, x; } a_data;

#define node(v) ((v_data *)((v)->data))
#define arc(a)  ((a_data *)((a)->data))

int main(void)
{     glp_graph *G;
      glp_vertex *v, *w;
      glp_arc *a;
      int i, s, t, ret;
      double sol;
      G = glp_create_graph(sizeof(v_data), sizeof(a_data));
      glp_read_maxflow(G, &s, &t, offsetof(a_data, cap),
         "sample.max");
      ret = glp_maxflow_ffalg(G, s, t, offsetof(a_data, cap),
         &sol, offsetof(a_data, x), offsetof(v_data, cut));
      printf("ret = %d; sol = %5g\n", ret, sol);
      for (i = 1; i <= G->nv; i++)
      {  v = G->v[i];
         for (a = v->out; a != NULL; a = a->t_next)
         {  w = a->head;
```

```
            printf("x[%d->%d] = %5g (%d)\n", v->i, w->i,
               arc(a)->x, node(v)->cut ^ node(w)->cut);
         }
      }
      glp_delete_graph(G);
      return 0;
}
```

If '`sample.max`' is the example data file from the subsection describing `glp_read_maxflow`, the output may look like follows:

```
Reading maximum flow problem data from 'sample.max'...
Flow network has 9 nodes and 14 arcs
24 lines were read
ret = 0; sol =    29
x[1->4] =    19 (0)
x[1->2] =    10 (0)
x[2->4] =     0 (0)
x[2->3] =    10 (1)
x[3->8] =    10 (0)
x[3->5] =     0 (1)
x[4->5] =    19 (0)
x[5->7] =     4 (1)
x[5->6] =    15 (0)
x[5->2] =     0 (0)
x[6->8] =     8 (1)
x[6->7] =     7 (1)
x[7->9] =    11 (0)
x[8->9] =    18 (0)
```

2.2.6 glp_rmfgen — Goldfarb's maximum flow problem generator

Synopsis

