Description of Cycle canceling algorithm for solving the Minimum cost flow problem

García Sanjuan, Fernando; Martínez Villaronga, Adrià A.; Puigcerver Pérez, Joan; Melzer, Michaela

{*fergars2,admarvil,joapuipe*}*@inf.upv.es*, *melzer.michaela@web.de*

December 18, 2009

Abstract

This document presents the description of cycle canceling algorithm used to solve the minimum cost flow problem. This work was done for the subject *Estructures Matemàtiques per a la Informàtica II* (also known as *Graphs and Combinatorics*).

Contents

1 Basic notions

In this section some basic notions about Networks and Flows are introduced. It is expected that the reader knows basic concepts and algorithms of Graph Theory. Algorithms used in this work and which are not explained are breadth-first search (which traverses all the edges of a graph) and Bellman-Ford (which can calculate the distance from one vertex to any other and can detect negative-weight cycles).

1.1 Networks and flows

Definition 1 (Network)**.** A network is a tuple $G = (V, E, s, t, c)$ where V is the set of vertices of a graph, E is the set of undirected edges, $s \in V$ is called the source, $t \in V$ is called the sink and c is a function which assigns to each edge one integer capacity (its profile is $c : E \to \mathbb{N}$).

The graph defined by (V, E) must be weakly connected.

Definition 2 (Flow)**.** A flow is a function $f : E \to \mathbb{N}$ which verifies the following conditions:

1. $0 \leq f(e) \leq c(e), \forall e \in E$.
2. $\sum_{\forall i \in \Gamma(j)} f((j,i)) = \sum_{\forall i \in \Gamma^{-1}(j)} f((i,j)), \forall j \in V - \{s,t\}$[1].

Definition 3 (Total flow of a network)**.** Given a network $G = (V, E, s, t, c)$ and a flow f, the total flow of the network $f(G)$ for the given f is defined as:

$$f(G) = \sum_{i \in \Gamma(s)} f((s,i)) = \sum_{i \in \Gamma^{-1}(t)} f((i,t))$$

1.2 Maximum flow and Edmonds-Karp algorithm

This concepts are not directly related with the minimum cost flow problem, but they are used in our implementation of cycle canceling algorithm.

Definition 4 (Maximum flow)**.** A flow f is maximum, if and only if, $f(G) \geq f'(G)$, for all possible flows f' that can be found in G.

Theorem 1 (Augmenting path theorem)**.** *A flow f is maximum if and only if the residual network $G(f)$ contains no augmenting paths between s and t. [AMO93]*

The strategy of Edmonds-Karp algorithm lies on looking for augmenting paths in the residual network and increase the minimum residual capacity through this path. If any augmenting path is used, the number of incrementations that can be done is f, where f is the maximum amount of flow in G. If breadth-first search is used in order to find the shortest augmenting path between s and t, then it can be proved that the number of incrementations is bounded by $O(|V||E|)$.

[1] $\Gamma(i)$ denotes the set of adjacent vertices of i, and $\Gamma^{-1}(i)$ denotes the set of vertices such that i is one of their adjacents.

Algorithm 1 Given $G = (V, E, s, t, c)$, it computes the maximum flow on G.

```
1: $x = 0$
2: Compute the residual network $G_x$
3: **while** $G_x$ has augmenting paths **do**
4:     Compute the shortest augmenting path $P$ of $G_x$. Let $n$ be $P$ length
5:     $\delta = \min\{c((v_i, v_{i+1}))\}, 1 \leq i < n, v_i \in P$
6:     $x((v_i, v_{i+1})) = x((v_i, v_{i+1})) + \delta, 1 \leq i < n, v_i \in P$
7: **end while**
8: **return** $x$
```

Theorem 2. *The number of iterations in Edmonds-Karp algorithm is bounded by* $O(|V||E|)$. *[CLRS90]*

By theorems 1 and 2 it is proved that Edmonds-Karp computes the maximum flow in a finite number of steps.

Since breath-first search runs on $O(|E|)$, the running time cost of Edmonds-Karp algorithm is $O(|V||E|^2)$.

Edmonds-Karp algorithm can be used to compute a feasible flow f of $f(G)$ units in G. Since f is a feasible flow, $f(G)$ must be lower or equal than the maximum flow on G. If $f(G)$ is equal to the maximum flow, then the direct application of Edmonds-Karp will compute a feasible flow.

Theorem 3. *Let be* f *the maximum flow of* G*, a flow of* f' *units can be computed using Edmonds-Karp algorithm, where* $0 \leq f' \leq f$.

Proof. Compute the flow such that $f' = 0$ is trivial.

