Solving Maximum Flow Problems in J

Michal Dobrogost *

1 Abstract

After a brief introduction to maximum flow problems, a solution is formulated in J. The existing family of push-relabel algorithms is refined in two ways to better suit the high-level nature of J. The focus is on correctness and pedagogy not outright execution efficiency. We conclude by tracing the execution over a specific maximum flow instance. A proficiency in the J programming language is assumed.

2 Introduction

Maximum flow problems were first formulated in 1954 by T.E. Harris to model Soviet railway traffic. The motivation was to find the maximum carrying capacity of the railway system between two cities. An additional motivation was the dual, minimum cut, problem which would allow one to determine how to disconnect a railway network most efficiently [1].

Solving maxflow has applications in bipartite matching, vertex cover, scheduling, baseball elimination, image segmentation and many others [6, 7, 4]. I was lead down the path of implementing a solver in J while exploring network robustness measures and the edge connectivity of a graph in particular. Network robustness is concerned with qualifying how reliable a given network is in the face of failing edges or nodes. Edge connectivity is one such measure and is defined as the minimum number of edges that need to be removed from a graph in order to disconnect it [5].

Maximum flow problems can be defined as follows: given a graph, two distinguished nodes, s and t, and a capacity constraint on each edge find the maximum amount of flow from s to t through the edges of the graph so that the flow through any edge does not exceed its capacity.

*michal dot dobrogost at gmail
www.cucave.net/papers/jmaxflow

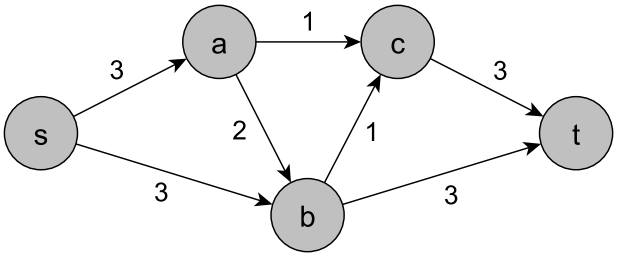


Figure 1: A maximum flow instance with edges labeled by their capacity. This graph will serve as a running example.

To be more precise, given a graph $G = (V, E)$, nodes $s, t \in V$ and a function $C \colon V \times V \to \mathbb{R}+$ find a function $F \colon V \times V \to \mathbb{R}$ such that:

$$\forall x, y \in V, F(x, y) \leq C(x, y) \qquad \text{(honour edge capacities)}$$

$$\forall x \in V \setminus \{s, t\}, \sum_{z \in V} F(z, x) - F(x, z) = 0 \qquad \text{(nodes conserve flow)}$$

$$\forall x, y \in V, F(x, y) = -F(y, x) \qquad \text{(skew symmetry)}$$

$$\arg\max_F |F| = \sum_{x \in V} F(s, x) = \sum_{x \in V} F(x, t) \qquad \text{(maximize flow amount)}$$

The instance in Figure 1 is essentially encapsulated in a single $|V| \times |V|$ matrix where the entry at C_{ij} represents the maximum amount of flow allowed along the edge from node i to node j.

```
   C;s;t
+---------+-+-+
|0 3 3 0 0|0|4|
|0 0 2 1 0| | |
|0 0 0 1 3| | |
|0 0 0 0 3| | |
|0 0 0 0 0| | |
+---------+-+-+
```

The first three properties and the summation of the fourth can be readily expressed in J. The maximization of the fourth, while satisfying the others, is the goal of this paper. The flow F is expressed as a matrix of the same shape as C.

```
edgeCapSatisified=.  *./ , F <: C
conserveFlowSatisified=. *./ 0= (-. (i.#F) e. s,t) # (+/ - +/"1) F
skewSymSatisified=. *./ 0= , F + |:F
flowAmount=. +/ {. F
```

