```
/**
 * \file flow_net.c
 *
 *
 *
 * \author jtd7
 * \date 07-11-2011
 */

#include "flow_net.h"
#include <limits.h>
#include <stdlib.h>
#include <assert.h>
#include "queue.h"

extern int dest, source;

void malloc_check(void* p);
int sumFlowIn(struct net* n, int id);
int sumFlowOut(struct net* n, int id);
void  checkAllFlow(struct net* n);

/*
 *   Adds an arc between 's' and 't' with a capacity of 'cap' in the
 *   given network.  If 'antip' is non-zero then antiparallel edges may
 *   be added to the network.  If 'antip' is zero then a dummy node will
 *   be placed in the network to prevent antiparallel edges.
 *
 *   Return 0 on success and 1 on failure.
 */
int net_add_arc(struct net *n, int antip, unsigned int s, unsigned int t,
                int cap){
        struct edge * tEdge;
    int dummyIndex = 0;

    tEdge = getFromEdgeList(&(n->nodes[t].out), t, s);
    if(antip == 0 && tEdge != NULL){
        /* add a dummy node */
        dummyIndex = n->regular_nodes;
        n->nodes[dummyIndex].id = n->regular_nodes;
        n->regular_nodes++;

        /* change t's edge to point to dummy node */
        tEdge->endNode = dummyIndex;
        addToEdgeList(&(n->nodes[dummyIndex].in), tEdge->startNode, tEdge->endNo
de, tEdge->capacity);

        /* remove t to s from s's in list */
        removeFromEdgeList(&(n->nodes[s].in), t, s);

        /* add an edge from dummyIndex to startNode */
        addToEdgeList(&(n->nodes[dummyIndex].out), dummyIndex, s, tEdge->capacit
y);
        addToEdgeList(&(n->nodes[s].in), dummyIndex, s, tEdge->capacity);

        /* add the edge to both ends normally */
        addToEdgeList(&(n->nodes[s].out), s, t, cap);
        addToEdgeList(&(n->nodes[t].in), s, t, cap);
    }
    else{
        /* add the edge to both ends normally */
        addToEdgeList(&(n->nodes[s].out), s, t, cap);
        addToEdgeList(&(n->nodes[t].in), s, t, cap);
    }
    return 0;
}

int net_free(struct net *n){
    return 1;
```



```
}

int init_flow_net(struct net* n, int nnodes, int nedges){
    int i = 0;
    n->nodes = malloc((nnodes + nedges / 2) * sizeof(struct node));
    for(i = 0; i < nnodes; i++){
        n->nodes[i].id = i;
        n->nodes[i].parent = -1;
    }

    n->regular_nodes = nnodes;
    n->super_normal_nodes = nnodes;
    return 0;
}

int compute_max_flow(struct net* n){
    int minCapacity = 0;
    int maxFlow = 0;
        while(findPath(n) != -1){
                /*fprintf(stderr, "found a path!\n");*/
                minCapacity = getMinCapacity(n);
                maxFlow += minCapacity;
                /*fprintf(stderr, "min capacity: %d\n", minCapacity);*/

                residualize(n, minCapacity);
            print_dot_graph(n);

                n->nodes[dest].parent = -1;
    }
        /* print out flows */
        print_flows(n);

        printf("%d\n", maxFlow);
    return 0;
}

void print_flows(struct net * n){
        int nnodes = n->regular_nodes;
        int normals = n->super_normal_nodes;
        struct edgeList * edge;

        int i = 0;

        struct edgeList * cur;

        for(i = 0; i < nnodes; i++){
                cur = n->nodes[i].out;
                while(cur){
                        /* if the edge is part of the final flow and not a dummy
 node */
                        if(cur->ed->partOfFlow && cur->ed->startNode < normals){
                                if(cur->ed->endNode >= normals){
                                        edge = n->nodes[cur->ed->endNode].out;
                                        while(edge){
                                                if(edge->ed->partOfFlow){
                                                        printf("%d %d %d\n", edge
->ed->endNode, cur->ed->startNode, cur->ed->capacity);
                                                }
                                                edge = edge->next;
                                        }
                                }
                                else{
                                        printf("%d %d %d\n", cur->ed->endNode, cu
r->ed->startNode, cur->ed->capacity);
                                }
