# "Djikstra Algorithm Base"

The input format for the djikstra algorithm is

<long node\_id> [<long node\_id> <float edge\_value>]\*

The output format of the djikstra algorithm is

<long node\_id> [<long root\_of\_spanning\_tree\_id> <long next\_node\_higher\_in\_spanning\_tree>]\*

roots do not list themselves

Run this command with:

mpiexec -n <N machines> --hostfile <hostfile> ./djikstra --graph <graph location> [--saveprefix <prefix to attach to output>]

The output describes the spanning trees constructed starting at up to 3000 nodes

## "Djikstra Algorithm Details"

Djikstra starts with a randomly selected (~3000) set of nodes using a constant time and space random process. Each gather collects the earlier spanning tree members. The apply step calculates the best path, storing it. The scatter step notifies all nodes other than nodes that have notifid it to execute now. This process runs concurrently for each starting node.

Note: this algorithm does not preserve correctness of the total cost at each node, only relative stregth between node choices. Re-walk the spanning tree to calculate this (see below for examples).

# "Betweeness Algorithms"

The input format for the betweeness algorithm is:

<long node\_id> [<long node\_id> <float edge\_value>]\*

The output format of the betweeness algorithm is

<long node\_id> <float betweeness score>

Run this command with:

mpiexec -n <N machines> --hostfile <hostfile> ./betweeness --graph <graph location> [--saveprefix <prefix to attach to output>]

The output estimates betweeness using ~3000 randomly selected spanning trees (typically +/-3% accuracy in the measure for each node.)

## "Betweeness Algorithm Details"

Djikstra sanning trees are calculated first, then the datsa structure is reset, then betweeness scores are calculated by walking the spanning trees from leaves to roots. Finally, the betweeness scores are collated from the various samples of spanning trees.

See djiksra\_imp for details on how spanning trees are calculated.

The next step resets all the bookkeeping on the spanning trees and sets costs to zero.

Finally, the betweeness is calculated. All nodes are started at first, but only nodes without another spanning tree node pointing to it have a non-null execution.

The gather step checks if all nodes in the spanning tree pointing to it have been calculated yet, silently skipping if this is not true. Otherwise, the betweeness scores are collected.

The apply step sums the betweeness score for this node and spanning tree.

The scatter step signals the next higher node in this spanning tree that a new betweeness score is ready.

When the graph is saved, it outputs the sum of all betweeness scores across all calculated spanning trees and estimates the expected final betweeness score.

# "Closeness Algorithm"

The input format for the closeness algorithm is:

<long node\_id> [<long node\_id> <float edge\_value>]\*

The output format of the betweeness algorithm is

<long node\_id> <float closeness score>

Run this command with:

mpiexec -n <N machines> --hostfile <hostfile> ./closeness --graph <graph location> [--saveprefix <prefix to attach to output>]

The output estimates closeness using ~3000 randomly selected spanning trees (typically +/-3% accuracy in the measure for each node.)

## "Closeness Algorithm Details"

Djikstra spanning trees are calculated first, then the datsa structure is reset, then closeness scores are calculated by walking the spanning trees from leaves to roots. Finally, the closeness scores are collated from the various samples of spanning trees.

See djiksra\_imp for details on how spanning trees are calculated, except the link direction is reversed.

The next step resets all the bookkeeping on the spanning trees and sets costs to zero.

Finally, the closeness is calculated. The starting node set is reused, and the spanning trees are all walked simultaneously from root to leaves.

The gather step collects the parent closeness score.

The apply step combines the parent's closeness score with the edge value and stores it.

The scatter step signals al child nodes.

When the graph is saved, it outputs the sum of all closeness scores across all calculated spanning trees and estimates the expected final closeness score.

