A parallel algorithm for the minimum flow problem in bipartite networks

LAURA CIUPALĂ
Department of Computer Science
University Transilvania of Brașov
Iuliu Maniu Street 50, Brașov
ROMANIA

ELEONOR CIUREA
Department of Computer Science
University Transilvania of Brașov
Iuliu Maniu Street 50, Brașov
ROMANIA

Abstract: - In this paper, we develop a parallel implementation of the deficit scaling algorithm for minimum flow in bipartite networks. This algorithm performs a pull from an active node with a large deficit and with the smallest distance label from $N_1$ at a time followed by pulls from several nodes in $N_2$ in parallel. It runs in $O(n_1^2 \log C \log p)$ time using $p = \lceil m/n_1 \rceil$ processors.

Key-Words: - Network flow; Parallel algorithms; Minimum flow problem; Scaling technique

1 Introduction

The literature on network flow problem is extensive. Over the past 50 years researchers have made continuous improvements to algorithms for solving several classes of problems. From the late 1940s through the 1950s, researchers designed many of the fundamental algorithms for network flow, including methods for maximum flow and minimum cost flow problems. In the next decades, there are many research contributions concerning improving the computational complexity of network flow algorithms by using enhanced data structures, techniques of scaling the problem data etc.

Although it has its own applications, the minimum flow problem was not dealt so often as the maximum flow ([1], [2], [3], [12], [13], [14], [15], [16]) and the minimum cost flow problem ([1], [6], [17]). There are many problems that occur in economy that can be reduced to minimum flow problems.

The minimum flow problem in a network an be solved in two phases:
(1) establishing a feasible flow, if there is one
(2) from a given feasible flow, establish the minimum flow.

The problem of determining a feasible flow can be reduced to a maximum flow problem (for details see [1]).

For the second phase of the minimum flow problem there are three approaches:
1. using decreasing path algorithms (see [8], [9])
2. using preflow algorithms (see [3], [5], [8], [9])
3. finding a maximum flow from the sink node to the source node in the residual network (see [2], [6]).

All these algorithms can be modified in order to become more efficient when applied on bipartite networks.

In this paper, we develop a parallel implementation of the deficit scaling algorithm for minimum flow in bipartite networks. This algorithm performs a pull from an active node with a large deficit and with the smallest distance label from $N_1$ at a time followed by pulls from several nodes in $N_2$ in parallel. On a PRAM with $p = \lceil m/n_1 \rceil$ processors, it runs in $O(n_1^2 \log C \log p)$ time.

2 Notation and definition

A network $G = (N, A)$ is called bipartite if its node set $N$ can be partitioned into two subsets $N_1$ and $N_2$, such that all arcs have one endpoint in $N_1$ and the other in $N_2$.

We consider a bipartite capacitated network $G = (N, A, l, c, s, t)$ with a nonnegative capacity $c(i, j)$ and with a nonnegative lower bound $l(i, j)$ associated with each arc $(i, j) \in A$. We distinguish two special nodes in the network $G$: a source node $s$ and a sink node $t$. We assume without loss of generality that $s \in N_1$ and $t \in N_2$.

Let $n = |N|$, $n_1 = |N_1|$, $n_2 = |N_2|$, $m = |A|$ and $C = \max \{ c(i, j) \mid (i, j) \in A \}$.

A flow is a function $f: A \to \mathbb{R}$, satisfying the next conditions:

\[ f(s, N) - f(N, s) = v \] (1)
\[ f(i, N) - f(N, i) = 0, \quad i \neq s, t \] (2)
\[ f(i, N) - f(N, i) = -v \] (3)
\[ l(i, j) \leq f(i, j) \leq c(i, j), \quad (i, j) \in A \] (4)

for some $v \geq 0$, where

\[ f(i, N) = \Sigma_j f(i, j), \quad i \in N \]
and

\[ f(N, i) = \Sigma_j f(j, i), \quad i \in N. \]

We refer to $v$ as the value of the flow $f$.