Suppose that we have a flow of f' units, where $f' < f$. Since f' is not the maximum flow, there exists an augmenting path from s to t of δ, where $1 \leq \delta \leq f - f'$. So, the flow f' can be augmented in 1 unit.

This procedure can be repeated up to $f' = f$, so a flow of f' units can be computed for all $0 \leq f' \leq f$. $\square$

Algorithm 2 Given $G = (V, E, s, t, c)$ and the desired f, it computes a flow of f units if it is possible. If not, it computes the maximum flow.

```
1: $x = 0$
2: Compute the residual network $G_x$
3: **while** $G_x$ has augmenting paths and $x < f$ **do**
4:     Compute the shortest augmenting path $P$ of $G_x$. Let $n$ be $P$ length
5:     $\delta = \min\{c((v_i, v_{i+1}))\}, 1 \leq i < n, v_i \in P$
6:     **if** $x + \delta > f$ **then**
7:         $\delta = f - x$
8:     **end if**
9:     $x((v_i, v_{i+1})) = x((v_i, v_{i+1})) + \delta, 1 \leq i < n, v_i \in P$
10: **end while**
11: **return** $x$
```

2 Description of the problem

Definition 5 (Flow cost)**.** A flow cost q is a function with profile $q : E \to N$ which assigns a cost of transportation to each unit of flow that "flows through an edge".

Thus, the definition of network can be extended to $G = (V, E, s, t, c, q)$ in the cases that a cost is assigned to each edge.

Definition 6 (Total flow cost)**.** The total flow cost for a given flow f is calculated as:

$$Q(f) = \sum_{\forall e \in E} q(e) \times f(e)$$

The statement of minimum cost flow problem is the next.

Definition 7 (Minimum cost flow problem)**.** Given a network $G = (V, E, s, t, c, q)$ and an amount of flow k, compute a flow f^* such that $f^*(G) = k$ and $Q(f^*) \leq Q(f), \forall f : f(G) = k$.

In the following section an algorithm for solving this problem known as cycle canceling algorithm is explained.

3 Cycle canceling algorithm

Cycle canceling algorithm is based on the Negative cycle optimality condition. In order to prove this theorem, we need introduce the concept of augmenting cycle and the augmenting cycle theorem.

Definition 8 (Augmenting cycle)**.** A cycle W in G is called an augmenting cycle with respect to the flow x if by augmenting a positive amount of flow f around the cycle, the flow remains feasible. The augmentation increases the flow on forward arcs in the cycle W and decreases the flow on backward arcs in the cycle. For example:

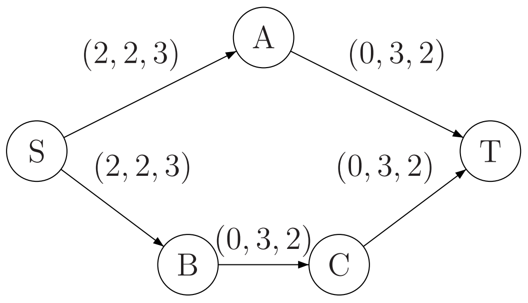


Figure 1: The label (f, c, q) of each edge represents the flow, the capacity and the cost.

In this network the non-directed cycle S, A, T, C, B, S is an augmenting cycle. If we decrease in 1 unit the flow through forward edges on the cycle ((S, A) and (S, T)) and increase in 1 unit the flow through backward edges, then the total amount of flow does not change.

Notice that a augmenting cycle with respect to a flow x corresponds to a directed cycle in a residual network G_x.

Theorem 4 (Augmenting cycle theorem)**.** *Let f and f' be any two feasible solutions of a network flow problem. Then f equals f' plus the flow on at most $|E|$ directed cycles in $G_{f'}$. Furthermore, the cost of f equals the cost of f' plus the cost of flow on these augmenting cycles. [AMO93]*

Theorem 5 (Negative cycle optimality condition)**.** *Let f^* be a feasible solution of a minimum cost flow problem. Then f^* is an optimal solution if and only if the residual network G_{f^*} contains no negative cost cycle.*

Proof. Suppose that f is a feasible flow and G_f contains a negative cycle. Then f cannot be an optimal flow, because we can augment the flow through the cycle improving the global cost. So, if $f*$ is an optimal flow, then G_{f^*} cannot contain a negative cycle.

Now, suppose that f is a feasible flow and G_f contains no negative cycle. Let f^* be an optimal flow such that $f^* \neq f$. Since f^* is an optimal flow, then by definition, the cost of f^* is lower or equal than f, this is $Q(f^*) \leq Q(f)$.

