G1: A High-Level Domain-Specific Language for Graph Algorithms

Samuel Triest

2019-04-27

1 Abstract

Graphs and efficient graph algorithms are a fundamental component of computer science that underlie many important problem domains, including network analysis, robotics, and others. As such, extensive research has been done in graph theory, both in the design of efficient algorithms and efficient implementations of those algorithms on modern hardware. However, while languages and accelerators [1, 2] exist to execute these algorithms on GPUs and specialized hardware, these methods are intrinsically tied to their respective hardware, leading to code that is difficult to implement efficiently, and difficult to port to more effective architectures as they become available.

As such, the goal of this research is to identify a set of fundamental graph primitives that are simple enough to be easily understood, but expressive enough to represent state-of-the-art implementations of graph algorithms, such as near-far shortest-path [3], concurrent connected components [4], and others. Such graph primitives are able to form the basis of a high-level language for graph algorithms. In addition to being simpler to write programs in, such a language would have the distinct advantage of being orthogonal to the underlying hardware, making it portable across current and future hardware backends.

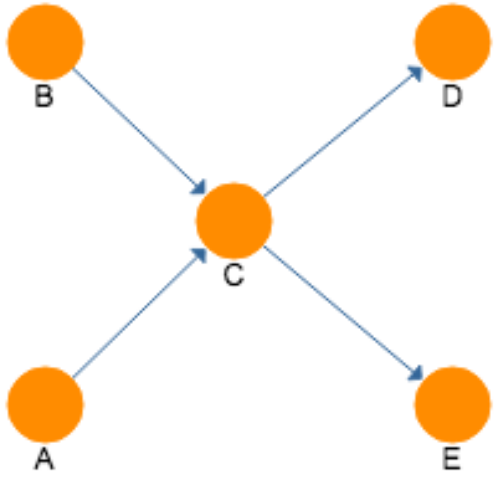
Contained in this report is the progress I've made on this research for my CSC200H class, including a description of the form that the primitives will take, a current list of primitives, and a prototype of a language and abstract syntax tree that can port these primitives into efficient GPU implementations.

2 Representation of Primitives and Graphs

As this language is designed to be high-level and independent of the underlying hardware, we make no assumptions as to the representation of the graph. Rather, we consider a graph as an abstract collection of edges and vertices ($G = (V, E)$), where an edge is a 2-tuple of vertices (and an optional weight). Additionally, many graph algorithms require the ability to store various information at a graph's vertices (e.g. whether or not a vertex has been explored in BFS, or current best distance in SSSP). As such, we will assume vertices have a predefined set of "properties" that can be referenced by the primitives.

In this form, graph primitives can be represented as functions on collections of vertices or edges that output other collections. This form has been explored by Pai et. al. [5], and has shown to be sufficient for breadth-first search.

To give an example of how these primitives are used, consider the following graph:



(a) Sample Graph

Consider a function **outneighbors(V)** that retrieves the out-neighbors of a given vertex. Clearly, **outneighbors(A)**={C} and **outneighbors(C)**={D, E}. We can also find all vertices two hops away from A by finding **outneighbors(outneighbors(A))**. That is, by repeatedly applying a function to the output of previous functions, we can perform various inquiries on graphs. The next sections will present a more complete set of functions/primitives, and express various algorithms as applications of these primitives.

3 A Representative Set of Algorithms and Their Decomposition into Primitives

In order to show completeness of the given set of primitives, the following algorithms will be considered as a representative set:

1) Breadth-first search
2) Single-source shortest path (Bellman-Ford and near-far)
3) Connected components
4) K-core decomposition

3.1 Breadth-first Search

Breadth-first search is considered as it is a simple traversal algorithm that is commonly used as a performance benchmark in graph processing. Breadth-first search is a procedure that starts with a frontier containing a root node, and iteratively expands the frontier to the out-neighbors of the current frontier until all vertices reachable from the root have been explored. Breadth-first search has a relatively straightforward mapping to the series of graph primitives, as it is essentially a repeated application of out-neighbors and filtering.

3.2 Single-source Shortest Path

Single-source shortest path (SSSP) is an class of algorithms that determine the minimum path length between a source vertex and all other vertices in a weighted graph. For this project, I will be considering Bellman-Ford and near-far; two implementations of SSSP. Both algorithms work by repeated application of relaxation steps, where

when considering an edge (u, v), dist(v) = min(dist(v), dist(u) + weight(u, v)). Central to these algorithms are a graph traversal (like in BFS), as well as the ability to have defined behavior for updates to the same vertex in parallel.