The minimum flow problem is to determine a flow $f$ for which $v$ is minimized.

For the minimum flow problem, a *preflow* is a
function $f: A \rightarrow \mathbb{R}_+$ satisfying the next conditions:

$$f(i, N) - f(N, i) \leq 0, \ i \neq s,t$$  \hspace{1cm} (5)
$$l(i, j) \leq f(i, j) \leq c(i, j), (i, j) \in A$$  \hspace{1cm} (6)

Let $f$ be a preflow. We define the deficit of a node $i \in N$ in the following manner:

$$e(i) = f(i, N) - f(N, i)$$  \hspace{1cm} (7)

Thus, for the minimum flow problem, for any preflow $f$, we have $e(i) \leq 0, \ i \in N \setminus \{s, t\}$.

We say that a node $i \in N \setminus \{s, t\}$ is active if $e(i) < 0$ and balanced if $e(i) = 0$.

A preflow $f$ for which $e(i) = 0, \ i \in N \setminus \{s, t\}$ is a flow. Consequently, a flow is a particular case of preflow.

For the minimum flow problem, the residual capacity $r(i, j)$ of any arc $(i, j) \in A$, with respect to a given preflow $f$, is given by

$$r(i, j) = c(j, i) - f(i, j) + f(i, j) - l(i, j).$$

By convention, if $(j, i) \notin A$ then we add arc $(j, i)$ to the set of arcs $A$ and we set $l(j, i) = 0$ and $c(j, i) = 0$. The residual capacity $r(i, j)$ of the arc $(i, j)$ represents the maximum amount of flow from the node $i$ to node $j$ that can be canceled. The network $G_f = (N, A_f)$ consisting only of the arcs with positive residual capacity is referred to as the residual network (with respect to preflow $f$).

In the residual network $G_f = (N, A_f)$ the distance function $d : N \rightarrow \mathbb{N}$ with respect to a given preflow $f$ is a function from the set of nodes to the nonnegative integers. We say that a distance function is valid if it satisfies the following conditions:

$$d(s) = 0$$
$$d(j) \leq d(i) + 1, \ \text{for every arc} \ (i, j) \in A_f.$$

We refer to $d(i)$ as the distance label of node $i$.

**Theorem 1** [5](a) If the distance labels are valid, the distance label $d(i)$ is a lower bound on the length of the shortest directed path from node $s$ to node $i$ in the residual network.

(b) If $d(t) \geq n$, the residual network contains no directed path from the source node to the sink node.

We say that the distance labels are exact if for each node $i$, $d(i)$ equals the length of the shortest path from node $s$ to node $i$ in the residual network.

We refer to an arc $(i, j)$ from the residual network as an admissible arc if $d(j) = d(i) + 1$; otherwise it is inadmissible.

We refer to a node $i$ with $e(i) < 0$ as an active node. We adopt the convention that the source node and the sink node are never active.

### 3 Deficit scaling algorithm

This algorithm is a special implementation of the generic preflow algorithm for minimum flow and, like all preflow algorithms for minimum flow, it maintains a preflow at every step and proceeds by pulling the deficits of the active nodes closer to the source node. For measuring closeness it uses the exact distance labels. Consequently, pulling the deficits from the active nodes closer to the source node means decreasing flow on admissible arcs.

Let $e_{\text{max}} = \max \{-e(i) \mid i \text{ is an active node}\}$. The **deficit dominating** is the smaller integer $\bar{F}$ that is a power of 2 and satisfies $e_{\text{max}} \leq \bar{F}$. We refer to a node $i$ with $e(i) \leq -\bar{F}/2$ as a node with large deficit and as a node with small deficit otherwise.

The scaling deficit algorithm for the minimum flow always pulls flow from active nodes with sufficiently large deficits to nodes with sufficiently small deficits in order to not allow that a deficit becomes too large.

