On Implementing the Goldberg-Tarjan's Push-Relabel Max-Flow Algorithm

Garrenlus de Souza
Informatics Institute
Federal University Of Rio Grande do Sul - UFRGS
Porto Alegre, Brazil
gsouza@inf.ufrgs.br

I. Introduction

A. The Algorithm

The Goldberg-Tarjan's approach on solving the max-flow problem in this algorithm goes in another direction when one takes into account algorithms like Ford-Fulkerson's, that strives for optimality while keeping feasibility throughout the entire execution, by guaranteeing some of the preconditions to optimality over a solution which it strives to make feasible, at which point it becomes optimal. More than that, this algorithm does not work by augmenting paths between the source and the sink (the s and t nodes), but by striving to reach the sink in a way that does not keep excess flow dammed along the way from source to sink.

One important aspect is that just like Ford-Fulkerson's, it works over what is called a residual graph, an artifact derived from the original graph of interest. For this problem, each and every arc a connecting two vertices has a given capacity c_a. The residual R_G graph can be generated by adding all the vertices and arcs from G with the additional step of adding an arc a' for every arc a in G in the opposite direction such that it's capacity is the same (so the *C* function must be updated accordingly for the residual graph). This residual graph is such that for every pair {*a, a'*} the flow function $f_{A'}$ over the arcs of R_G, at any point of the algorithm, restricted to the following rule.

$$f_a + f_{a'} = c_a \in A \tag{1}$$

This way when flow is increased in one "direction" (therefore reducing the available capacity), it must be decreased from the capacity of the arc going the other way.

Goldberg and Tarjan's approach introduces a ranking function (we'll call it d_v) that can be found by many names in the literature, height, potential, distance and maybe others. This function, defined over the nodes with its range in $[0, 2 * |V|]$, is employed among other things to define how flow can be transmitted from one node to the next, and as could be concluded through the classes over the algorithm, helps greatly when it comes to ensuring some invariants as well as deriving upper bound limits for run-time complexity.

The purpose of this writing is not of getting into all the details and arguments that back up the correctness and optimality of the algorithm, although some of the details entailed by these arguments may come in handy when presenting the implementation of the algorithm as well as its inherent rationale.

The reference pseudo-code for the algorithm is depicted next. Notice that little to no detail is given about how to actually track the active vertices, let alone choose the one with the greatest value for the function d_v.

Algorithm 1 Goldberg & Tarjan's push-relabel

Require: a graph $G = (V, A, \mathcal{C})$ such that c_a denotes the capacity of arc $a \in A$ and a pair of vertices $s, t \in V$ such that an arbitrary amount of flow units can be spawned and absorbed from s, by t, respectively.
Ensure: The maximum flow from s to t.

```
1:  d_s ← |V|; d_v ← 0; ∀v ∈ V except s
2:  f_a ← c_a, ∀a ∈ N+(s) else f_a ← 0
3:  while there are active nodes do
4:      v ← the active node with the greatest d_v
5:      while while v is active do
6:          if ∃{v, w} ∈ R_G with d_w = d_v − 1 then
7:              push(v, w)
8:          else
9:              relabel(v)
10:         end if
11:     end while
12: end while
13: return f
```

The $push$ and $relabel$ operations are key for this algorithm.

push: This operation reduces the capacity of the given arc v, w by the minimum between the remaining capacity of the edge and the excess flow available at v and increases the arc's counterpart (added when the residual graph was created). So if we call e_v the excess flow at v, then:

$$\delta = min(c_{\{v,w\}}, e_v) \tag{2}$$

This amount δ is then used to update the excess flow at both v and w as well as the arcs themselves.

$$e_v = e_v - \delta \tag{3}$$

$$c_{\{v,w\}} = c_{\{v,w\}} - \delta \tag{4}$$

relabel: When applied over a vertex v this procedure increases the value of d_v by one, effectively allowing for nodes in its neighborhood to receive flow in the next iteration provided the fact that there are arcs from v with available flow capacity.

$$d_v = d_v + 1 \tag{5}$$

Though Equation 3 is depicted to reflect the next state of e_v it must be stated that e_w is also updated, though by adding δ. Something analogous happens with Equation 4, as the capacity for the arc counterpart must be adjusted to reflect the increase of capacity after the push operation is completed. It is also interesting to highlight the fact that the flow f is being implicitly interpreted as the "current load" of an edge, or the absence of capacity (remember Equations 1 and 4).

