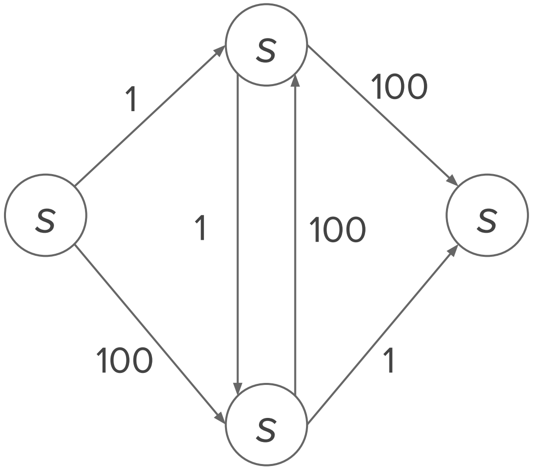
The Push-Relabel algorithm

Michele Papale - Marco Varrone

Advanced Algorithms 2017/2018

Introduction to the algorithm

Introduction: problem definition



- **Graph** G with vertices V and edges E
- **Source** vertex $s \in V$ with no incoming edges
- **Sink** vertex $t \in V$ with no outgoing edges
- Nonnegative and integral **capacity** u_e for each edge $e \in E$

Introduction: goal definition

The feasible solutions are the **flows** in the network

A flow f is a nonnegative vector of length $|E|$ subject to:

- **Capacity constraint**: $f_e \leq u_e$ for every edge $e \in E$
- **Conservation constraint**: for every vertex v other than s and t,

$$\sum_i f_{iv} = \sum_i f_{vi} \text{ for every other vertex } i \in V$$

Our goal is to compute a **maximum flow**

Introduction: Ford-Fulkerson algorithm

One of the most known algorithm for computing maximum flow.

The time complexity of Ford-Fulkerson is bounded by

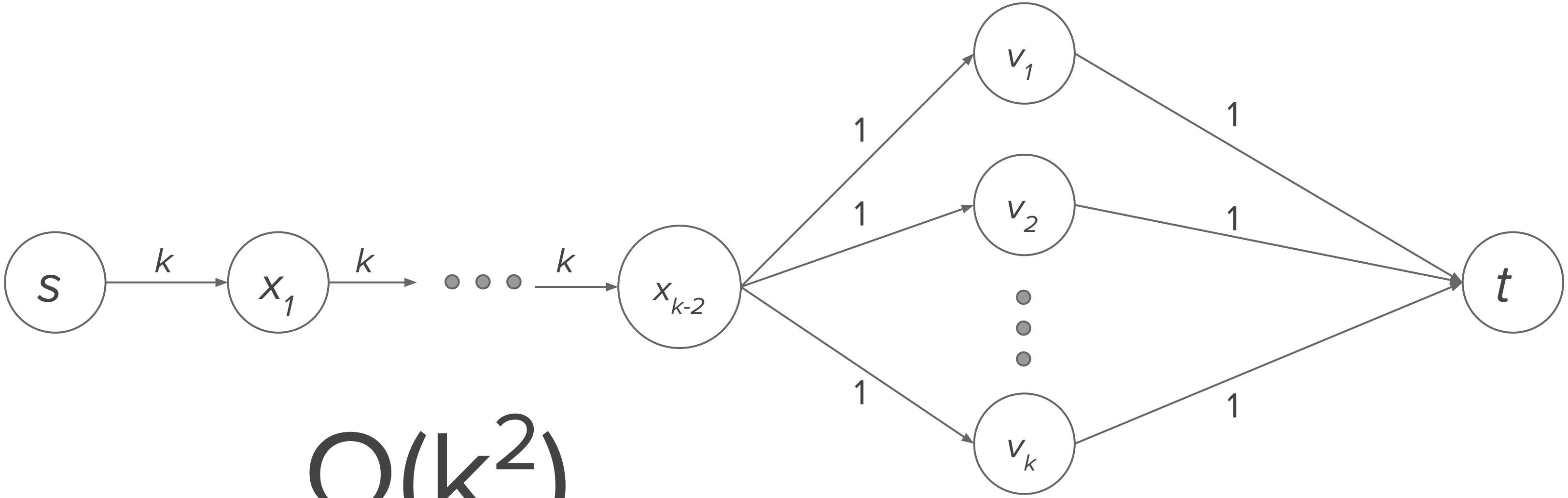
$$O(|E| \cdot f)$$

where f is the maximum flow of the graph

The algorithm is based on **augmenting paths**

Introduction: Ford-Fulkerson limitations

1. Non-terminating instances
2. Excessive computation for some instances, e.g.



$$\Omega(k^2)$$

The Push-Relabel algorithm

The algorithm: constraint relaxation

Relaxed conservation constraint

for every vertex v other than s, $\sum_i f_{iv} \geq \sum_i f_{vi}$ for every other vertex $i \in V$

$$\text{flow} \longrightarrow \textbf{preflow}$$

Excess of vertex v and flow f where $v \neq s, t$:

$$e_f(v) = \sum_i f_{iv} - \sum_i f_{vi}$$

The algorithm: push step

Push(*v*)

