Goldberg's algorithm implementation and benchmarking in Python

Davide Bergamaschi

Politecnico di Milano

2018

Index

Implementation analysis

Push operation

```
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
```

Push operation

```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
```

Compute Δ

Push operation

```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
```

Compute Δ

Update preflow

Push operation

```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
```

Compute Δ

Update preflow

Update excess

Push operation

```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
```

Compute Δ

Update preflow

Update excess

Time complexity: $O(1)$

Relabel operation

```
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
```

Relabel operation

```
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
```

Find non-saturated edges

Relabel operation

```
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
```

Find non-saturated edges

Find minimum distance among corresponding vertices and relabel accordingly

Relabel operation

```
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
```

Find non-saturated edges

Find minimum distance among corresponding vertices and relabel accordingly

May scan all edges
Time complexity: $O(E)$

Initialization

```
def _create_residual_edges(graph, capacity):
    reverse_edges = graph.new_edge_property("object")

    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

def stack_push_relabel(graph, source, target, capacity):
    reverse_edges, preflow, distance, excess = helper.create_maps(graph, capacity)

    actives = structure.Stack()

    #    continues...    #
```

Initialization

```
def _create_residual_edges(graph, capacity):
    reverse_edges = graph.new_edge_property("object")

    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

def stack_push_relabel(graph, source, target, capacity):
    reverse_edges, preflow, distance, excess = helper.create_maps(graph, capacity)

    actives = structure.Stack()

    #    continues ...    #
```

Add reverse edges to graph

Initialization

```
def _create_residual_edges(graph, capacity):
    reverse_edges = graph.new_edge_property("object")

    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

def stack_push_relabel(graph, source, target, capacity):
    reverse_edges, preflow, distance, excess = helper.create_maps(graph, capacity)

    actives = structure.Stack()

    #    continues...    #
```

Add reverse edges to graph

Create run maps (calls above func.)

Initialize active stack

Initialization

```python
def _create_residual_edges(graph, capacity):
    reverse_edges = graph.new_edge_property("object")

    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

def stack_push_relabel(graph, source, target, capacity):
    reverse_edges, preflow, distance, excess = helper.create_maps(graph, capacity)

    actives = structure.Stack()

   #    continues...    #
```

Add reverse edges to graph

Create run maps (calls above func.)

Initialize active stack

All edges are scanned, E reverse edges are created along with maps
Time complexity: $O(E)$, space complexity: $O(V+E)$

Initialization

```
# ...continues from push_relabel function

for v in graph.vertices():
    distance[v] = 0
    excess[v] = 0
    is_active[v] = False
distance[source] = graph.num_vertices()

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

for s_out in source.out_edges():
    cap = capacity[s_out]

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

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

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

#    continues...    #
```

Initialization

```
# ...continues from push_relabel function

for v in graph.vertices():
    distance[v] = 0
    excess[v] = 0
    is_active[v] = False
distance[source] = graph.num_vertices()

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

for s_out in source.out_edges():
    cap = capacity[s_out]

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

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

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

#    continues...    #
```

Initialize node and edge values

Distance of source is set to V

Initialization

```
# ...continues from push_relabel function

for v in graph.vertices():
    distance[v] = 0
    excess[v] = 0
    is_active[v] = False
distance[source] = graph.num_vertices()

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

for s_out in source.out_edges():
    cap = capacity[s_out]

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

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

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

#    continues...    #
```

Initialize node and edge values
Distance of source is set to V

Push flow to source neighbors

Initialization

```
# ...continues from push_relabel function

for v in graph.vertices():
    distance[v] = 0
    excess[v] = 0
    is_active[v] = False
distance[source] = graph.num_vertices()

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

for s_out in source.out_edges():
    cap = capacity[s_out]

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

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

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

#    continues...    #
```

Initialize node and edge values
Distance of source is set to V

Push flow to source neighbors

Add to stack if not present

Initialization

```
# ...continues from push_relabel function

for v in graph.vertices():
    distance[v] = 0
    excess[v] = 0
    is_active[v] = False
distance[source] = graph.num_vertices()

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

for s_out in source.out_edges():
    cap = capacity[s_out]

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

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

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

#   continues...   #
```

Initialize node and edge values
Distance of source is set to V

Push flow to source neighbors

Add to stack if not present

First edges, then vertices are iterated
Time complexity: $O(V + E)$

Main loop

```
# ...continues from push_relabel function

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

    for out_e in cur_v.out_edges():
        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

            if (excess[cur_v] <= 0):
                break

    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
```

Main loop

```
# ...continues from push_relabel function

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

    for out_e in cur_v.out_edges():
        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

            if (excess[cur_v] <= 0):
                break

    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
```

At each step a node is popped from the stack

Main loop

```
# ...continues from push_relabel function

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

    for out_e in cur_v.out_edges():
        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

            if (excess[cur_v] <= 0):
                break

    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
```

At each step a node is popped from the stack

All possible pushes from it are performed...

Main loop

```
# ...continues from push_relabel function

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

    for out_e in cur_v.out_edges():
        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

            if (excess[cur_v] <= 0):
                break

    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
```

At each step a node is popped from the stack

All possible pushes from it are performed...

...keeping track of any node that becomes active

Main loop

```
# ...continues from push_relabel function

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

    for out_e in cur_v.out_edges():
        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

            if (excess[cur_v] <= 0):
                break

    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
```

At each step a node is popped from the stack

All possible pushes from it are performed...