```
int glp_rmfgen(glp_graph *G, int *s, int *t, int a_cap, const int parm[1+5]);
```

Description

The routine `glp_rmfgen` is a GLPK version of the maximum flow problem generator developed by D. Goldfarb and M. Grigoriadis.[10,11,12]

The parameter `G` specifies the graph object, to which the generated problem data have to be stored. Note that on entry the graph object is erased with the routine `glp_erase_graph`.

The pointers `s` and `t` specify locations, to which the routine stores the source and sink node numbers, respectively. If `s` or `t` is `NULL`, corresponding node number is not stored.

The parameter `a_cap` specifies an offset of the field of type `double` in the arc data block, to which the routine stores the arc capacity. If `a_cap` < 0, the capacity is not stored.

[10] D. Goldfarb and M. D. Grigoriadis, "A computational comparison of the Dinic and network simplex methods for maximum flow." Annals of Op. Res. 13 (1988), pp. 83-123.

[11] U. Derigs and W. Meier, "Implementing Goldberg's max-flow algorithm: A computational investigation." Zeitschrift für Operations Research 33 (1989), pp. 383-403.

[12] The original code of RMFGEN implemented by Tamas Badics is publicly available from ftp://dimacs.rutgers.edu/pub/netflow/generators/network/genrmf.

The array `parm` contains description of the network to be generated:

`parm[0]`		not used
`parm[1]`	`seed`	random number seed (a positive integer)
`parm[2]`	`a`	frame size
`parm[3]`	`b`	depth
`parm[4]`	`c1`	minimal arc capacity
`parm[5]`	`c2`	maximal arc capacity

Returns

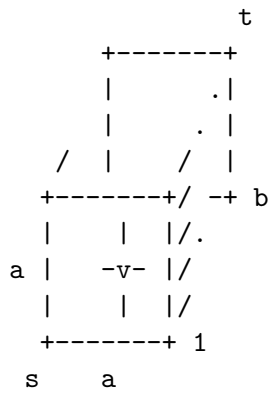
If the instance was successfully generated, the routine `glp_netgen` returns zero; otherwise, if specified parameters are inconsistent, the routine returns a non-zero error code.

Comments[13]

The generated network is as follows. It has b pieces of frames of size $a \times a$. (So alltogether the number of vertices is $a \times a \times b$.)

In each frame all the vertices are connected with their neighbours (forth and back). In addition the vertices of a frame are connected one to one with the vertices of next frame using a random permutation of those vertices.

The source is the lower left vertex of the first frame, the sink is the upper right vertex of the b-th frame.



The capacities are randomly chosen integers from the range of $[c_1, c_2]$ in the case of interconnecting edges, and $c_2 \cdot a^2$ for the in-frame edges.

[13] This material is based on comments to the original version of RMFGEN.

2.3 Assignment problem

2.3.1 Background

Let there be given an undirected bipartite graph $G = (R \cup S, E)$, where R and S are disjoint sets of vertices (nodes), and $E \subseteq R \times S$ is a set of edges. Let also for each edge $e = (i, j) \in E$ there be given its cost c_{ij}. A *matching* (which in case of bipartite graph is also called *assignment*) $M \subseteq E$ in G is a set of pairwise non-adjacent edges, that is, no two edges in M share a common vertex. A matching, which matches all vertices of the graph, is called a *perfect matching*. Obviously, a perfect matching in bipartite graph $G = (R \cup S, E)$ defines some bijection $R \leftrightarrow S$.

The *assignment problem* has two different variants. In the first variant the problem is to find matching (assignment) M, which maximizes the sum:

$$\sum_{(i,j)\in M} c_{ij} \tag{10}$$

(so this variant is also called the *maximum weighted bipartite matching problem* or, if all $c_{ij} = 1$, the *maximum cardinality bipartite matching problem*). In the second, classic variant the problem is to find *perfect* matching (assignment) M, which minimizes or maximizes the sum (10).

An example of the assignment problem, which is the maximum weighted bipartite matching problem, is shown on Fig. 3.

The maximum weighted bipartite matching problem can be naturally formulated as the following LP problem:

maximize

$$z = \sum_{(i,j)\in E} c_{ij} x_{ij} \tag{11}$$

subject to

$$\sum_{(i,j)\in E} x_{ij} \le 1 \quad \text{for all } i \in R \tag{12}$$

$$\sum_{(i,j)\in E} x_{ij} \le 1 \quad \text{for all } j \in S \tag{13}$$

$$0 \le x_{ij} \le 1 \quad \text{for all } (i,j) \in E \tag{14}$$

where $x_{ij} = 1$ means that $(i, j) \in M$, and $x_{ij} = 0$ means that $(i, j) \notin M$.[14]

[14] The constraint matrix of LP formulation (11)—(14) is totally unimodular, due to which $x_{ij} \in \{0, 1\}$ for any basic solution.

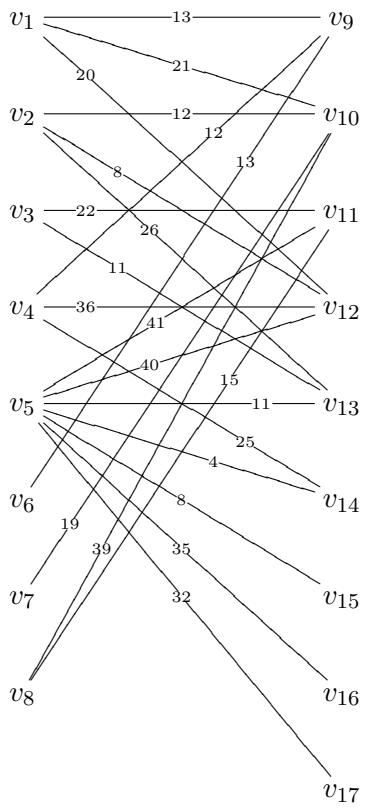


Fig. 3. An example of the assignment problem.

Similarly, the perfect assignment problem can be naturally formulated as the following LP problem:

minimize (or maximize)

$$z = \sum_{(i,j)\in E} c_{ij} x_{ij} \tag{15}$$

subject to

$$\sum_{(i,j)\in E} x_{ij} = 1 \quad \text{for all } i \in R \tag{16}$$

$$\sum_{(i,j)\in E} x_{ij} = 1 \quad \text{for all } j \in S \tag{17}$$

$$0 \le x_{ij} \le 1 \quad \text{for all } (i,j) \in E \tag{18}$$

where variables x_{ij} have the same meaning as for (11)—(14) above.

In GLPK an undirected bipartite graph $G = (R \cup S, E)$ is represented as directed graph $\overline{G} = (V, A)$, where $V = R \cup S$ and $A = \{(i,j) : (i,j) \in E\}$, i.e. every edge $(i,j) \in E$ in G corresponds to arc $(i \to j) \in A$ in $\overline{G}$.

2.3.2 glp_read_asnprob — read assignment problem data in DIMACS format

Synopsis

```
int glp_read_asnprob(glp_graph *G, int v_set, int a_cost, const char *fname);
```

Description

The routine `glp_read_asnprob` reads the assignment problem data from a text file in DIMACS format.

The parameter `G` specifies the graph object, to which the problem data have to be stored. Note that before reading data the current content of the graph object is completely erased with the routine `glp_erase_graph`.

The parameter `v_set` specifies an offset of the field of type `int` in the vertex data block, to which the routine stores the node set indicator:

0 — the node is in set R;

1 — the node is in set S.

If `v_set` < 0, the node set indicator is not stored.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, to which the routine stores the edge cost c_{ij}. If `a_cost` < 0, the edge cost is not stored.

The character string `fname` specifies the name of a text file to be read in. (If the file name name ends with the suffix '`.gz`', the file is assumed to be compressed, in which case the routine decompresses it "on the fly".)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

Example. Below here is an example program that read the assignment problem data in DIMACS format from a text file '`sample.asn`'.