1. choose and outgoing edge **(*v*, *w*)** of *v* in the graph
2. **δ** = min{$e_f(v)$, c_{vw} - f_{vw}} where c_{vw} is the capacity of edge (*v*, *w*)
3. push δ units along (*v*, *w*)

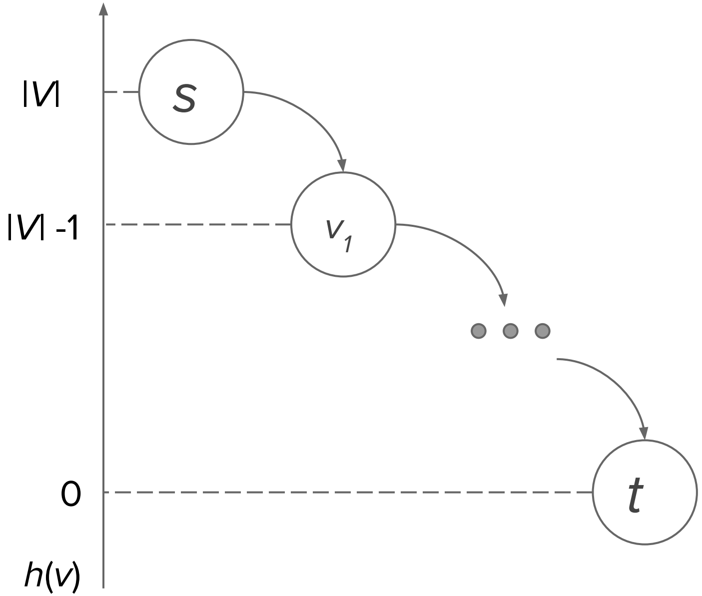
The algorithm: relabel step

Goal: avoid pushing flow along cycles forever

Solution: define a **height** $h(v)$ for each vertex v

- $h(s) = |V|$
- $h(t) = 0$

Push only if $h(v) = h(w) + 1$



The algorithm: relabel step

Relabel(v)

if no push can be performed from vertex v

$$h(v) = \min_{(v,\, w)}\{h(w) + 1\}$$

The algorithm: termination

- Push and relabel are performed as long as

there is a vertex $v \neq s, t$ with $e_f(v) > 0$

- When all these vertices have zero excess:

preflow is a maximum flow

Pseudocode and alternative implementations

Pseudocode - terminology

- active vertex
- admissible edge
- residual graph

Pseudocode

```
def get_max_flow():
    init(source, sink)
    while active = get_active_vertex()
            if not push(active):
                    relabel(active)
    return excess(sink)
```

```
def init(self, source):
    height(source) = n
    for e in G:
        res(e) = capacity(e)
    for e in source.out_edges():
        send_flow(source, e.target(), capacity(e))
```

```
def push(vertex)
    for edge in admissible_out_edges:
        delta = min(excess(vertex), res(edge))
        send_flow(vertex, edge.target(), delta)
    return if a push has been performed
```

```
def relabel(vertex):
    height(vertex) = min(v, w){h(w) + 1}
```

```
def send_flow(source, target, value):
    edge = edge(source, target)
    if there is a corresponding reverse_edge
    not present in the initial graph:
        reverse_edge = found_edge
    else:
        reverse_edge = add_edge(target, source)

    decrease_res(edge, value)
    increase_res(reverse_edge, value)
    increase_excess(target, value)
    decrease_excess(source, value)
```

Wave implementation

Alternative implementation (Lift-to-front)

When a vertex v gets relabeled,

then v is placed at the beginning of the active node list.

