Programming Workbench

Compressed Sparse Row Format for Representing Graphs

TERENCE KELLY

Terence Kelly studied computer science at Princeton and the University of Michigan, followed by a long stint at Hewlett-Packard Laboratories. Kelly now writes code and documentation promoting persistent memory programming and other programming techniques. His past publications—some of tragicomic interest only—are listed at http://ai.eecs.umich.edu/~tpkelly/papers/. tpkelly@eecs.umich.edu.

Welcome to the second installment of Programming Workbench. Today's topic is *compressed sparse row* (CSR) format, a compact and efficient way to represent graphs in memory. As usual, all example code is available in machine-readable form [6].

Graphs provide a generic abstraction that finds numerous applications for modeling *connectedness* and *ordering* in computing systems. Undirected graphs, for example, can represent communications links among computers; directed graphs can encode dependencies or precedence constraints in software compilation, software package installation, and job scheduling problems. Top computer science textbooks emphasize two ways of representing graphs in memory: adjacency matrices and adjacency lists [1, 8]. Today we'll consider other options that offer different tradeoffs and sometimes provide significant advantages. In particular we'll see that *compressed sparse row* (CSR) format—a compact and memory-hierarchy-friendly graph representation—is sometimes the format of choice. Understanding CSR in detail rounds out a programmer's education and informs the buy-or-build decisions that routinely confront practitioners.

We'll begin in the next section by reviewing ways of representing graphs, including CSR. Then we'll walk through a working C11 program that converts an edge list representation of a graph into CSR format. Finally we'll conclude by suggesting extensions and exercises to help better understand the tradeoffs surrounding CSR. For brevity, we'll restrict attention to unweighted directed graphs, but we thereby lose little generality: an *undirected* edge can be represented by two *directed* edges in opposite directions, and adding edge weights to a CSR representation is easy.

Graph Representations

Figure 1(a) shows a directed graph that we'll use as a running example. We follow the convention that vertexIDs range from 1 to V inclusive, where V is the total number of vertices. The example graph contains V=9 vertices and E=9 directed edges. For example, there's a directed edge from vertex 2 to vertex 1, shown as an arrow near the top of Figure 1(a). Vertices 5 and 9 have in-degree zero and out-degree zero, i.e., they have neither incoming nor outgoing edges. Zero-degree vertices arise naturally in applications; for example, they may represent software packages with no dependencies or compute jobs with no precedence constraints.

Rather than treating zero-degree vertices as special cases, removing them and/or handling them "out of band," we'll take the simpler approach of representing them straightforwardly. *Self edges*, i.e., edges that point from a vertex to itself, do not appear in our example, but they pose no special difficulties for the graph representations discussed below. We omit self edges for brevity; they arise relatively infrequently in applications of practical interest.

In many practical applications, a graph is given as a file that essentially contains an *edge list* of "from"/"to" vertexID pairs, possibly mummified in a more elaborate format such as XML or JSON. Figure 1(b) shows an edge list representation of our example graph. The first line in the list, "2 1," represents the directed edge from vertex 2 to vertex 1. Zero-degree vertices, such as 5 and 9 in our example, don't appear in an edge list, so metadata accompanying the