```
typedef struct
{     /* vertex data block */
      ...
      int set;
      ...
} v_data;

typedef struct
{     /* arc data block */
      ...
      double cost;
      ...
} a_data;

int main(void)
{     glp_graph *G;
      int ret;
      G = glp_create_graph(sizeof(v_data), sizeof(a_data));
      ret = glp_read_asnprob(G, offsetof(v_data, set),
         offsetof(a_data, cost), "sample.asn");
      if (ret != 0) goto ...
      ...
}
```

DIMACS assignment problem format[15]

The DIMACS input file is a plain ASCII text file. It contains *lines* of several types described below. A line is terminated with an end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

Note that DIMACS requires all numerical quantities to be integers in the range $[-2^{31},\ 2^{31}-1]$ while GLPK allows the quantities to be floating-point numbers.

Comment lines. Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character `c`.

```
c This is a comment line
```

Problem line. There is one problem line per data file. The problem line must appear before any node or arc descriptor lines. It has the following format:

```
p asn NODES EDGES
```

The lower-case character `p` signifies that this is a problem line. The three-character problem designator `asn` identifies the file as containing specification information for the assignment problem. The `NODES` field contains an integer value specifying the total number of nodes in the graph (i.e. in both sets R and S). The `EDGES` field contains an integer value specifying the number of edges in the graph.

Node descriptors. All node descriptor lines must appear before all edge descriptor lines. The node descriptor lines lists the nodes in set R only, and all other nodes are assumed to be in set S. There is one node descriptor line for each such node, with the following format:

```
n ID
```

The lower-case character `n` signifies that this is a node descriptor line. The `ID` field gives a node identification number, an integer between 1 and `NODES`.

Edge descriptors. There is one edge descriptor line for each edge in the graph. Edge descriptor lines are of the following format:

```
a SRC DST COST
```

The lower-case character `a` signifies that this is an edge descriptor line. For each edge (i, j), where $i \in R$ and $j \in S$, the `SRC` field gives the identification number of vertex i, and the `DST` field gives the identification number of vertex j. Identification numbers are integers between 1 and `NODES`. The `COST` field contains the cost of edge (i, j).

Example. Below here is an example of the data file in DIMACS format corresponding to the assignment problem shown on Fig 3.

```
c sample.asn
c
c This is an example of the assignment problem data
c in DIMACS format.
c
p asn 17 22
c
n 1
n 2
```

[15] This material is based on the paper "The First DIMACS International Algorithm Implementation Challenge: Problem Definitions and Specifications", which is publicly available at http://dimacs.rutgers.edu/Challenges/.

```
n 3
n 4
n 5
n 6
n 7
n 8
c
a 1  9 13
a 1 10 21
a 1 12 20
a 2 10 12
a 2 12  8
a 2 13 26
a 3 11 22
a 3 13 11
a 4  9 12
a 4 12 36
a 4 14 25
a 5 11 41
a 5 12 40
a 5 13 11
a 5 14  4
a 5 15  8
a 5 16 35
a 5 17 32
a 6  9 13
a 7 10 19
a 8 10 39
a 8 11 15
c
c eof
```

2.3.3 glp_write_asnprob — write assignment problem data in DIMACS format

Synopsis

```
int glp_write_asnprob(glp_graph *G, int v_set, int a_cost, const char *fname);
```

Description

The routine `glp_write_asnprob` writes the assignment problem data to a text file in DIMACS format.

The parameter `G` is the graph program object, which specifies the assignment problem instance.

The parameter `v_set` specifies an offset of the field of type `int` in the vertex data block, which contains the node set indicator:

0 — the node is in set R;

1 — the node is in set S.

If `v_set` < 0, it is assumed that a node having no incoming arcs is in set R, and a node having no outgoing arcs is in set S.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, which contains c_{ij}, the edge cost. If `a_cost` < 0, it is assumed that $c_{ij} = 1$ for all edges.

The character string `fname` specifies a name of the text file to be written out. (If the file name ends with suffix '`.gz`', the file is assumed to be compressed, in which case the routine performs automatic compression on writing it.)

Note

The routine `glp_write_asnprob` does not check that the specified graph object correctly represents a bipartite graph. To make sure that the problem data are correct, use the routine `glp_check_asnprob`.

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

2.3.4 glp_check_asnprob — check correctness of assignment problem data

Synopsis

```
int glp_check_asnprob(glp_graph *G, int v_set);
```

Description

The routine `glp_check_asnprob` checks that the specified graph object `G` correctly represents a bipartite graph.

The parameter `v_set` specifies an offset of the field of type `int` in the vertex data block, which contains the node set indicator:

0 — the node is in set R;

1 — the node is in set S.

If `v_set` < 0, it is assumed that a node having no incoming arcs is in set R, and a node having no outgoing arcs is in set S.

Returns

0 — the data are correct;

1 — the set indicator of some node is 0, however, that node has one or more incoming arcs;

2 — the set indicator of some node is 1, however, that node has one or more outgoing arcs;

3 — the set indicator of some node is invalid (neither 0 nor 1);

4 — some node has both incoming and outgoing arcs.

2.3.5 glp_asnprob_lp — convert assignment problem to LP

Synopsis