# "Prestige Algorithm"

The input format for the prestige algorithm is:

<long node\_id> [<long node\_id> <float edge\_value>]\*

The output format of the betweeness algorithm is

<long node\_id> <float prestige score>

Run this command with:

mpiexec -n <N machines> --hostfile <hostfile> ./prestige --graph <graph location> [--saveprefix <prefix to attach to output>]

The output estimates prestige using ~3000 randomly selected spanning trees (typically +/-3% accuracy in the measure for each node.)

## "Prestige Algorithm Details"

Djikstra spanning trees are calculated first, then the datsa structure is reset, then prestige scores are calculated by walking the spanning trees from leaves to roots. Finally, the prestige scores are collated from the various samples of spanning trees.

See djiksra\_imp for details on how spanning trees are calculated.

The next step resets all the bookkeeping on the spanning trees and sets costs to zero.

Finally, the prestige is calculated. The starting node set is reused, and the spanning trees are all walked simultaneously from root to leaves.

The gather step collects the parent prestige score.

The apply step combines the parent's prestige score with the edge value and stores it.

The scatter step signals al child nodes.

When the graph is saved, it outputs the sum of all prestige scores across all calculated spanning trees and estimates the expected final prestige score.

# Undirected Triangle Counting

The undirected triangle counting program can count the total number of triangles in a graph, and can also, with little more time, count the number of triangles passing through each vertex in the graph.

It implements the edge-iterator algorithm described in

T. Schank. Algorithmic Aspects of Triangle-Based Network Analysis. Phd in computer science, University Karlsruhe, 2007.

with several optimizations.

The input to the system is a graph in any of the Portable graph format described in [Graph File Formats](file:///C:\Users\nakashim\Desktop\graph_formats.html). It is important that the input be "cleaned" and that reverse edges are removed: i.e. if edge 1–>5 exists, edge 5–>1 should not exist. (The program will run without these edge removed. But numbers may be erroneous).

To count the total number of triangles in a graph, the minimal set of options required are:

> ./undirected\_triangle\_count --graph=[graph prefix] --format=[format]

Output looks like:

Number of vertices: 875713

Number of edges: 4322051

Counting Triangles...

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 875713

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

Counted in 1.16463 seconds

13391903 Triangles

To count the number of triangles on each vertex, the minimal set of options are:

> ./undirected\_triangle\_count --graph=[graph prefix] --format=[format] --per\_vertex=[output prefix]

Tne output prefix is where the output counts will be written. This may be located on HDFS. For instance, if the output\_prefix is "v\_out", the output files will be written to:

v\_out\_1\_of\_16

v\_out\_2\_of\_16

...

v\_out\_16\_of\_16

Each line in the output file contains two numbers: a Vertex ID, and the number of triangles intersecting the vertex.

This program can also run distributed by using

> mpiexec -n [N machines] --hostfile [host file] ./undirected\_triangle\_count ....

See your MPI documentation for details on how to launch this job. All machines must have access to the input graph location and the output graph location. Graphs may be on HDFS. If you have problems loading HDFS files, see the [FAQ](file:///C:\Users\nakashim\Desktop\FAQ.html).

## Options

Relevant options are:

* **–graph** (Required). The prefix from which to load the graph data
* **–format** (Required). The format of the input graph
* **–per\_vertex** (Optional. Default ""). If set, will write the output counts.
* **–ncpus** (Optional. Default 2) The number of processors that will be used for computation.
* **–ht** (Optional. Default 64) The implementation uses a mix of vectors and hash sets to optimize set intersection computation. This parameter sets the capacity limit below which, vectors are used, and above which, hash sets are used.
* **–-graph\_opts** (Optional, Default empty) Any additional graph options. See [graphlab::distributed\_graph](file:///C:\Users\nakashim\Desktop\classgraphlab_1_1distributed__graph.html) a list of options.

# Directed Triangle Counting

The directed triangle counting program counts the total number of directed triangles in a graph of each type, and can also output the number of triangles of each type passing through each vertex in the graph.