                        }
                        cur = cur->next;
                }
        }
```

```
}

void residualize(struct net * n, int minCapacity){
        struct edge * edge;
        struct edge * cancel;
        int cur, parent = 0;

        /*fprintf(stderr, "residualizing\n");*/
        cur = dest;
        parent = n->nodes[cur].parent;
        while(n->nodes[cur].id != source){
                /* get parents edge going to child */
                edge = getFromEdgeList(&(n->nodes[parent].out), parent, n->nodes
[cur].id);

                /* subtract flow from capacity */
                if(edge->capacity != minCapacity){
                        edge->capacity = edge->capacity - minCapacity;
                        if(edge->capacity == 0){
                                /* delete the edge */
                                removeFromEdgeList(&(n->nodes[parent].out), pare
nt, n->nodes[cur].id);
                                removeFromEdgeList(&(n->nodes[cur].in), parent,
n->nodes[cur].id);
                        }
                }
                else{
                        /* delete edge */
                        removeFromEdgeList(&(n->nodes[parent].out), parent, n->n
odes[cur].id);
                        removeFromEdgeList(&(n->nodes[cur].in), parent, n->nodes
[cur].id);
                }

                /* if a cancellation edge was already created, don't create anot
her one */
                cancel = getFromEdgeList(&(n->nodes[cur].out), n->nodes[cur].id,
 parent);
                if (cancel != NULL){
                        cancel->capacity = cancel->capacity + minCapacity;
                }
                else{
                        /* add cancellation edge */
                        addToEdgeList(&(n->nodes[cur].out), n->nodes[cur].id, pa
rent, minCapacity);
                        addToEdgeList(&(n->nodes[parent].in), n->nodes[cur].id,
parent, minCapacity);

                        cancel = getFromEdgeList(&(n->nodes[cur].out), n->nodes[
cur].id, parent);
                        cancel->partOfFlow = 1;
                }

                /* move up and reset cur and parent */
                cur = n->nodes[cur].parent;
                parent = n->nodes[cur].parent;
        }

}

int getMinCapacity(struct net * n){
        int cur, parent = 0;
        struct edge * edge;
        int minCapacity = INT_MAX;

        /*fprintf(stderr, "getting min capacity\n");*/

        cur = dest;
        parent = n->nodes[cur].parent;
```



```
        while(n->nodes[cur].id != source){
                /*fprintf(stderr, "%d\n", cur);*/
                /* get parents edge going to child */
                edge = getFromEdgeList(&(n->nodes[parent].out), parent, n->nodes
[cur].id);

                /* set min capacity */
                if(edge->capacity < minCapacity){
                        minCapacity = edge->capacity;
                }


                /* move up and reset cur and parent */
                cur = n->nodes[cur].parent;
                parent = n->nodes[cur].parent;
        }
        /*fprintf(stderr, "%d\n", source);*/
        return minCapacity;
}

int findPath(struct net * n){
        struct nodeList * popped;
        struct edgeList * cur;
        struct nodeList * queue = NULL;
        int i, neighbor, nnodes = 0;

        /* label all nodes unvisited */
        nnodes = n->regular_nodes;
        for(i = 0; i < nnodes; i++){
                n->nodes[i].visited = 0;
        }

        /* enqueue the source */
        queue = enqueue(queue, &(n->nodes[source]));
        n->nodes[source].visited = 1;

        while(queue != NULL){
                /* pop of queue */
                popped = dequeue(&queue);

                cur = popped->node->out;
                while(cur){
                        neighbor = cur->ed->endNode;
                        if(n->nodes[neighbor].visited == 0){
                                n->nodes[neighbor].visited = 1;
                                n->nodes[neighbor].parent = popped->node->id;

                                /* if found dest */
                                if(n->nodes[neighbor].id == dest){
