GPU Accelerated Graph Processing in Python

Getting Started Cheat Sheet

High-Performance Jupyter Notebooks – Managed GPU environment.
Scale up as needed. Get started in minutes with no setup required.
Try the free notebook with examples here: https://cutt.ly/rapids-cheatsheets-cugraph

For additional cheat sheets go to: **nvidia.com/rapids-kit/**

CREATE GRAPH

Create graphs from adjacency or edge lists.

```
from scipy.sparse import coo_matrix

values = [1,1,1,1,1]
sources = [0,0,0,1,2]
destinations = [1,2,3,2,3]
adj_list = coo_matrix((values, (sources, destinations))).tocsr()
g = cugraph.Graph()
```

g.from_cudf_adjlist(cudf.Series(adj_list.indptr), cudf.Series(adj_list.indices)) - Create a graph from an adjacency list. The adjacency list is derived from the adjacency matrix. The adjacency matrix is a VxV (V being the number of vertices in the graph) sparse representation of graph edges: A value at a location in the matrix indicates an edge; 0 signifies the lack of. The adjacency list is a compressed sparse row (CSR)-formatted representation of the adjacency matrix with two lists describing the edges: One is a full list of destination vertices in the graph (adj_list.indices in the example here), and the other one (the adj_list.indptr) specifies the starting position of edge destinations for each source vertex in the graph.

```
edges = cudf.DataFrame([
    (0, 1, 1)
  , (0, 2, 1)
  , (0, 3, 1)
  , (1, 2, 1)
  , (2, 3, 1)
], columns=['src', 'dst', 'weight'])
g = cugraph.Graph()
g.from_cudf_edgelist(
  edges
  , source='src'
  , destination='dst'
  , edge_attr='weight'
  , renumber=False
)
```

- Create a graph from an edge list. Edge list contains one element per edge, normally in a form of (source, destination, <weight>) (weight is optional).

```
edges = cudf.DataFrame([
  (0,10)
  ,(0,11)
  ,(0,12)
  ,(1,10)
  ,(1,12)
], columns=['src', 'dst'])
nodes = [None, None]
nodes[0] = cudf.Series([0,1])
nodes[1] = cudf.Series([10,11,12])
g = cugraph.Graph()
g.add_nodes_from(nodes[0], bipartite=True)
g.add_nodes_from(nodes[1], bipartite=True)
g.from_cudf_edgelist(edges, source='src', destination='dst', edge_attr=None)
```

- Create a bipartite graph where nodes are put into two independent sets, and no edge connection exists between nodes in the same set.

GRAPH PROPERTIES

Explore the properties of a graph.

g.degree() - Calculate how many edges each vertex has.

g.degrees() - Calculate how many incoming and outgoing edges each vertex has. These two numbers will be the same for undirected graphs and can be different in directed graphs.

g.edges() - Retrieve a cuDF DataFrame with all edges and two columns representing edge source and destination.

g.has_edge(0,1) - Check if a connection between two nodes exists (here, between node 0 and 1).

g.has_node(1) - Check if graph has a node (here, node 1).

g.in_degree() - Calculate how many incoming edges each vertex has.

g.is_bipartite() - Check if the graph is bipartite. Note that this method does not explicitly check if the graph is bipartite but rather relies on setting the bipartite parameter to True when adding nodes.

g.is_directed() - Check if the graph is directed, i.e. edges have a direction of flow.

g.is_multipartite() - Check if the graph is multipartite. Note that this method does not explicitly check if the graph is bipartite but rather relies on setting the bipartite parameter to True when adding nodes.

g.neighbors(0) - Retrieve a cuDF Series containing a full list of all the destination nodes connected to the specified node (here, node 0).

g.nodes() - Extract a cuDF Series of all the nodes in the graph.

g.number_of_edges() - Get the total number of edges in the graph.

g.number_of_nodes() - Get the total number of nodes in the graph.

g.number_of_vertices() - Get the total number of nodes in the graph.

g.out_degree() - Calculate how many outgoing edges each vertex has.

g.sets() - Retrieve a set of nodes in a bi- or multipartite graph as a list of cuDF Series. Note that this will return an empty list if the nodes were added to the graph with the bipartite and multipartite parameters set to False (default).

g.to_directed() - Convert a graph to a directed graph.

g.to_undirected() - Convert a graph to an undirected graph.

g.view_adj_list() - Retrieve a list of cuDF Series representing the adjacency list of a graph.

g.view_edge_list() - Retrieve a cuDF DataFrame with a list of all edges with their weights.

TRANSFORM GRAPH

```
g.add_internal_vertex_id(
  nodes
  , external_column_name='index'
  , internal_column_name='mapped_vertex_id'
)
```

- Retrieve internal node IDs and match them to the original ones.

TRANSFORM GRAPH

g.clear() - Clear the whole graph.

g.delete_adj_list() - Remove all the edges from your graph via removing the adjacency list.

g.delete_edge_list() - Remove all the edges from your graph via removing the edge list.

g.get_two_hop_neighbors() - Retrieve a list of all second-degree neighbors for all the nodes in the graph.

g.lookup_internal_vertex_id(cudf.Series([0,1,2,3])) - Retrieve a list of internal node IDs for the specified list of nodes.

