Goldberg's algorithm implementation and benchmarking in Python

Davide Bergamaschi

2018

# Abstract

We hereby analyze the features of our implementation of Goldberg's push-relabel algorithm and, utilizing benchmark results, show that the performance of such implementation is essentially consistent with the theoretical results from the literature.

# The implementation

## Technology

The algorithm was implemented in Python (version 3), relying on the following external modules:

- graph_tool, for graph representation and manipulation; this module relies directly on the Boost Graph Library
- memory_profiler, to collect memory usage statistics

## Algorithm

This implementation is meant to be as close as possible to Goldberg's general push-relabel algorithm, without employing elaborate optimization. It uses a simple stack structure to keep track of active nodes. In the following section we show and discuss its most significant parts.

### Push / relabel routines

The implementation here is very straightforward, and naturally leads to a complexity of $O(1)$ for pushes and of $O(E)$ for relabel operations (since a relabel might result in visiting all edges in the worst case).

```python
def push(edge, excess, capacity, preflow, reverse_edges):
    origin = edge.source()
    dest = edge.target()

    delta = min(excess[origin], capacity[edge] - preflow[edge])

    preflow[edge] = preflow[edge] + delta
    rev = reverse_edges[edge]
    preflow[rev] = preflow[rev] - delta

    excess[origin] = excess[origin] - delta
    excess[dest] = excess[dest] + delta

def relabel(vertex, distance, capacity, preflow):
    suitable_edges = filter(
        lambda edge : capacity[edge] - preflow[edge] > 0, vertex.out_edges()
    )

    dists = map(
        lambda edge : distance[edge.target()], suitable_edges
    )

    new_d = min(dists) + 1

    distance[vertex] = new_d
```

## Initialization

The algorithm begins by adding reverse arcs to the graph and by initializing the necessary structures to store run variables (temporary preflow, distance and excess), as well as a map to easily access the reverse of an edge in constant time. Then a simple stack is created to store active nodes, together with another map to enhance the stack with fast vertex membership test.

The only non constant-time operation here is the edge creation, which takes $O(E)$ time. Moreover, due to edge iterator issues, a further list of vertex couples has to be kept. Said list has length $E$. This could be avoided by employing a lazy initialization mechanism, but this direction has not been taken for the sake of code clarity.

```python
def stack_push_relabel(graph, source, target, capacity):
    # Initializing data maps
    reverse_edges, preflow, distance, excess = helper.create_maps(graph, capacity)

    # Initializing stack to keep active node
    actives = structure.Stack()

    # Initializing active map
    is_active = graph.new_vertex_property("bool")

    #    ...    #

def _create_residual_edges(graph, capacity):
    #    ...    #

    newlist = []
    for edge in graph.edges():
        newlist.append((edge, edge.target(), edge.source()))

    for entry in newlist:
```

```
        new = graph.add_edge(entry[1], entry[2])
        capacity[new] = 0
        reverse_edges[entry[0]] = new
        reverse_edges[new] = entry[0]

    return reverse_edges
```

The algorithm then proceeds with the usual push-relabel initialization, with the only precaution of adding activated nodes to the active list during the initial saturating pushes. Vertices first and then edges are iterated, hence time complexity will be $O\left(V+E\right)$.

```
#     ...     #
# continues from push_relabel function

# Initializing distance, excess and active property
for v in graph.vertices():
    distance[v] = 0
    excess[v] = 0
    is_active[v] = False
distance[source] = graph.num_vertices()

# Initializing preflow
for edge in graph.edges():
    preflow[edge] = 0

# Saturate edges outgoing from source
for s_out in source.out_edges():
    cap = capacity[s_out]

    # If capacity is 0, nothing to push
    # Probably an added residual arc
    # Skip cycle just for optimization
    if cap == 0:
        continue

    preflow[s_out] = cap
    preflow[reverse_edges[s_out]] = - cap

    excess[s_out.target()] = excess[s_out.target()] + cap
    excess[source] = excess[source] - cap

    # Since node has become active, add it to active stack
    active = s_out.target()
    if active != target and is_active[active] == False:
        actives.push(active)
        is_active[active] = True
```

## Main loop

This is were the sequence of push and relabel actions that give the algorithm its name takes place.

At each step of the cycle, the last activated node is popped from the stack. All possible pushes from the selected node are performed, adding any target node that becomes active to the active set. Lastly the selected node is relabeled if it is still active.

To analyze complexity, we observe that each time a vertex is selected for the main cycle, its outgoing edge list is possibly scanned twice, one time for pushing and one for relabeling. Since the maximum number of relabels for a vertex is $2V-1$, the maximum number of scans for each vertex is $4V-1$ (one for each relabeling, one for the pushes before each relabeling and one for the pushes after the last relabeling). Hence the global time spent to process each node $v$ is $O(V \times deg_{out}(v)) + O(1) \times n_{pushes}(v)$. Summing over all vertices, and recalling that the global number of pushes from all nodes is $O\left(V^2E\right)$, we find that the global execution time of the main loop is $O\left(V^2E\right)$ as well, which coincides with the global time complexity of our implementation, since it dominates the initialization complexity.

```
#     ...     #
# continues from push_relabel function

cur_v = actives.pop()
while cur_v:
    is_active[cur_v] = False

    # Look for admissible edges
    for out_e in cur_v.out_edges():
        # If admissible, push flow
        if (distance[cur_v] > distance[out_e.target()]
            and capacity[out_e] - preflow[out_e] > 0):
            helper.push(out_e, excess, capacity, preflow, reverse_edges)

            active = out_e.target()
            if active != source and active != target and is_active[active] == False:
                actives.push(active)
                is_active[active] = True

            # Node not active anymore
            if (excess[cur_v] <= 0):
                break

    # No more admissible edges
    # Relabel if still active
    if (excess[cur_v] > 0):
        helper.relabel(cur_v, distance, capacity, preflow)
        if not is_active[cur_v]:
            actives.push(cur_v)
            is_active[cur_v] = True

    cur_v = actives.pop()

return preflow
```