                                        return dest;
                                }
                                queue = enqueue(queue, &(n->nodes[neighbor]));
                        }
                        cur = cur->next;
                }
        }
        return -1;
}


struct edge * newEdge(int s, int t, int cap){
    struct edge * e = malloc(sizeof(struct edge));
    malloc_check(e);
    e->startNode = s;
    e->endNode = t;
    e->capacity = cap;
    e->partOfFlow = 0;
    return e;
}
```

```
struct edge * getFromEdgeList(struct edgeList ** eList, int s, int t){
        struct edgeList * cur = *eList;
        while(cur){
                if(cur->ed->startNode == s && cur->ed->endNode == t)
                        return cur->ed;
                cur = cur->next;
        }
        return NULL;
}

void removeFromEdgeList(struct edgeList ** eList, int s, int t){
        struct edgeList * prev = NULL;
        struct edgeList * cur = *eList;

        /* remove head */
        if(cur->ed->startNode == s && cur->ed->endNode == t){
                *eList = (*eList)->next;
                free(cur);
                return;
        }
        while(cur){
                if(cur->ed->startNode == s && cur->ed->endNode == t){
                        prev->next = cur->next;
                        free(cur);
                        return;
                }
                prev = cur;
                cur = cur->next;
        }
}

void addToEdgeList(struct edgeList ** eList, int s, int t, int cap){
    struct edge * e = newEdge(s, t, cap);
    struct edgeList * cur;

    /* add to head */
    if(*eList == NULL){
        *eList = malloc(sizeof(struct edgeList));
        (*eList)->ed = e;
        (*eList)->next = NULL;
        return;
    }

    /* add to end */
    cur = *eList;
    while(cur->next){
        cur = cur->next;
    }
    cur->next = malloc(sizeof(struct edgeList));
    cur->next->ed = e;
    cur->next->next = NULL;
}

void print_graph(struct net * n){
/*    int i = 0;
    int nnodes = n->regular_nodes;
    struct edgeList * cur;

    for(i = 0; i < nnodes; i++){
        fprintf(stderr, "node %d\n", n->nodes[i].id);
        cur = n->nodes[i].in;
        fprintf(stderr, "\tin edges\n");
        while(cur){
            fprintf(stderr, "\t\t%d %d %d\n", cur->ed->startNode, cur->ed->endNo
de, cur->ed->capacity);
            cur = cur->next;
        }
        cur = n->nodes[i].out;
```



```
        fprintf(stderr, "\tout edges\n");
        while(cur){
            fprintf(stderr, "\t\t%d %d %d\n", cur->ed->startNode, cur->ed->endNo
de, cur->ed->capacity);
            cur = cur->next;
        }
    }*/
}

void print_dot_graph(struct net * n){
    /*int i = 0;
    struct edgeList * cur;
    printf("digraph {\n");
    int nnodes = n->regular_nodes;
    for(i = 0; i < nnodes; i++){
        cur = n->nodes[i].out;
        while(cur){
            printf("\t\t %d -> %d[label=\"%d\",weight=\"%d\"];\n", cur->ed->star
tNode, cur->ed->endNode, cur->ed->capacity, cur->ed->capacity);
            cur = cur->next;
        }
    }
    printf("}\n");*/
}
```

```
/**

Christopher Wilt
Computer Science 758 Skeleton

Flow Networks

This program is supposed to output the max flow for the network.
Networks come in on standard in, in the following format:

<num_nodes>
<num_edges>
<edges, 1 per line>

edges are as follows:
start end weight

As an example:

4
6
0 1 1
1 2 3
2 3 2
3 1 2
2 1 3
0 3 2

It is expected to output the flow in the following format:

<edge utilization, 1 per line>
<cost>

Edge utilization is formatted as follows:
start end flow

where start is the start vertex, end is the end vertex, and flow is
the amount of flow through that edge.

As an example:

0 3 2
0 1 1
3 1 2
3

Node 0 is defined as the source, and node 1 is defined as the sink.

IMPORTANT NOTE:

You will probably want to eliminate antiparallel edges.  This is fine,
but you will need to make sure your program prints out something that
corresponds to the ORIGINAL graph.  This means that if you elect to
add nodes to the graph, if your final flow goes through nodes you
added, you will need to eliminate the extra nodes.