On the other hand, f^* can be transformed to f using at most $|E|$ augmenting cycles (by theorem 4), which are directed cycles on G_f. Let W_i be these directed cycles, then $Q(f^*) = Q(f) + \sum_{i=1}^{|E|} Q(W_i)$. Since G_f does not contain negative cost cycles, then $\sum_{i=1}^{|E|} Q(W_i) \geq 0$ and $Q(f^*) \geq Q(f)$.

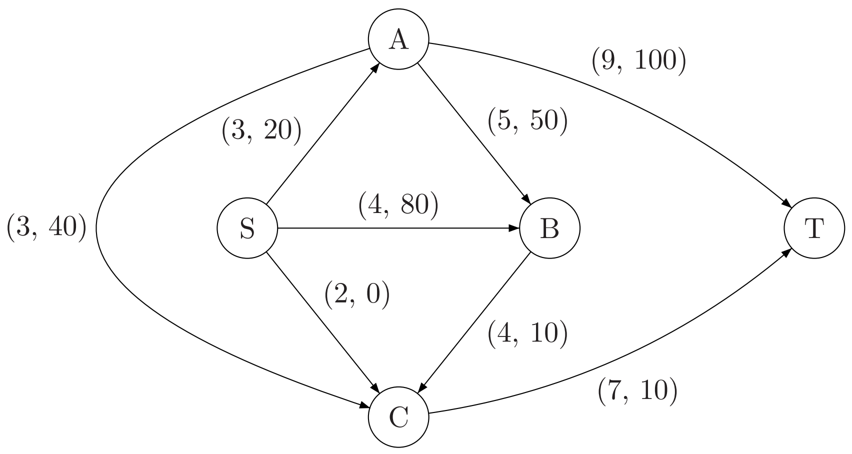
If $Q(f^*) \leq Q(f)$ and $Q(f^*) \geq Q(f)$, then $Q(f^*) = Q(f)$. So, f is optimal too. Thus, it is proved that if G_f contains no negative cycle, then f is optimal. □

The negative cycle optimality condition shows a simple algorithm for solving the minimum cost flow problem.

Algorithm 3 Given $G = (V, E, s, t, c, q)$ and f, it computes the minimum cost flow x of f units in G.

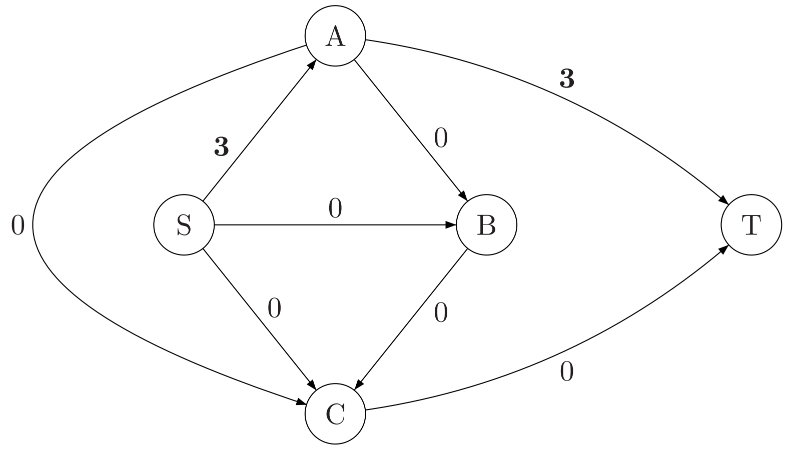
```
Require: $V$ is the set of vertices, $E$ is the set of edges, $s, t \in V$, $c : E \to \mathbb{N}$, $q : E \to \mathbb{N}$
    and $f \in \mathbb{N}$.
 1: Establish a feasible flow $x$ of $f$ units in $G$.
 2: $E_1 = \{(i,j) \in E : f((i,j)) < c((i,j))\}$
 3: $E_2 = \{(i,j) \in E : f((j,i)) > 0\}$
 4: $G_x = (V, E_1 \cup E_2, s, t, c, q)$
 5: while $G_x$ has a negative-cost cycle do
 6:   Compute the negative-cost cycle $C$
 7:   $d = \min\{c((i,j))\}, \forall (i,j) \in C$
 8:   $x((i,j)) = x((i,j)) + d, \forall (i,j) \in E : (i,j) \in E_1 \cap C$
 9:   $x((i,j)) = x((i,j)) - d, \forall (i,j) \in E : (j,i) \in E_2 \cap C$
10: end while
11: return $x$
```

