Listing 1: Successive Shortest Path

```cpp
#include "ssp.h"
#include <iostream>
#include <list>
#include <assert.h>

#define INFTY std::numeric_limits<long>::max() / 2

/* for a node with positive b value find a sp to a node w with negative b value.
Augment the v -> w path. If the b value of b got positive insert it.
If the b value of v got negative extract it.*/
bool Ssp::Process(node_iter v) {
  node_iter w = pqueue->Dijkstra(v);
  if (w == g.nodes.end())
    return false;
  Augment(v, w);
  if (w->b > 0)
    positive.push_front(w);

  if (v->b <= 0)
    positive.pop_back();
  return true;
}

void Ssp::Augment(node_iter v, node_iter w) {
  node_iter a = w;
  edge_iter e = a->current;

  /* finding the minimum */
  long min = -(w->b);
  while (a != v){
    if (min > e->resCap)
      min = e->resCap;
    a = a->parent;
    e = a->current;
  }
  if (v->b < min)
    min = v->b;

  /* augmenting path v ~> w by min */
  a = w;
  e = a->current;

  while (a != v) {
    e->resCap -= min;
    e->rev->resCap += min;

    a = a->parent;
    e = a->current;
  }

  v->b -= min;
  w->b += min;
}

Ssp::Ssp(resGraph &g_f) : g(g_f) {
  for (auto v = g.nodes.begin(); v != g.nodes.end() - 1; v++) {
    if (v->b > 0)
      positive.push_back(v);
  }
```

```
  pqueue = new Plist(g);
}

bool Ssp::Run() {
  while (not positive.empty()) {
    bool cc = Process(positive.back());
    if (not cc)
      return cc;
  }
  return true;
}

void Ssp::bPrint() {
  long cost = 0;
  for (auto i : g.flow)
    cost += (g.edges[i].c * g.edges[i].rev->resCap);

  // assert(isBFlow());
  // MBF();
  // node_iter v = negCycle();
  // assert(v == g.nodes.end());
  g.print();
  std::cout << cost << std::endl;
}

// This can be done way faster.
node_iter Ssp::GetTail(edge_iter edge) {
  for (auto v = g.nodes.begin(); v != g.nodes.end() - 1; v++)
    for (auto e = v->first; e != (v + 1)->first; e++)
      if (e == edge)
        return v;
  return g.nodes.end();
}

bool Ssp::isBFlow() {
  node_iter v;
  edge_iter e;
  for (auto i : g.flow) {
    e = g.edges.begin() + i;
    v = GetTail(e);
    g.b_values[std::distance(g.nodes.begin(), v)] -= e->rev->resCap;
    g.b_values[std::distance(g.nodes.begin(), e->head)] += e->rev->resCap;
  }
  for (auto b : g.b_values)
    if (b != 0)
      return false;
  return true;
}

/* checking minimality by searching for negative cycles in the residual graph */
void Ssp::MBF() {
  for (auto &v : g.nodes)
    v.dist = INFTY;
  g.nodes[0].dist = 0;

  for (int i = 0; i < g.n - 1; i++) {
    for (auto v = g.nodes.begin(); v != g.nodes.end() - 1; v++) {
      for (auto e = v->first; e != (v + 1)->first; e++) {
        if (e->resCap > 0 and v->dist + e->c < e->head->dist) {
          e->head->dist = v->dist + e->c;
          e->head->parent = v;
```

```
          e->head->current = e;
        }
      }
    }
  }
}

node_iter Ssp::negCycle() {
  for (auto v = g.nodes.begin(); v != g.nodes.end() - 1; v++) {
    for (auto e = v->first; e != (v + 1)->first; e++) {
      if (e->resCap > 0 and v->dist + e->c < e->head->dist) {
        std::vector<bool> visited(g.n, false);
        int iNode = std::distance(g.nodes.begin(), e->head);
        visited[iNode] = true;
        node_iter w = v;
        iNode = std::distance(g.nodes.begin(), v);