```
int glp_asnprob_lp(glp_prob *P, int form, glp_graph *G, int names, int v_set,
                   int a_cost);
```

Description

The routine `glp_asnprob_lp` builds LP problem, which corresponds to the specified assignment problem.

The parameter `P` is the resultant LP problem object to be built. Note that on entry its current content is erased with the routine `glp_erase_prob`.

The parameter `form` defines which LP formulation should be used:

`GLP_ASN_MIN` — perfect matching (15)—(18), minimization;

`GLP_ASN_MAX` — perfect matching (15)—(18), maximization;

`GLP_ASN_MMP` — maximum weighted matching (11)—(14).

The parameter `G` is the graph program object, which specifies the assignment problem instance.

The parameter `names` is a flag. If it is `GLP_ON`, the routine uses symbolic names of the graph object components to assign symbolic names to the LP problem object components. If the `flag` is `GLP_OFF`, no symbolic names are assigned.

The parameter `v_set` specifies an offset of the field of type `int` in the vertex data block, which contains the node set indicator:

0 — the node is in set R;

1 — the node is in set S.

If `v_set` < 0, it is assumed that a node having no incoming arcs is in set R, and a node having no outgoing arcs is in set S.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, which contains c_{ij}, the edge cost. If `a_cost` < 0, it is assumed that $c_{ij} = 1$ for all edges.

Returns

If the LP problem has been successfully built, the routine `glp_asnprob_lp` returns zero, otherwise, non-zero (see the routine `glp_check_asnprob`).

Example

The example program below reads the assignment problem instance in DIMACS format from file '`sample.asn`', converts the instance to LP (11)—(14), and writes the resultant LP in CPLEX format to file '`matching.lp`'.

```
#include <stddef.h>
#include <glpk.h>

typedef struct { int set; } v_data;
typedef struct { double cost; } a_data;

int main(void)
{     glp_graph *G;
      glp_prob *P;
      G = glp_create_graph(sizeof(v_data), sizeof(a_data));
      glp_read_asnprob(G, offsetof(v_data, set),
         offsetof(a_data, cost), "sample.asn");
      P = glp_create_prob();
      glp_asnprob_lp(P, GLP_ASN_MMP, G, GLP_ON,
         offsetof(v_data, set), offsetof(a_data, cost));
      glp_delete_graph(G);
      glp_write_lp(P, NULL, "matching.lp");
      glp_delete_prob(P);
      return 0;
}
```

If '`sample.asn`' is the example data file from the subsection describing `glp_read_asnprob`, file '`matching.lp`' may look like follows:

```
Maximize
 obj: + 20 x(1,12) + 21 x(1,10) + 13 x(1,9) + 26 x(2,13) + 8 x(2,12)
 + 12 x(2,10) + 11 x(3,13) + 22 x(3,11) + 25 x(4,14) + 36 x(4,12)
 + 12 x(4,9) + 32 x(5,17) + 35 x(5,16) + 8 x(5,15) + 4 x(5,14)
 + 11 x(5,13) + 40 x(5,12) + 41 x(5,11) + 13 x(6,9) + 19 x(7,10)
 + 15 x(8,11) + 39 x(8,10)

Subject To
 r_1: + x(1,9) + x(1,10) + x(1,12) <= 1
 r_2: + x(2,10) + x(2,12) + x(2,13) <= 1
 r_3: + x(3,11) + x(3,13) <= 1
 r_4: + x(4,9) + x(4,12) + x(4,14) <= 1
 r_5: + x(5,11) + x(5,12) + x(5,13) + x(5,14) + x(5,15) + x(5,16)
 + x(5,17) <= 1
 r_6: + x(6,9) <= 1
 r_7: + x(7,10) <= 1
 r_8: + x(8,10) + x(8,11) <= 1
 r_9: + x(6,9) + x(4,9) + x(1,9) <= 1
 r_10: + x(8,10) + x(7,10) + x(2,10) + x(1,10) <= 1
 r_11: + x(8,11) + x(5,11) + x(3,11) <= 1
 r_12: + x(5,12) + x(4,12) + x(2,12) + x(1,12) <= 1
 r_13: + x(5,13) + x(3,13) + x(2,13) <= 1
 r_14: + x(5,14) + x(4,14) <= 1
 r_15: + x(5,15) <= 1
 r_16: + x(5,16) <= 1
 r_17: + x(5,17) <= 1