We show the 4 possible types of triangles here: In each case, the vertex being evaluated is the green vertex labeled "A". A dotted edge means that the direction of the edge do not matter.

|  |  |
| --- | --- |
| **Triangle Name** | **Triangle Pattern** |
| In Triangle |  |
| Out Triangle |  |
| Through Triangle |  |
| Cycle Triangle |  |

The input to the system is a graph in any of the Portable graph format described in [Graph File Formats](file:///C:\Users\nakashim\Desktop\graph_formats.html).

To count the total number of triangles in a graph, the minimal set of options required are:

> ./directed\_triangle\_count --graph=[graph prefix] --format=[format]

Output looks like this:

Number of vertices: 875713

Number of edges: 5105039

Counting Triangles...

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 875713

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

Counted in 1.962 seconds

Collecting results ...

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 875713

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

28198954 In triangles

28198954 Out triangles

28198954 Through triangles

11669313 Cycle triangles

Observe that the number of In, Out and Through triangles are identical. This is because every In-triangle necessarily forms one Out and one Through triangle, (and similarly for the rest). Also the number of Cycle Triangles must be divisible by 3 since every cycle is counted 3 times, once on each vertex in the cycle.

To count the number of triangles on each vertex, the minimal set of options are:

> ./directed\_triangle\_count --graph=[graph prefix] --format=[format] --per\_vertex=[output prefix]

Tne output prefix is where the output counts will be written. This may be located on HDFS. For instance, if the output\_prefix is "v\_out", the output files will be written to:

v\_out\_1\_of\_16

v\_out\_2\_of\_16

...

v\_out\_16\_of\_16

Each line in the output file has the following format:

[vid] [in triangles] [out triangles] [through triangles] [cycle\_triangles] [#out edges] [#in edges]

This program can also run distributed by using

> mpiexec -n [N machines] --hostfile [host file] ./directed\_triangle\_count ....

See your MPI documentation for details on how to launch this job. All machines must have access to the input graph location and the output graph location. Graphs may be on HDFS. If you have problems loading HDFS files, see the [FAQ](file:///C:\Users\nakashim\Desktop\FAQ.html).

## Options

Relevant options are:

* **–graph** (Required). The prefix from which to load the graph data
* **–format** (Required). The format of the input graph
* **–per\_vertex** (Optional. Default ""). If set, will write the output counts.
* **–ncpus** (Optional. Default 2) The number of processors that will be used for computation.
* **–ht** (Optional. Default 64) The implementation uses a mix of vectors and hash sets to optimize set intersection computation. This parameter sets the capacity limit below which, vectors are used, and above which, hash sets are used.
* **-–graph\_opts** (Optional, Default empty) Any additional graph options. See [graphlab::distributed\_graph](file:///C:\Users\nakashim\Desktop\classgraphlab_1_1distributed__graph.html) a list of options.

# PageRank

The PageRank program computes the pagerank of each vertex. See the [Wikipedia article](http://en.wikipedia.org/wiki/PageRank) for details of the algorithm.

## Input

The input to the system is a graph in any of the Portable graph format described in [Graph File Formats](file:///C:\Users\nakashim\Desktop\graph_formats.html).

> ./pagerank --graph=[graph prefix] --format=[format]

Alternatively, a synthetic power law graph of an arbitrary number of vertices can be generated using:

> ./pagerank --powerlaw=[nvertices]

The resultant graph will have powerlaw out-degree, and nearly constant in-degree. The actual generation process draws vertex degree from a truncated power-law distribution with alpha=2.1. The distribution is truncated at maximum out-degree 100M to avoid allocating massive amounts of memory for creating the sampling distribution.

## Type

There are several modes of computation that are supported. All will eventually obtain the same solutions.