3.1 Example

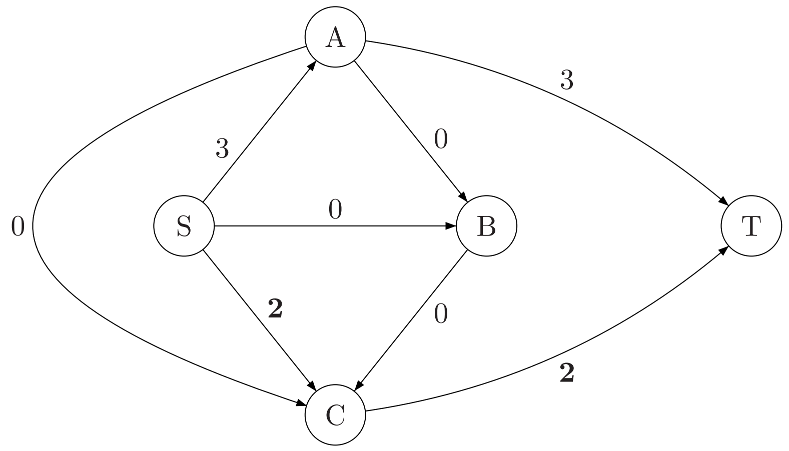


where the weights of the edges represent a tuple (c_i, q_i) where c_i is the capacity and q_i the cost of sending one unit of flow by the edge i. The objective is to find the flow of value 6 with the minimum cost.

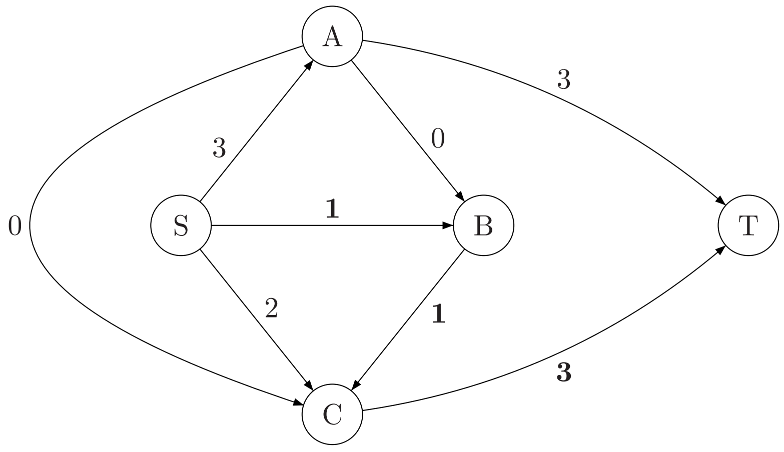
STEP 1: Edmonds-Karp algorithm allows to find a possible flow with the desired value on this network:
A flow of 3 units can be sent choosing the path $\{(S, A), (A, T)\}$:



Now, using the path $\{(S, C), (C, T)\}$ it is allowed to increase the flow in 2 units:



At this point, the total flow on the network is 5. Finally, it is possible to obtain a flow of 6 sending a flow of 1 by the f-augmenting path $\{(S,B),(B,C),(C,T)\}$:



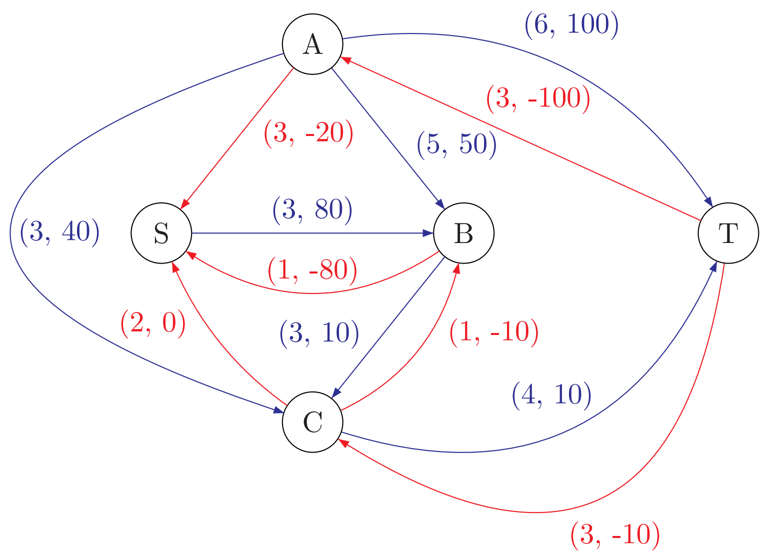
With this flow, the total cost is 480. The way to calculate it is:

$$\sum_{i=1}^{|E|} f_i \cdot q_i$$

So:
$(S,A) : 3 \cdot 20 = 60$
$(S,B) : 1 \cdot 80 = 80$
$(S,C) : 2 \cdot 0 = 0$
$(A,T) : 3 \cdot 100 = 300$
$(B,C) : 1 \cdot 10 = 10$
$(C,T) : 3 \cdot 10 = 30$
TOTAL COST: 60 + 80 + 0 + 300 + 10 + 30 = 480

That cost is not the minimum. It is necessary to improve it going on with the algorithm.

