**OR 215 Spring 1999**

**Network Flows M. Hartmann**

**THE EXCESS SCALING ALGORITHM**

**FOR THE MAXIMUM FLOW PROBLEM**

The Capacity Scaling Algorithm

The Excess Scaling Algorithm

Proof of Polynomial Time

Extensions

# THE GENERIC SCALING ALGORITHM

**Input:** A problem instance ***P****.*

* Let ***P\**** be a very rough approximation to ***P***.
* Solve problem ***P\****, possibly approximately.
* Replace ***P\**** by a less rough approximation to ***P***.
* Solve problem ***P\****, possibly approximately, starting with the previous solution.

Iterate until problem ***P*** is solved optimally.

**IMPROVEMENT IN THE FORD FULKERSON AUGMENTING PATH ALGORITHM**

Augment along the path that maximizes (P), the residual capacity of the path.

* Number of augmentations is O(m log U).
* Running time is O(m2 log U).

**A scaling variant:**

* Select a target value Δ. Initially Δ = U/2.
* Augment along a path p with (P) ≥ Δ. If no such path exists, replace Δ by Δ/2.

The number of augmenting paths is O(m log U), or O(m) between successive divisions of Δ by 2.

**Running time:**  O(nm log U) if implemented well, or

O(nm) between successive divisions of Δ by 2.

**BOUNDING THE NUMBER**

**OF AUGMENTATIONS**

At the end of the scaling iteration, the residual capacity from S to T is less than mΔ:

< Δ

< Δ

< Δ

T

S

Hence the number of augmentations at next iteration is less than 2m.

**AN ALGORITHM FOR WHICH THE NUMBER OF**

**NON-SATURATING PUSHES IS O(n2 log U).**

Let K satisfy U ≤ 2K and let emax = max { e(i) : i∈N }.

**algorithm** EXCESS-SCALING;

**begin**

PREPROCESS;

K := ⎡log 2 U⎤ ;

**for** k := K **down to** 0 **do**

**begin** { Δ-scaling phase }

Δ := 2k ;

**while** there is a node i with e(i) > Δ/2 **do**

push/relabel(i,Δ);

**end;**

**end**

push/relabeL must be modified to ensure that no node excess exceeds Δ in the Δ-scaling phase.

# PRELIMINARIES

The Δ in the scaling phase is referred to as the *excess-dominator*. The scaling phase is also called the Δ-scaling phase.

* The number of scaling phases is O(log U).
* At the Δ-scaling phase, Δ/2 < emax ≤Δ.
* Each scaling phase reduces Δ by a factor of 2.
* After K +1 scaling phases, emax is reduced to 0.

**New data structures:**

ActNode(Δ,k) = { i ∈ N : d(i) = k, e(i) > Δ/2 }

ActSet(Δ) = { k : ActNode(Δ) ≠ ∅ }.

We store the labels in ActSet(Δ) in increasing order.)

**procedure** SELECT;

**begin**

**if**  ActSet(Δ) = ∅ **then**

go to the Δ/2 scaling phase

**else**

**begin**

select the minimum distance label k in ActSet(Δ);

select i ∈ ActNode(Δ,k);

PUSH/RELABEL(i,Δ);

**end;**

**end**

**procedure** PUSH/RELABEL(i,Δ);

**begin**

**if** there is an admissible arc (i,j) **then**

push  := min { e(i),rij,Δ-e(j) }units of flow from i to j

**else**

d(i) := min { d(j) +1 : (i,j) ∈ A(i) and rij > 0 };

**end**

# EXAMPLE

2

3

4

2

3

5

1

e(2) = 0

e(3) = 5

e(1) = 3

2

3

4

2

3

5

1

After preprocessing;

*nodes* are labelled with distances,

*arcs* are labelled with residual capacities.