Notice that in Algorithm 1, step 2, the respective arcs are updated according to the equations for the $push$ operation except for the excess flow at s if necessary as its value is not of the interest of the author in the present work.

II. Methodology

A. Environment

For the current study, the applied methodology consisted primarily into the analysis of running time and key data-structure related statistics made over a reference implementation written in C++. The operating system is Ubuntu 20.04 LTS, running over a 32 GB RAM, 11th Gen Intel Core i7-1185G7 at 3.00GHz clock-speed system. All the code was compiled using g++ 9.4.0 (built for this very platform) through a CMake enabled build pipeline set to *"Release"* mode. The graphs used as input are generated using [1] and the author's implementation was tested against Boost's [2] *push_relabel_max_flow* procedure in order to address inconsistencies throughout the development process. The timing fixtures are native to C++ in order to reduce noise that could be possibly introduced by timing the process through external tools for example. This alone does not guarantee the absence of external interference, but seemed to be enough for the purposes of this work. All the data structures employed come from the *C++ STL* unless otherwise stated.

The employed graphs are in DIMACS [3] format and its generation parameters are provided in the employed code, including the seeds used to instantiate random number generators, this was done to improve the replicability of the results, as well as to favor the consistency in execution of the tests by minimizing the amount of human interference. Some automation scripts were written (*bash* and *Python 3*) to orchestrate the statistics generation, which in this case consists of "csv" files.

B. Implementation

One could point out that the algorithm presented so far does not specify how to implement any of the operations. This in turn allow us to take this freedom and explore some of the possible solutions. Though disposing of some leeway on the implementation of said steps of the algorithm, one must respect the run-time upper bounds for each of them. These limits are said to be $O(1)$ for both *push* and *relabel* and $O(n)$ [4] for the selection of the active node v with the highest value for d_v. Two implementations are going to be evaluated [5], they differ by the implementation of step 4 in the algorithm by employing a linear search and a special data-structure called HL to do so.

In order to address the implementation of the algorithm, and through a healthy amount of experimenting, some object oriented abstractions were designed to model the graph in memory, hopefully allowing for a good strike between space and time trade-offs. The residual graph is generated at parsing time so no distinction between synthetic and base edges must be made at run-time, further simplifying the algorithm.

1) Vertices: A vertex/node holds information about its outgoing edges, the amount of flow excess and a pointer to the next arc through which it can push flow outwards (given it has some in the first place).

2) Edges: Arcs are stored as edges and hold information about their destination (as an index) and the current capacity. Another key information held by edges is the position its synthetic counterpart is stored in the arc collection of the node it points to. This is key on implementing a *push* in $O(1)$.

3) Operations: Although not strictly necessary for the current discussion, below are some of the details about the key procedures involved in the presented implementation. The *push* and *relabel* operations are implemented quite literally under the premises of C++ so we'll not be discussing them in detail. The test for node "activity" is quite straightforward and its concerned exclusively with the amount of excess at the node of interest.

$$active(v) = e_v > 0 \tag{6}$$

Another detail left out in Algorithm I is how to choose the next edge one can use to perform a push. By taking note of the last exhausted arc from a given node v we can potentially avoid checking all its arcs every time, this way edges that come before said arc can be classified (with decreasing precedence) into:

- Edges to nodes w with d_w that break the transition requirement for a push to happen as stated in the 6th step of Algotithm I.
- They have no available capacity, i.e. $f_{\{v,w\}} - c_{\{v,w\}} = 0$.

That way the selection can progress more efficiently. In the cases in which all the eligible arcs were exhausted and there's excess flow remaining at v, this pointer is reset to the first arc in v's collection of edges as the information we have is bound to the value of d_v and therefore must be discarded after a relabel.

C. Maximizing the active node selection

1) Linear Search: This approach is quite straightforward and just sweeps a static list of the nodes v testing if 1) v is active and 2) if the d_v is the greatest found until that point of the sweep.