### Classical

To get classical PageRank iterations, adding the option

> --iterations=[N Iterations]

### Dynamic Synchronous (default)

The dynamic synchronous computation only performs computation on vertices that have not yet converged to the desired tolerance. The default tolerance is 0.001. This can be modified by adding the option

> --tol=[tolerance]

### Dynamic Asynchronous

The dynamic asynchronous computation only performs computation on vertices that have not yet converged to the desired tolerance. This uses the asynchronous engine. The default tolerance is 0.001. This can be modified by adding the option

> --tol=[tolerance]

Note

This is known to be slow! PageRank does not benefit from the consistency guaranteed by the asynchronous engine. A new engine is in development with weaker consistency semantics, but sufficient for pagerank.

## Output

To save the resultant pagerank of each vertex, include the option

> --saveprefix=[output prefix]

Tne output prefix is where the output counts will be written. This may be located on HDFS. For instance, if the output\_prefix is "v\_out", the output files will be written to:

v\_out\_1\_of\_16

v\_out\_2\_of\_16

...

v\_out\_16\_of\_16

Each line in the output file contains two numbers: a Vertex ID, and the computed PageRank. Note that the output vector is NOT normalized, namely computed entries do not sum into one.

This program can also run distributed by using

> mpiexec -n [N machines] --hostfile [host file] ./pagerank ....

See your MPI documentation for details on how to launch this job. All machines must have access to the input graph location and the output graph location. Graphs may be on HDFS. If you have problems loading HDFS files, see the [FAQ](file:///C:\Users\nakashim\Desktop\FAQ.html).

## Options

Relevant options are:

* **–graph** (Optional). The prefix from which to load the graph data
* **–format** (Optional). The format of the input graph
* **–powerlaw** (Optional. Default 0). If set, generates synthetic powerlaw graph with the specified number of vertices.
* **–saveprefix** (Optional. Default ""). If set, will write the output counts.
* **–tol** (Optional. Default=1E-3). Changes the convergence tolerance for the Dynamic computation modes.
* **–iterations** (Optional. Default 0). If set, runs classical PageRank iterations for the specified number of iterations.
* **-–graph\_opts** (Optional, Default empty) Any additional graph options. See [graphlab::distributed\_graph](file:///C:\Users\nakashim\Desktop\classgraphlab_1_1distributed__graph.html) a list of options.
* **–ncpus** (Optional. Default 2) The number of processors that will be used for computation.
* **-–engine** (Optional, Default "synchronous") Sets the engine type. Must be either "synchronous" or "asynchronous"
* **-–engine** (Optional, Default "synchronous") Sets the engine options. Available options depend on the engine type. See [graphlab::async\_consistent\_engine](file:///C:\Users\nakashim\Desktop\classgraphlab_1_1async__consistent__engine.html) and [graphlab::synchronous\_engine](file:///C:\Users\nakashim\Desktop\classgraphlab_1_1synchronous__engine.html) for details.

# KCore Decomposition degree>=Kのノードのみからなるサブグラフを取り出すグラフ分割

This program iteratively finds the KCore of the network.

## Input

The input to the system is a graph in any of the Portable graph format described in [Graph File Formats](file:///C:\Users\nakashim\Desktop\graph_formats.html).

> ./kcore --graph=[graph prefix] --format=[format]

Output may look like:

K=0: #V = 875713 #E = 4322051

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 0

K=1: #V = 875713 #E = 4322051

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 153407

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

K=2: #V = 711870 #E = 4160100

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 108715

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

K=3: #V = 581712 #E = 3915291

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 69907

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

K=4: #V = 492655 #E = 3668104

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 52123

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

K=5: #V = 424155 #E = 3416251

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 41269

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

K=6: #V = 367361 #E = 3158776

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 33444

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

K=7: #V = 319194 #E = 2902138

INFO: synchronous\_engine.hpp(start:1213): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1257): Active vertices: 29201

INFO: synchronous\_engine.hpp(start:1307): Running Aggregators

K=8: #V = 274457 #E = 2629033

......

To just get the informative lines:

> ./kcore --graph=[graph prefix] --format=[format] > k\_out.txt

...