Bounds
 0 <= x(1,12) <= 1
 0 <= x(1,10) <= 1
 0 <= x(1,9) <= 1
 0 <= x(2,13) <= 1
 0 <= x(2,12) <= 1
 0 <= x(2,10) <= 1
 0 <= x(3,13) <= 1
 0 <= x(3,11) <= 1
 0 <= x(4,14) <= 1
 0 <= x(4,12) <= 1
 0 <= x(4,9) <= 1
 0 <= x(5,17) <= 1
 0 <= x(5,16) <= 1
 0 <= x(5,15) <= 1
 0 <= x(5,14) <= 1
 0 <= x(5,13) <= 1
 0 <= x(5,12) <= 1
 0 <= x(5,11) <= 1
 0 <= x(6,9) <= 1
 0 <= x(7,10) <= 1
 0 <= x(8,11) <= 1
 0 <= x(8,10) <= 1

End
```

2.3.6 glp_asnprob_okalg — solve assignment problem with out-of-kilter algorithm

Synopsis

```
int glp_asnprob_okalg(int form, glp_graph *G, int v_set, int a_cost,
                      double *sol, int a_x);
```

Description

The routine `glp_mincost_okalg` finds optimal solution to the assignment problem with the out-of-kilter algorithm.[16] Note that this routine requires all the problem data to be integer-valued.

The parameter `form` defines which LP formulation should be used:

`GLP_ASN_MIN` — perfect matching (15)—(18), minimization;

`GLP_ASN_MAX` — perfect matching (15)—(18), maximization;

`GLP_ASN_MMP` — maximum weighted matching (11)—(14).

The parameter `G` is the graph program object, which specifies the assignment problem instance.

The parameter `v_set` specifies an offset of the field of type `int` in the vertex data block, which contains the node set indicator:

0 — the node is in set R;

1 — the node is in set S.

If `v_set` < 0, it is assumed that a node having no incoming arcs is in set R, and a node having no outgoing arcs is in set S.

The parameter `a_cost` specifies an offset of the field of type `double` in the arc data block, which contains c_{ij}, the edge cost. This value must be integer in the range [`-INT_MAX`, `+INT_MAX`]. If `a_cost` < 0, it is assumed that $c_{ij} = 1$ for all edges.

The parameter `sol` specifies a location, to which the routine stores the objective value (that is, the total cost) found. If `sol` is `NULL`, the objective value is not stored.

The parameter `a_x` specifies an offset of the field of type `int` in the arc data block, to which the routine stores x_{ij}. If `a_x` < 0, this value is not stored.

Returns

0	Optimal solution found.
`GLP_ENOPFS`	No (primal) feasible solution exists.
`GLP_EDATA`	Unable to start the search, because the assignment problem data are either incorrect (this error is detected by the routine `glp_check_asnprob`), not integer-valued or out of range.
`GLP_ERANGE`	The search was prematurely terminated because of integer overflow.
`GLP_EFAIL`	An error has been detected in the program logic. (If this code is returned for your problem instance, please report to `<bug-glpk@gnu.org>`.)

[16] GLPK implementation of the out-of-kilter algorithm is based on the following book: L. R. Ford, Jr., and D. R. Fulkerson, "Flows in Networks," The RAND Corp., Report R-375-PR (August 1962), Chap. III "Minimal Cost Flow Problems," pp. 113-26.

Comments

Since the out-of-kilter algorithm is designed to find a minimal cost circulation, the routine `glp_asnprob_okalg` converts the original graph to a network suitable for this algorithm in the following way:[17]

1) it replaces each edge (i, j) by arc $(i \rightarrow j)$, flow x_{ij} through which has zero lower bound ($l_{ij} = 0$), unity upper bound ($u_{ij} = 1$), and per-unit cost $+c_{ij}$ (in case of `GLP_ASN_MIN`), or $-c_{ij}$ (in case of `GLP_ASN_MAX` and `GLP_ASN_MMP`);

2) then it adds one auxiliary feedback node k;

3) for each original node $i \in R$ the routine adds auxiliary supply arc $(k \rightarrow i)$, flow x_{ki} through which is costless ($c_{ki} = 0$) and either fixed at 1 ($l_{ki} = u_{ki} = 1$, in case of `GLP_ASN_MIN` and `GLP_ASN_MAX`) or has zero lower bound and unity upper bound ($l_{ij} = 0$, $u_{ij} = 1$, in case of `GLP_ASN_MMP`);

4) similarly, for each original node $j \in S$ the routine adds auxiliary demand arc $(j \rightarrow k)$, flow x_{jk} through which is costless ($c_{jk} = 0$) and either fixed at 1 ($l_{jk} = u_{jk} = 1$, in case of `GLP_ASN_MIN` and `GLP_ASN_MAX`) or has zero lower bound and unity upper bound ($l_{jk} = 0$, $u_{jk} = 1$, in case of `GLP_ASN_MMP`).

Example

The example program shown below reads the assignment problem instance in DIMACS format from file '`sample.asn`', solves it by using the routine `glp_asnprob_okalg`, and writes the solution found to the standard output.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>

typedef struct { int set; } v_data;
typedef struct { double cost; int x; } e_data;

#define node(v) ((v_data *)((v)->data))
#define edge(e) ((e_data *)((e)->data))

int main(void)
{     glp_graph *G;
      glp_vertex *v;
      glp_arc *e;
      int i, ret;
      double sol;
      G = glp_create_graph(sizeof(v_data), sizeof(e_data));
      glp_read_asnprob(G, offsetof(v_data, set),
         offsetof(e_data, cost), "sample.asn");
      ret = glp_asnprob_okalg(GLP_ASN_MMP, G,
         offsetof(v_data, set), offsetof(e_data, cost), &sol,
         offsetof(e_data, x));
      printf("ret = %d; sol = %5g\n", ret, sol);
      for (i = 1; i <= G->nv; i++)
      {  v = G->v[i];
         for (e = v->out; e != NULL; e = e->t_next)
            printf("edge %2d %2d: x = %d; c = %g\n",
```