CENTRALITY

Compute centrality metrics for a graph.

cugraph.betweenness_centrality(g, k=20, normalized=True, weight=None) - Calculate a betweenness centrality metric for every node in the graph. The betweenness centrality metric is the number of shortest paths that pass through a vertex, e.g., in a social graph, this signifies how many times a person is asked to introduce or propagate the introduction further to their connections.

cugraph.edge_betweenness_centrality(g, k=20, normalized=True, weight=None) - Calculate a betweenness centrality metric for every edge in the graph. The edge betweenness centrality is the number of times an edge is a part of a shortest path, e.g., in a social graph, this can be viewed as a particular relationship with someone who has many other relationships (a high degree in graph terms).

cugraph.katz_centrality(g, alpha=0.05, max_iter=200, tol=1e-07) - Calculate a Katz centrality metric for every vertex in the graph. The Katz centrality metric measures influence by taking into account the total number of walks between a pair of actors, e.g., in a social graph, how many different ways one person can reach another and how many times each node occurs in such travels/walks.

COMMUNITY

Find communities in a graph.

cugraph.ecg(g, min_weight=0.05, ensemble_size=20) - Compute the ensemble clustering for graphs (ECG) partition of the input graph. ECG runs a truncated Louvain method on an ensemble of permutations of the input graph to determine weights that are then used for running the Louvain method on the full graph.

cugraph.ktruss_subgraph(g, 3, use_weights=True).edges() - Return a k-truss subgraph of a graph for a specified k. A k-truss is a subgraph where each edge is part of at least (k-2) triangles. These subgraphs are used for finding close-knit groups of nodes in a graph.

parts, modularity_score = cugraph.leiden(g, max_iter=10) - Detect communities in a graph using the Leiden algorithm, which is an extension of the Louvain algorithm. The Leiden algorithm guarantees that the communities are well-connected.

parts, modularity_score = cugraph.louvain(g, max_iter=10) - Detect communities in a graph using the Louvain algorithm. It aims at optimizing the modularity; modularity measures the relative density of edges inside a community with respect to edges outside the community.

g_sub = cugraph.subgraph(g, cudf.Series([0,1,3])) - Extract a subgraph of the input graph given a cuDF Series of nodes to retrieve with their corresponding edges.

cugraph.triangles(g) - Compute the number of triangles in the input graph.

spectral = cugraph.spectralBalancedCutClustering(g, 5) - Compute a clustering/partitioning of the input graph using the spectral balanced cut method.

spectral = cugraph.spectralBalancedCutClustering(g, 5)
cugraph.analyzeClustering_edge_cut(g, 5, spectral) - Calculate the edge cut score for a spectral clustering.

spectral = cugraph.spectralBalancedCutClustering(g, 5)
cugraph.analyzeClustering_modularity(g, 5, spectral) - Calculate the modularity score for a spectral clustering.

COMMUNITY

Find communities in a graph.

spectral = cugraph.spectralBalancedCutClustering(g, 5)
cugraph.analyzeClustering_ratio_cut(g, 5, spectral) - Calculate the ratio cut score for a spectral clustering.

```
cugraph.spectralModularityMaximizationClustering(
   g
   , num_clusters=5
   , num_eigen_vects=3
   , evs_max_iter=200
   , kmean_tolerance=1e-06
   , kmean_max_iter=200
)
```
- Compute a clustering/partitioning of the input graph using the spectral modularity maximization method.

COMPONENTS

Extract strong and weak components of a graph.

cugraph.strongly_connected_components(g) - Retrieve a cuDF DataFrame of strongly connected components. Strongly connected components of a directed graph is a subset of nodes. By starting from one node, it is possible to reach any of the other nodes identified in a strongly connected components subgraph.

cugraph.weakly_connected_components(g) - Retrieve a cuDF DataFrame of weakly connected components. Weakly connected components are sets of connected nodes in an undirected graph where each node is reachable from any other node in the same set.

LINKS

Estimate quality and similarity of nodes.

cugraph.hits(g, max_iter=50) - Compute Hyperlink-Induced Topic Search (HITS) hubs and authorities. Hubs estimate the node value based on outgoing links, while authorities estimate the node value based on the incoming links.

cugraph.pagerank(g, alpha = 0.85, max_iter = 500, tol = 1.0e-05) - Compute the PageRank score for each node in a graph. PageRank counts the number and quality of the link to a node to determine how important it is.

pairs = g.get_two_hop_neighbors()
cugraph.jaccard(g, pairs) - Compute the Jaccard similarity score between each pair of vertices connected by an edge. The Jaccard similarity score is defined between two sets as the ratio of the volume of their intersection divided by the volume of their union.

cugraph.jaccard_coefficient(g) - Compute the Jaccard similarity score between each pair of vertices connected by an edge. Equivalent of cugraph.jaccard(...)

cugraph.jaccard_w(g, weights=edges['value']) - Compute the weighted Jaccard similarity score between each pair of vertices connected by an edge.

cugraph.overlap(g) - Compute the overlap coefficient between each pair of vertices connected by an edge. The overlap coefficient is defined between two sets as the ratio of the volume of their intersection divided by the smaller of their two volumes.

cugraph.overlap_coefficient(g) - Compute the overlap coefficient between each pair of vertices connected by an edge. Equivalent of cugraph.overlap(...)

cugraph.overlap_w(g, weights=edges['value']) - Compute the weighted overlap coefficient between each pair of vertices connected by an edge.

TRAVERSAL

Find different ways of traversing a graph.

cugraph.bfs(g, start=0) - Find the distances and predecessors for a breadth-first traversal of a graph. The algorithm explores all the neighbors of a node before moving to its neighbors and continuing the search.

cugraph.shortest_path(g, 0) - Compute the distance and predecessors for shortest paths from the specified source to all the vertices in the graph.