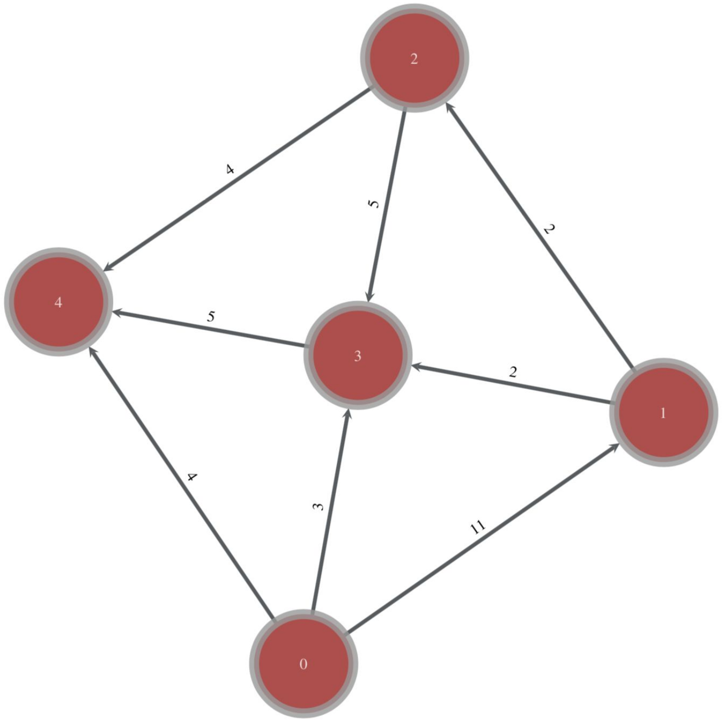
Third implementation: highest label preflow

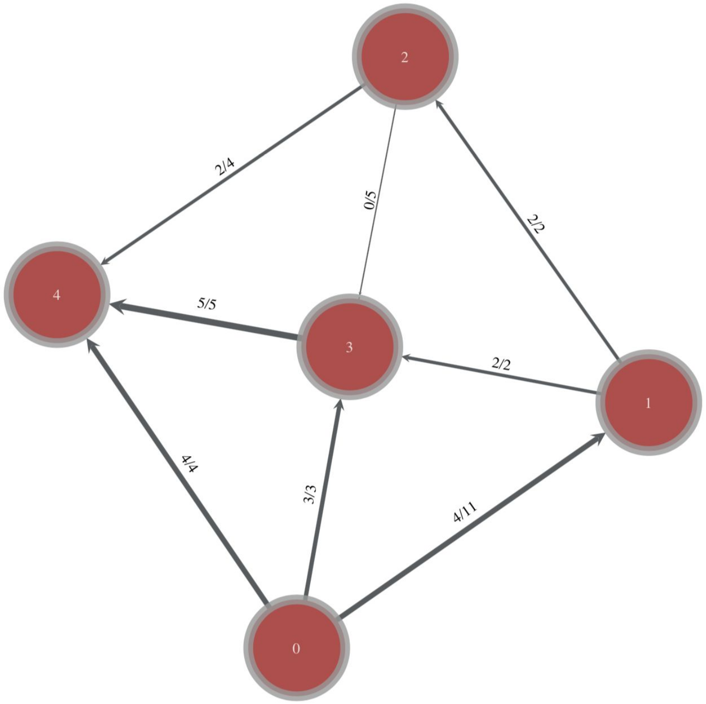
Alternative implementation

As active vertex to push or relabel,

pick the one having the largest height $h(v)$

Running Goldberg 5 nodes graph (0 source, 4 target)





Complexity analysis

Implementation details and improvements

All the complexities have been sampled by running a **cythonized** version of python scripts (in order to compile it in C and to increase the performance)

To build and manipulate graphs, the library **graph-tool** has been used.

Performance has been further increased by using **numpy** module.

Temporal complexity - Goldberg implementation

Relabels

- Total number of relabels = $O(n^2)$
- Complexity of all relabels = $O(mn)$

Pushes

- Total number of saturating pushes = $O(mn)$
- Total number of non-saturating pushes = $O(n^2m)$