[17] The conversion is performed internally and does not change the original graph program object passed to the routine.

```
            e->tail->i, e->head->i, edge(e)->x, edge(e)->cost);
      }
      glp_delete_graph(G);
      return 0;
}
```

If '`sample.asn`' is the example data file from the subsection describing `glp_read_asnprob`, the output may look like follows:

```
Reading assignment problem data from 'sample.asn'...
Assignment problem has 8 + 9 = 17 nodes and 22 arcs
38 lines were read
ret = 0; sol =   180
edge  1 12: x = 1; c = 20
edge  1 10: x = 0; c = 21
edge  1  9: x = 0; c = 13
edge  2 13: x = 1; c = 26
edge  2 12: x = 0; c = 8
edge  2 10: x = 0; c = 12
edge  3 13: x = 0; c = 11
edge  3 11: x = 1; c = 22
edge  4 14: x = 1; c = 25
edge  4 12: x = 0; c = 36
edge  4  9: x = 0; c = 12
edge  5 17: x = 0; c = 32
edge  5 16: x = 1; c = 35
edge  5 15: x = 0; c = 8
edge  5 14: x = 0; c = 4
edge  5 13: x = 0; c = 11
edge  5 12: x = 0; c = 40
edge  5 11: x = 0; c = 41
edge  6  9: x = 1; c = 13
edge  7 10: x = 0; c = 19
edge  8 11: x = 0; c = 15
edge  8 10: x = 1; c = 39
```

2.3.7 glp_asnprob_hall — find bipartite matching of maximum cardinality

Synopsis

```
int glp_asnprob_hall(glp_graph *G, int v_set, int a_x);
```

Description

The routine `glp_asnprob_hall` finds a matching of maximal cardinality in the specified bipartite graph. It uses a version of the Fortran routine `MC21A` developed by I. S. Duff[18], which implements Hall's algorithm.[19]

The parameter `G` is a pointer to the graph program object.

The parameter `v_set` specifies an offset of the field of type `int` in the vertex data block, which contains the node set indicator:

0 — the node is in set R;

1 — the node is in set S.

[18] I. S. Duff, Algorithm 575: Permutations for zero-free diagonal, ACM Trans. on Math. Softw. 7 (1981), pp. 387-390.

[19] M. Hall, "An Algorithm for Distinct Representatives," Am. Math. Monthly 63 (1956), pp. 716-717.

If `v_set` < 0, it is assumed that a node having no incoming arcs is in set R, and a node having no outgoing arcs is in set S.

The parameter `a_x` specifies an offset of the field of type `int` in the arc data block, to which the routine stores x_{ij}. If `a_x` < 0, this value is not stored.

Returns

The routine `glp_asnprob_hall` returns the cardinality of the matching found. However, if the specified graph is incorrect (as detected by the routine `glp_check_asnprob`), this routine returns a negative value.

Comments

The same solution may be obtained with the routine `glp_asnprob_okalg` (for LP formulation `GLP_ASN_MMP` and all edge costs equal to 1). However, the routine `glp_asnprob_hall` is much faster.

Example

The example program shown below reads the assignment problem instance in DIMACS format from file '`sample.asn`', finds a bipartite matching of maximal cardinality by using the routine `glp_asnprob_hall`, and writes the solution found to the standard output.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>

typedef struct { int set; } v_data;
typedef struct { int x;   } e_data;

#define node(v) ((v_data *)((v)->data))
#define edge(e) ((e_data *)((e)->data))

int main(void)
{     glp_graph *G;
      glp_vertex *v;
      glp_arc *e;
      int i, card;
      G = glp_create_graph(sizeof(v_data), sizeof(e_data));
      glp_read_asnprob(G, offsetof(v_data, set), -1,
         "sample.asn");
      card = glp_asnprob_hall(G, offsetof(v_data, set),
         offsetof(e_data, x));
      printf("card = %d\n", card);
      for (i = 1; i <= G->nv; i++)
      {  v = G->v[i];
         for (e = v->out; e != NULL; e = e->t_next)
            printf("edge %2d %2d: x = %d\n",
               e->tail->i, e->head->i, edge(e)->x);
      }
      glp_delete_graph(G);
      return 0;
}
```

If '`sample.asn`' is the example data file from the subsection describing `glp_read_asnprob`, the output may look like follows:

