Chapter 1

Graph Exploration and Breadth First Search

Contributed by Peter Kogge

1.1 Introduction

A common step within many graph applications is to "explore" the reachable region around some vertex to identify some "community." This exploration involves following edges from vertexes already identified and identifying when new vertexes are part of the desired community. A typical exploration application may be a variation of finding a *spanning tree*, where starting at some *root vertex* we report all reachable vertices, but with no repeats.

One example may involve searching air flights. For some airport, a query may be to generate a list of all other reachable airports, optionally with the minimum number of hops needed, or using just some subset of possible airlines, only traversing through some maximum number of countries, or only using certain classes of equipment.

Another example exploration problem is the classic *Six Degrees of Kevin Bacon*[1] problem, which found, using the Internet Movie Database (IMDb)[2] of actors and movies they worked in, that every other actor had at most five movies between him/herself and Kevin Bacon. There is a website[3] that implements a version of this search starting and ending at any arbitrary actor, and returning the number of movie links between them.

At least two basic graph explorations exist: **depth first** and **breadth first**. The former starts at a root vertex and, following edges, dives as deep in a graph as it can, stopping only when no further vertexes can be reached. In this case, the search backs up to the vertex explored just prior to the current one and tries again on alternate edges. The latter keeps a "frontier" of newly touched vertexes, and looks only one level deeper from all of them before starting with a new frontier. This latter, **Breadth First Search (BFS)**, is the kernel investigated in this paper.

Other possible exploration problems[4] for which BFS is applicatle include:

- Search for neighbors in peer-peer networks
- Search engine web crawlers
- Social networks distance k friends

[1]https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

[2]https://www.imdb.com/

[3]http://oracleofbacon.org/

[4]https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/

- GPS navigation to find neighboring locations
- Patterns for broadcasting in networks

From Wikipedia[5] comes the following additional BFS-related problems problems:

- Community Detection
- Maze running
- Routing of wires in circuits
- Finding Connected components
- Copying garbage collection, Cheney's algorithm
- Shortest path between two nodes u and v
- CuthillMcKee mesh numbering
- Maximum flow in a flow network
- Serialization/Deserialization of a binary tree
- Construction of the failure function of the Aho-Corasick pattern matcher.
- Testing bipartiteness of a graph

1.2 The Problem as a Graph

All of these problems can be expressed as graphs with different characteristics. Each vertex represents some sort of entity and each edge some relationship between them. As discussed in Section 1.3 such graphs can become very large, they can have multiple classes of vertexes and edges, and that search criteria may limit which vertexes and/or edges are candidates for inclusion in an exploration.

1.3 Some Realistic Data Sets

There are a wide variety of data sets that are relevant. For example, for the airport routing problem there are over 17,000 airports, over 5,000 airlines, and over 100,000 flights a day. Thus a valid graph representation would include over 17,000 vertices and at least 100,000 edges with about 5,000 different airline labels on them. The vertexes may have different properties, such as what country they are in, and/or do they have an international designation. Edges may also have additional properties such as: flight number, distance, equipment type, etc. These edges are directional, in that a flight possible from u to v does not guarantee that there is a flight from v to u.

A different kind of graph can be seen in the IMDB graph[6] where there are multiple classes of vertexes (8,702,001 people, and 4,734,693 titles of 10 types) and multiple classes of again directed edges (34 different roles people could have in a title).

Even bigger graphs are possible. Consider for example that there are over 3 billion human DNA base pairs. Even bigger, consider a graph of every person who has ever lived (estimated to be about 107 billion), every address they have ever called home, every contact they have ever made, who their relatives are, what did they purchase, etc.

1.4 BFS-A Key Graph Kernel

The range of applications for a breadth first search kernel is quite large, with a variety of different kinds of search options and termination criteria. By far, however, the most widely referenced

[5]https://en.wikipedia.org/wiki/Breadth-first_search

[6]https://www.imdb.com/pressroom/stats/

Class	Scale	N (Billions)	M (Billions)	Size (TB)
Toy	26	0.064	1	0.017
Mini	29	0.5	4	0.14
Small	32	4	64	1.1
Medium	36	64	1024	17.6
Large	39	512	4,096	140.8
Huge	42	4,096	65,000	1126
Edgefactor is 16 for all these classes. Size assumes vertex and its edges take about 282B				

Table 1.1: GRAPH500 Graph Classes.

variant is the one defined and used as a benchmark by the GRAPH500 website, and implemented with a wide range of algorithms on a wide range of hardware platforms. Literally thousands of performance reports are available for it.

The benchmark definition of the BFS kernel as defined in GRAPH500 [1] assumes as input an undirected graph along with a root vertex. A valid implementation will assign an integer label to each vertex in the graph, with a value representing the minimum number of edges that reach it via a path from the root vertex. The root's label is "0." If a vertex is unreachable from the root, its label is left at some arbitrarily large number.

1.4.1 Input Graph Construction

The GRAPH500 defines a spectrum of graphs that are suitable for use in submissions as GRAPH500 performance reports. Such graphs have only one class of N vertexes, and all M edges are undirected (so if (u,v) is an edge, so is (v,u)). The generator for a GRAPH500-compatible graph, has two parameters:

- Scale: The base two logarithm of the number of vertices. Thus $N = 2^Scale$.
- Edgefactor: the ratio of the edge count to the vertex count. Thus the average degree of a vertex is $2 * Edgefactor$. For GRAPH500 this value is set to 16. Thus $M = Edgefactor * N = 16N$

A typical graph generator is based on the R-MAT scale-free graph generator algorithm [3, 6, 5]. A first-pass untimed part of the GRAPH500 benchmark removes duplicate edges and self-loops, and may reformat the graph description in whatever format is best for the chosen BFS algorithm, with some limitations to assure unbiased performance results. The GRAPH500 website includes several reference codes for generating such matrices.

GRAPH500 defines a variety of graph "size ranges" that are used in comparing different implementations, as summarized in Table 1.1.

1.4.2 Timing and Performance Metrics

The performance metric selected by GRAPH500 is **Traversed Edges per Second** or **TEPS**, which is roughly computed by the number of edges in the graph divided by the execution time.

Timing for a standardized execution of a GRAPH500 implementation consists of taking the average of 64 runs of the kernel on the same graph, with a different randomly-selected root used for each run.

1.4.3 Typical Algorithms

There are at least two major algorithms for implementing the GRAPH500 BFS kernel, and a hybrid that combines the two. Most modern implementations use the hybrid.

1.4.3.1 Top Down BFS

The **Forward BFS** algorithm recursively tracks the current "frontier" of newly touched vertexes, and explores just those vertexes on each pass. The level label assigned to all newly touched vertexes on a pass is one more than the level assigned to the vertexes in the frontier. Algorithm 1 outlines such an algorithm, with the procedure *TopDownPass* performing the inner loop. The labels for the vertices are assumed to be a large vector indexed by vertex name, but keeping track of the frontier and the already touched vertexes is described as just set operations (clearly a real world implementation would use a more concrete data structure).

Algorithm 1 Top Down BFS:
V is the set of vertices; E a set of edges

```
 1: procedure TOPDOWN-BFS(G,ROOT)
 2:     $Touched \leftarrow \{root\}$
 3:     $Frontier \leftarrow \{root\}$
 4:     $Labels \leftarrow$ N-vector of a large integer
 5:     $Label[root] \leftarrow 0$
 6:     $Level \leftarrow 0$
 7:     while $Frontier$ not empty do
 8:         $Level += 1$
 9:         $TopDown - Pass(Frontier,\ Touched,\ Level)$X
10:     return
11: procedure TOPDOWNPASS(TOUCHED, LEVEL)
12:     $Next \leftarrow \{\}$
13:     for $u$ in $Frontier$ do
14:         for all edges $(u, v)$ in E do
15:             if v not in $Touched$ then
16:                 $Touched \leftarrow Touched \cup \{v\}$
17:                 $Next \leftarrow Next \cup \{v\}$
18:                 $Label[v] \leftarrow Level$
19:     $Frontier \leftarrow Next$
20:     return
```

In terms of complexity, each vertex is added at most once to *Touched*, and thus at most once to *Frontier*. This means that each edge in the innermost loop, line 14, is referenced at most once. If the complexity of the other operations are constant, then the time complexity of this algorithm is $O(M)$. Likewise, the space complexity is $O(N + M)$.

1.4.3.2 Bottom Up BFS

As shown in Algorithm 2, the **Backward BFS** algorithm runs the other way. At each iteration of the inner loop (procedure *BottonUpPass*), all non-touched vertexes are scanned to see if there is any edge from a touched vertex to them. If so, they are marked as touched.

In terms of complexity, it is possible for each iteration to add at most one vertex to the touched set, so that the inner procedure is called N times, with the inner loop executing over all the remaining vertexes. This results in a complexity of as high as $O(NM)$. As before, the particulars of individual tests, especially the test for v not in $Touched$ in line 13, can complicate and increase this complexity.

Algorithm 2 BottomUp BFS:
V is the set of vertices; E a set of edges

```
 1: procedure BottomUp-BFS(G,root)
 2:     $Touched \leftarrow \{root\}$
 3:     $Labels \leftarrow$ N-vector of a large integer
 4:     $Label[root] \leftarrow 0$
 5:     $Level \leftarrow 0$
 6:     $TouchedFlag \leftarrow True$
 7:     while $TouchedFlag$ do
 8:         $Level += 1$
 9:         $TouchedFlag \leftarrow BackwardPass(Touched, Level)$
10:     return
11: procedure BottomUpPass(inout Touched, Level)
12:     $TouchedFlag \leftarrow False$
13:     for $v$ not in $Touched$ do
14:         for all edges $(u, v)$ in E do
15:             if u in $Touched$ then
16:                 $TouchedFlag \leftarrow True$
17:                 $Touched \leftarrow Touched \cup v$
18:                 $Label[v] \leftarrow Level$
19:     return $TouchedFlag$
```

1.4.3.3 Hybrid Algorithm

An observation by Beamer [2] was that when exploring bottom-up, as soon as any edge incoming to a vertex v is found to come from some previously touched vertex, then v can be marked as "touched," with no other edges to v needing to be explored. This is in contrast to the top-down pass where all edges from a frontier u must be explored. This can represent a significant reduction in edges explored, and thus possibly in time.

Beamer then used this observation to design a hybrid algorithm that switches from top-down to bottom-up when the number of edges from the new frontier is greater than the number of edges into as yet as untouched vertexes, and then switches back when the number of vertexes on the frontier becomes relatively small.

Following the notation from Beamer, we define the following terms:

- N_f = the number of vertices on the current frontier,
- M_f = the number of edges exiting from vertexes in the current frontier,
- M_u = the number of edges into vertexes that are currently untouched,
- α = the factor by which the number of actual edges that need be searched in a bottom-up pass can be reduced when determining when to switch from top-down to bottom-up.
- β = the factor by which the number of vertexes should be reduced when checking for the switch back from bottom-up to top-down.

The terms $N_f,\ M_f,\ M_u$ for the tests for the next step are all simply computable as a pass proceeds. The switch from top-down to bottom-up should occur when:

$$M_f \ < \ M_u/\alpha \tag{1.1}$$

Also, the switch back from bottom-up to top-down should be performed when:

$$N_f \ < \ N/\beta \tag{1.2}$$

Beamer suggests an α of 14 and a β of 24 works well.

Algorithm 3 outlines this approach. It includes the continual re-computation of the necessary decision parameters.

1.5 Prior and Related Work

As mentioned above, Ang et al [1] discuss the functional definition of the BFS kernel as defined for the GRAPH500 benchmark. Beamer [2] introduced the hybrid forward/backward BFS algorithm using in virtually all modern implementations. Kogge [4] performed an early analysis of how the performance of GRAPH500-listed implementation reports was a function of the architecture or hardware resources used to run the problem.

Algorithm 3 Hybrid BFS:
V is the set of vertices; E a set of edges

```
procedure HYBRID-BFS(G,ROOT)
    $Touched \leftarrow \{root\}$
    $Frontier \leftarrow \{root\}$
    $Labels \leftarrow$ N-vector of a large integer
    $Label[root] \leftarrow 0$
    $Level \leftarrow 0$
    $N_f \leftarrow 1$
    $M_f \leftarrow outdegree(root)$
    $M_u \leftarrow M - M_f$
    while $Frontier$ not empty do
        $Level += 1$
        $Next \leftarrow \{\}$
        $N_f \leftarrow 0$
        $M_f \leftarrow 0$
        if $(M_f \geq M_u/\alpha)$or$(N_f < N/\beta)$ then
            for all $u$ in $Frontier$ do                ▷ This pass is top-down
                for all edges $(u, v)$ in E do
                    if v not in $Touched$ then
                        $Touched \leftarrow Touched \cup \{v\}$
                        $Next \leftarrow Next \cup \{v\}$
                        $Label[v] \leftarrow Level$
                        $N_f += 1$
                        $M_f += outdegree(v)$
                        $M_u -= indegree(v)$
        else                                            ▷ This pass is bottom-up
            for all $v$ not in $Touched$ do
                for all edges $(u, v)$ in E do
                    if u in $Touched$ then
                        $Next \leftarrow Next \cup v$
                        $Label[v] \leftarrow Level$
                        $N_f += 1$
                        $M_f += outdegree(v)$
                        $M_u -= indegree(v)$
            $Touched \leftarrow Touched \cup Next$
        $Frontier \leftarrow Next$
    return
```

Bibliography

[1] James Ang, Brian Barrett, Kyle Wheeler, and Richard Murphy. Introducing the graph 500. 01 2010.

[2] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[3] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *In In Fourth SIAM International Conference on Data Mining*, 2004.

[4] Peter Kogge. Performance analysis of a large memory application on multiple architectures. In *7th Int. Conference on PGAS Programming Models*, 2013.

[5] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Inf. Process. Lett.*, 67(6):305–309, September 1998.

[6] Robert Sedgewick. *Algorithms in C*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.