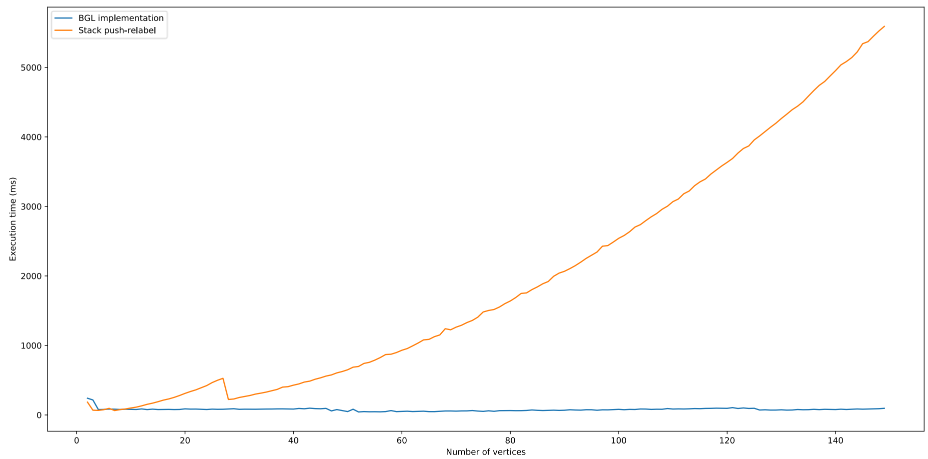
# Benchmarks

The performance of our implementation has been empirically measured both in terms of running time and memory consumption. In the following section, benchmark results are displayed and compared with those obtained by running the same tests on the Boost Graph Library implementation of the push-relabel algorithm.

All tests have been executed on a Celeron-based machine running a 64-bit GNU/Linux system.

# Execution time

The observed results empirically confirm that the execution time of our implementation is polynomial and grows superlinearly with the number of vertices and linearly with the number of edges.



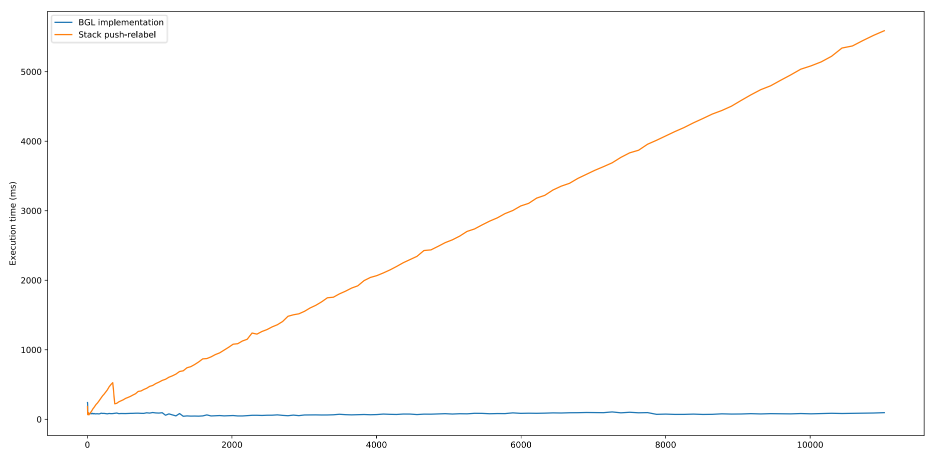




Figure 1: Execution time plots

# Memory usage

The empirical results show that memory usage indeed grows linearly with the number of edges, since our implementation always creates $E$ reverse arcs to perform its computation.

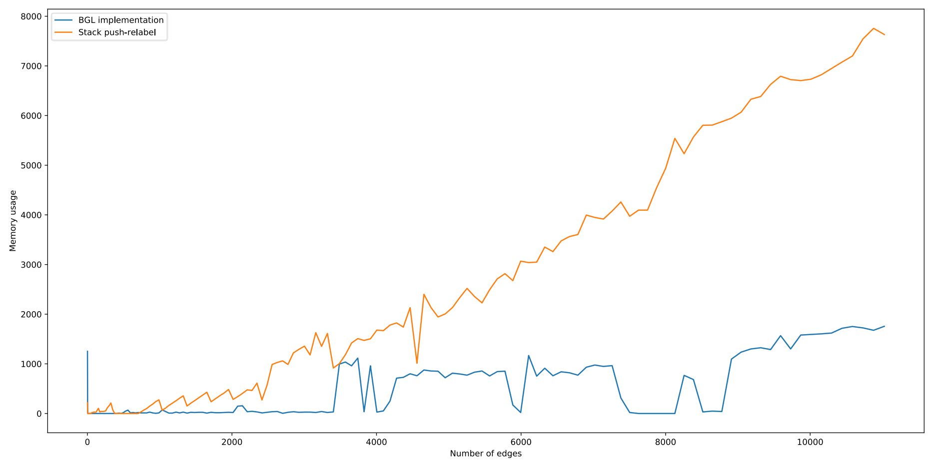



Figure 2: Memory usage plots