        while (not visited[iNode]) {
          visited[iNode] = true;
          v = v->parent;
          iNode = std::distance(g.nodes.begin(), v);
        }
        if (v == e->head) {
          v->parent = w;
          v->current = e;
        }
        return v;
      }
    }
  }
  return g.nodes.end();
}
```

Listing 2: Dijkstra

```
#include "plist.h"
#include <limits>
#include <assert.h>
#define INFTY std::numeric_limits<long>::max()

Plist::Plist(resGraph &g_f) : g(g_f), size(0) { perm.reserve(g.n); }

void Plist::Insert(node_iter v) {
  v->inHeap = true;
  size++;
}

/* only to be applied to a nonempty plist */
node_iter Plist::RemoveMin() {
  node_iter minNode = g.nodes.begin();
  for (; not minNode->inHeap; minNode++)
    ;
  for (auto v = minNode + 1; v != g.nodes.end() - 1; v++) {
    if (v->dist < minNode->dist and v->inHeap)
      minNode = v;
  }
  minNode->inHeap = false;
  size--;
  return minNode;
}

/* performs dijkstra from v and return first node that has negative b value if
   existent
```

```
otherwise returns end of node vector */
node_iter Plist::Dijkstra(node_iter v) {
  node_iter currentNode, newNode;
  edge_iter last;

  /* setting distance labels to infinity */
  for (auto w = g.nodes.begin(); w != g.nodes.end(); w++) {
    w->dist = INFTY;
  }
  v->dist = 0;

  Empty();
  Insert(v);

  while (not IsEmpty()) {
    currentNode = RemoveMin();
    perm.push_back(currentNode);
    if (currentNode->b < 0) {
      /* current Node works, so we clean up the Plist,
      update the node potentials and return the current Node*/
      long distB = currentNode->dist;
      for (auto w = g.nodes.begin(); w != g.nodes.end() - 1; w++)
        w->inHeap = false;
      for (auto &w : perm)
        w->pot += (distB - w->dist);
      return currentNode;
    }
    last = (currentNode + 1)->first;
    for (auto e = currentNode->first; e != last; e++) {
      newNode = e->head;
      if (e->resCap > 0 and
          currentNode->dist + (e->c + e->head->pot - currentNode->pot) <
              newNode->dist) {
        /* this way is shorter. Hence we update the sp tree */
        // assert((e->c + e -> head -> pot - currentNode -> pot) >= 0);
        newNode->dist =
            currentNode->dist + (e->c + e->head->pot - currentNode->pot);
        newNode->parent = currentNode;
        newNode->current = e;
        if (not InList(newNode))
          Insert(newNode);
      }
    }
  }
  /* there is no node with negative b reachabel from v.
  This means that there is no b-flow. */
  return g.nodes.end();
}
```

Listing 3: Residual Graph

```
#include <fstream>
#include <sstream>
#include <algorithm>
#include <iostream>
#include "resGraph.h"

/* assumes input format is as in the second programming exercise.
 returns the residual graph corresponding to the zero flow */
resGraph::resGraph(std::ifstream &file) {
  /* internal arrays */
  std::vector<long> edge_tail;
```

```
std::vector<long> first;
std::vector<long> index;

/* ------------------------------------------------ reading input file
   ------------------------------------------------ */

// reading first line which is the number of nodes.
long num;
std::string line;
std::getline(file, line);
std::stringstream ss(line);
ss >> num;
n = num;

// allocating memory.