 */

#include <stdlib.h>
#include <stdio.h>

#include "flow_net.h"


const int source = 0;
const int dest = 1;
int node_count = 0;
int edge_count = 0;
```



```
void malloc_check(void* p){
    if(p == NULL){
        fprintf(stderr, "malloc failed\n");
        exit(1);
    }
}

int main(int argc, char** argv){
    if(argc > 1){
        fprintf(stderr, "this program does not take command line arguments\n");
        return 1;
    }
    if(!scanf("%d\n", &node_count)){
        fprintf(stderr, "failed to read number of vertices\n");
        return 1;
    }
    if(!scanf("%d\n", &edge_count)){
        fprintf(stderr, "failed to read number of edges\n");
        return 1;
    }
    struct net* f_net = malloc(sizeof(struct net));
    init_flow_net(f_net, node_count, edge_count);


    malloc_check(f_net);

    int edgeStart, edgeEnd, edgeWeight;
    int num_scanned = 3;

    while(num_scanned == 3){

        num_scanned = scanf("%d %d %d\n", &edgeStart, &edgeEnd, &edgeWeight);
        if(num_scanned != 3){
            break;
        }
        int ret = net_add_arc(f_net, 0, edgeStart, edgeEnd, edgeWeight);
        if(ret != 0){
            fprintf(stderr, "error readng graph\n");
            return 1;
        }
    }

    /*print_graph(f_net);*/
    print_dot_graph(f_net);
    int err;
    err = compute_max_flow(f_net);


    net_free(f_net);
    free(f_net);

    return err;
}
```

```
/**
 * Queue.c
 */

#include "flow_net.h"
#include <limits.h>
#include <stdlib.h>
#include <assert.h>
#include "queue.h"

struct nodeList * enqueue(struct nodeList * head, struct node * n){
    struct nodeList * cur;

    /*fprintf(stderr, "enqueing %d\n", n->id);*/
        /* add to head */
    if(head == NULL){
        /*fprintf(stderr, "adding to head\n");*/
        head = malloc(sizeof(struct nodeList));
        head->node = n;
        head->next = NULL;
        return head;
    }

    /* add to end */
    cur = head;
    while(cur->next){
        cur = cur->next;
    }
    cur->next = malloc(sizeof(struct nodeList));
    cur->next->node = n;
    cur->next->next = NULL;
    return head;
}

struct nodeList * dequeue(struct nodeList ** head){
        struct nodeList * retval = *head;

    /*fprintf(stderr, "dequeing %d\n", retval->node->id);*/

        *head = (*head)->next;

        return retval;
}
```

```
/**
 * \file flow_net.h
 *
 * A simple network for computing max flow.
 *
 * \author Christopher Wilt
 */

#if !defined(_FLOW_NET_H_)
#define _FLOW_NET_H_

#include <stdio.h>

struct edge {
    int startNode;
    int endNode;
    int capacity;
    int partOfFlow;
};

struct edgeList {
    struct edge* ed;
    struct edgeList* next;
};

struct node {
    int id;
    int parent;
    int visited;
    struct edgeList* out;
    struct edgeList* in;
};

struct nodeList {
    struct node * node;
    struct nodeList * next;
};


struct net {
    int regular_nodes;
    int super_normal_nodes;
    struct node* nodes;
};

/*
 * Adds an arc between 's' and 't' with a capacity of 'cap' in the
 * given network.  If 'antip' is non-zero then antiparallel edges may
 * be added to the network.  If 'antip' is zero then a dummy node will
 * be placed in the network to prevent antiparallel edges.
 *
 * Return 0 on success and 1 on failure.
 */
int net_add_arc(struct net *n, int antip, unsigned int s, unsigned int t,
                int cap);

/* free the network, put not the pointer to it */
int net_free(struct net *n);

/*
  Should initialize the flow network.
 */
int init_flow_net(struct net* n, int nodes, int edges);

/*
  Stub that should calculate the max flow of a network.
 */
int compute_max_flow(struct net* n);
```



```
struct edge * newEdge(int s, int t, int cap);
void addToEdgeList(struct edgeList ** eList, int s, int t, int cap);
void print_graph(struct net * n);
void print_dot_graph(struct net * n);
struct edge * getFromEdgeList(struct edgeList ** eList, int s, int t);
int addDummyNode(struct net * n);
void removeFromEdgeList(struct edgeList ** eList, int s, int t);
int findPath(struct net * n);
int getMinCapacity(struct net * n);
void residualize(struct net * n, int minCapacity);
void print_flows(struct net * n);

#endif /* !_FLOW_NET_H_ */
```

```
/**
        queue.h
*/
#if !defined(_QUEUE_H_)
#define _QUEUE_H_

#include <stdio.h>
#include "flow_net.h"

struct nodeList * enqueue(struct nodeList * head, struct node * n);
struct nodeList * dequeue(struct nodeList ** head);

#endif
```

```
CC=gcc
CFLAGS=-lm -std=c99 -Wall -pedantic

SRC= flow_net.c queue.c

all: default

default: $(SRC) main.c
	$(CC) -DNDEBUG -O3 $^ -o flow $(CFLAGS)
	$(CC) -g $^ -o flow_debug $(CFLAGS)

clean:
	rm -f flow
	rm -f *.o
```