2) HL: The implemented HL structure consists in an array of $|V|$ buckets, implemented as stacks. Indexed from 0 to $|V|-1$, each bucket holds values that map to its respective index in the structure. In the case of this work, nodes v were mapped to buckets according to the function d_v. In this implementation only the active nodes are stored into the structure, and only to a certain value of d since nodes v that reach a value $d_v = |V|$ would eventually discharge their excess flow to the source, not contributing to the solution of the problem (i.e. increasing the amount of flow that is received at t). In this case a pointer to the non-empty bucket with the highest value of d is kept so it can be more efficiently accessed. This pointer must be updated as elements are removed or inserted into the structure.

D. Input

The graphs later fed to the implementations of the algorithm were generated through the code provided by the course instructor [1] and are subdivided in ten different classes.

TABLE I
GRAPH CLASSES

Class	Parameters	n	m
Mesh	r,c	rc+2	3r(c-1)
Random level	r,c	rc+2	3r(c-1)
Random 2-level	r,c	rc+2	3r(c-1)
Matching	n,d	2n+2	n(d+2)
Square Mesh	d,D	d*d+2	(d-1)dD+2d
BasicLine	n,m,D	nm+2	nmD+2m
ExpLine	n,m,D	nm+2	nmD+2m
DExpLine	n,m,D	nm+2	nmD+2m
DinicBad	n	n	2n-3
GoldBad	n	3n+3	4n+1

For the first three classes, the values for r and c were drawn from a set $v = \{16, 32, 64, 128, 256\}$. For the Matching set the parameter n was drawn also from v but d was taken from $w = \{2, 4, 8, 16\}$. For the Square Mesh case the values were drawn from v with the exception that $D < d$. From BasicLine to DiExpLine n and m were drawn from v and D was in the w set. The last two classes' parameters were drawn from a new set, $u = \{512, 1024, 2048, 4096, 8192\}$. The capacity function was limited to the range $[0, 1000]$. The choice for this values was made on the premises of generating test cases that would not (*a priori*) take too much time to run a few times per query on the max-flow from some arbitrary, randomly selected pair of nodes. It turns out that when it came to actually executing the whole pipeline, the graph generation step was surprisingly quick, taking less than a minute to complete. On the other hand the execution of the algorithm per se did not seemed to follow, with test-cases that easily surpassed two minutes. For the purposes of this work running a test-case more than once was a requirement, so a decision was made in order to try and tame this time management issue. After some testing it was concluded that graphs under 200KB in size rendered the best trade-off between the time available to analyze the results (from both implementations) and the range of different classes available.

E. The setup

Each test-case consisted in a pair of randomly selected pairs, a source and a sink. Each pair was then fed into a max-flow query five times. Each graph was tested against eleven different pairs. The results presented next depict values aggregated in the y axis the a $median$ function. The amount of results generated was overwhelming for the amount of time available, so it became a problem trying to depict the behavior of every single instance generated. Instead some of the most representative or curiously counter intuitive results are going to be presented, although the reader is for sure well equipped to visit the references section and try by himself executing the whole pipeline, end-to-end, exactly as done for the current work.

III. RESULTS

A. Pushes & Relabels

The first three classes behavior when it came to the number of pushes and relabels was quite curious as a result of how the relabel function was implemented.

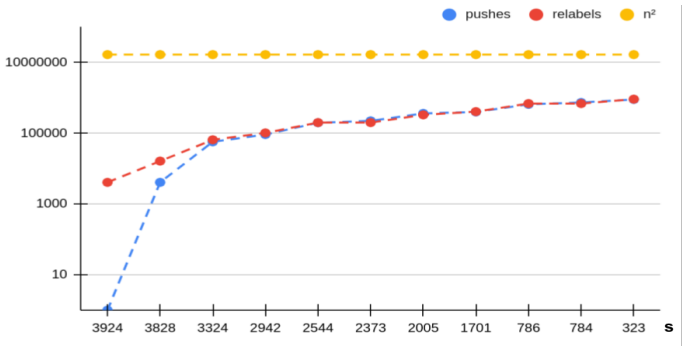

Fig. 1. The number of pushes and relabels for an instance from Mesh with $n = 256$ and $r = 16$ under the execution of the list-based implementation.