nodes.assign(n + 1, Node());
first.resize(n + 1);
b_values.resize(n);

source = nodes.begin();
sink = ++nodes.begin();

// reading supplies
for (int i = 0; i < n; i++) {
  std::getline(file, line);
  std::stringstream ss(line);
  long supply;
  ss >> supply;
  nodes[i].b = b_values[i] = supply;
}

// reading number of edges
std::getline(file, line);
ss.clear();
ss.str(std::string());
ss << line;
ss >> num;
m = 2 * num;

for (int i = 0; i < m / 2; i++) {
  std::getline(file, line);
  std::stringstream ss(line);
  long head, tail;
  ss >> tail >> head;
  long cap;
  ss >> cap;
  long cost;
  ss >> cost;

  edges.push_back(Edge(nodes.begin() + head, cap, cost));
  edges.push_back(Edge(nodes.begin() + tail, 0, -cost));

  edge_tail.push_back(tail);
  edge_tail.push_back(head);
  first[tail + 1]++;
  first[head + 1]++;

  index.push_back(i);
  index.push_back(-1);
}
```

```
for (long i = 0; i < m; i += 2) {
  edges[i].rev = edges.begin() + i + 1;
  edges[i + 1].rev = edges.begin() + i;
}

/* ---------------- linear time algorithm for sorting edges ---------------- */

/* at this moment the i+1'th enty of first contains the out degree of the i'th
 node
 after the next loop the i'th entry contains the index of the first edge
 leaving the i'th node */
nodes[0].first = nodes[0].current = edges.begin();
for (int i = 1; i < n + 1; ++i) {
  first[i] += first[i - 1];
  nodes[i].first = nodes[i].current = edges.begin() + first[i];
}

/* temprorary variables */
int tail, last, edge_num, edge_new_num;
edge_iter edge_current, edge_new;

/* When I wrote this, only God and I understood what I was doing.
Now, God only knows */
for (int i = 0; i < n - 1; i++) /* scanning all the nodes
                                     exept the last*/
{

  last = std::distance(edges.begin(), nodes[i + 1].first);
  /* edges outgoing from v must be cited
   from position first[v] to the position
   equal to initial value of first[v+1]-1 */

  for (edge_num = first[i]; edge_num < last; edge_num++) {
    tail = edge_tail[edge_num];

    while (tail != i)
    /* the edge no  edge_num  is not in place because edge cited here
     must go out from i;
     we'll put it to its place and continue this process
     until an edge in this position would go out from i */

    {
      edge_new_num = first[tail];
      edge_current = edges.begin() + edge_num;
      edge_new = edges.begin() + edge_new_num;

      /* keeping track of original index for output */
      std::swap(index[edge_num], index[edge_new_num]);

      /* edge_current must be cited in the position edge_new
       swapping these edge:                                  */
      std::swap(edge_current->head, edge_new->head);
      std::swap(edge_current->resCap, edge_new->resCap);
      std::swap(edge_current->c, edge_new->c);

      if (edge_new != edge_current->rev) {
        std::swap(edge_current->rev, edge_new->rev);
        (edge_current->rev)->rev = edge_current;
        (edge_new->rev)->rev = edge_new;
      }
```

```
      edge_tail[edge_num] = edge_tail[edge_new_num];
      edge_tail[edge_new_num] = tail;

      /* we increase first[tail]  */
      first[tail]++;

      tail = edge_tail[edge_num];
    }
  }
  /* all edges outgoing from  i  are in place */
}

/* gathering indices of original edges of the graph in flow */
for (int i = 0; i < m; i++) {
  if (index[i] >= 0)
    flow.push_back(i);
}
/* after the sort procedure the i'th entry of flow is the index
 of the edge with original index i */
std::sort(flow.begin(), flow.end(), [&](const int a, const int b) -> bool {
  return index[a] < index[b];
});

/* finally done */
}

void resGraph::print() {
  for (int i = 0; i < m / 2; ++i) {
    long flowValue = (edges[flow[i]].rev->resCap);
    if (flowValue > 0)
      std::cout << i << " " << flowValue << std::endl;
  }
}

void resGraph::readFlow(std::ifstream &file) {
  std::string line;
  std::getline(file, line);
  while (std::getline(file, line)) {
    std::stringstream ss(line);
    long i, value;
    ss >> i >> value;
    edges[flow[i]].resCap -= value;
    edges[flow[i]].rev->resCap += value;
  }
}
```