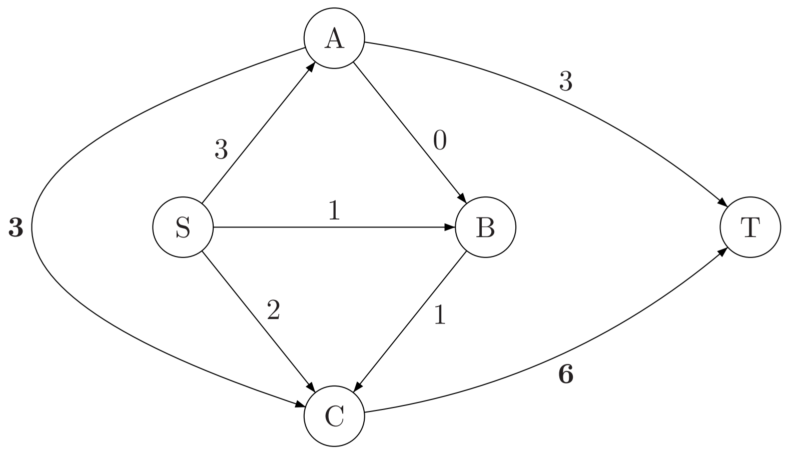
STEP 2: Find the marginal network:
$E_1 = \{(S,B),(A,B),(A,C),(A,T),(B,C),(C,T)\}$ (drawed in blue)
$E_2 = \{(A,S),(B,S),(C,S),(C,B),(T,A),(T,C)\}$ (drawed in red)



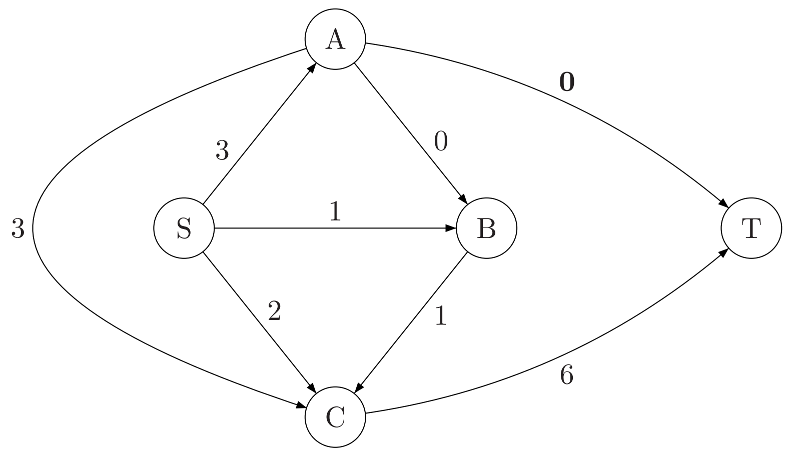
STEP 3: Bellman-Ford algorithm detects this cycle on the marginal network: $C = \{(T, A), (A, C), (C, T)\}$, which cost is negative: $-100 + 40 + 10 = -50 < 0$.

STEP 4: Calculate the minimum capacity of the cycle C:
$minCap = min(3, 3, 4) = 3$.

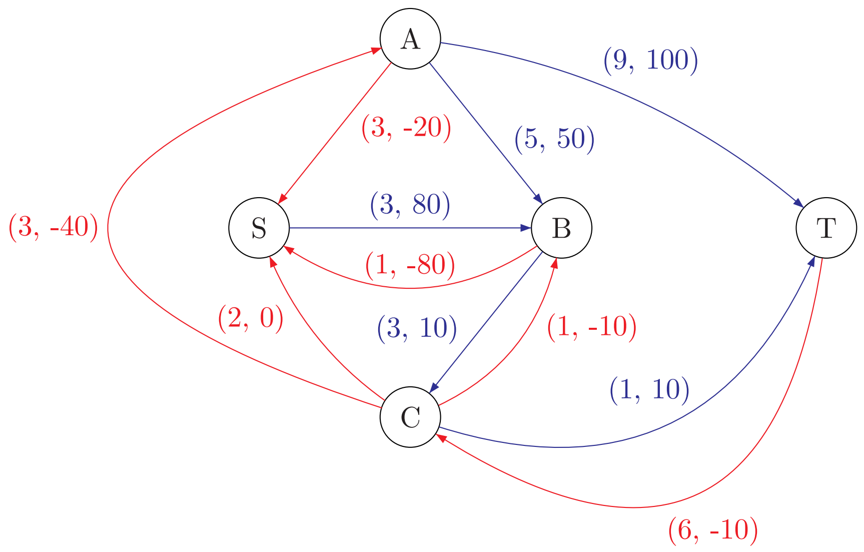
STEP 5: Update the flow acording that $minCap$ that has been just calculated:
5.1) $f(i, j) = f(i, j) + minCap, \forall(i, j) \in E : (i, j) \in E_1 \bigcap C$, where $f(i, j)$ is the flow of the edge (i, j) and E is the set of edges.