> cat k\_out.txt

Computes a k-core decomposition of a graph.

Number of vertices: 875713

Number of edges: 4322051

K=0: #V = 875713 #E = 4322051

K=1: #V = 875713 #E = 4322051

K=2: #V = 711870 #E = 4160100

K=3: #V = 581712 #E = 3915291

K=4: #V = 492655 #E = 3668104

K=5: #V = 424155 #E = 3416251

K=6: #V = 367361 #E = 3158776

K=7: #V = 319194 #E = 2902138

K=8: #V = 274457 #E = 2629033

K=9: #V = 231775 #E = 2335154

K=10: #V = 193406 #E = 2040738

K=11: #V = 159020 #E = 1753273

K=12: #V = 131362 #E = 1500517

K=13: #V = 106572 #E = 1256952

K=14: #V = 86302 #E = 1047053

K=15: #V = 68409 #E = 849471

K=16: #V = 53459 #E = 676076

K=17: #V = 40488 #E = 519077

...

The program can also save a copy of the graph at each stage by adding an option.

> --savecores=[prefix]

The resultant graphs will be saved with prefixes [prefix].K For instance if prefix is out, The 0-Core graph may be saved in

out.0.1\_of\_4

out.0.2\_of\_4

out.0.3\_of\_4

out.0.4\_of\_4

The 5-Core graph will be saved in

out.5.1\_of\_4

out.5.2\_of\_4

out.5.3\_of\_4

out.5.4\_of\_4

and so on.

The range of k-Core graphs to compute can be controlled by the kmin and the kmax option described below.

This program can also run distributed by using

> mpiexec -n [N machines] --hostfile [host file] ./kcore....

See your MPI documentation for details on how to launch this job. All machines must have access to the input graph location and the output graph location. Graphs may be on HDFS. If you have problems loading HDFS files, see the [FAQ](file:///C:\Users\nakashim\Desktop\FAQ.html).

## Options

Relevant options are:

* **–graph** (Required). The prefix from which to load the graph data
* **–format** (Required). The format of the input graph
* **–ncpus** (Optional. Default 2) The number of processors that will be used for computation.
* **–savecores** (Optional. Default ""). The target prefix to save the resultant K-core graphs.
* **–kmin** (Optional. Default 0). Only output result for the K-core graph starting at K=kmin
* **–kmax** (Optional. Default Inf). Only output result for the K-core graph up to K=kmax

# Graph Coloring　隣接頂点が異なる色になるよう塗り分ける問題

The graph coloring program implements a really simple graph coloring procedure: each vertex reads the colors of its neighbors and takes on the smallest possible color which does not conflict with its neighbors.

The procedure necessarily uses the asynchronous engine (it will never converge with the synchronous engine).

The input to the system is a graph in any of the Portable graph format described in [Graph File Formats](file:///C:\Users\nakashim\Desktop\graph_formats.html). It is important that the input be "cleaned" and that reverse edges are removed: i.e. if edge 1–>5 exists, edge 5–>1 should not exist. (The program will run without these edge removed. But numbers may be erroneous).

To color a graph, the minimal set of options required are:

> ./simple\_coloring --graph=[graph prefix] --format=[format] --output=[output prefix]

Output looks like:

Number of vertices: 875713

Number of edges: 5105039

Coloring...

Completed Tasks: 875713

Issued Tasks: 875713

Blocked Issues: 0

------------------

Joined Tasks: 0

Colored in 42.3684 seconds

Metrics server stopping.

Observe that the number of Completed Tasks is identical to the number of vertices. This is a result of the consistency model which ensures that the entire vertex update is peformed "atomically".

Tne output prefix is where the output counts will be written. This may be located on HDFS. For instance, if the output\_prefix is "v\_out", the output files will be written to:

v\_out\_1\_of\_16

v\_out\_2\_of\_16

...

v\_out\_16\_of\_16

Each line in the output file contains two numbers: a Vertex ID, and the number color of the vertex.

