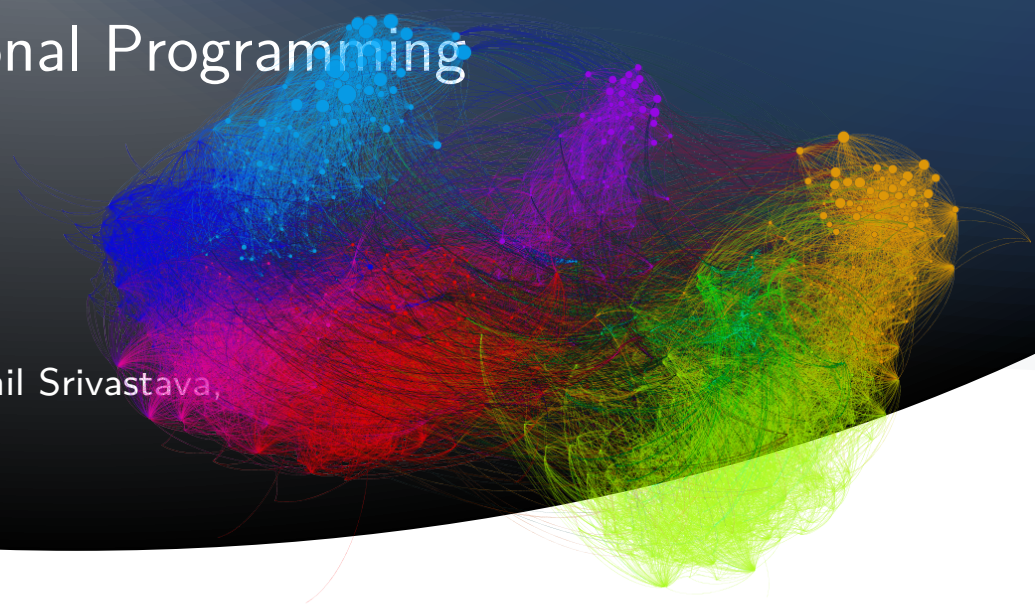
Graphs in Machine Learning

GraphLab Abstraction

Immutable Graphs and Functional Programming

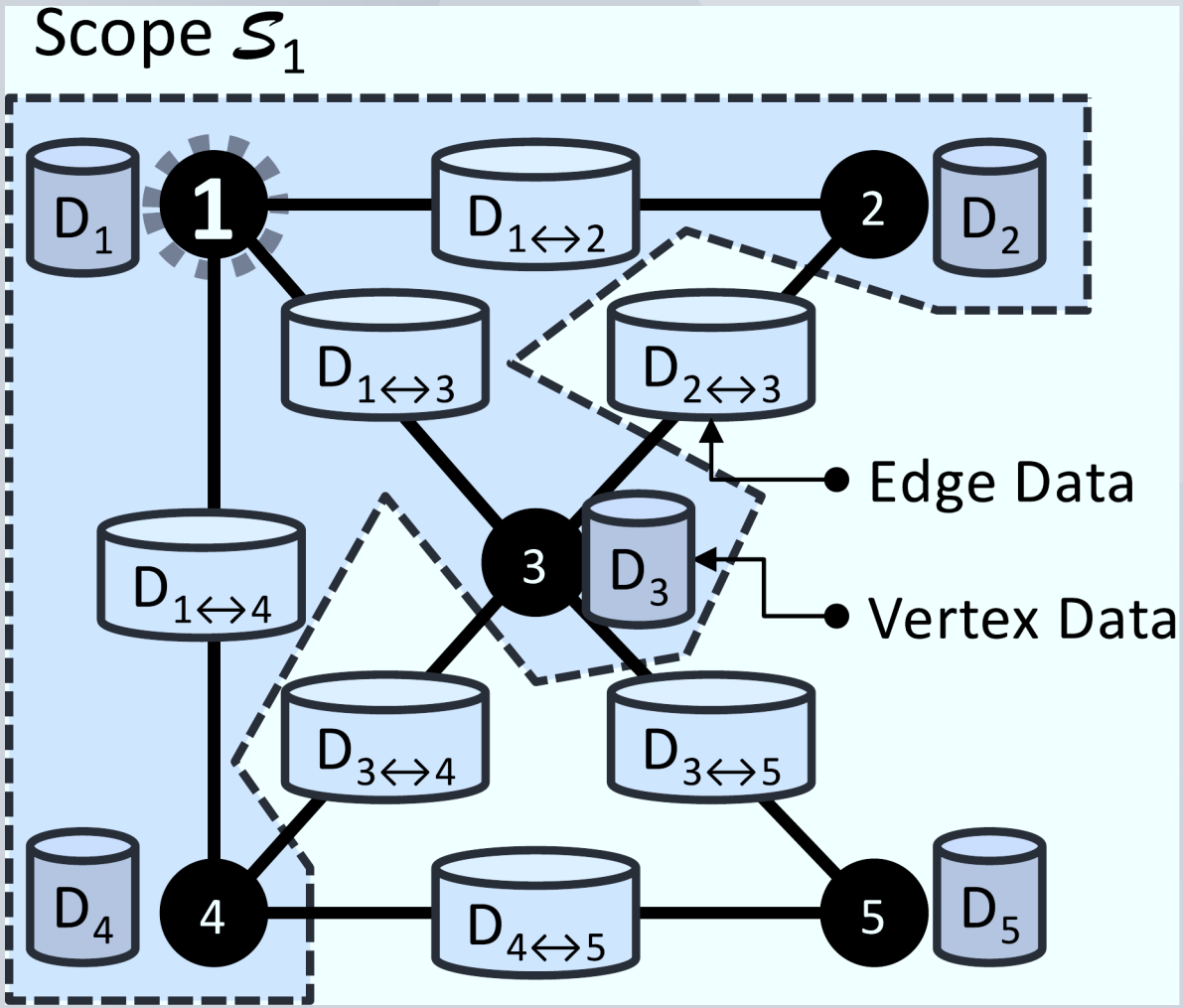
Michal Valko

Inria & ENS Paris-Saclay, MVA



Partially based on material by: Rob Fergus, Nikhil Srivastava, [illegible]is Koutis, Joshua Batson, Daniel Spielman

The GraphLab abstraction



The GraphLab abstraction

```
In [1]: import sframe

In [2]: edges = sframe.SFrame.read_csv('/media/sf_share/td3_example_edges.csv')

In [3]: vertices = sframe.SFrame.read_csv('/media/sf_share/td3_example_vertices.csv')

In [4]: G = sframe.SGraph(edges= edges, vertices=vertices, src_field='src', dst_field='dst')

In [5]: G

Out[5]: SGraph({'num_edges': 26, 'num_vertices': 9})
        Vertex Fields:['__id', 'f']
        Edge Fields:['__src_id', '__dst_id', 'weight']
```

The GraphLab abstraction

Under the hood: tabular representation

```
Columns:
    __id int
    f float

Rows: 9

Data:
+------+------+
| __id |  f   |
+------+------+
|  5   | 0.51 |
|  7   | 0.82 |
|  10  | 0.08 |
|  2   | 0.82 |
|  6   | 0.85 |
|  9   | 0.83 |
|  3   | 0.18 |
|  1   | 0.35 |
|  4   | 0.36 |
+------+------+
[9 rows x 2 columns]
```

```
Columns:
    __src_id int
    __dst_id int
    weight float

Rows: 26

Data:
+----------+----------+----------+
| __src_id | __dst_id |  weight  |
+----------+----------+----------+
|    7     |    5     | 0.13185  |
|    5     |    7     | 0.13185  |
|    7     |    7     | 0.026779 |
|    10    |    7     | 0.57121  |
|    7     |    10    | 0.57121  |
|    10    |    2     | 0.94047  |
|    7     |    6     | 0.64528  |
|    5     |    3     | 0.93374  |
|    10    |    3     | 0.31713  |
|    5     |    1     | 0.57796  |
+----------+----------+----------+
[26 rows x 3 columns]
Note: Only the head of the SFrame is printed.
```