The deficit scaling algorithm for the minimum flow is the following:

**Deficit scaling algorithm;**

begin

```
if $f$ is not labeled then $f$ is a minimum flow
else begin
  for each arc $(i, t) \in A$ do $f(i, t) := l(i, t); \ d(t) := n$;
  $\bar{F} := 2^\lceil \log C \rceil$;
  while $\bar{F} \geq 1$ do
    begin
      while the network contains an active node with a large deficit do
        begin
          among active nodes with large deficits, select a node $j$ with the smallest distance label;
          pull_relabel($j$);
        end
      $\bar{F} := \bar{F} / 2$;
    end
  end
```

end

```
Procedure pull_relabel($j$)
begin
  if the network contains an admissible arc $(i, j)$ then
    if $i \neq t$ then
      pull $g = \min \{-e(j), r(i, j), \bar{F} + e(i)\}$ units of flow from node $j$ to node $i$;
    else

  end

  return
```

end
When applied on bipartite networks, the deficit scaling algorithm determines a minimum flow.

Actually, the algorithm terminates with optimal residual capacities. From these residual capacities we can determine a minimum flow in several ways. For example, we can make a variable change: For all arcs \((i, j)\), let

\[
c'(i, j) = c(i, j) - l(i, j),
\]

\[
r'(i, j) = r(i, j),
\]

\[
f'(i, j) = f(i, j) - l(i, j).
\]

The residual capacity of arc \((i, j)\) is

\[
r(i, j) = c(i, j) - f(j, i) + f(i, j) - l(i, j)
\]

Equivalently,

\[
r(i, j) = c'(i, j) - f'(j, i) + f'(i, j).
\]

We can compute the value of \(f'\) in the following way:

\[
f'(i, j) = \max\{r'(i, j) - c'(j, i), 0\}.
\]

Converting back into the original variables, we obtain the following expression:

\[
f(i, j) = l(i, j) + \max\{r(i, j) - c(j, i) + l(j, i), 0\}.
\]

**Theorem 3** [5] The deficit scaling algorithm runs in \(O(nm+n^2\log C)\) time.

When applied on bipartite networks, the deficit scaling algorithm can be improved by imposing the rule that we pull flow from two adjacent arcs in order maintain all the nodes in \(N_1\) balanced, only nodes in \(N_1\) could have deficits. We refer to these operations as **bipulls**. By replacing the **pull_relabel** procedure with the **bipull_relabel** procedure described below, one obtains a minimum flow in a bipartite network in \(O(nm+n_1^2\log C)\) time.

**Procedure pull_relabel()**

**begin**

**if** the network contains an admissible arc \((i, j)\) **then**

**if** \(h \neq t\) **then**

\[
pull g = \min\{-e(j), r(i, j), r(h, i), F + e(h)\} \text{ units of flow along the path } h \rightarrow i \rightarrow j;
\]

**else**

\[
pull g = \min\{-e(j), r(i, j), r(h, i)\} \text{ units of flow along the path } h \rightarrow i \rightarrow j;
\]

**else**

\[
d(h) := \min\{d(h) | (h, i) \in A_f\} + 1;
\]

**else**

\[
d(j) := \min\{d(i) | (i, j) \in A_f\} + 1;
\]

**end**

**4 Parallel deficit scaling algorithm for minimum flow in bipartite networks**

We develop a parallel implementation of the deficit scaling algorithm for minimum flow in bipartite networks on a EREW PRAM using \(p = \lceil m/n_1 \rceil\) processors. This algorithm works on networks in which any node has both in-degree and out-degree no greater than \(p\). This restriction implies no loss of generality because any bipartite network with \(m\) arcs, \(n_1\) nodes in \(N_1\) and \(n_2\) nodes in \(N_2\) can be transformed in an equivalent bipartite network with \(O(m)\) arcs, \(O(n_1)\) nodes in \(N_1\) and \(O(n_2)\) nodes in \(N_2\) in which any node has both in-degree and out-degree no greater than \(p\) (for details see [2]).