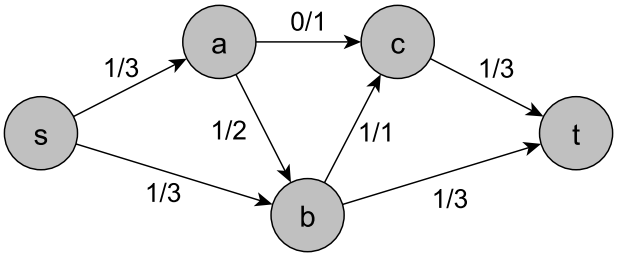


Figure 2: A flow is shown. Edges are labeled with their *flow/capacity.*

3 The Residual Graph

An implicit choice was made in that the domains of C and F were defined as $V \times V$ and not E. Suppose you have two nodes that have a single edge between them (for example a and b). Once flow is sent along that edge, from a to b, you can now send flow in the other direction, from b to a, by decreasing the amount of flow on the original edge. Essentially, any flow F will induce a residual graph in such a way. This is demonstrated in Figures 2 and 3.

```
   ] F=. 5 5 $ 0 1 1 0 0, 0 0 1 0 0, 0 0 0 1 1, 0 0 0 0 1, 0 0 0 0 0
0 1 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
```

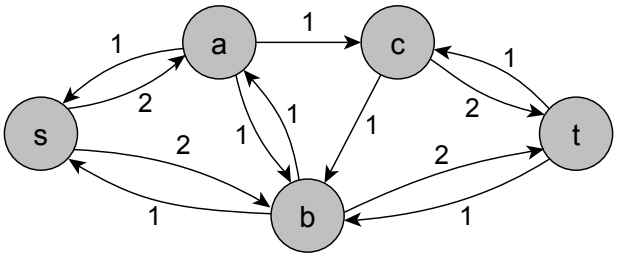


Figure 3: The residual graph induced by the flow in Figure 2. The edges are labeled with their residual capacity. This is the only time that the residual graph is made explicit.

```
   ] F =. (skewSym=. - |:) F
 0  1  1  0 0
_1  0  1  0 0
_1 _1  0  1 1
 0  0 _1  0 1
 0  0 _1 _1 0
   ] Residual=. C - F
0 2 2 0 0
1 0 1 1 0
1 1 0 0 2
0 0 1 0 2
0 0 1 1 0
```

During the execution of the algorithm it is useful to be able to update only overflowing nodes. A skew symmetric update is required for which we introduce the `updateSkewSym` function. In this example we push an additional unit of flow from nodes s and a, to b.

```
   updateSkewSym=. 4 : 0
    'f'=. x
    'vs fDelta'=. y
    f + skewSym fDelta vs} ($f) $ 0
)
   F ; F updateSkewSym (0,1);(2 5 $ 0 0 1 0 0)
+-------------+-------------+
| 0  1  1  0 0| 0  1  2  0 0|
|_1  0  1  0 0|_1  0  2  0 0|
|_1 _1  0  1 1|_2 _2  0  1 1|
| 0  0 _1  0 1| 0  0 _1  0 1|
| 0  0 _1 _1 0| 0  0 _1 _1 0|
+-------------+-------------+
```

4 Algorithm Prerequisites

There are many possible approaches to solving maxflow (an overview is presented in [2]). In practice push-relabel algorithms are the fastest [3] and an instance of this family is presented here.

Push relabel algorithms relax the flow conservation property by maintaining a preflow instead. A preflow has the requirement that, for internal nodes, more flow may come in than leave and if this is the case the node is referred to as overflowing.

$$\forall x \in V \setminus \{s,t\}, \sum_{z \in V} F(z,x) - F(x,z) \geq 0 \qquad \text{(preflow)}$$

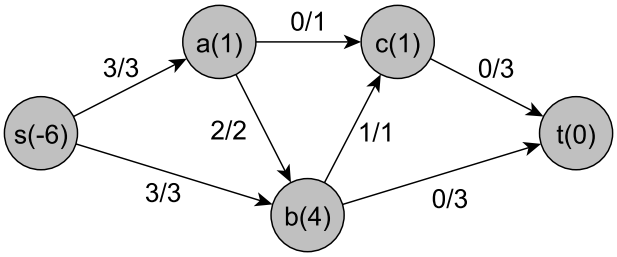


Figure 4: A preflow is shown. Edges are labeled with *flow/capacity*. Nodes are labeled with *name(excess)*.