The GraphLab abstraction

```
In [1]: import sframe

In [2]: G = sframe.SGraph()

In [3]: G

Out[3]: SGraph({'num_edges': 0, 'num_vertices': 0})
        Vertex Fields:['__id']
        Edge Fields:['__src_id', '__dst_id']
```

The GraphLab abstraction

```
In [1]: import sframe

In [2]: G = sframe.SGraph()

In [3]: G

Out[3]: SGraph({'num_edges': 0, 'num_vertices': 0})
        Vertex Fields:['__id']
        Edge Fields:['__src_id', '__dst_id']

In [4]: G.add_edges(sframe.Edge(1,2))

Out[4]: SGraph({'num_edges': 1, 'num_vertices': 2})
        Vertex Fields:['__id']
        Edge Fields:['__src_id', '__dst_id']
```

The GraphLab abstraction

```
In [1]: import sframe

In [2]: G = sframe.SGraph()

In [3]: G

Out[3]: SGraph({'num_edges': 0, 'num_vertices': 0})
        Vertex Fields:['__id']
        Edge Fields:['__src_id', '__dst_id']

In [4]: G.add_edges(sframe.Edge(1,2))

Out[4]: SGraph({'num_edges': 1, 'num_vertices': 2})
        Vertex Fields:['__id']
        Edge Fields:['__src_id', '__dst_id']

In [5]: G

Out[5]: SGraph({'num_edges': 0, 'num_vertices': 0})
        Vertex Fields:['__id']
        Edge Fields:['__src_id', '__dst_id']
```

The GraphLab abstraction

- The graph is immutable. why?

The GraphLab abstraction

- The graph is immutable. why?
- All computations are executed asyncronously

The GraphLab abstraction

- The graph is immutable. why?
- All computations are executed asyncronously
  - ↳ We do not know the order of execution

The GraphLab abstraction

- The graph is immutable. why?
- All computations are executed asyncronously
  - ↳ We do not know the order of execution
    We do not even know where the node is stored
    what data can we access?

The GraphLab abstraction

- The graph is immutable. why?
- All computations are executed asyncronously
  - ↳ We do not know the order of execution
    We do not even know where the node is stored
    what data can we access?
- The data is stored in the graph itself

The GraphLab abstraction

- The graph is immutable. why?
- All computations are executed asyncronously
  - ↳ We do not know the order of execution
    We do not even know where the node is stored
    what data can we access?
- The data is stored in the graph itself
  - ↳ only access local data

The GraphLab abstraction

- The graph is immutable. why?
- All computations are executed asyncronously
  - ↳ We do not know the order of execution
    We do not even know where the node is stored
    what data can we access?
- The data is stored in the graph itself
  - ↳ only access local data
- Functional programming approach

The GraphLab abstraction

```
triple_apply(triple_apply_fn, mutated_fields, input_fields=None)
```

processes all edges asyncronously and in parallel

```
>>> PARALLEL FOR (source, edge, target) AS triple in G:
...     LOCK (triple.source, triple.target)
...     (source, edge, target) = triple_apply_fn(triple)
...     UNLOCK (triple.source, triple.target)
... END PARALLEL FOR
```

- No guarantees on order of execution

The GraphLab abstraction

```
triple_apply(triple_apply_fn, mutated_fields, input_fields=None)
```

processes all edges asyncronously and in parallel

```
>>> PARALLEL FOR (source, edge, target) AS triple in G:
...     LOCK (triple.source, triple.target)
...     (source, edge, target) = triple_apply_fn(triple)
...     UNLOCK (triple.source, triple.target)
... END PARALLEL FOR
```

- No guarantees on order of execution
- Updating `(src,edge,dst)` violates immutability

The GraphLab abstraction

```
triple_apply(triple_apply_fn, mutated_fields, input_fields=None)
```

processes all edges asyncronously and in parallel

```
>>> PARALLEL FOR (source, edge, target) AS triple in G:
...     LOCK (triple.source, triple.target)
...     (source, edge, target) = triple_apply_fn(triple)
...     UNLOCK (triple.source, triple.target)
... END PARALLEL FOR
```