3.3 Connected Components

Connected components algorithms are used to determine which vertices in an undirected graph can reach each other. When considering a parallel implementation of connected components, the standard algorithm is a variant of the union-find procedure to assign a representative vertex to each component, where each vertex points to exactly one representative vertex. If two components happen to share an edge, one representative vertex will point to the other, thus connecting the components. After a number of steps, the algorithm will converge to the correct set of components.

This algorithm is considered especially because it is not a traversal, but rather operates on the graph as a whole for a number of iterations. This algorithmic form is central to other algorithms such as maximum independent set and minimum spanning tree.

3.4 K-core Decomposition

K-core decomposition relies on the following definition of a k-core:

A vertex is a k-core iff. it has at least k neighbors of degree k or more.

The algorithm for k-core decomposition works by pruning the 1-cores from the graph, then the 2-cores, 3-cores, etc. until there are no vertices remaining. This pruning procedure differs significantly from the previous algorithms in that it is not a traversal, and relies heavily on preserving information at previous pruning stages. Many other network analysis algorithms, such as PageRank [6], rely on this preservation of information as well.

4 Graph Primitives

The set of graph primitives explored in this paper are generally functions that can be applied to a given graph. These primitives will typically take in some collection of graph elements (such as edges and vertices), and output some other collection of graph elements. In abstraction, these primitives bear a significant resemblance to the streams library in Java 10.

These primitives follow a hierarchy of abstraction: Algorithm → Graph → Graph Element

4.1 Algorithm-Level Primitives

Algorithm-level primitives are the highest level of abstraction. These primitives need not consider a specific graph and instead deal with preprocessing of graph elements and algorithm execution.

Primitive	Input	Output	Description
evaluate	Graph G	Graph G´	Executes the algorithm on Graph G
add_property	Graph G, String name	Graph G´	Adds the property 'name' to all vertices in G

Table 1: Algorithm-level Primitives

4.2 Graph-Level Primitives

Graph-level primitives operate at the graph level. That is, graph primitives can be used to get information about the graph, but make no reference to graph elements themselves. In general, these primitives are used to generate collections of graph elements to process by graph element-level primitives. Note that while no guarantee is made as to the specific ordering of these collections, the primitives should guarantee a consistent ordering per graph.

Primitive	Input	Output	Description
all_vertices	Graph G	Vertex[]	Returns a collection containing all vertices in G
random_vertex	Graph G, Int k	Vertex[]	Returns a collection of k vertices in G, chosen at random
pick_vertex	Graph G, String id	Vertex[]	Returns a singleton collection containing the vertex with the given id, $\emptyset$ if no such vertex exists
all_edges	Graph G	Edge[]	Returns a collection of all edges in G
random_edge	Graph G, Int k	Edge[]	Returns a collection of k edges in G, chosen at random
pick_edge	Graph G, String id	Edge[]	Returns a singleton collection containing the edge with the given id, $\emptyset$ if no such edge exists
n_V	Graph G	Int N	Returns the number of vertices in G (scalar)
n_E	Graph G, String name	Int N	Returns the number of edges in G (scalar)

Table 2: Graph-level Primitives

4.3 Graph Element-Level Primitives

Graph element-level primitives operate at the level of vertices and edges. These primitives are generally responsible for the majority of computation in graph algorithms.

Primitive	Input	Output	Description
edges	Vertex[] v	Edge[]	Returns all edges of the form: $< v, - >$
out_neighbors	Vertex[] v	Vertex[]	Returns the out-neighbors of all vertices in v
in_neighbors	Vertex[] v	Vertex[]	Returns the in-neighbors of all vertices in v
filter	Vertex[] v, F:Vertex → Boolean	Vertex[]	Returns all vertices in v where $F(v)$ is true
broadcast	Vertex[] v, Object x, String p	Vertex[]	Assigns x to the property p for all vertices in v
update	Vertex[] v, Object[] x, String p	Vertex[]	Assigns the i-th value in x to property p of the i-th vertex in v. Note that there is no defined behavior if there are duplicate v.
compare_and_update	Vertex[] v, Object[] x, String p, Cmp:Object → Int	Vertex[]	Assigns the i-th value in x to property p of the i-th vertex in v, but only if the comparator value of the object is less than the current property for v. This function has defined behavior for duplicate v.
src	Edge[] e	Vertex[]	Returns a collection of all vertices that the edges in e point from
dst	Edge[] e	Vertex[]	Returns a collection of all vertices that the edges in e point to
filter	Edge[] e, F:Edge → Boolean	Edge[]	Returns all vertices in e where $F(e)$ is true

Table 3: Graph Element-level Primitives

4.4 Worklist-Level Primitives

Worklist-level primitives typically deal with storing the output of primitives in memory for future primitives to call. Worklists are a fundamental abstraction in many GPU implementations of efficient algorithms, as a large number of graph algorithms work on only a subsection of the graph at a time. Things like the frontier in BFS, or active edges in connected components are implemented easily as worklists. As such, this language supports worklists as a fundamental abstraction.