```
Reading assignment problem data from 'sample.asn'...
Assignment problem has 8 + 9 = 17 nodes and 22 arcs
38 lines were read
card = 7
edge  1 12: x = 1
edge  1 10: x = 0
edge  1  9: x = 0
edge  2 13: x = 1
edge  2 12: x = 0
edge  2 10: x = 0
edge  3 13: x = 0
edge  3 11: x = 1
edge  4 14: x = 1
edge  4 12: x = 0
edge  4  9: x = 0
edge  5 17: x = 1
edge  5 16: x = 0
edge  5 15: x = 0
edge  5 14: x = 0
edge  5 13: x = 0
edge  5 12: x = 0
edge  5 11: x = 0
edge  6  9: x = 1
edge  7 10: x = 1
edge  8 11: x = 0
edge  8 10: x = 0
```

2.4 Critical path problem

2.4.1 Background

The *critical path problem* (CPP) is stated as follows. Let there be given a project J, which is a set of jobs (tasks, activities, etc.). Performing each job $i \in J$ requires time $t_i \geq 0$. Besides, over the set J there is given a precedence relation $R \subseteq J \times J$, where $(i, j) \in R$ means that job i immediately precedes job j, i.e. performing job j cannot start until job i has been completely performed. The problem is to find starting times x_i for each job $i \in J$, which satisfy to the precedence relation and minimize the total duration (makespan) of the project.

The following is an example of the critical path problem:

Job	Desription	Time	Predecessors
A	Excavate	3	—
B	Lay foundation	4	A
C	Rough plumbing	3	B
D	Frame	10	B
E	Finish exterior	8	D
F	Install HVAC	4	D
G	Rough electric	6	D
H	Sheet rock	8	C, E, F, G
I	Install cabinets	5	H
J	Paint	5	H
K	Final plumbing	4	I
L	Final electric	2	J
M	Install flooring	4	K, L

Obviously, the project along with the precedence relation can be represented as a directed graph $G = (J, R)$ called *project network*, where each node $i \in J$ corresponds to a job, and arc $(i \to j) \in R$ means that job i immediately precedes job j.[20] The project network for the example above is shown on Fig. 4.

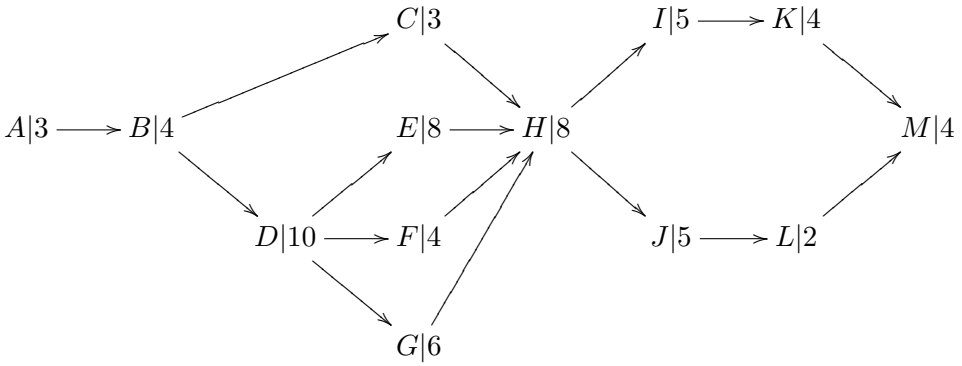


Fig. 4. An example of the project network.

[20]There exists another network representation of the critical path problem, where jobs correspond to arcs while nodes correspond to events introduced to express the precedence relation. That representation, however, is much less convenient than the one, where jobs are represented as nodes of the network.

May note that the project network must be acyclic; otherwise, it would be impossible to satisfy to the precedence relation for any job that belongs to a cycle.

The critical path problem can be naturally formulated as the following LP problem:

minimize

$$z \tag{19}$$

subject to

$$x_i + t_i \le z \quad \text{for all } i \in J \tag{20}$$

$$x_i + t_i \le x_j \quad \text{for all } (i,j) \in R \tag{21}$$

$$x_i \ge 0 \quad \text{for all } i \in J \tag{22}$$

The inequality constraints (21), which are active in the optimal solution, define so called *critical path* having the following property: the minimal project duration z can be decreased only by decreasing the times t_j for jobs on the critical path, and delaying any critical job delays the entire project.

2.4.2 glp_cpp — solve critical path problem

Synopsis