Total temporal complexity

$$O(n^2m)$$

Temporal complexity - Goldberg implementation

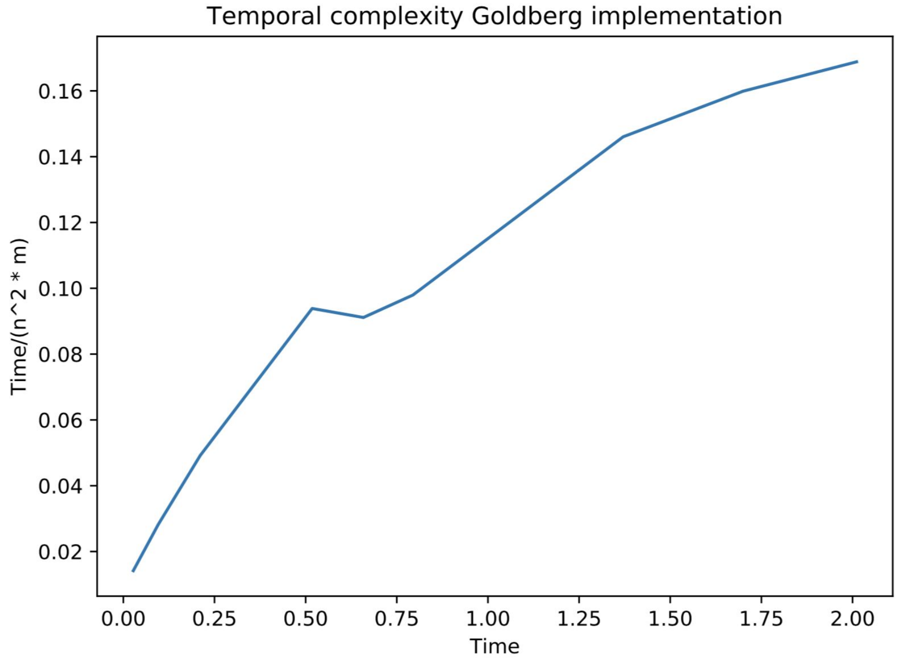
Several executions: from 10 nodes graphs to 90 nodes as input.

But also, each input has been sampled multiple times.

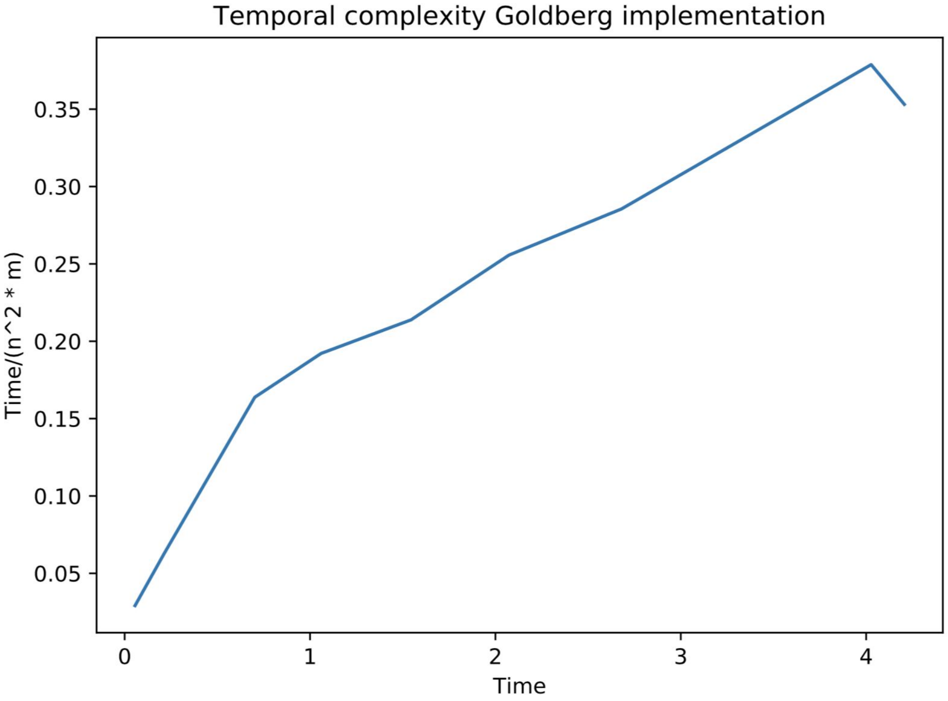
Considered:

- Average case
- Max case

Average case



Max case



Temporal complexity - alternative implementations

Wave:

Total temporal complexity

$$O(n^3)$$

Relabels

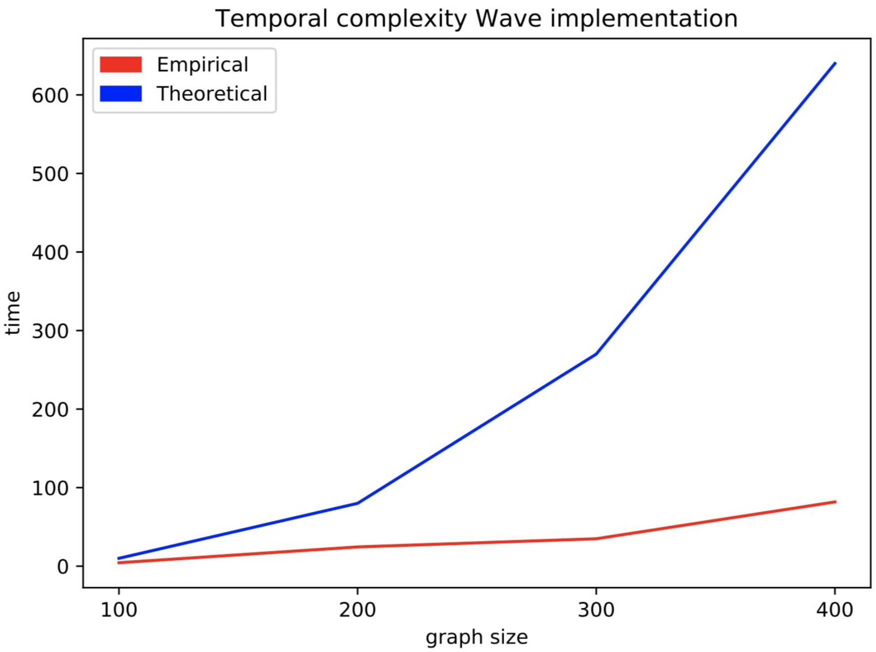
- like Goldberg implementation

Pushes

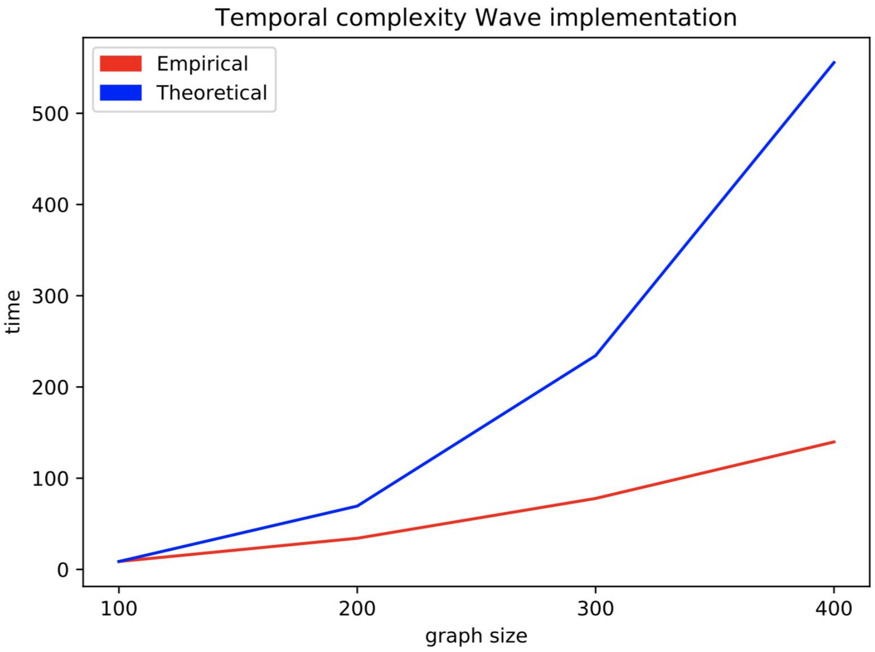
- Total number of non-saturating pushes = $O(n^3)$