...keeping track of any node that becomes active

Current node is relabeled if still active

Main loop

```
# ...continues from push_relabel function

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

    for out_e in cur_v.out_edges():
        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

            if (excess[cur_v] <= 0):
                break

    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
```

At each step a node is popped from the stack

All possible pushes from it are performed...

...keeping track of any node that becomes active

Current node is relabeled if still active

When no active node is left the algorithm terminates returning the computed flow map

Main loop complexity

- Max number of relabels per node: $2V - 1$

Main loop complexity

- Max number of relabels per node: $2V - 1$
- Max number of scans of outgoing edges per vertex: $4V + 1$

Main loop complexity

- Max number of relabels per node: $2V-1$
- Max number of scans of outgoing edges per vertex: $4V+1$
- Global time spent to process a node v:

$$O(V \times deg_{out}(v)) + O(1) \times n_{pushes}(v)$$

Main loop complexity

- Max number of relabels per node: $2V - 1$
- Max number of scans of outgoing edges per vertex: $4V + 1$
- Global time spent to process a node v:

$$O(V \times deg_{out}(v)) + O(1) \times n_{pushes}(v)$$

- Summing over all vertices and recalling that the total number of pushes is $O(V^2E)$ we obtain:

$$O(V \sum_{v} deg_{out}(v)) + O(1) \times \sum_{v} n_{pushes}(v) = O(VE) + O(V^2E)$$

Main loop complexity

- Max number of relabels per node: $2V - 1$
- Max number of scans of outgoing edges per vertex: $4V + 1$
- Global time spent to process a node v:

$$O(V \times deg_{out}(v)) + O(1) \times n_{pushes}(v)$$

- Summing over all vertices and recalling that the total number of pushes is $O(V^2E)$ we obtain:

$$O(V \sum_v deg_{out}(v)) + O(1) \times \sum_v n_{pushes}(v) = O(VE) + O(V^2E)$$

- Hence main loop complexity is itself $O(V^2E)$

Overall complexity

- Implementation time complexity: $O(V^2E)$

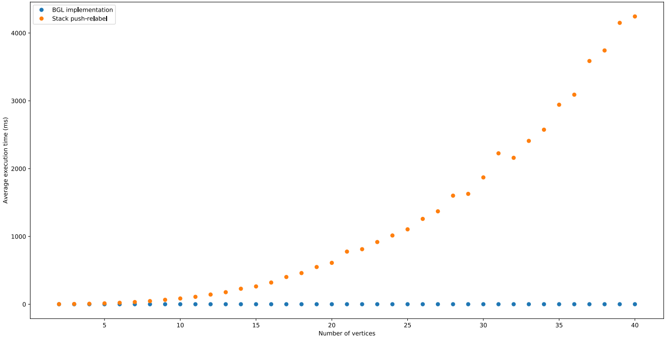
Overall complexity

- Implementation time complexity: $O(V^2 E)$

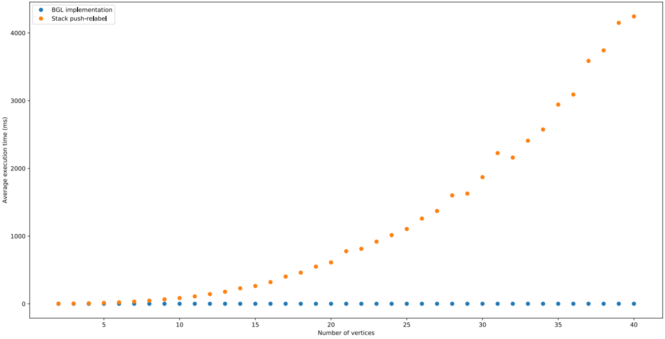
- Implementation space complexity: $O(V + E)$

Benchmark results

Execution time vs number of vertices

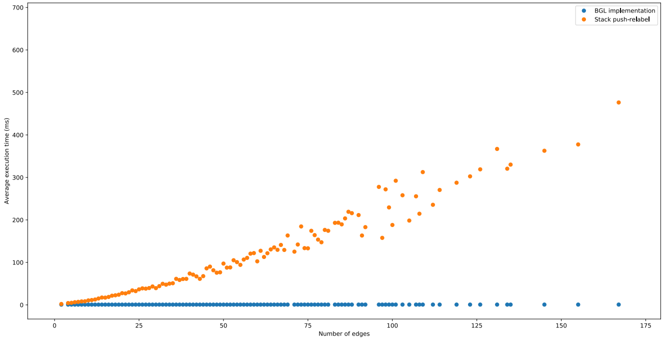


Execution time vs number of vertices

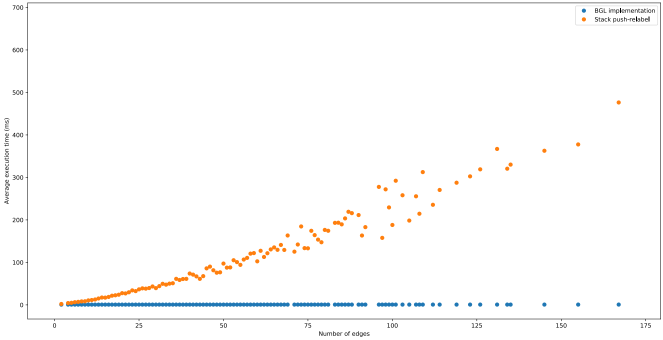


Execution time superlinear in the number of vertices...

Execution time vs number of edges



Execution time vs number of edges



...and linear in the number of edges