This program can also run distributed by using

> mpiexec -n [N machines] --hostfile [host file] ./simple\_coloring ....

See your MPI documentation for details on how to launch this job. All machines must have access to the input graph location and the output graph location. Graphs may be on HDFS. If you have problems loading HDFS files, see the [FAQ](file:///C:\Users\nakashim\Desktop\FAQ.html).

## Options

Relevant options are:

* **–graph** (Required). The prefix from which to load the graph data
* **–format** (Required). The format of the input graph
* **–ncpus** (Optional. Default 2) The number of processors that will be used for computation.
* **–-graph\_opts** (Optional, Default empty) Any additional graph options. See –graph\_help a list of options.
* **–-engine\_opts** (Optional, Default empty) Any additional engine options. See –engine\_help a list of options.

A particularly relevant option is

--engine\_opts="factorized=true"

This uses a weaker consistency setting which only guarantees that individual "gather/apply/scatter" operations are atomic, but does not guarantee atomicity of the entire update. As a result, this may require more updates to complete, but could in practice run significantly faster.

# Connected Component　　分割可能グラフの各分割ごとにノード数や最小ノード番号を調べる

The connected component program can find all connected components in a graph, and can also count the number of vertices (size) of each connected component.

The input to the system is a graph in any of the Portable Graph formats described in [Graph File Formats](file:///C:\Users\nakashim\Desktop\graph_formats.html).

To find connected components in a graph, the minimal set of options required are:

> ./connected\_component --graph=[graph prefix] --format=[format]

Here is a toy example, graph with 6 nodes and 5 edges:

# example graph

# vertices: 6 edges: 5

1 2

2 3

4 5

4 6

5 6

Assume file name is toy\_graph, the command used for running connected compnents is

> ./connected\_component --graph=toy\_graph --format=tsv --saveprefix=out

When you set –saveprefix=output\_prefix, the pairs of a Vertex ID and a Component ID will be written to a sequence of files with prefix output\_prefix. This may be located on HDFS. For instance, if the output\_prefix is "v\_out", the output files will be written to:

out\_1\_of\_4

out\_2\_of\_4

out\_3\_of\_4

out\_4\_of\_4

Let's examine the output. The first column is the node id, while the second column is it's assigned component number (which is also the lowest node id in this component). In our case:

1,1

2,1

3,1

4,4

5,4

6,4

There are two components. The first compoent is 1,2,3 and the second component is 4,5,6

Note that this program can also run distributed by using

> mpiexec -n [N machines] --hostfile [host file] ./connected\_component ....

See your MPI documentation for details on how to launch this job. All machines must have access to the input graph location and the output graph location. Graphs may be on HDFS. If you have problems loading HDFS files, see the [FAQ](file:///C:\Users\nakashim\Desktop\FAQ.html).

## Options

Relevant options are:

* **–graph** (Required). The prefix from which to load the graph data
* **–format** (Required). The format of the input graph
* **–saveprefix** (Optional). If set, pairs of a Vertex ID and a Component ID will be saved to a sequence of files with the given prefix.
* **–ncpus** (Optional. Default 2). The number of processors that will be used for computation.
* **–graph\_opts** (Optional, Default empty). Any additional graph options. See [graphlab::distributed\_graph](file:///C:\Users\nakashim\Desktop\classgraphlab_1_1distributed__graph.html) a list of options.

connected\_components\_stats is a helper utility, which computes histogram of component sizes.

Using our toy example

> ./connected\_component\_stats --graph=out

Connected Component

INFO: mpi\_tools.hpp(init:63): MPI Support was not compiled.

INFO: dc.cpp(init:573): Cluster of 1 instances created.