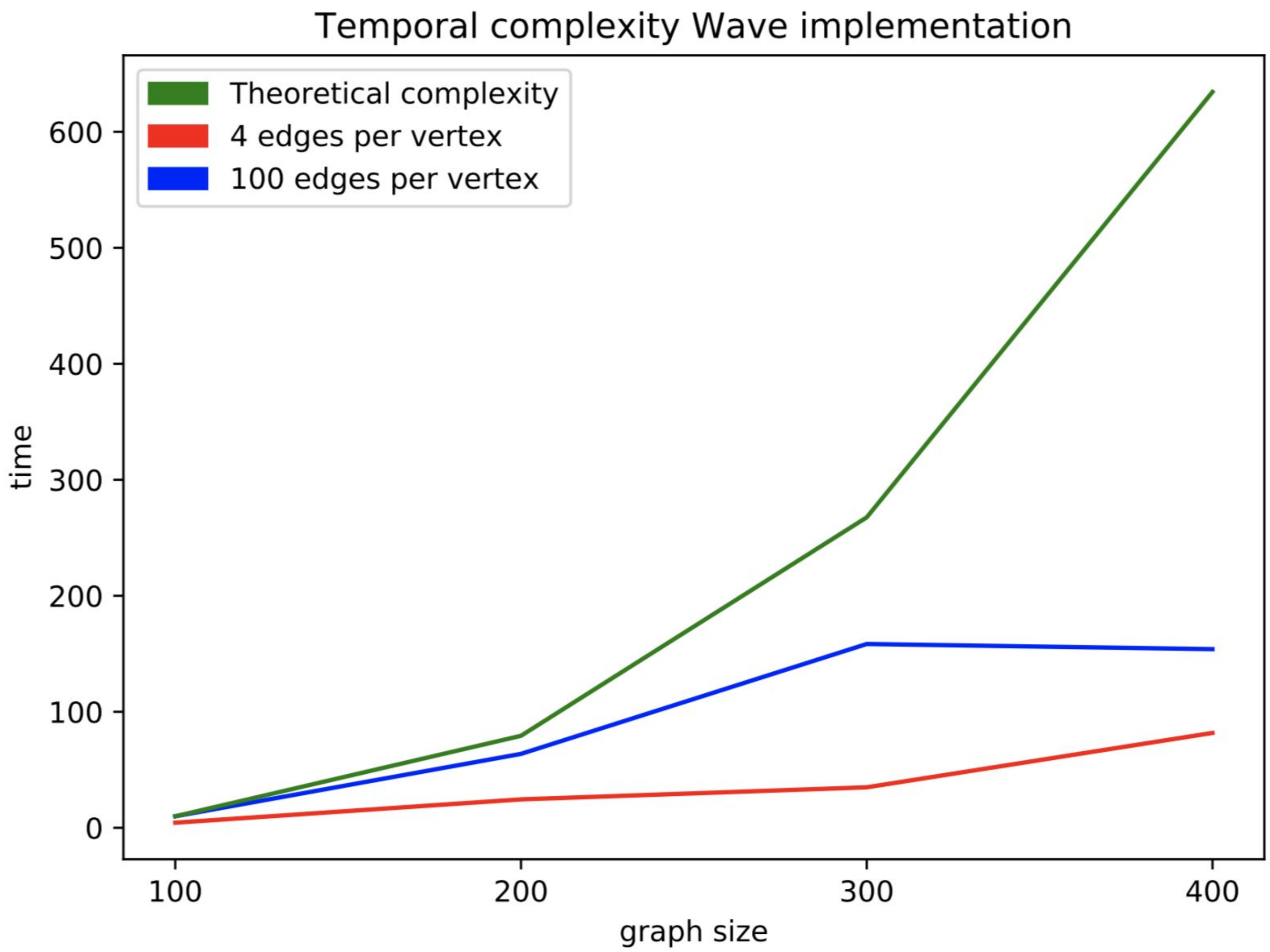
Highest label preflow: $O(m^{1/2} n^2)$

Using the avg case

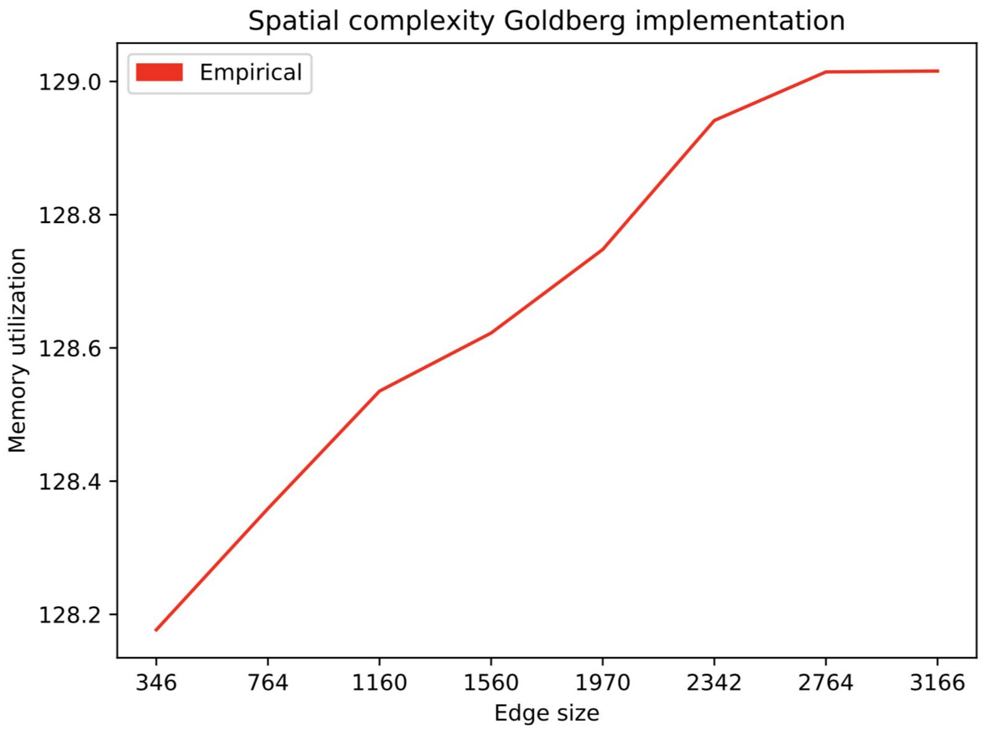


Using the max case




Temporal complexity Wave implementation
Theoretical complexity
4 edges per vertex
100 edges per vertex
time
0
100
200
300
400
500
600
100
200
300
400
graph size

Spatial complexity - Goldberg implementation



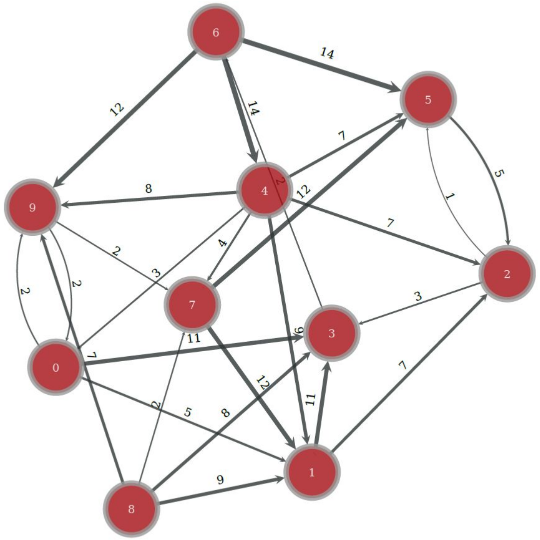
Graph types, tests and applications

Graph types: random graph

- Add *N* vertices to the graph

- Generate M distinct couples of integers between 0 and *N*-1, using a discrete uniform distribution