For any node \(j \in N\), let \(N(j) = \{i \in N | (i, j) \in A_f\}\). We assume that nodes in \(N(j)\) are denoted by \(j_1, j_2, \ldots, j_s\), where \(k = |N(j)|\). Let \(N'(j) = \{i \in N | (i, j) \in A_f\}\) and \((i, j)\) is an admissible arc.

For each node \(j \in N_2\), we refer to

\[
r'(j) = \sum_{i \in N_1} r(i, j)
\]

to be the effective residual capacity of node \(j\).

We define the effective residual capacity \(r'(i, j)\) of arc \((i, j)\) in the following manner:

\[
r'(i, j) = 0 \text{ if } (i, j) \text{ is not an admissible arc}
\]

\[
r'(i, j) = r(i, j) \text{ if } (i, j) \text{ is an admissible arc and}
\]

\[
i \in N_2 \text{ and } j \in N_1
\]

\[
r'(i, j) = \min\{r(i, j), r'(i)\} \text{ if } (i, j) \text{ is an admissible arc and } i \in N_2 \text{ and } j \in N_1
\]

The parallel deficit scaling algorithm for minimum flow in bipartite networks pulls flow from a node \(j \in N_1\) at a time and then it pulls flow from several nodes from \(N_2\) in parallel. Pulling \(r'(i, j)\) units of flow on any arc \((i, j)\) with \(i \in N_2 \text{ and } j \in N_1\), we can be sure that we never pull more flow into a node \(i \in N_2\) than its effective residual capacity. Consequently, all the deficit of node \(i\) can be pulled out prior to a relabel of node \(i\).

For an efficient allocation of the processors to the arcs, we will use the following functions:

**Current(j)** will return the current arc entering into \(j\)

**NextCurrent(j, g)** will return \(|N(j)| + 1\) if after pulling \(g\) units of flow from node \(j\) all admissible arc entering in \(j\) will be dropped from the residual network. Otherwise, it will return the index of the arc that will be current arc after pulling \(g\) units of flow from node \(j\).

**NextDecrement(j, g)** will return the amount of flow that will be pull on arc **NextCurrent(j, g)** when pulling flow from node \(j\).

**Allocate(j, D)** takes as an input a node \(j\) and a \(p\)-dimensional array of demands of processors from the
nodes in \(N(j)\) and returns a vector \(\text{proc}\), where \(\text{proc}(k)\) is the set of processors allocated to the node \(j_k\) from \(N(j)\).

Assuming that \(|N(j)|\) is a power of 2, we can associate to any node \(j\) a complete binary tree \(T(j)\) whose leaves are the indexes of the nodes in \(N(j)\). The key of the leaf \(k\) is \(r'(j_k, j)\) and the key of each internal node of the binary tree is the sum of the keys of its descendent leaves.

When a node \(j\) is relabeled, to each node \(j_k\) of \(N(j)\) is assigned a processor and its binary tree is updated. This assignment of processors takes \(O(\log p)\) steps per relabel. Moreover, each processor updates its binary tree in \(O(\log p)\) steps.

When a pull of flow from a node \(j\) is performed, the binary tree for the node \(j\) must be updated. If \(k\) processors are assigned, the \(\text{Current}(j)\) is increased by at most \(k\) and the updating can be accomplished with \(k\) processors in \(O(\log p)\) time.

In order to compute \(\text{NextCurrent}(j, g)\), we start at the root of the binary tree corresponding to the node \(j\) and we select the right child or the left child depending on whether \(g\) is less than or greater than the key of the right child. We then recur on the selected child. We also can compute \(\text{NextDecrement}(j, g)\) in this manner. Obviously, both functions \(\text{NextCurrent}(j, g)\) and \(\text{NextDecrement}(j, g)\) can be computed using one processor in \(O(\log p)\) time.

The parallel bipartite deficit scaling algorithm performs at a time a pull from an active node with a large deficit and with the smallest distance label from \(N_1\) followed by pulls from several nodes in \(N_2\) in parallel.