5.2) $f(i, j) = f(i, j) - minCap, \forall(i, j) \in E : (j, i) \in E_2 \bigcap C$.



STEP 2: It is time to recalculate the marginal network:
$E_1 = \{(S, B), (A, B), (A, T), (B, C), (C, T)\}$ (drawed in blue)
$E_2 = \{(A, S), (B, S), (C, S), (C, A), (C, B), (T, C)\}$ (drawed in red)



STEP 3: At this point, it can be seen that there is no negative cycles on the marginal network. Therefore, the algorithm is over.

The result is a flow with a value of 6 and which cost is the minimum possible (330). The way to calculate the cost is the following:
$(S, A) : 3 \cdot 20 = 60$
$(S, B) : 1 \cdot 80 = 80$
$(S, C) : 2 \cdot 0 = 0$
$(A, C) : 3 \cdot 40 = 120$
$(B, C) : 1 \cdot 10 = 10$
$(C, T) : 6 \cdot 10 = 60$
TOTAL COST: $60 + 80 + 0 + 120 + 10 + 60 = 330$

3.2 Correctness

The algorithm is a direct application of theorem 5, so if the algorithm ends, then the solution is optimal.

If all capacities are integer, then a feasible flow x can be found using the modification of Edmonds-Karp algorithm as we proved in theorem 3.

It is proved that after $|E|CQ$ (where C is the maximum capacity and Q is the maximum cost in the network) iterations, all the cycles are canceled and the algorithm ends. [AMO93]

3.3 Implementation

In our implementation of cycle canceling algorithm we used Edmonds-Karp to find a feasible flow as we explained in section 1.2.

In order to find negative cycles, Bellman-Ford algorithm can be used. When a negative cycle is detected by Bellman-Ford algorithm, this can be rebuilt using the predecessors list computed by the algorithm. We can use the definition of cycle and trace the predecessors of each vertex in order to obtain the cycle. This can be done in $O(|V|^2)$ since there are at most $|V| - 2$ vertices in a path between two vertices and $|V|$ paths are traversed. There are more efficient ways to obtain the negative cycle, but this is one easy.

A detailed implementation of the main algorithms is attached as an appendix.

3.4 Computational Cost

Let's analyze the algorithm step by step to determinate it's total cost:
First of all, it's necessary to find an admissible flow for the specified value. Edmonds-Karp can solve this problem in $O(|V||E|^2)$ as seen in section 1.2.
Once this is done, negative cost cycles must be found in order to redirect the flow through that cycle. As we explained in section 3.2 at most $|E|CQ$ iterations are done.
In order to find the cycle, it's necessary to build the residual graph and apply Bellman-Ford. The resiudal graph is built in $O(|E|)$ and Bellman-Ford has a cost of $O(|V||E|)$ [AMO93].
Updating the flow through the network is the last step. It costs $O(|E|)$ since the flow must be updated in each edge.

$$\begin{gathered}
O(|V| \cdot |E|^2) + |E|CQ \cdot (O(|E|) + O(|V| \cdot |E|) + O(|E|) \\
O(|V| \cdot |E|^2) + |E|CQ \cdot (O(|V| \cdot |E|)) \\
O(|V| \cdot |E|^2) + O(|V||E|^2 CQ) \\
O(|V||E|^2 CQ)
\end{gathered}$$

The global cost of that algorith is $\mathbf{O}(|\mathbf{V}||\mathbf{E}|^{\mathbf{2}}\mathbf{CQ})$.

4 Applications

4.1 An immediate application

Given a network of pipes, where each one has a maximum capacity, is desired to carry a flow of water f from A to B. The problem is that these pipes have a maintenance cost which is directly proportional to the flow passing through them. Obviously, the transportation cost is want to be minimal.

This situation can be modeled as a graph where each edge corresponds to a pipe. It is a very basic and very direct application of the generic problem, but not less important.