- Use these values as vertex indices for adding new edges

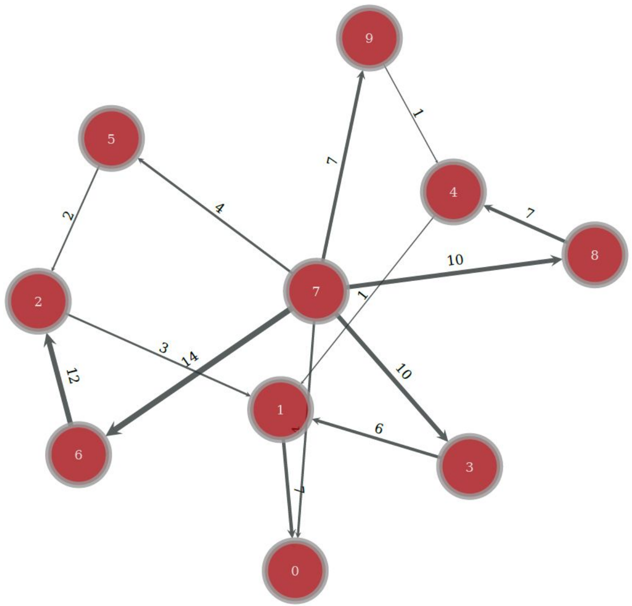


Graph types: triangular network

- *N* points are generated over a 2D plane using a uniform distribution

- Each point is a vertex

- The edges are generated through a technique called triangulation



- Two possible types of triangulation: *simple* and *delaunay*

Graph types: scale-free network

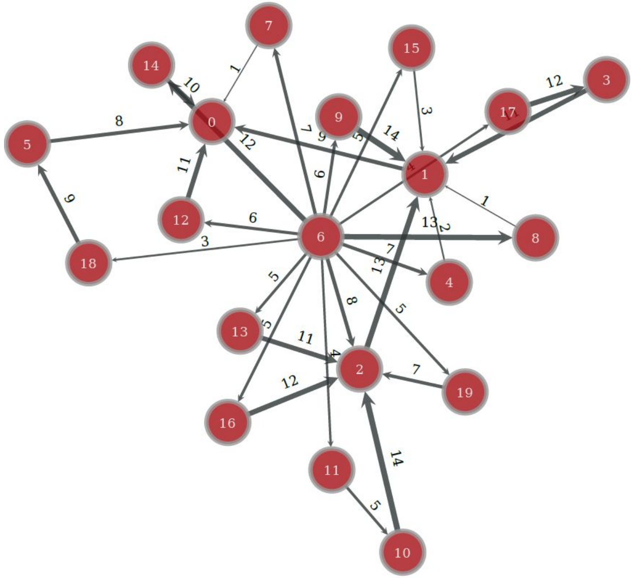
The degree distributions follows a power law

There are few vertices (*hubs*) with very high degree

(number of incident edges)

A very common type of network

(biology, social networks, ...)