```
double glp_cpp(glp_graph *G, int v_t, int v_es, int v_ls);
```

Description

The routine `glp_cpp` solves the critical path problem represented in the form of the project network.

The parameter `G` is a pointer to the graph object, which specifies the project network. This graph must be acyclic. Multiple arcs are allowed being considered as single arcs.

The parameter `v_t` specifies an offset of the field of type `double` in the vertex data block, which contains time $t_i \ge 0$ needed to perform corresponding job $j \in J$. If `v_t` < 0, it is assumed that $t_i = 1$ for all jobs.

The parameter `v_es` specifies an offset of the field of type `double` in the vertex data block, to which the routine stores the *earliest start time* for corresponding job. If `v_es` < 0, this time is not stored.

The parameter `v_ls` specifies an offset of the field of type `double` in the vertex data block, to which the routine stores the *latest start time* for corresponding job. If `v_ls` < 0, this time is not stored.

The difference between the latest and earliest start times of some job is called its *time reserve*. Delaying a job within its time reserve does not affect the project duration, so if the time reserve is zero, the corresponding job is critical.

Returns

The routine `glp_cpp` returns the minimal project duration, i.e. minimal time needed to perform all jobs in the project.

Example

The example program below solves the critical path problem shown on Fig. 4 by using the routine `glp_cpp` and writes the solution found on the standard output.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>

typedef struct { double t, es, ls; } v_data;

#define node(v) ((v_data *)((v)->data))

int main(void)
{     glp_graph *G;
      int i;
      double t, es, ef, ls, lf, total;
      G = glp_create_graph(sizeof(v_data), 0);
      glp_add_vertices(G, 13);
      node(G->v[1])->t = 3;   /* A: Excavate */
      node(G->v[2])->t = 4;   /* B: Lay foundation */
      node(G->v[3])->t = 3;   /* C: Rough plumbing */
      node(G->v[4])->t = 10;  /* D: Frame */
      node(G->v[5])->t = 8;   /* E: Finish exterior */
      node(G->v[6])->t = 4;   /* F: Install HVAC */
      node(G->v[7])->t = 6;   /* G: Rough elecrtic */
      node(G->v[8])->t = 8;   /* H: Sheet rock */
      node(G->v[9])->t = 5;   /* I: Install cabinets */
      node(G->v[10])->t = 5;  /* J: Paint */
      node(G->v[11])->t = 4;  /* K: Final plumbing */
      node(G->v[12])->t = 2;  /* L: Final electric */
      node(G->v[13])->t = 4;  /* M: Install flooring */
      glp_add_arc(G, 1, 2);   /* A precedes B */
      glp_add_arc(G, 2, 3);   /* B precedes C */
      glp_add_arc(G, 2, 4);   /* B precedes D */
      glp_add_arc(G, 4, 5);   /* D precedes E */
      glp_add_arc(G, 4, 6);   /* D precedes F */
      glp_add_arc(G, 4, 7);   /* D precedes G */
      glp_add_arc(G, 3, 8);   /* C precedes H */
      glp_add_arc(G, 5, 8);   /* E precedes H */
      glp_add_arc(G, 6, 8);   /* F precedes H */
      glp_add_arc(G, 7, 8);   /* G precedes H */
      glp_add_arc(G, 8, 9);   /* H precedes I */
      glp_add_arc(G, 8, 10);  /* H precedes J */
      glp_add_arc(G, 9, 11);  /* I precedes K */
      glp_add_arc(G, 10, 12); /* J precedes L */
      glp_add_arc(G, 11, 13); /* K precedes M */
      glp_add_arc(G, 12, 13); /* L precedes M */
      total = glp_cpp(G, offsetof(v_data, t), offsetof(v_data, es),
         offsetof(v_data, ls));
      printf("Minimal project duration is %.2f\n\n", total);
      printf("Job  Time      ES     EF     LS     LF\n");
      printf("--- ------   ------ ------ ------ ------\n");
      for (i = 1; i <= G->nv; i++)
      {  t = node(G->v[i])->t;
         es = node(G->v[i])->es;
         ef = es + node(G->v[i])->t;
         ls = node(G->v[i])->ls;
```

```
         lf = ls + node(G->v[i])->t;
         printf("%3d %6.2f %s %6.2f %6.2f %6.2f %6.2f\n",
            i, t, ls - es < 0.001 ? "*" : " ", es, ef, ls, lf);
      }
      glp_delete_graph(G);
      return 0;
}
```

The output from the example program shown below includes job number, the time needed to perform a job, earliest start time (ES), earliest finish time (EF), latest start time (LS), and latest finish time (LF) for each job in the project. Critical jobs are marked by asterisks.

```
Minimal project duration is 46.00

Job  Time      ES     EF     LS     LF
--- ------   ------ ------ ------ ------
  1   3.00 *   0.00   3.00   0.00   3.00
  2   4.00 *   3.00   7.00   3.00   7.00
  3   3.00     7.00  10.00  22.00  25.00
  4  10.00 *   7.00  17.00   7.00  17.00
  5   8.00 *  17.00  25.00  17.00  25.00
  6   4.00    17.00  21.00  21.00  25.00
  7   6.00    17.00  23.00  19.00  25.00
  8   8.00 *  25.00  33.00  25.00  33.00
  9   5.00 *  33.00  38.00  33.00  38.00
 10   5.00    33.00  38.00  35.00  40.00
 11   4.00 *  38.00  42.00  38.00  42.00
 12   2.00    38.00  40.00  40.00  42.00
 13   4.00 *  42.00  46.00  42.00  46.00
```