INFO: distributed\_graph.hpp(set\_ingress\_method:3200): Automatically determine ingress method: grid

Loading graph in format: adj

INFO: distributed\_graph.hpp(load\_from\_posixfs:2189): Loading graph from file: ./out\_1\_of\_4

INFO: distributed\_graph.hpp(load\_from\_posixfs:2189): Loading graph from file: ./out\_2\_of\_4

INFO: distributed\_graph.hpp(load\_from\_posixfs:2189): Loading graph from file: ./out\_3\_of\_4

INFO: distributed\_graph.hpp(load\_from\_posixfs:2189): Loading graph from file: ./out\_4\_of\_4

INFO: distributed\_ingress\_base.hpp(finalize:185): Finalizing Graph...

INFO: distributed\_ingress\_base.hpp(exchange\_global\_info:519): Graph info:

nverts: 2

nedges: 0

nreplicas: 2

replication factor: 1

Complete Finalization in 0.001965

graph calculation time is 2.4e-05 sec

RESULT:

size count

3 2

As expected, there are two components of size 3.

# Approximate Diameter　直径推定

The approximate diameter program can estimate a diameter of a graph. The implemented algorithm is based on the work,

U Kang, Charalampos Tsourakakis, Ana Paula Appel, Christos Faloutsos and Jure Leskovec, HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop (2008).

The input to the system is a graph in any of the Portable Graph formats described in [Graph File Formats](file:///C:\Users\nakashim\Desktop\graph_formats.html).

To compute an approximate diameter of a graph, the minimal set of options required are:

> ./approximate\_diameter --graph=[graph prefix] --format=[format]

Output looks like:

Approximate graph diameter

INFO: synchronous\_engine.hpp(start:1263): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1312): Active vertices: 1271950

INFO: synchronous\_engine.hpp(start:1361): Running Aggregators

1-th hop: 12895307 vertex pairs are reached

INFO: synchronous\_engine.hpp(start:1263): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1312): Active vertices: 1271950

INFO: synchronous\_engine.hpp(start:1361): Running Aggregators

2-th hop: 319726269 vertex pairs are reached

INFO: synchronous\_engine.hpp(start:1263): 0: Starting iteration: 0

INFO: synchronous\_engine.hpp(start:1312): Active vertices: 1271950

INFO: synchronous\_engine.hpp(start:1361): Running Aggregators

3-th hop: 319769151 vertex pairs are reached

converge

graph calculation time is 40 sec

approximate diameter is 2

This program can also run distributed by using

> mpiexec -n [N machines] --hostfile [host file] ./approximate\_diameter ....

See your MPI documentation for details on how to launch this job. All machines must have access to the input graph location and the output graph location. Graphs may be on HDFS. If you have problems loading HDFS files, see the [FAQ](file:///C:\Users\nakashim\Desktop\FAQ.html).

## Options

Relevant options are:

* **–graph** (Required). The prefix from which to load the graph data
* **–format** (Required). The format of the input graph
* **–tol** (Optional. Default=1E-4). Changes the convergence tolerance for the number of reached vertex pairs at each hop.
* **–use-sketch** (Optional. Default=1). If true, will use Flajolet & Martin bitmask to approximately count numbers of reached vertex pairs, and will require a smaller memory. If false, will count exact numbers of reached vertex pairs. But this will need a huge memory and be slow.
* **–ncpus** (Optional. Default 2). The number of processors that will be used for computation.
* **–graph\_opts** (Optional, Default empty). Any additional graph options. See [graphlab::distributed\_graph](file:///C:\Users\nakashim\Desktop\classgraphlab_1_1distributed__graph.html) a list of options.

# Graph Partitioning

This program can partition a graph by using normalized cut.

The input to the system is a graph in any of the Portable Graph formats described in [Graph File Formats](file:///C:\Users\nakashim\Desktop\graph_formats.html). You can also give weights to edges with the weight format. For instance in this weight format file, there are 5 edges:

1 2 4.0

2 3 1.0

3 4 5.0

4 5 2.0

5 3 3.0

To partition a graph, the minimal set of options required are:

> ./partitioning --graph=[graph prefix] --format=[format]

**This program uses svd in Graphlab Collaborative Filtering Toolkit and kmeans in Graphlab Clustering Toolkit.** The paths to the directories are specified by –svd-dir and –kmeans-dir, respectively.