e(2) = 0

e(3) = 5

e(1) = 3

2

3

4

2

3

5

1

e(2) = 0

e(3) = 3

e(1) = 3

2

3

4

2

3

5

1

e(2) = 0

e(3) = 3

e(1) = 3

2

3

4

2

3

5

1

e(2) = 3

e(3) = 0

e(1) = 3

2

3

1

2

3

5

3

1

e(2) = 0

e(3) = 0

e(1) = 3

2

3

1

2

3

5

3

1

e(3) = 0

e(2) = 2

e(1) = 1

2

3

1

2

3

5

3

1

e(2) = 2

e(1) = 1

2

3

1

2

3

5

3

1

e(3) = 0

e(2) = 1

e(1) = 2

1

3

1

2

3

5

3

1

e(3) = 0

1

e(2) = 1

e(1) = 2

1

3

1

2

3

5

3

1

e(3) = 0

1

# COMPLEXITY ANALYSIS

**Lemma.**  *The algorithm satisfies the following two conditions:*

*1. Each non-saturating push from a node i to a node j sends at least* Δ */2 units of flow.*

*2. No excess ever exceeds* Δ *.*

**Proof:** Suppose that we perform a non-saturating push

on (i,j). Then e(i) > Δ/2 because i ∈ ActNode(Δ,d(i)).

Also e(j) ≤ Δ/2 because d(j) ∉ ActList(Δ). [Recall that

d(i) has the minimum distance label in ActList(Δ).] Thus

 = min { e(i),Δ-e(j) } ≥ Δ/2.

**Theorem.** *The excess-scaling algorithm performs*

*O(n2) non-saturating pushes per scaling iteration and*

*O(n2 log U) pushes in total.*

**PROOF.** Consider the potential function

 = i∈N e(i)d(i) /Δ.

(Think of d(i) as the height of a node, and e(i)/Δ as its weight measured in units of Δ. Then  is its gravita- tional potential.)

**Questions:**

* What is the effect on  of a saturating push? (Does it go up? Does it go down?)
* What is the effect on  of a non-saturating push?
* What is the total impact on  of the distance increases of a specific node i over all iterations?

**A MORE FORMAL PROOF OF THE TIME BOUND**

Let ΔD, R, SAT and NS be the steps during which an

excess dominator decrease (Δ := Δ/2), relabel, saturating push or non-saturating push occurs, respectively. Note that |ΔD| ≤ ⎡log 2 U⎤ +1, |R| ≤ 2n2 and |SAT| ≤ nm.

Let K be the last iteration. Each step k ∈ ΔD, R, SAT or NS, so

(K) - (0) =  k ∈ D (k)-(k –1) +  k ∈ R (k)-(k –1)

 k ∈ SAT (k)-(k –1) +  k ∈ NS (k)-(k -1)

Next we bound the relevant terms:

* (0) ≤ n2and(K) = 0
* if k ∈ ΔD, then (k) -(k -1) ≤ n2
* if k ∈ R, then (k) -(k -1) ≤ increase in d(i)
* if k ∈ SAT, then (k) -(k -1) ≤ 0
* if k ∈ NS, then (k) -(k -1) ≤ -1/2

Thus |NS|/2 ≤ n2 + 2n2 log U + 2n2 = O(n2 log U).

**ADDITIONAL COMMENTS ON EXCESS SCALING**

1. The algorithm can be modified (substantively) so that the running time is O(nm + n2 log1/2 U).
2. The algorithm can be modified (a little) so that *any* node i with large excess may be selected for pushing, but if we try to push to a node j that has large excess, then we put i on a stack and try to push from j.
3. The algorithm works quite well in practice. (But highest level pushing is a little better.)

**FURTHER RESULTS**

1. Using the Dynamic Tree data structure, the running time of the pre-flow push algorithm can be reduced to O(nm log(n2/m)), but the algorithm is not very practical.
2. In the case of unit capacities, the max-flow problem can be solved in O(n2/3 m) time. If at most one unit of flow can pass through each node, the running time is O(n1/2 m).
3. In a bipartite network with n = n1+ n2 nodes, almost all pre-flow push algorithms can replace n by n1 in the complexity.
4. In a planar network, the max-flow problem can be solved in O(n3/2 log n) time using planar separators.
5. The arc connectivity of a network (the number of arcs whose removal [strongly] disconnects the network) can be determined in O(nm) time.
6. Hao and Orlin (1994) show that the overall minimum capacity cut in a network can be determined in time O(nm log(n2/m)).