Primitive	Input	Output	Description
→	Worklist w	Object[]	Outputs the contents of the worklist
←	Object[] o	Worklist	Appends the contents of o to the worklist
:=	Object[] o	Worklist	Replaces the contents of the worklist with o

Table 4: Worklist-level Primitives

5 Algorithm Decompositions

Below are implementations of the representative set of algorithms as graph primitives. They are implemented in Python, where each primitive is a Python function.

5.1 Breadth-First Search

Breath-first search is a fundamental graph algorithm that repeatedly expands a frontier of vertices along edges until the frontier is empty. In this decomposition, we first denote all vertices as unexplored by setting its `level` property to -1. We then populate the frontier as a worklist with a single random vertex. BFS converges when this worklist is empty. At each iteration, we want to populate the worklist with all unexplored neighbors of the current worklist. This is achieved by first getting all out neighbors of the worklist and filtering. We then denote these vertices as explored by replacing updating their `level` property.

```
def BFS(G):

#init
 broadcast(G, all_vertices(G), -1)

 wl = broadcast(G, random_vertex(G), 0)

 while wl: #convergence condition
   #kernel
     p_prev = wl[0].prop
     wl = broadcast(G, vertex_filter(G, out_neighbors(G, wl),lambda v: v.prop == -1),
         p_prev + 1)
```

5.2 Bellman-Ford SSSP

Bellman-Ford is the classical example of a parallel SSSP algorithm. In contrast to Dijkstra's algorithm, which is highly serialized due to its priority queue, Bellman-Ford relies on each vertex streaming its current best distance to its out-neighbors. If that best distance (plus the edge weight) is less than the recipient's current best distance, the recipient will update its best distance. In the absence of negative cycles, Bellman-Ford will converge in at most V iterations. Otherwise, a lack of convergence can be detected on the $(V+1)$-th cycle.

In primitives, we first initialize each vertex with properties for current distance and previous vertex in path. This will always start as (∞, v) for all vertices v in the graph. The source vertex will start as (0, v). We then execute the distance streaming step for V iterations. Key to this decomposition is the `compare_and_update` primitive, which utilizes a comparator to make sure the smallest update is applied when multiple updates stream to the same vertex.

```
def SSSP_bf(G, start):
#init
 update(G, all_vertices(G), zip([float('inf')]*len(all_vertices(G)), all_vertices(G)))
 broadcast(G, pick_vertex(G, start), (0, pick_vertex(G, start)[0]))

 wl = all_edges(G)
 cnt = len(all_vertices(G)) - 1
 while cnt > 0: #convergence condition
   #kernel
     cnt -= 1
     compare_and_update(G, [e.v for e in wl], [(e.u.prop[0] + e.w, e.u) for e in wl],
         cmp = lambda p1, p2: p1[0] < p2[0])
```

5.3 Near-Far SSSP

SSSP near-far is an efficient parallel SSSP algorithm that is distinct from Bellman-Ford in two significant ways:

1) Instead of processing each vertex at each iteration, only process vertices that were updated in the previous iteration, as those vertices are the only ones that can cause updates.

2) Bin these updates into a "near" and "far" bins based on some threshold for current best distance. Only process updates in the near bin, as any solution found by processing the near bin will be better than any solution found by processing the far bin. If the near bin is empty, increase the threshold and move updates in the far bin below the threshold into the near bin. This prevents potentially redundant updates to large portions of the graph, as lower-weighted updates will propagate first.

Convergence is achieved when both the near and far bins are empty.