From the first three classes, only the $n = 256$, $r = 16$ instances were taken into account as the goal was to assess how the statistics would relate to n at its limit (in the current experimental setting). In 2 we can see that the number of operations varies considerably depending on the source node one chooses. One interesting details is that in some cases the amount of pushes is actually smaller than the amount of relabels, a run-time artifact observed in cases were the algorithm inevitably hit dead ends and is forced to perform relabels (implemented under a 1-step a time policy) repeatedly until the flow can be sent back. This situation could be resolved by introducing the value for d_w into the calculation of how much an active node v would need to increase is value in d in order to perform the next feasible $push$ to another node w. When it comes to the results for the HL implementation, this distortion between pushes and relabels is even more pronounced, though both end up converging as the number of operations grew larger for classes 1, 2 and 3.

The $Matching$ class of graphs turned out to provide a curious behavior. Although the $s-t$ node pairs were not kept

between the analysis of each implementation, the numbers for pushes were quite close to each other. One other interesting aspect is the stability of such values throughout the experiments (Fig. 2, something that to the author's knowledge can only stems from the inherent regularity of the instances drawn from such class.

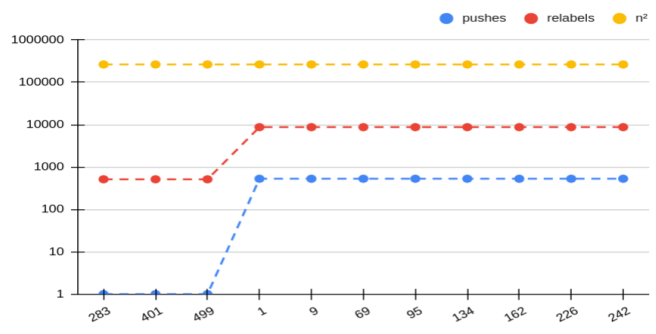

Fig. 2. The number of pushes and relabels for an instance from Matching with $n = 256$ and $d = 16$ under the execution of the HL-based implementation.

The $SquareMesh$ class instance analyzed had its parameters adjusted as so the graph file size could fit into the $200KB$ limit previously mentioned. Instead of using the value 16 for the D parameter, the value 15 was employed instead. This was the first time in which the amount of pushes and relabels was observed to diverge between the implementations. As expected the HL based approach rendered itself considerably more efficient as one can see in Fig. 4.

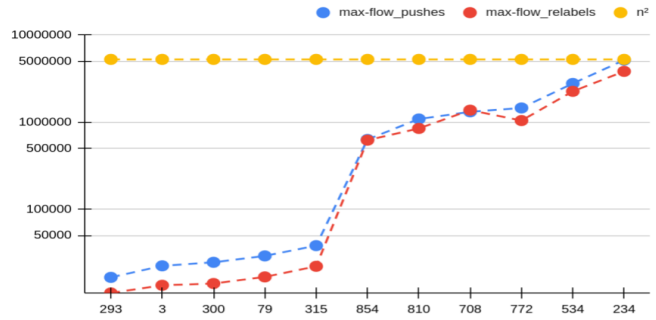

Fig. 3. The number of pushes and relabels for an instance from Square Mesh with $n = 256$ and $D = 15$ under the execution of the $list$-based implementation.

Notice the number of relabels for the list-based approach ends up close to the proposed estimate for worst case scenario while on the other hand the HL-based's numbers stay reasonably distant from this upper bound. Another interesting point could be made about the relationship between pushes and relabels as while using a list the amount of pushes is consistently higher than the amount of relabels, something not quite present with the HL-based approach.

The observed overall behavior of the Basic Line, ExpLine and DExpLine classes was identical and can be summarized by push and relabel counts quite close to each other, as well

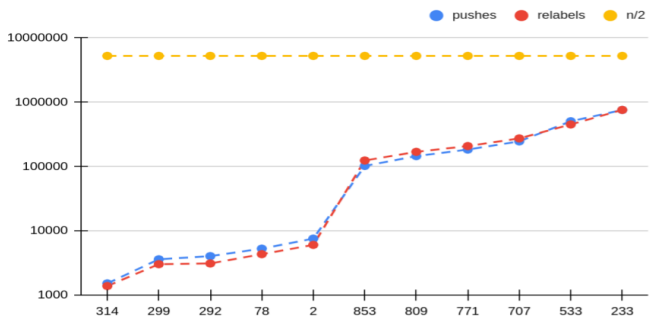

Fig. 4. The number of pushes and relabels for an instance from Square Mesh with $n = 256$ and $D = 15$ under the execution of the HL-based implementation.

as reasonably close to the n^2 upper bound as can be seen in Fig. 5. For this comparison, the biggest, same generation parameters were taken into account for all the instances mentioned.