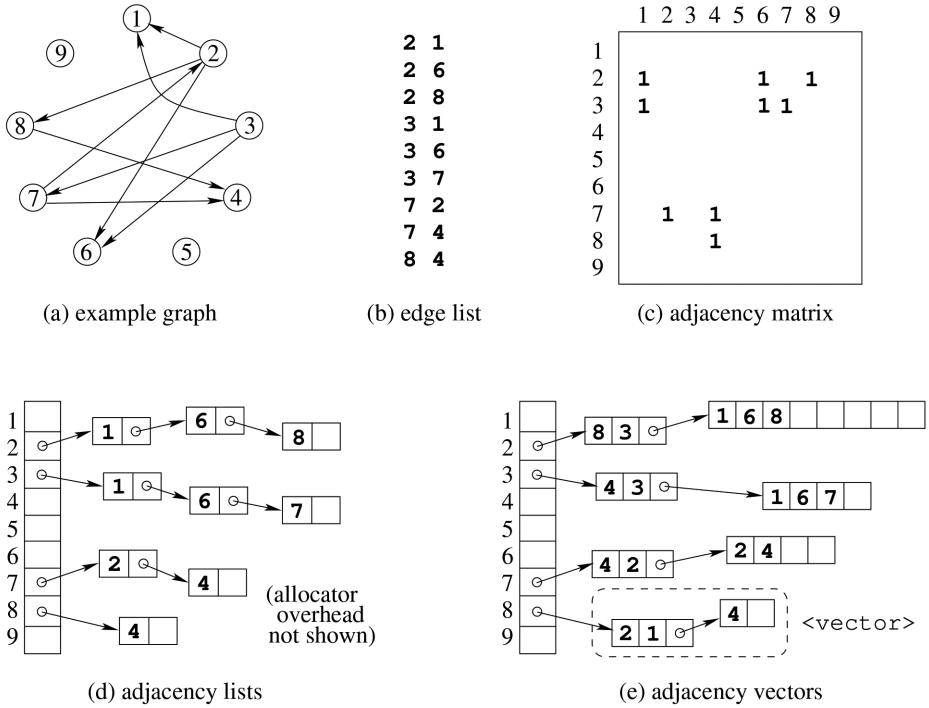

Figure 1: Textbook representations of running example, a directed graph with nine vertices and nine edges. The C++ STL `<vector>` depicted within the dashed oval in Figure 1(e) is a two-part data structure: a partially filled data array, on the right, located via the header on the left, which contains the *capacity* of the data array, the number of positions in the array occupied by user data (which may be less than the capacity, as shown here), and a pointer to the data array itself. The header of the `<vector>` enclosed by the dashed oval indicates that the data array can hold two integers but is currently holding only one. This `<vector>` represents the adjacencies of vertex 8, and the lone integer contained in the data array corresponds to the directed edge from vertex 8 to vertex 4, i.e., the last line of the edge list in Figure 1(b).

edge list must ensure that zero-degree vertices don't go missing: Thanks to our vertexID convention, simply knowing V ensures that we don't overlook zero-degree vertices. For clarity, Figure 1(b) shows a sorted edge list, but edge lists seldom arrive sorted in practical applications.

Figure 1(c) depicts a standard textbook *adjacency matrix* representation of our example graph. A directed edge from vertex i to vertex j appears as a "*1*" at row i, column j of the adjacency matrix; all other matrix entries are zero (not shown for clarity). An adjacency matrix is efficient for some operations, such as testing in constant time whether an edge connects a given pair of vertices. The major downside of adjacency matrix representation is that it requires $O(V^2)$ bits even for *sparse* graphs in which most vertex pairs are not connected by an edge. Sparse graphs arise frequently in practice, and for large sparse graphs an adjacency matrix wastes too much memory on zero entries.

The representation that most textbooks recommend for sparse graphs uses *adjacency lists*, shown in Figure 1(d). On the left is an array of pointers indexed by "from" vertexID; each pointer is the head of a singly linked list of "to" vertexIDs. Adjacency lists are flexible—it's easy to add or delete vertices—and they are indeed more compact than adjacency matrices for sparse graphs. However they entail unfortunate time and space overheads of their own: space overheads include the "`next`" pointer in every list node; list nodes will also carry allocator overheads if a general-purpose allocator like `malloc()` creates them. We suffer time overheads when we traverse an adjacency list because we must chase pointers across the address space, creating random memory accesses that today's computers penalize heavily compared with sequential accesses. If we transform an unsorted edge list representation into dynamically allocated adjacency lists in the straightforward way, the list nodes for each adjacency list will be scattered across the heap, exacerbating the pointer-chasing problem.