Unit tests

Tests performed by

- Number of nodes
- Graph type
- Push-relabel implementation

```
test_random.py ..........                                          [  7%]
test_random_height.py ..........                                   [ 14%]
test_random_wave.py ..........                                     [ 22%]
test_scale_free.py ..........                                      [ 29%]
test_scale_free_height.py ..........                               [ 37%]
test_scale_free_wave.py ..........                                 [ 44%]
test_source_sinks.py ...............                               [ 55%]
test_triangulation.py ....................                         [ 70%]
test_triangulation_height.py ....................                  [ 85%]
test_triangulation_wave.py ....................                    [100%]

========== 135 passed, 891 warnings in 95.14 seconds ==========
```

To check:

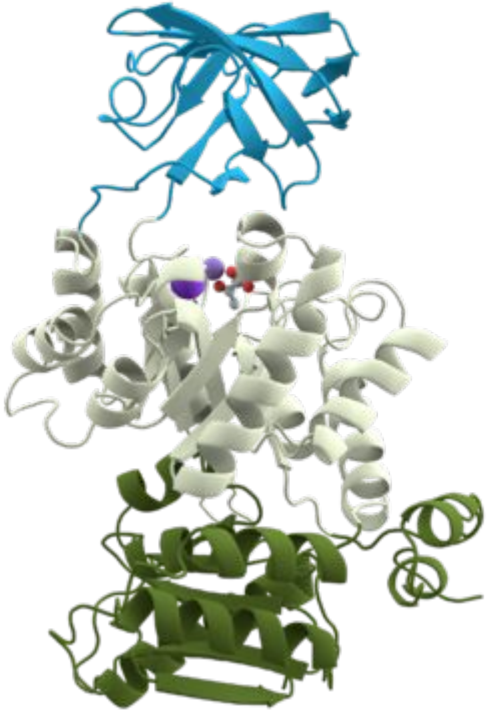
- correctness (max flow = graph-tool)
- Only 1 source and 1 sink

Application: protein domain decomposition

Identify protein domains

- Each residue is a vertex
- Each residue-residue interaction is an edge
- Capacity depends on how close residues are

Find a **minimum cut**



Application: Baseball elimination

- Baseball teams play competition
- Only one team wins (the one who will win the most number of games)
- Every fan wants to know if his favourite team can still win

Example:

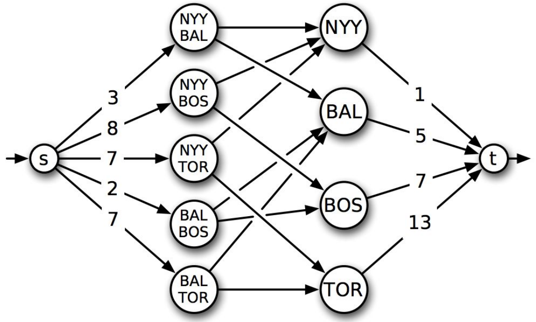
- Detroit can still win if it will win 27 games...
- ...but, all other teams has to lose, infeasible!

Team	Won–Lost
New York Yankees	75–59
Baltimore Orioles	71–63
Boston Red Sox	69–66
Toronto Blue Jays	63–72
Detroit Tigers	49–86

Application: Baseball elimination

From source to game nodes: capacity = #remaining games

From team nodes to sink: capacity = W[n] + R[n] − W[i]



Theorem: Team n can end the season in first place if and only if there is a feasible flow in this graph that saturates every edge leaving s.

References

http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap27.htm

https://theory.stanford.edu/~tim/w16/l/l3.pdf

https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm

https://www.youtube.com/watch?v=0hI89H39USg&t=223s

http://jeffe.cs.illinois.edu/teaching/algorithms/notes/24-maxflowapps.pdf

http://www.stat.wsu.edu/math/faculty/bkrishna/FilesMath574/Papers/MaxFlowDomainDecomp.pdf

Lecture Notes for IEOR 266: Graph Algorithms and Network Flows - Professor Dorit S. Hochbaum