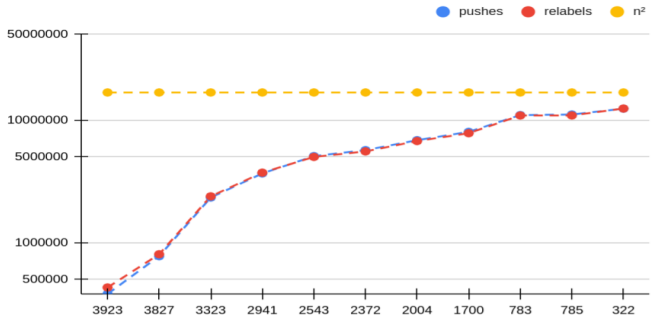

Fig. 5. The number of pushes and relabels for an instance from Basic Line with $n = 256$ and $m = 16$ and $D = 2$ under the execution of the HL-based implementation.

For the instances of DinicBad there were no representative divergences between the two implementations in terms of pushes and relabels with curves that quite resemble each other as can be seen in Fig. 6. The GoldBad class instances followed suit and did not present any kind of divergence between the two implementations. The ratio between pushes and relabels ($pushes/relabels$) varied between 99% and 100% for the GoldBad instances evaluated while in the DinicBad case this value ranged from 10% all the way to 110%.

B. Lapses

A lapse is defined in the current work as a non-definitive (it advances the search state but is not conclusive) step when it comes to selecting the next node to be subject of a *discharge* (steps 3 to 11 in Algorithm I-A). This non-productive step although a little different between the two implementations, works by the same principle at each one of them.

In the list-based implementation the selection of the next node to receive a discharge was made purely by scanning the nodes in a linear manner in the same container they were being held to represent the graph as a whole. This way for each

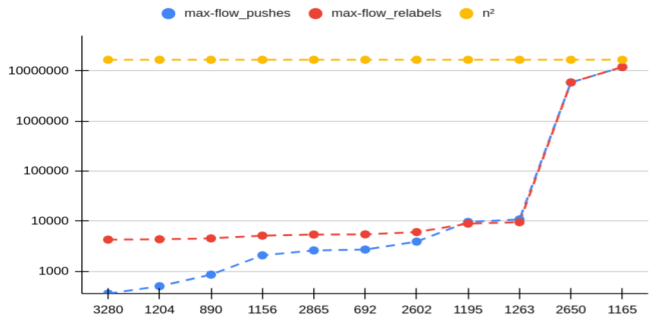

Fig. 6. The number of pushes and relabels for an instance from DinicBad with $n = 4096$ under the execution of the list-based implementation.

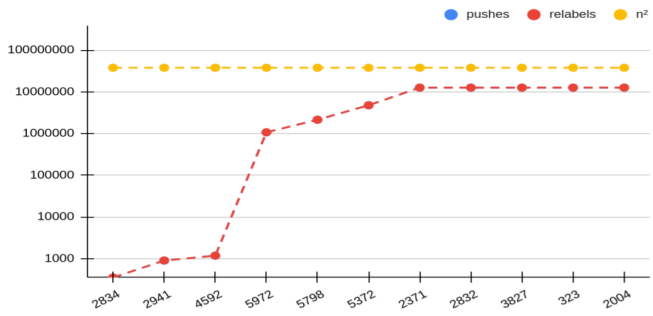

Fig. 7. The number of pushes and relabels for an instance from GoldBad with $n = 4096$ under the execution of the HL-based implementation.

discharge the algorithm is guaranteed to find the next candidate node in at most $O(n)$ between two *relabel* operations, thus under the worst-case upper bound of $O(n^3)$. In this scenario the amount of lapses would be the amount of iterations or queries to this node container, each query performed in order to assess if the value d_v at hand is actually greater than the one found to this point in the search. The evaluated graphs did not in any case surpass the n^3 estimated boundary for the number of pushes, so as to follow the trends observed in the previous section these numbers did not even got relatively close to that upper limit.