The excess flow coming into a node may be computed using a difference of the flow matrix to its transpose. The definition of `vertexExcess` takes a skew symmetric flow as input and uses the rank changing operator `"1` instead of transpose.

```
   ] F=. 5 5 $ 0 3 3 0 0, 0 0 2 0 0, 0 0 0 1 0, 0 0 0 0 0, 0 0 0 0 0
0 3 3 0 0
0 0 2 0 0
0 0 0 1 0
0 0 0 0 0
0 0 0 0 0
   (vertexExcess=. 0.5 * (+/) - (+/"1)) skewSym F
_6 1 4 1 0
```

A height function $H\colon V \to \mathbb{N}_0$ labels the nodes and restricts where an overflowing node's flow may go next. The intuition is that flow may only go downhill from a node of higher height to a node of lower height.

$$\begin{aligned} H(s) &= |V| \\ H(t) &= 0 \\ (C-F)(u,v) > 0 \implies H(u) &\leq H(v)+1 \end{aligned}$$

A utility function is used for restricting a vector up to a given cumulative sum:

```
   untilAddTo=. 4 : 0
    (<.&y) &.: (+/\"1) x
)
   (i.2 7) ([ ; untilAddTo ; (+/"1 @: untilAddTo)) 7,21
+-----------------+-------------+----+
|0 1 2  3  4  5  6|0 1 2 3 1 0 0|7 21|
|7 8 9 10 11 12 13|7 8 6 0 0 0 0|    |
+-----------------+-------------+----+
```

5 Algorithm

The algorithm is initialized by pushing as much flow as possible from s to all adjacent nodes. All heights are initialized to zero except for $H(s) = |V|$. The return type here serves as the input and return type for the other verbs in this section, this can be seen as the current state of the execution.

```
   NB. Initialize push-relabel data structures
   NB. Returns (flow, residual capacities, vertex heights, vertex excess)
   init=. 3 : 0
    'C s t'=. y   NB. (original capacities, index of s, index of t)
    n=. #c

    NB. Initialize flow by sending out as much as possible from s.
    F=. (($C)$0) updateSkewSym s ; s{C

    NB. Initalize heights by setting s height to n, all others to 0.
    H=. (n,0) (s,t)} n $ 0

    NB. Put them all together
    F;(C - F);H;(vertexExcess F)
)
```

The remainder of the algorithm is a repetition of two operations: `relabel` and `push`. In `relabel` overflowing nodes are found whose height is insufficient to push out excess flow. The height is then increased to the maximum allowed by the definition of the height function which is governed by the residual neighbors of a node.

```
   relabel=. 4 : 0
    'C s t'=. x
    NB. (original capacities, index of s, index of t)
    'F R H E'=. y   NB. (flow, residual, vertex heights, vertex excess)

    NB. Active (overflowing) nodes
    a=. 0 0 (s,t)} E > 0
    aIx=. a # i. #c

    NB. Update height of active nodes by one more than their min neighbor
    H2=. (1+ (0 < a#R) (<./ @ #)"1 H) aIx} H

    F;R;H2;E
)
```

In `push` as much excess flow as possible is pushed out of an overflowing node. Only neighboring nodes to which there is a residual edge and whose height is one less than the overflowing node's

height may accept the excess flow. Because of the constraints on the height function, a height difference of one is the same as pushing excess flow to neighboring nodes of lower height. We use `untilAddTo` to push the excess flow to as many neighbors as possible - limited by the excess amount overflowing or by the residual capacity of the outgoing edges.

```
   push=. 4 : 0
    'C s t'=. x     NB. (original capacities, index of s, index of t)
    'F R H E'=. y   NB. (flow, residual, vertex heights, vertex excess)

    NB. Active (overflowing) nodes
    a=. 0 0 (s,t)} E > 0
    aIx=. a # i. #c

    NB. Selection of active vertex edges which have positive residual capacity
    edgesPosCapacity=. 0 < a#R

    NB. Selection of active vertex edges to nodes of height smaller by one
    edgesHDiff=. 1 = (a#H) (-"0 _) H

    NB. Residual capacity of active vertex edges which can take more flow
    edges=. (a#R) * edgesPosCapacity * edgesHDiff

    NB. Add flow along the first residual edges but limited by the excess amount
    flowToAdd=. edges untilAddTo (a#E)

    NB. Add the flow
    F2=. F updateSkewSym aIx;flowToAdd

    NB. Find qualifying edges for each active vertex
    F2;(C - F2);H;(vertexExcess F2)
)
```

Finally we provide a wrapper that initializes the data structures and then alternatively runs `relabel` and `push` operations until the data structures stop changing. Once that happens, progress has stopped and the algorithm is complete.

```
   maxflow=: 4 : 0
    'C'=. x
    's t'=. y
    'F R H E'=. ((C;s;t)&push @: ((C;s;t)&relabel))^:_ init (C;s;t)
    F
)
```

```
   c maxflow s;t
 0  2  3  0 0
_2  0  1  1 0
_3 _1  0  1 3
 0 _1 _1  0 2
 0  0 _3 _2 0
```

6 Correctness

The algorithm was tested against *solver-5* from the *First DIMACS Implementation Challenge*[1]. In all tested cases the resulting flow amount matched that of the other solver. A DIMACS format reader in J has also been implemented [2].