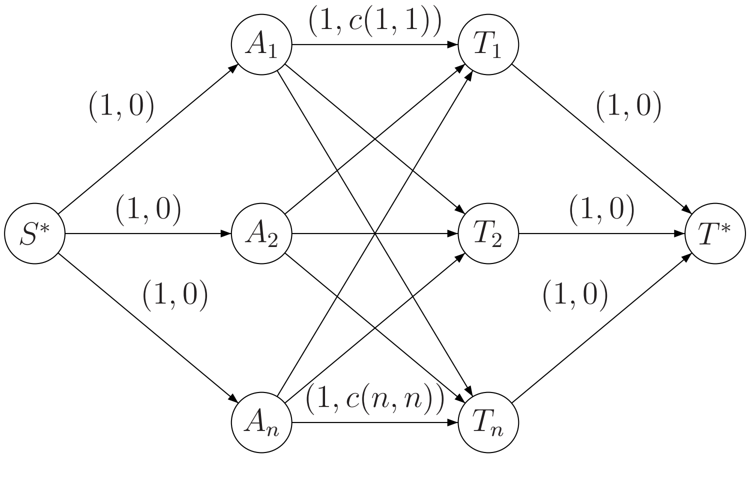
4.2 The assignment problem

Given a set of A agents and a set of T tasks, where $|A| = |T|$, each agent must be assigned to a task incurring some cost $c(A_i, T_i)$ which depends on the agent-task assignment. The assignment problem lies on assign each task to each agent incurring the minimum cost.

It is possible to represent the statement of this problem using a network with one vertex for each agent a task and two extra vertices which are the source and the sink. Each vertex representing an agent will be connected to all the vertices representing a task.

The capacity of each edge connecting one pair of agent-task is 1 and the cost is the specified by the function c.

The capacity of the edges connecting the source and the sink with other vertices is 1 and the cost is 0.



In order to find a solution for the assignment problem a flow of n units must be sent from S^* to T^*. If we want to minimize the cost of this flow, we can apply the algorithm seen before.

5 Other approaches

There are many other strategies to solve this problem. Cycle canceling is old-known and easy, but not the most efficient.

Bellman-Ford algorithm has one of the best asymptotic running time to find negative cycles, but there are more efficient algorithms which run faster in practice [CG96].

Other completely different approaches to solve this problem exists too. For example "Successive Shortest Path and Capacity Scaling" [EK72] or "Cost Scaling" methods.

The most efficient approach is based on linear programming since the minimum cost flow problem can be stated as a optimization problem. The algorithm used to solve this kind of problem is known as Network Simplex algorithm. This is the most used and it is implemented in programming libraries like Lemon Graph Library[2].

A Source code

A.1 Bellman-Ford algorithm

```
/* Bellman-Ford algorithm. O(|V| |E|) */
bool Bellman_Ford (int * pred, int * capacity) const
{
  assert(pred != 0);
  assert(capacity != 0);

  // O(|V|)
  int * distance = new int [N];
  for(size_t i = 0; i < N; ++i) {
    distance[i] = INT_MAX;
    capacity[i] = INT_MAX;
    pred[i] = -1;
  }

  unsigned int s = 0;
  while( adjList[s] == 0 ) s++;
  distance[s] = 0;

  // O(|V| + |V|*|E|)
  for(size_t i = 1; i < N; ++i){
    for(size_t u = 0; u < N; ++u)
      for(Edge * e = adjList[u]; e != 0; e = e->next)
        if( distance[u] != INT_MAX &&
            distance[e->v] > distance[u] + e->cost ){
          distance[e->v] = distance[u] + e->cost;
          pred[e->v] = u;
          capacity[e->v] = e->capa;
        }
  }

  // O(|V| + |E|)
  bool negative_cycle = false;
  for(size_t u = 0; u < N && !negative_cycle; ++u)
    for(Edge * e = adjList[u]; e != 0; e = e->next)
      if( distance[u] != INT_MAX &&
          e->cost + distance[u] < distance[e->v]){
```

[2]http://lemon.cs.elte.hu/trac/lemon

```
        negative_cycle = true;
        break;
      }


  delete [] distance;
  return negative_cycle;
}
```