The program will create some intermediate files. The final partitioning result is written in files named [graph prefix].result with suffix, for example [graph prefix].result\_1\_of\_4. The partitioning result data consists of two columns: one for the ids and the other for the assigned partitions. For instance:

1 0

2 0

3 1

4 1

5 1

**NOTE:** To run this program in a distributed setting, you must use the "mpi-args" option, not like other graphlab toolkits. The graph partitioning calls other graphlab programs. When "--mpi-args" is set, these graphlab programs are called with "mpiexec" and the string written after the "mpi-args" option. For example, if you set –mpi-args="-n 4 --hostfile host", the program calls the other graphlab programs with "mpiexec -n 4 --hostfile host".

## Options

Relevant options are:

* **–graph** (Required). The prefix from which to load the graph data
* **–format** (Required). The format of the input graph. If "weight" is set, the program will read the data file where each line holds [id1] [id2] [weight].
* **–partitions** (Optional. Default 2). The number of partitions
* **–svd-dir** (Optional. Default ../collaborative\_filtering/). Path to the directory where Graphlab svd is located
* **–kmeans-dir** (Optional. Default ../clustering/). Path to the directory where Graphlab kmeans is located
* **–ncpus** (Optional. Default 2). The number of processors that will be used for computation.
* **–graph\_opts** (Optional, Default empty). Any additional graph options. See [graphlab::distributed\_graph](file:///C:\Users\nakashim\Desktop\classgraphlab_1_1distributed__graph.html) a list of options.
* **–mpi-args** (Optional, Default empty). If set, will execute mipexec with the given string.

# "Total Subgraph Centrality"

Total subgraph centrality was implemented by Jacob Kesinger, see additional details in his [blog post](http://jacobkesinger.tumblr.com/post/64338572799/total-subgraph-centrality). Total Subgraph Communicability is a new centrality measure due to Benzi&Klymco [1]. For a directed graph with adjacenty matrix A,

TSC\_i = sum\_j exp(A)\_{ij} = (exp(A)\*1)\_i.

This code calculates the TSC using an Arnoldi iteration on the Krylov subspace {b, Ab,A\*Ab, A\*A\*Ab, ...} due to Saad[1], and using the new warp engine from Graphlab 2.2 (without which this would have been, at best, very challenging).

Small components of large graphs will have bogus answers due to floating point issues. To find the exact TSC for a particular node i, run with "--column i" to find exp(A)\*e\_i; you will have to sum the resulting output yourself, however.

SAMPLE INPUT:

0 1

1 2

1 3

2 4

3 4

1 0

2 1

3 1

4 2

4 3

OUTPUT:

0 5.17784

1 10.3319

2 8.49789

3 8.49789

4 7.96807

You can verify this in python as:

import scipy

import scipy.linalg

A = scipy.array([[0,1,0,0,0],[1,0,1,1,0],[0,1,0,0,1],[0,1,0,0,1],[0,0,1,1,0]])

scipy.linalg.expm2(A).sum(axis=1)

[1]: Benzi, Michele, and Christine Klymko. Total Communicability as a Centrality Measure. ArXiv e-print, February 27, 2013. [arxiv](http://arxiv.org/abs/1302.6770)

[2]: Saad, Yousef. “Analysis of Some Krylov Subspace Approximations to the Matrix Exponential Operator.” SIAM Journal on Numerical Analysis 29, no. 1 (1992): 209–228.

* [GraphLab Toolkits](file:///C:\Users\nakashim\Desktop\toolkits.html)
* Generated on Fri Sep 19 2014 13:06:45 for GraphLab: Distributed Graph-Parallel API by 1.8.6