Full correctness proofs for the push-relabel family of algorithms can be found in [3]. Here we present the atomic description of `push` and `relabel` as given in the full proofs. The atomic operations apply to a single node or edge and can be applied so long as preconditions are met. Correctness is proved here by showing that extending these operations to work over all possible inputs they apply to is also correct.

`relabel(u)`
Precondition: u is overflowing, $(C - F)(u, v) > 0 \implies H(u) \leq H(v)$.
Effect: $H(u)$ is increased to one more than the minimum $H(v)$ where $(C - F)(u, v) > 0$.

`push(u,v)`
Precondition: u is overflowing, $(C - F)(u, v) > 0 \implies H(u) = H(v) + 1$.
Effect: push the minimum of $E(u)$ or $(C - F)(u, v)$ units of flow from u to v.

Relabeling multiple nodes

It is correct to apply `relabel` over all nodes to which it applies.

The major difference is that we do not check the second precondition, however, this has no effect except for additional runtime costs. By the definition of H we have that $(C - F)(u, v) > 0 \implies H(u) \leq H(v) + 1$. Since the height is updated to one more than the minimum we know that the height will not change at all (if $H(u) = H(v) - 1$) or will increase otherwise. So the operation may be applied and produce no change, but that is the same as not applying it at all because the precondition is not met. This is the same progress condition as the atomic `relabel` in [3] provides.

Since the atomic operation may be applied in any order, and only a single node's entry in H is updated by the atomic version, it is safe to apply the operation to all overflowing nodes at once and is equivalent to any ordering of sequential `relabel` operations.

[1] `http://dimacs.rutgers.edu/Challenges/`
[2] `www.cucave.net/papers/jmaxflow`

Pushing excess out of multiple nodes

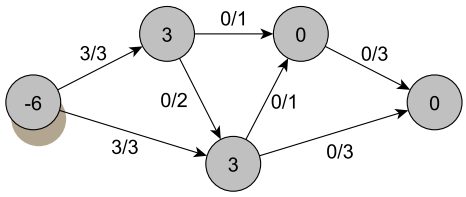
It is correct to apply `push` over all nodes to which it applies.

The preconditions are checked as listed for the scalar case. Pushing can remove and add edges in the residual graph as well as modify their capacities. The task is to ensure that pushing out of multiple nodes will not cause conflicts.

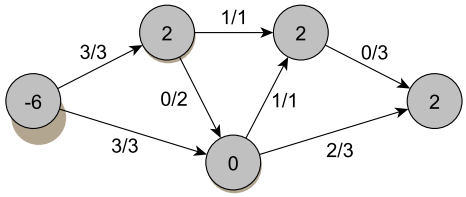
The only type of edge that is removed has its starting point in the overflowing node. The only type of edge that is added has its end point in the overflowing node. Both types of edges can be modified. Since a height difference of one is required only one node out of any pair of nodes connected in the residual graph will satisfy the preconditions to push flow to the receiving node. This is enough to ensure that each scalar operation affects a different set of edges than any other scalar operation that applies at the same time.

Pushing does not change the height of a node and so it does not affect its own preconditions that way. However, decreasing the excess or making a choice between multiple different targets of the push may affect what pushes are possible in the future. This is an arbitrary choice and so there exists a set of atomic push operations which will have the same effect as applying them all at once so long as the same choices of source and target nodes is made.

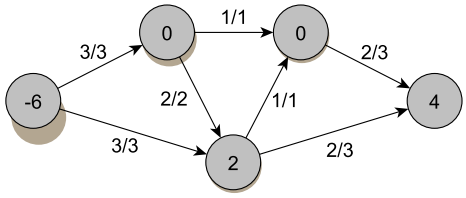
7 Example



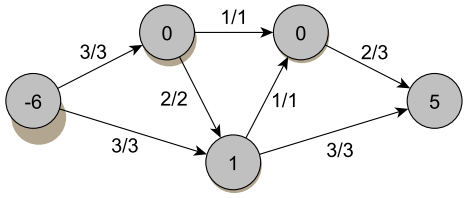
(a) State after initialization.