A.2 Edmonds-Karp algorithm

```
/* Look for a f-augmenting path.  O(|E|) */
bool f_augmenting_path(unsigned int source, unsigned int sink,
                       unsigned int& f, int * path) const
{

  bool find = false;
  queue<unsigned int> Q;
  int * inc_flow = new int [N];

  memset(inc_flow, 0x00, sizeof(int)*N);
  memset(path, 0xFF, sizeof(int)*N);

  inc_flow[source] = INT_MAX;
  path[source] = 0;
  Q.push(source);
  while( !Q.empty() ){
    unsigned int u = Q.front(); Q.pop();
    if(u == sink){ find = true; break; }

    for( Edge * e = adjList[u]; e != 0; e = e->next ){
      if( path[e->v] < 0 && (e->capa - e->flow) ){
        path[e->v] = u;
        inc_flow[e->v] = MIN(inc_flow[path[e->v]],
                             e->capa - e->flow);
        Q.push(e->v);
      }
    }
  }

  delete [] inc_flow;
  f = inc_flow[sink];
  return find;
}

/* Increase the flow between source and sink in 'inc' units. */
void increaseFlow(unsigned int source, unsigned int sink,
                  int * path, int inc)
{
  for(unsigned int v = sink; v != source; v = path[v]) {
    Edge * e = adjList[path[v]];
    for(; e->v != v; e = e->next);
    e->flow += inc;
  }
}
```

```
/* Edmonds-Karp algorithm.   O(|V| |E|^2)*/
unsigned int Edmonds_Karp(unsigned int source, unsigned int sink,
                          unsigned int desired_flow)
{
  assert(source < N && sink < N);

  reset_flows();

  int * path = new int [N];
  bool stop = false;
  unsigned int flow = 0, inc_flow = 0;
  while( f_augmenting_path(source, sink, inc_flow, path) && !stop){

    if(flow + inc_flow > desired_flow){
      inc_flow = desired_flow - flow;
      stop = true;
    }

    increaseFlow(source, sink, path, inc_flow);
    flow += inc_flow;

  }

  delete [] path;
  return flow;
}
```

A.3 Cycle canceling algorithm

```
/* Cycle canceling algorithm. O(|V| |E|^2 Q C) */
int Cycle_Cancelling(unsigned int source, unsigned int sink,
                     unsigned int desired_flow, bool verbose ) {

  // Step 1       O( |V| * |E|^2 )
  // Compute a feasible flow
  if ( Edmonds_Karp(source, sink, desired_flow) != desired_flow ){
    cerr << "It's impossible to find a flow of "
         << desired_flow << " units" << endl;
    return -1;
  }
  if(verbose) write_flow("flow_0");

  Graph residual;
  char name[100];
  int * cycle = new int [N];
  int * capacity = new int [N];
  bool * visited = new bool [N];
  int it = 1;
  while ( 1 ) {

    // Step 2       O(|V| + |E|)
    // Compute the residual graph
    residual = residual_graph();
    if(verbose){
      sprintf(name, "residual_%d", it -1);
      residual.write(name);
```

```
}

// Step 3      O(|V| + |V|*|E|)
// Check if the residual graph contains a negative-weight cycle
if( !residual.Bellman_Ford(cycle, capacity) ){
  unsigned int cost = 0;
  for(size_t u = 0; u < N; ++u)
    for(Edge *e = adjList[u]; e != 0; e = e->next)
      cost += e->flow * e->cost;

  delete [] cycle; delete [] capacity; delete [] visited;
  return cost;
}


// Step 3.1       O(|V|^2)
// Obtain the negative cycle
int first = -1;
for(size_t u = 0; u < N && first < 0; ++u){
  memset(visited, 0x00, sizeof(bool) * N);
  size_t v = u, i = 0;
  while( i < N ){
    visited[v] = true;
    v = cycle[v]; ++i;

    if( v < 0 ) break;
    if( visited[v] ) {
      first = v;
      break;
    }
  }
}


// Step 4 O(|V|)
// Obtain the minimum capacity through the cycle
int min_cap = INT_MAX;
size_t v = first;
do {
  if( capacity[v] < min_cap ) min_cap = capacity[v];
  v = cycle[v];
} while (v != first);


// Step 5 O(|V| + |E|)
// Update the flow
v = first;
Edge *e = 0;
do {
  /* E1 */
  e = adjList[cycle[v]];
  while( e != 0 && e->v != v ){ e = e->next; }
  if( e != 0 && e->flow < e->capa ) e->flow += min_cap;

  /* E2 */
```

```
          e = adjList[v];
          while( e != 0 && e->v != cycle[v] ){ e = e->next; }
          if( e != 0 && e->flow > 0 ) e->flow -= min_cap;

          v = cycle[v];
        } while (v != first);


        if (verbose){
          sprintf(name, "flow_%d", it);
          write_flow(name);
        }
        ++it;
      }
    }

};
```

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, algorithms and applications*. Prentice-Hall, 1993.

[CG96] Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. Technical report, NEC Research Institute, 1996.

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithm*. MIT Press, 1990.

[EK72] Jack R. Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the Association for Computing Machinery 19*, 1972.