A key consequence of this algorithm is the necessity of a sub-kernel, where the main loop requires part of its code to be executed repeatedly until convergence. In Python, this is implemented using nested loops, but must be accounted for in the final language implementation.

```
def SSSP_nf(G, start, d):
   #init
    update(G, all_vertices(G), zip([float('inf')] * len(all_vertices(G)),
        all_vertices(G)))
    broadcast(G, pick_vertex(G, start), (0, pick_vertex(G, start)[0]))

    near = pick_vertex(G, start)
    far = []

    delta = d
    i = 1
    while near or far: #convergence condition
      #kernel
        while near: #subconvergence condition
         #subkernel
            v_update = compare_and_update(G, [e.v for e in out_edges(G, near)],
                [(e.u.prop[0] + e.w, e.u) for e in out_edges(G, near)], cmp = lambda p1,
                p2: p1[0] < p2[0])
            near = vertex_filter(G, v_update, lambda v: v.prop[0] <= delta)
            far.extend(vertex_filter(G, v_update, lambda v: v.prop[0] > delta))
            i += 1
        delta += d
        near = vertex_filter(G, far, lambda v: v.prop[0] <= delta)
        far = vertex_filter(G, far, lambda v: v.prop[0] > delta)
```

5.4 Connected Components

Connected components is implemented in parallel as a variant of the union-find procedure. That is, each component has a representative vertex (typically the vertex in the cluster with the minimum label, though this is arbitrary). If two components are found to be connected, the smaller of the two representative vertices becomes the representative vertex of the other. This essentially creates a tree out of vertices in a component, which is periodically flattened (pointer jumping).

In primitives, each vertex is first defined to be in its own component, with itself as its representative. All edges are passed into a worklist. In an iteration, all edges in the worklist are considered, and if an edge spans two components (the src and dst of the edge have differing representative vertices), the components are coalesced. If an edge does not span a component, it can be safely discarded. Convergence is achieved when there are no edges left in the worklist.

```
def CC(G):
   #init
    update(G, all_vertices(G), all_vertices(G))
    wl = all_edges(G)
    prev = False
    while wl: #convergence condition
      #kernel
        prev = not prev
        wl = edge_filter(G, wl, lambda e: e.u.prop != e.v.prop)
        if prev:
            compare_and_update(G, [e.u.prop for e in wl], [e.v.prop for e in wl], cmp =
                lambda u, v: u.prop < v.prop)
        else:
            compare_and_update(G, [e.u.prop for e in wl], [e.v.prop for e in wl], cmp =
                lambda u, v: u.prop > v.prop)

        update(G, all_vertices(G), [v.prop.prop for v in all_vertices(G)])
    print(G)
```

5.5 K-Core Decomposition

K-Core decomposition is a network analysis algorithm that approximates the connectedness of vertices in a graph. Given the definition of a k-core as a vertex with k or more neighbors of degree k or more, k-core decomposition identifies the maximal k for each vertex. This is done by finding vertices below a given "soreness" and pruning them.

As primitives, all vertices have a property describing their "soreness", and a property indicating if they have pruned. Finding k for each vertex is achieved by gradually increasing a threshold, and filtering out vertices who have fewer non-pruned out-neighbors than the threshold. All remaining vertices can be assigned a "soreness" equal to the threshold. Convergence can be attained when the threshold hits V, or if no vertices are left. Like near-far, k-core also requires sub-kernels.

```
    #Need to rewrite this using a worklist for all vertices and another for current
       level.

def KCore(G):
   #init
    update(G, all_vertices(G), zip([True]*num_v(G), [0]*num_v(G)))

    for level in range(num_v(G)): #convergence condition
      #kernel
        while True: #subconvergence condition
         #subkernel
            updates = vertex_filter(G,
                                    vertex_filter(G, all_vertices(G), lambda v: v.prop[0]),
                                    lambda v: len(vertex_filter(G, out_neighbors(G, [v]),
                                        lambda u: u.prop[0])) <= level)
            broadcast(G, updates, (False, level))
            if not updates:
                break
```

6 Language Prototype

6.1 Graph Algorithms as Applications of Kernels

In general, it seems that the majority of graph algorithms can be expressed as an instantiation of a general fixed-point iteration algorithm. That is, all the representative algorithms can be expressed in the following form:

```
G = (V, E)
K = A list of graph functions
init()
while !F(G) do
    for k ϵK do
        k(G)
    end
end
```

Algorithm 1: General graph algorithm

Essentially, most graph algorithms are comprised of an initialization step, followed by repeated applications of the same functions until some convergence condition is reached. In BFS, for example, **init()** is setting all vertices to be unexplored, and adding the start vertex to the worklist, **F** is a check for elements in the worklist, and **k** is an exploration of neighbors in the worklist. An implementation of this language in the following form is provided below.

As mentioned in the previous sections, some of the algorithms in the representative set require the use of sub-kernels. In order to account for this, we allow **k(G)** to call other elements in the set of kernels, provided that those kernels have their own convergence conditions.

6.2 Kernels as Repeated Applications of Primitives

In general, each primitive takes a collection of graph elements as input, and out outputs another collection of graph elements. As such, we can treat the collection of graph elements as an implicit parameter in these primitive calls. In order to simplify notation, and to take advantage of the structure of repeated primitive calls, we choose to use the $\rightarrow$ symbol to denote that the output of the primitive should be used as the input to the next primitive.

6.3 Implementation of Representative Set in G1

Provided below are implementations of the representative algorithm set in G1. The implementations follow quite clearly from the Python implementations, as code outside the while loop is mapped to `\_init`, the termination condition of the while loop maps to `\_F`, and the code inside the while loop maps to various kernels.

```
Algorithm BFS {
    _worklists = [wl]
    _graph = g
    _kernels = [bfs_kernel]
    _F = wl -> isEmpty()
    _init = {
        all_vertices() -> broadcast(-1, 'level')
    }

    Kernel bfs_kernel {
        wl <- (wl -> out_neighbors -> filter(lambda v: v -> get('level') == -1) ->
            broadcast(wl[0] + 1, 'level'))
    }
}
```

```
Algorithm ConnectedComponents {
    _worklists = [wl]
    _graph = g
    _kernels = [root_find, pointer_jump]
    _F = wl -> isEmpty()
    _init = {
        all_vertices() -> update(all_vertices(), 'root')
        wl <- all_edges()
    }

    Kernel root_find{
        wl <- edge_filter(lambda e: src(e) -> get('root') != dst(e) -> get('root'))
        compare_and_update(wl -> src(), wl -> dst() -> get('root'), lambda u, v: u ->
            get('root') > v)
    }

    Kernel pointer_jump{
        all_vertices() -> update(all_vertices() -> get('root') -> get('root'), 'root')
    }
}
```

```
Algorithm KCore {
    _worklists = [wl, updates]
    _graph = g
    _kernels = [kcore]
    _F = level == num_V() # or no vertices are active
    _init = {
        all_vertices() -> update(0, 'coreness')
        all_vertices() -> update(True, 'active')
        level = 0
    }

    Kernel kcore{
        update()
        level += 1
    }

    Subkernel update{
        _F {
            updates -> isEmpty()
        }
        updates = all_vertices -> filter(lambda v: v -> get('active')) -> filter(lambda
            v: v -> out_neighbors() -> filter(lambda u: u -> get('active')) -> size() >
            level)
        updates -> update(level) -> update(False, 'active')
    }


}
```

```
Algorithm SSSP_nf(start) {
    _worklists = [near, far, v_update]
    _graph = g
    _kernels = [nf_kernel]
    _F = (near -> isEmpty()) && (far -> isEmpty())
    _init = {
        all_vertices() -> update(all_vertices(), 'prev')
        all_vertices() -> update(inf, 'prev')
        pick_vertex(start) -> update(0, 'dist')

        near <- pick_vertex(start)
        delta = d

    }

    Kernel nf_kernel{
        n_kernel()
        delta += d
        near <- v_update -> filter(lambda x: x < delta)
        far <- v_update -> filter(lambda x: x >= delta)
    }

    Subkernel n_kernel{
        _F {
            near -> isEmpty()
        }
        _Kernel {
            v_update <- compare_and_update(near -> out_neighbors() -> get('dist'), near
                -> edges() -> dst() -> get('dist') + near -> edges() -> weight(), lambda
                u, v: u[0] < v[0])
            near <- v_update -> filter(lambda x: x < delta)
            far <- v_update -> filter(lambda x: x >= delta)
        }
    }
}
```

7 Abstract Syntax Tree

The abstract syntax tree for this language follows quite readily from the language design mentioned in the previous section. Since each statement in this language is a repeated application of primitives to some initial collection, we can treat the initial collection as a leaf node, and parse the statement such that a node's parent is the next primitive in the sequence. Some primitives such as `filter` that accept arbitrary lambdas need extra computation branches. However, their initial collections need to be passed from the leftmost branch of the parse tree. That is, the collection that the lambda refers to is the leftmost child of the lambda node (which is a collection, as all primitives output collections). This collection can easily be referred to using an LL parse of the AST.