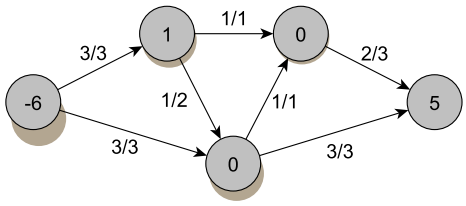
(b) Node B splits its excess onto both of its neighbors. This is because node c comes before t in the node ordering.



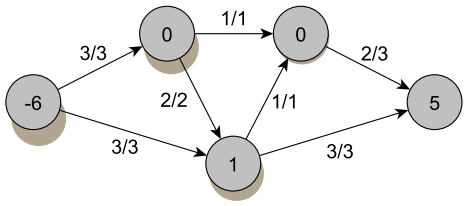
(c) Node A can push to B now that there is a height differential.



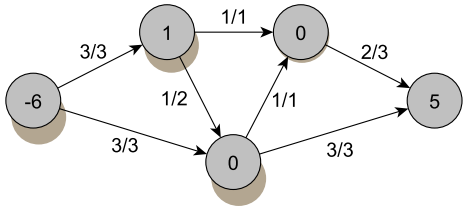
(d) Node B pushes remaining flow to T as limited by the edge capacity.



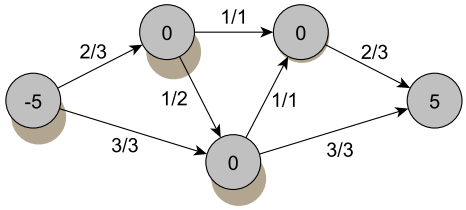
(e) Nodes A and B now swap excess until they achieve sufficient height.



(f) Nodes A and B swapping.



(g) Nodes A and B swapping.



(h) Node A has achieved sufficient height to push back to S.

Figure 1: Tracing a complete run. Edges are labeled with *flow/capacity* while nodes are labeled with *excess*. Vertex height is indicated by the length of a node's shadow. Each step consists of one relabel and one push operation.

8 Conclusion

An extension of push-relabel algorithms was made to apply the atomic operations to multiple nodes at once. This is a requirement for efficient implementation in J. An arbitrary ordering of operations results in a $\mathrm{O}\left(|V|^2|E|\right)$ time bound for push-relabel algorithms. Better bounds exist when the order of scalar operations is carefully controlled. This implementation is worse than the baseline bound because some individual steps were implemented for clarity:

- `updateSkewSym` builds an entire flow matrix instead of updating individual rows of the current matrix.
- `untilAddTo` has to run over a representation where all nodes are listed and not just possible residual neighbors. A more sparse graph representation would need to be chosen.
- `vertexExcess` recomputes the excess amount by summing all edges of each node, a more incremental approach would be more efficient.

Writing a solver directly in J has illuminated the details of push-relabel algorithms. It has also shaped the initial thinking - elucidating the idea of preflow before I dove into existing literature. Framing the algorithm in J naturally resulted in identifying potential areas of parallelism.

References

[1] Alexander Schrijver *On the history of the transportation and maximum flow problems.* Mathematical Programming, February 2002, Volume 91 Issue 3, pages 437-445.

[2] Andrew V. Goldberg, Robert E. Tarjan *A new approach to the maximum-flow problem.* Journal of the ACM Oct. 1988, Volume 35 Issue 4, pages 921-940.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms (*2^{nd} *edition)* MIT Press and McGraw-Hill 2001.

[4] Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin *Network Flows: Theory, Algorithms, and Applications* Prentice-Hall, Inc. Upper Saddle River, NJ, USA 1993.

[5] Wendy Ellens *Effective resistance and other graph measures for network robustness* PhD Thesis, Mathematisch Instituut at the University of Leiden 2011.

[6] Jeff Erickson *Algorithms Lecture 23 : Applications of Maximum Flow* `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/23-maxflowapps.pdf` Fall 2010.

[7] Nikhil Bansal *Maximum Flow Applications* `http://www.win.tue.nl/~nikhil/courses/2WO08/max-flow-applications-4up.pdf` Winter 2008.