In the HL structure, lapses are the steps required to find the next non-empty bucket with the greatest value for d. This happens every time there is a gap between non-empty buckets, which in turn can be caused by nodes that need to return flow throughout the execution of the algorithm. The use of this structure allows for some clever optimizations. One of them allows for speed ups when it comes to the problem of just getting to know the max-flow that ends up at the sink, ignoring entirely the flow values for the paths that deviate from the ones that start at s and end at t. This implementation guarantees that the next *discharge* candidate can be found in at most n lapses which in turn also stands under the $O(n^3)$ upper bound. One interesting aspect is that over the explored graph instances the total amount of lapses ended up being amortized by the number of insertions made to the HL structure (this container was used only to hold nodes v with $e_v > 0$). As the total number of *pushes* equals the number of *pop* operations, **the cost of *pop* was observed to be $O(1)$ amortized**. The distribution of lapses varied consistently between different classes of graph instances and never surpassed the number of *push* operations.

C. Runtime

When it comes to run time, the *HL* implementation showed promise for better performance. As a case study, the $r = \{16, 32, 64, 128, 256\}, c = 16$ from the *Mesh* class was taken. The times are depicted in Fig 8. One can see that the improvements range from 10x to 50x.

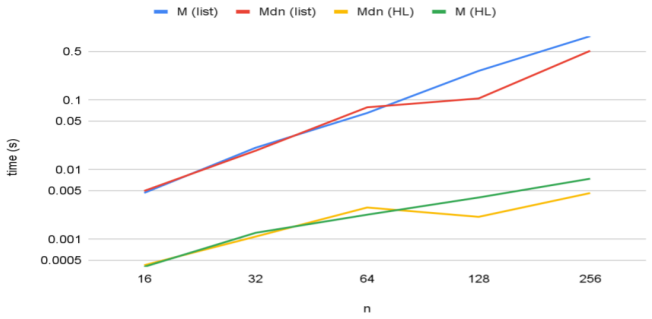

Fig. 8. The average and median of the runtime found for $r = \{16, 32, 64, 128, 256\}, c = 16$ under the Mesh class.

IV. Discussion

It was perceived that different classes produced variations in overall behavior when it comes to the number of pushes and relabels. Another relevant aspect is the difference on the run time measurements between the different implementations, something that positioned the HL based implementation in a way better situation with speed ups that reached 50x times for the presented test-case.

V. Conclusion

Dealing with the unknown can be hard. The biggest challenges in this work were concentrated around the idea of exploring how different input and data-structures could end up affecting the overall performance of the algorithm. The volume of data generated for the test cases proved overwhelming for the purposes of the work [6] and ended up cluttering the whole experimental environment as well as adding unexpected delays to the schedule. Although the amortized constant time of the operations made over the HL structure were not formally demonstrated, the empirical results ended up establishing some sort of confidence on the use of such resort when it comes to extracting performance.

References

[1] "New washington graph generator," https://www.inf.ufrgs.br/~mrpritt/lib/exe/fetch.php?tok=f9b838&media=http\%3A\%2F\%2Fwww.inf.ufrgs.br\%2F~mrpritt\%2Faa\%2Fnew_washington_test_dir.tar.gz, Accessed: 2023-2-2.

[2] “Dimacs implementation challenges,” https://www.boost.org/doc/libs/1_43_0/libs/graph/doc/push_relabel_max_flow.html, Accessed: 2023-2-2.
[3] “Dimacs implementation challenges,” http://archive.dimacs.rutgers.edu/Challenges/, Accessed: 2023-2-2.
[4] “Inf05010 – algoritmos avançados, notas de aula,” https://www.inf.ufrgs.br/~mrpritt/lib/exe/fetch.php?media=inf05016:notas-33a0c8.pdf, Accessed: 2023-2-2.
[5] “Github repository,” https://github.com/GarrenSouza/inf05016-push_relabel, Accessed: 2023-2-2.
[6] “Analysis data,” https://drive.google.com/drive/folders/1qdFHpc4fP-t9iO2nH-7I7gnVvYQvKhms?usp=share_link, Accessed: 2023-2-2.