- No guarantees on order of execution
- Updating `(src,edge,dst)` violates immutability
- `triple_apply_fn` receives a copy of `(src,edge,dst)`

The GraphLab abstraction

```
triple_apply(triple_apply_fn, mutated_fields, input_fields=None)
```

processes all edges asyncronously and in parallel

```
>>> PARALLEL FOR (source, edge, target) AS triple in G:
...     LOCK (triple.source, triple.target)
...     (source, edge, target) = triple_apply_fn(triple)
...     UNLOCK (triple.source, triple.target)
... END PARALLEL FOR
```

- No guarantees on order of execution
- Updating `(src,edge,dst)` violates immutability
- `triple_apply_fn` receives a copy of `(src,edge,dst)`
  - ↳ returns an updated `(src',edge',dst')`

The GraphLab abstraction

```
triple_apply(triple_apply_fn, mutated_fields, input_fields=None)
```

processes all edges asyncronously and in parallel

```
>>> PARALLEL FOR (source, edge, target) AS triple in G:
...     LOCK (triple.source, triple.target)
...     (source, edge, target) = triple_apply_fn(triple)
...     UNLOCK (triple.source, triple.target)
... END PARALLEL FOR
```

- No guarantees on order of execution
- Updating `(src,edge,dst)` violates immutability
- `triple_apply_fn` receives a copy of `(src,edge,dst)`
  - ↳ returns an updated `(src',edge',dst')`
    use return values to build a new graph

The GraphLab abstraction

`triple_apply_fn` is a pure function

Function in the mathematical sense, same input gives same output.

```
1  def triple_apply_fn(src, edge, dst):
2      #can only access data stored in src, edge, and dst,
3      #three dictionaries containing a copy of the
4      #fields indicated in mutated_fields
5      f = dst['f']
6
7      #inputs are copies, this does not change original edge
8      edge['weight'] = g(f)
9
10     return ({'f': dst['f']}, edge, dst)
```

The GraphLab abstraction

An example, computing degree of nodes

```
1 def degree_count_fn (src, edge, dst):
2     src['degree'] += 1
3     dst['degree'] += 1
4     return (src, edge, dst)
5
6 G_count = G.triple_apply(degree_count_fn, 'degree')
```

The GraphLab abstraction

Slightly more complicated example, suboptimal pagerank

```
#assume each node in G has a field 'degree' and 'pagerank'
#initialize 'pagerank' = 1/n for all nodes

def weight_count_fn (src, edge, dst):
    dst['degree'] += edge['weight']
    return (src, edge, dst)

def pagerank_step_fn (src, edge, dst):
    dst['pagerank'] += (edge['weight']*src['pagerank']
                                      /dst['degree'])
    return (src, edge, dst)

G_pagerank = G.triple_apply(weight_count_fn, 'degree')

while not converged(G_pagerank):
    G_pagerank = G_pagerank.triple_apply(
                          pagerank_step_fn, 'pagerank')
```

The GraphLab abstraction

Slightly more complicated example, suboptimal pagerank

```
#assume each node in G has a field 'degree' and 'pagerank'
#initialize 'pagerank' = 1/n for all nodes

def weight_count_fn (src, edge, dst):
    dst['degree'] += edge['weight']
    return (src, edge, dst)

def pagerank_step_fn (src, edge, dst):
    dst['pagerank'] += (edge['weight']*src['pagerank']
                                      /dst['degree'])
    return (src, edge, dst)

G_pagerank = G.triple_apply(weight_count_fn, 'degree')

while not converged(G_pagerank):
    G_pagerank = G_pagerank.triple_apply(
                          pagerank_step_fn, 'pagerank')
```

How many iterations to convergence?

Bibliography I

Michal Valko

michal.valko@inria.fr

Inria & ENS Paris-Saclay, MVA

https://misovalko.github.io/mva-ml-graphs.html