Using C++ Standard Template Library `<vector>`s instead of linked lists might seem like one way to reduce the overheads of adjacency lists. Figure 1(e) shows the resulting *adjacency vectors* representation. As with adjacency lists, an array indexed by "from" vertexID contains entry points to `<vector>`s of "to" vertexIDs. The dashed oval at the bottom of Figure 1(e) encloses the `<vector>` of vertexIDs adjacent to vertex 8. A `<vector>` is typically implemented as a two-part structure consisting of a

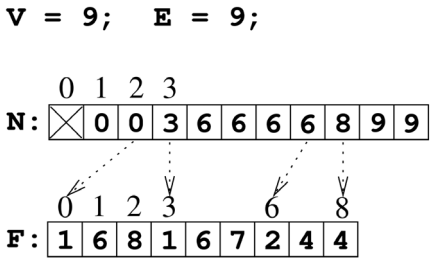

Figure 2: Compressed sparse row (CSR) representation of example graph. The vertices adjacent to vertex a are stored in positions N[a] through N[a+1]-1 of array F. For example, consider vertex 2 from Figure 1(a): directed edges extend from vertex 2 to vertices 1, 6, and 8. N[2] is zero, so the adjacencies of vertex 2 start at F[0]; N[2+1]-1 is 2, so they extend through F[2]. F[0] through F[2] contain the expected vertexIDs: 1, 6, and 8.

header containing the number of *allocated* entries, the number of *occupied* entries, and a pointer to an array of the entries themselves [10]. If we read a graph given as an edge list into adjacency <vector>s in the straightforward way, each <vector> grows as vertexIDs are added to it. Implementations typically *double* allocated capacity each time a <vector> fills up as it grows. The result is that up to roughly half of the allocated capacity of each vector can be unused; this waste may erode the benefits of reducing pointer and allocator overheads compared with adjacency lists. On the positive side, <vector>s reduce the time overhead of chasing pointers because they store adjacent vertexIDs in compact arrays.

The representations shown in Figure 1 don't exhaust all of the possibilities. For example, we sometimes need fast access to the *incoming* as well as the outgoing edges of a vertex, which is easy to arrange by associating a second adjacency list with each vertex. And nothing prevents us from using *both* an adjacency matrix and adjacency lists or <vector>s simultaneously, if we have sufficient memory. Using both representations yields the strengths of both: constant-time queries to test the existence of an edge between a given pair of vertices, and efficient access to the adjacent vertices of a given vertex.

Compressed Sparse Row Format

CSR originated in high-performance scientific computing as a way to represent sparse matrices, whose rows contain mostly zeros. The basic idea is to pack the column indices of non-zero entries into a dense array. CSR is more compact and is laid out more contiguously in memory than adjacency lists and adjacency <vector>s, eliminating nearly all space overheads and reducing random memory accesses compared with these other formats. The price we pay for CSR's advantages is reduced flexibility: adding new edges to a graph in CSR format is inefficient, so CSR is suitable for graphs whose structure is fixed and given all at once. CSR also carries a *cognitive* overhead: it's trickier than the other formats we've reviewed, and it uses arrays in FORTRANesque ways seldom seen in systems-y C/C++ code or in mainstream Java code. We'll walk through it slowly.

Figure 2 depicts the CSR representation of our example graph. First we'll consider the specifics of how CSR encodes a handful of the example graph's structural features, and then we'll describe CSR in more general terms. Like the textbook sparse-graph representations discussed earlier, CSR facilitates finding the adjacencies of a given vertex, i.e., the vertices at the "to" ends of edges emanating out of a given "from" vertex. CSR finds adjacencies using two layers of array indexing.

The CSR depicted in Figure 2 contains V, E, and two arrays of integers, N and F. Notice that F presents horizontally the same sequence of "to" vertexIDs that appear vertically in the right-hand column of the sorted edge list of Figure 1(b). Given a "from" vertexID, we find all corresponding "to" vertexIDs by indexing into F via N. We'll walk through the process of finding the adjacencies of the first three vertices in our example graph to gain intuition for how CSR encodes graph structure.