**Parallel bipartite deficit scaling algorithm:**

\[
\begin{align*}
\text{begin} & \\
\text{let } f & \text{ be a feasible flow in network } G; \\
\text{compute the exact distance labels } d(\cdot) \text{ in the residual network } G_f \text{ by applying the BFS parallel algorithm from the source node } s; \\
\text{if } t \text{ is not labeled then } f \text{ is a minimum flow} & \\
\text{else begin} & \\
\text{for each arc } (i, t) \in A \text{ do in parallel} & \\
& f(i, t) := h(i, t); \\
& d(t) := n; \\
& \bar{F} := 2^\log C; \\
\text{while } \bar{F} \geq 1 \text{ do} & \\
\text{while the network contains an active node with a large deficit} & \\
\text{do} & \\
\text{begin} & \\
& \text{determine in parallel } d(j) = \min\{d(i) \mid i \text{ is an active node with large deficit}\}; \\
& \text{parallel bipull\_relabel}(j, \bar{F} / 2, \text{proc}(j)); \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\end{align*}
\]

\[
\bar{F} := \bar{F} / 2;
\]

**Procedure parallel bipull\_relabel**(\(j, g, P\))

\[
\begin{align*}
\text{parallel pull}(j, g, P); & \\
\text{while } e(j_k) < 0 \text{ for some } j_k \in N(j) \text{ do} & \\
\text{begin} & \\
& \text{for } k = 1 \text{ to } p \text{ do in parallel} & \\
& D(j_k) = \text{NextCurrent}(j_k, -e(j_k)) - \text{Current}(j_k) + 1; & \\
& \text{proc} = \text{Allocate}(j, D); & \\
& \text{for } k = 1 \text{ to } p \text{ do in parallel} & \\
& \text{parallel pull}(j_k, -e(j_k), \text{proc}(k)); & \\
& \text{update data structures} & \\
\text{end}; & \\
& \text{for each } j_k \in N(j) \text{ do} & \\
& \text{if } \text{Current}(j_k) = |N(j_k)| + 1 \text{ then} & \\
& \text{relabel the node } j_k; & \\
& \text{if } \text{Current}(j) = |N(j)| + 1 \text{ then} & \\
& \text{relabel the node } j; & \\
\text{end}; & \\
\text{end}; & \\
\end{align*}
\]

**Procedure parallel pull**(\(j, g, P\))

\[
\begin{align*}
& c = \text{Current}(j); \\
& k = \text{NextCurrent}(j, g); \\
& s = |P|; \\
& \text{for } i = c \text{ to min}(k-1, c+s-1) \text{ do in parallel} & \\
& \text{pull } r'(j_i, j) \text{ units of flow on arc } (j_i, j) \text{ and update } r'; & \\
& \text{if } s \geq k+c+1 \text{ and } k \leq |N(j)| \text{ then} & \\
& \text{pull NextDecrement}(j, g) \text{ units of flow} & \\
& \text{on arc } (j_k, j) \text{ and update } r'; & \\
& \text{Current}(j) = \text{NextCurrent}(j, g); & \\
\text{end}; & \\
\end{align*}
\]

**Theorem 4** The parallel bipartite deficit scaling algorithm determines a minimum flow in a bipartite network in \(O(n_1^2 \log C \log p)\) time using \(p = \lfloor m/n_1 \rfloor\) processors.

This theorem can be proved in a similar manner with the complexity theorem of the parallel bipartite deficit scaling algorithm for maximum flow in bipartite networks (for details see [2]).

5 Conclusion
In this paper, we developed a parallel implementation of the deficit scaling algorithm for minimum flow in bipartite networks. This algorithm performs a pull from an active node with a large deficit and with the smallest distance label from $N_1$ at a time followed by pulls from several nodes in $N_2$ in parallel. Consequently, it runs in $O(n_1^2 \log C \log p)$ time using $p = \left\lceil \frac{m}{n_1} \right\rceil$ processors.

References: