Efficient Graph Search

STOP WHEN DONE

TERENCE KELLY

Welcome to *Drill Bits*, a new column about programming that aims to augment your toolbox and help you write better software. Most installments will include both bits and drills: example code and exercises to hone your skills.

As I write these words, much of humanity is under COVID-19 pandemic lockdown; nonessential activity is banned across the globe. Appropriately enough, this pilot episode of Drill Bits borrows from the zeitgeist the principle of eliminating needless work.

GRAPH SEARCH REVISITED

Graphs provide a versatile, unified abstraction for an exceptionally wide range of practical systems, from electronic circuits to social networks. Graph search is fundamental to analyzing graphs and the real-world systems they represent. Do standard graph-search algorithms leave room for improvement?

This column drills down on BFS (breadth-first search), which is useful in its own right and as a building block for more sophisticated analyses.⁹ After a quick refresher on the classic BFS algorithm, shown in figure 1, I'll show how to improve its efficiency with a technique that can similarly improve other graph analyses. BFS can handle directed graphs, but for brevity we will consider only undirected

FIGURE 1: CLASSIC BFS

```

1  s.distance = 0
2  s.parent = s    // source is defined to be its own parent
3  Enqueue(s)
4  while (Queue not empty) {
5      u = Dequeue()
6      for each vertex v adjacent to u {
7          if (v.parent == 0) {          // v found for the first time
8              v.distance = u.distance + 1
9              v.parent = u
10             Enqueue(v)
11         }
12     }
13 }

```

graphs. Until further notice, we'll restrict attention to connected graphs, in which every pair of vertices is joined by some sequence of edges.

The input to BFS is an unweighted graph whose vertices are numbered 1 through V . Adjacent vertices ("neighbors") of each vertex u are joined to it by undirected edges of the form $\{u, v\}$. E denotes the number of edges, and s denotes the *source* vertex at which search begins.

BFS computes for each vertex v two related outputs⁴: distance from the source (i.e., the number of edges on a shortest path between s and v), and a *parent* vertex chosen from v 's neighbors. Moving to v 's parent, grandparent, great-grandparent, etc. traces a shortest path to s ; backtracking yields a shortest path from s to v . The distance and parent fields of all vertices are initially zero. There's no separate field to indicate that BFS has found a vertex, because zero is not a valid vertex ID; discovered

vertices are those assigned non-zero parents. On connected graphs, BFS from any source finds all vertices.

The central data structure of BFS is a FIFO (first-in, first-out) queue containing the IDs of discovered vertices whose neighbors await inspection. Initially the queue is empty.

BFS begins by initializing and enqueueing the source vertex s , which is at distance zero from itself and is defined to be its own parent. An outer loop (line 4 of figure 1) repeatedly dequeues a vertex u and inspects its neighbors in an inner loop (line 6). If a neighbor v has not yet been found, its distance is set to one greater than that of u , its parent field is set to u , and v enters the queue. Classic BFS terminates when line 4 finds the queue empty.

EFFICIENT SEARCH

Programmers often want to make software faster. Unfortunately, mechanisms that directly increase speed often reduce efficiency, just as jet-engine afterburners raise MPH but reduce MPG. In contrast, eliminating wasted effort typically makes software faster as well, so we should first

strive for the double win of improved efficiency. Can BFS compute the same outputs with less work?

Consider the termination test in figure 1. The `while` loop on line 4 exits after the last vertex has emerged from

the queue, its neighbors have been inspected, and none of the neighbors has been enqueued, leaving the queue empty. Is that the moment when the output reaches its final state (i.e., the instant when every vertex has been assigned a distance and a parent)? Clearly not: Immediately

三十六计，走为上计

**Of all the thirty-six stratagems,
to know when to quit is the best.**

— Chinese military maxim

after assigning a distance and a parent to the last vertex, *classic BFS inserts that vertex into the queue*, thereby thwarting the termination test.

To appreciate how inefficient classic BFS can be, apply it to a *complete* graph, in which all $V \times (V - 1)/2$ possible edges are present and every vertex is adjacent to every other. On the very first trip through the `while` loop, the `for` loop on line 6 iterates over all neighbors of the source (i.e., all other vertices in the graph) and assigns final values to their distance and parent fields. *At this point the entire output of the computation has reached its final state*. Alas, all $V - 1$ neighbors of the source enter the queue; as they are dequeued, all of *their* neighbors are inspected. On a complete graph, classic BFS finalizes its output after performing work proportional to V , then blindly bustles through the busywork remaining in the queue, ultimately performing a total amount of work proportional to V^2 .

Fortunately, a succinct amendment to classic BFS eliminates all wasted effort. Figure 2 shows the enhancement, E-BFS (Efficient BFS). Variable `f` counts vertices with finalized output fields. Line 3 initializes `f` to 1, for the source vertex. After the output fields of a newfound vertex are set, the counter is incremented and compared to V on line 11. When `f` equals V , all output has reached a quiescent state and it's safe to terminate, regardless of what remains in the queue; line 12 breaks out of both loops by jumping straight to line 17. An implementation learns V while reading the input graph, and if the graph is connected, `f` must eventually equal V , so the test on line 11 eventually succeeds.

Remarkably, the improved BFS of figure 2 appears to be

FIGURE 2: EFFICIENT BFS

```

1  s.distance = 0
2  s.parent = s    // source is defined to be its own parent
3  f = 1           // number of vertices found & finalized
4  Enqueue(s)
5  while (Queue not empty) {
6      u = Dequeue()
7      for each vertex v adjacent to u {
8          if (v.parent == 0) {           // v found for the first time
9              v.distance = u.distance + 1
10             v.parent = u
11             if (++f == V)               // if output is finalized...
12                 goto end               // ...we're done
13             Enqueue(v)
14         }
15     }
16 }
17 end:

```

neither widely known nor widely used. Classic BFS alone appears in top computer science textbooks,^{4,11} advanced texts on graphs and networks,^{1,9} Web resources for students,^{6,12} books for professional programmers,^{7,10} industrial-strength graph software,³ and research papers too numerous to mention.

EVALUATION

Asymptotic analysis of classic BFS is straightforward. In figure 1, the outer `while` loop on line 4 iterates once per vertex, and the inner `for` loop on line 6 iterates twice per edge (once from each endpoint). Therefore, the runtime of classic BFS on a connected graph with V vertices and E edges is tightly bounded from below and from above by $V + E$, regardless of the particulars of the input graph. In the lingo of asymptotic analysis, the runtime of classic BFS

is both $\Omega(V + E)$ and $O(V + E)$; equivalently, the runtime is $\Theta(V + E)$.

In contrast, with E-BFS the particulars of the input graph *do* affect asymptotic runtime: The *lower* bound is better. As shown in figure 2, the inner `for` loop on line 7 iterates once for every neighbor of the source vertex, which may have up to $V - 1$ neighbors, so the runtime of E-BFS is bounded from below by V . E-BFS attains this lower bound on a complete graph because it terminates after inspecting every neighbor of the source, so the runtime is $\Omega(V)$. E-BFS has the same asymptotic *upper* bound on runtime as classic BFS— $O(V + E)$ —because some inputs compel E-BFS to inspect every edge [e.g., attach a small tree to a large complete graph and choose a source on the latter].

Does the asymptotic runtime difference matter in practice? To find out, I implemented both BFS variants and ran two experiments: The first uses complete graphs, which bring out the worst in classic BFS; the second uses *sparse* graphs containing only a small fraction of possible edges, which are more commonly encountered in practical applications.

Figure 3a shows the ratio of classic BFS mean runtime to E-BFS mean runtime for complete graphs of varying sizes. Averages are taken over 21 searches from different sources; plotting medians rather than means yields a similar picture. Classic BFS performs work proportional to V^2 and E-BFS performs work proportional to V , so the ratio of runtimes increases in proportion to V . For a complete graph with 2,000 vertices (and thus 1,999,000 edges), E-BFS is more than 9,400 times faster than classic BFS. For $V=10,000$ (and thus $E = 49,995,000$), E-BFS

FIGURE 3: CLASSIC/E-BFS EXPERIMENTAL PERFORMANCE COMPARISON

wins by nearly 73,000 $\times$.

Sparse graphs, in which E is far less than $V \times (V - 1)/2$, arise often in practice. Figure 3b shows the classic/E-BFS runtime ratio for sparse random connected graphs. Each graph has 2^{25} edges (roughly 33.5 million); the number of vertices, and thus their average degree, varies. The first $V - 1$ edges join all vertices to a single connected component: Each vertex ID v in the range $[2..V]$ has a neighbor ID u drawn with uniform probability from $[1..(v - 1)]$. Subsequent edges are (u, v) pairs where both IDs are chosen with uniform probability from $[1..V]$ and $u \neq v$. Figure 3b shows that for graphs with very low average degree—graphs so sparse that they’re practically trees—E-BFS is slightly *slower* than classic BFS. For such graphs classic BFS is reasonably efficient, and the added overhead of incrementing and comparing the E-BFS counter f slows search by one to four percent compared with classic BFS.

efficiency
haiku

*to skip needless work
fix the termination test
stop when you are done*

But as average degree increases beyond roughly 16, E-BFS begins to win; it's 40 times faster when mean degree is 512.

Both theory and experiment point to the same conclusion: Efficient BFS is never much slower than classic BFS, and sometimes it's much faster. The basic principle of

E-BFS—stop when output is finalized—is just common sense. Implementing E-BFS adds a few simple lines of code to classic BFS. Given these advantages, E-BFS deserves more attention than it has received to date. Henceforth it should be regarded as the standard baseline BFS algorithm.

GOING FURTHER

E-BFS generalizes beyond connected graphs. If a graph is not connected, BFS from any source will find fewer than V vertices, so the comparison on line 11 of figure 2 will never succeed. Fortunately, it's easy to restore the efficacy of E-BFS by associating with each vertex the size of its connected component and terminating when counter f equals the size of the component containing the source. You can obtain component sizes by running BFS once within each. In practice, input graphs are typically given as edge lists, and BFS implementations represent graphs using adjacency lists. The cost of running classic BFS once per component is, to within a constant factor, the same as that of building an adjacency list representation from an edge list. If you must run BFS from many source vertices on the same graph, the upfront cost of precomputing component sizes may later pay for itself via E-BFS savings.

As a concrete example of how E-BFS can exploit component sizes, consider applying it to a social-contact

graph representing possible paths of disease contagion. If tests for the disease are in short supply, they must be reserved for persons most likely to spread infection widely (e.g., because they have many contacts or because their contacts do). *Closeness centrality* is a graph metric that identifies such individuals.⁹ Computing closeness centrality requires running BFS from every vertex. If the given social contact graph is not connected, E-BFS doesn't streamline the first search in each connected component. It's cheap and easy, however, to associate the size of a component with all of its vertices as a side benefit of the first search, so subsequent searches in a component benefit from E-BFS.

Finally, BFS isn't the only algorithm that can benefit from the technique of figure 2. I've used it in a serial MIS (maximal independent set) program, where it substantially improved efficiency and speed. A DFS (depth-first search) program that computes the same outputs as BFS—per-vertex distance and parent fields—can use the technique as advantageously as BFS. DFS, however, is often used to compute additional outputs, e.g., to classify all *edges* of a graph, which inherently requires work proportional to E ; classic DFS is reasonably efficient for this purpose.

BITS

All source code used in my experiments is available at https://queue.acm.org/downloads/2020/Drill_Bits_01_sample_code.tar.gz. Two C programs generate connected undirected graphs: One generates complete graphs, and another generates sparse random graphs. The BFS program, also implemented in C, runs classic BFS by default; a compile-time option enables E-BFS. A shell

script shows how I compiled the C programs and ran experiments; a second script post-processes the output of the run script to a concise summary.

DRILLS

There's no substitute for hands-on experience to build intuition. Here are a few exercises in ascending order of difficulty and ambition, from minor tweaks to major projects:

1. The FIFO queue in the example BFS program is an array, `Q`, accessed via integer subscripts, as in `Q[i]`. Are pointers faster? Clearer?
2. Instrument classic BFS to quantify work performed after the output is finalized. How many vertices remain in the queue? How many neighbors do they have?
3. The use of `goto`, as in figure 2, can trigger an allergic reaction.⁵ Implement E-BFS without `goto` by changing both loops. Alternatively, `goto` greater lengths to offend purists by breaking out of the nested loops via `longjmp()` or `throw`.
4. Run experiments on different kinds of graphs (e.g., synthetic small-world graphs⁹ or real-world graphs). Does E-BFS beat classic BFS for these graphs?
5. Try different graph representations (e.g., adjacency matrix or compressed sparse row). Does performance improve compared with adjacency lists? If so, is it worth the effort?
6. Compare the performance of serial classic BFS, serial E-BFS, and a scale-out distributed BFS implementation. Are your results consistent with past evaluations of scalable data analyses² and scalable graph analyses⁸?

7. When is E-BFS effective on *directed* graphs? What preprocessing helps?
8. Use E-BFS to determine the Erdős number of Kevin Bacon and the Bacon number of Paul Erdős.¹¹ Which is harder: getting the graph search algorithm right or obtaining suitable input graphs?
9. Consider graphs on which E-BFS does *not* outperform classic BFS. What countermeasures (e.g., graph preprocessing steps) put E-BFS back in the lead?
10. Find another classic algorithm that performs needless work after its output is finalized. Fix the algorithm and measure the improvement.
11. Inform public health efforts by analyzing paths of possible contagion in social-contact graphs. Use efficient graph search to save the world.

Acknowledgments

Timothy Chow and Professor Robert E. Tarjan of Princeton University reviewed early drafts of this work and provided valuable feedback.

References

1. Ahuja, R. K., Magnanti, T. L., Orlin, J. B. 1993. *Network Flows* (page 76). Upper Saddle River, NJ: Prentice Hall.
2. Anderson, E., Tucek, J. 2010. Efficiency matters! *SIGOPS Operating Systems Review* 44(1):40–45; <https://doi.org/10.1145/1740390.1740400>.
3. Boost graph library; https://dl.bintray.com/boostorg/release/1.73.0/source/boost_1_73_0.tar.bz2. [Boost's BFS closely follows the Cormen et al. text.⁴]
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.

2009. *Introduction to Algorithms*, third edition (page 595). Cambridge, MA: The MIT Press.
5. Dijkstra, E. W. 1968. GOTO statement considered harmful. *Communications of the ACM* 11[3]:147–148; <https://doi.org/10.1145/362929.362947>.
 6. GeeksforGeeks. Breadth-first search or BFS for a graph; <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>.
 7. McDowell, G. L. 2016. *Cracking the Coding Interview*, sixth edition (page 108). Palo Alto, CA: CareerCup.
 8. McSherry, F., Isard, M., Murray, D. G. 2015. Scalability! But at what COST? Usenix HotOS XV; <https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-mcsherry.pdf>.
 9. Newman, M. 2010. *Networks: An Introduction* (page 320). New York, NY: Oxford University Press.
 10. Orwant, J., Hietaniemi, J., and John Macdonald, J. 1999. *Mastering Algorithms with Perl* (page 307). O'Reilly.
 11. Sedgewick, R., Wayne, K. 2011. *Algorithms*, fourth edition (page 540). Addison-Wesley Professional.
 12. Wikipedia. Breadth-first search; https://en.wikipedia.org/wiki/Breadth-first_search

Terence Kelly (tpkelly@acm.org) is a Distinguished Member and a Lifetime Member of the ACM. He studied computer science at Princeton and the University of Michigan followed by 16 years as an industrial researcher (HP Labs) and software engineer. Kelly avoids all unnecessary work and much of the other kind.

Copyright © 2020 held by owner/author. Publication rights licensed to ACM.