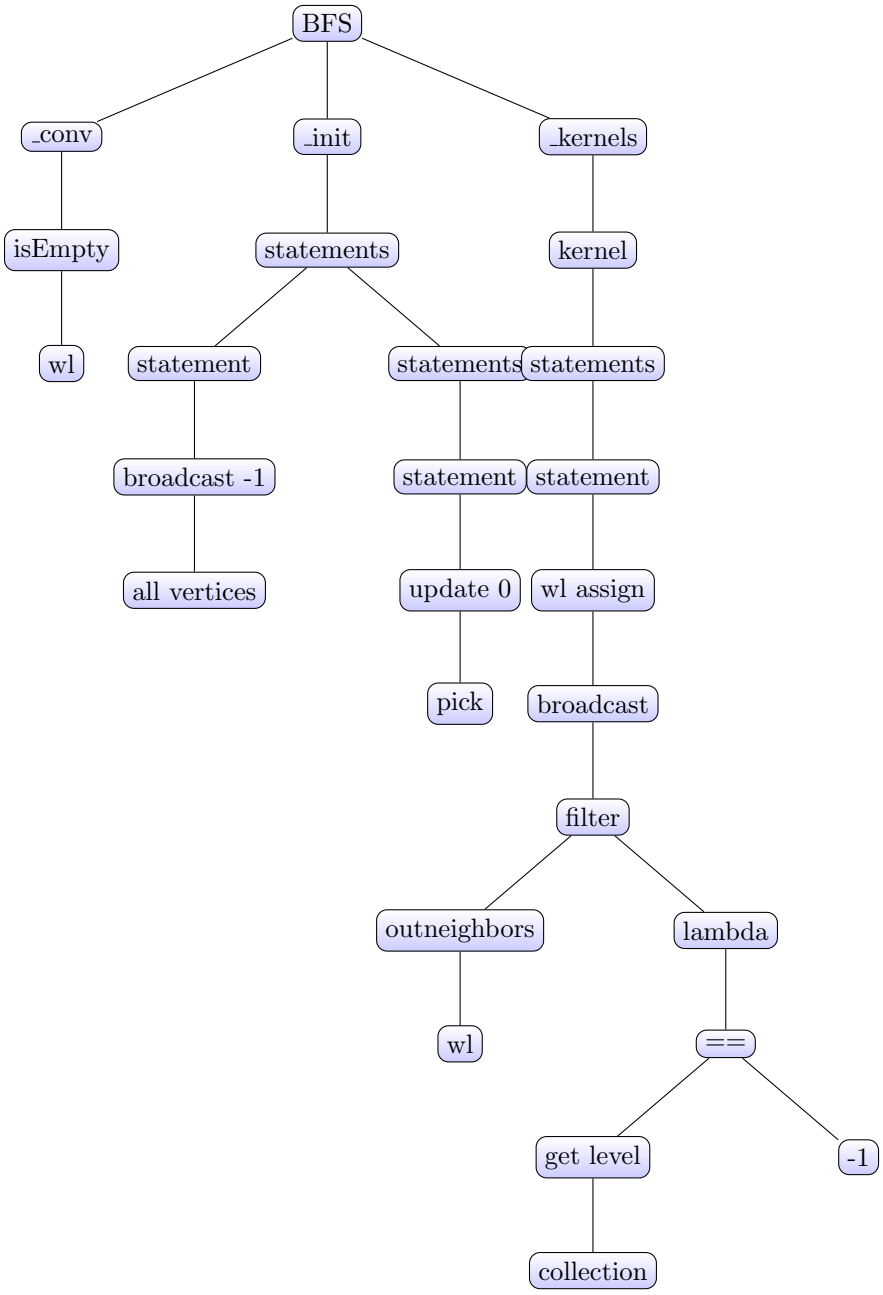


Figure 2: AST for BFS (the collection at the rightmost banch of the lambda is the "outneighbors" node)

8 Future Work

Future work on this project will primarily include GPU implementations of these primitives, and benchmarking of performance as compared to tother graph processing software.

References

[1] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *ACM SIGPLAN Notices*, volume 51, pages 1–19. ACM, 2016.

[2] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[3] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE, 2014.

[4] Sixue Liu and Robert E Tarjan. Simple concurrent labeling algorithms for connected components. *arXiv preprint arXiv:1812.06177*, 2018.

[5] Sreepathi Pai, M Amber Hassaan, and Keshav Pingali. An operational performance model of breadth-first search. In *Proceedings of the 1st International Workshop on Architecture for Graph Processing*, 2017.

[6] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.