The out-degree of vertex 1 is encoded as the *difference* between N[1] and N[2]. Since N[1] *equals* N[2]—both are zero—the out-degree of vertex 1 is zero, so there are no adjacent vertices to be found. The out-degree of vertex 2 is N[2] subtracted from N[3], which is 3. The IDs of the three vertices adjacent to vertex 2 are in array F starting at position N[2], i.e., at F[0], as indicated by the dotted arrow in Figure 2 from N[2] to F[0]. The out-degree of vertex 3 is the difference between N[3] and N[4], which is three; the IDs of the three vertices adjacent to vertex 3 begin at position N[3] in F, i.e., at F[3], as shown by a second dotted arrow in Figure 2. The figure contains a dotted arrow for every vertex with out-degree greater than zero; the arrowheads partition F into four sub-arrays of adjacencies.

In general, the out-degree of any vertex a is N[a+1] minus N[a]. The IDs of the vertices adjacent to a are located in array F starting at F[N[a]] and continuing through F[N[a+1]-1] inclusive. In other words, the entries of N, indexed by "from" vertexID, "point to" contiguous regions of F containing the adjacent "to" vertexIDs. Array F contains E entries, one for each edge. Array N contains V+2 entries: N[0] is unused and N[V+1] contains E. The total amount of memory required for CSR format is almost exactly equal to sizeof(int) multiplied by (V+E), so it's easy to determine if available memory is adequate based on a graph's size parameters.

If E exceeds INT_MAX, a larger integer type, e.g., int64_t, must be used for the elements of N, because the entries of N refer to positions in E-long array F and the last entry of N contains E. Similarly, F must use a sufficiently large type to represent vertexIDs up to V. Moreover it's actually best to choose a type for vertexIDs

such that *V* is strictly *less* than the MAX of the type, because *V*+1 is used as an index into N. Unsigned integer types may be used for arrays N and F, though signed integer types might be preferable, e.g., if we want to catch signed overflow errors at runtime with a compiler flag like GCC's -ftrapv.

CSR offers different tradeoffs than alternative formats. On the positive side, it eliminates memory allocation overheads almost completely. Furthermore, while array N contains the moral equivalent of pointers, they can be smaller than conventional pointers (32 vs. 64 bits), depending on the size of *E* and the relative sizes of ints and ordinary C pointers. The IDs of vertices adjacent to a given vertex are contiguous in F, so visiting all adjacent vertices involves zooming through an array, which is much faster on modern computers than chasing pointers down an adjacency list. While *adding* a new edge to a CSR representation isn't efficient—it would require insertion into the middle of F, which would take *O(E)* time on conventional memory [3]—*deleting* an edge is quick and easy: to delete an edge, simply set its entry in F to zero, which is not a valid vertexID, and ignore zero entries in F.

CSR isn't magic. When applied to the kinds of graphs that arise naturally in practical applications, many important graph algorithms, including traversal algorithms such as breadth-first search and depth-first search, must inevitably perform random memory accesses. CSR can't eliminate random memory accesses that are inherent to the computational task at hand; it can merely avoid introducing additional random accesses that arise as side effects of the format.

The Code

The C11 program listed in this section, "el2csr.c," converts an edge list representation of an unweighted directed graph to CSR format; the source code is available at [6]. We'll discuss everything substantive, skipping boilerplate like #includes. The purpose of the example code is to illustrate CSR format, so it avoids niceties for brevity and clarity.

The macros below handle error checking. BAIL() prints an error message prefixed by the file name and line number where it is called then exit()s. CAL() calls calloc() and bails if allocation fails.

```
#define ERRSTR strerror(errno)
#define                          S1(s) #s
#define                    S2(s) S1(s)
#define COORDS __FILE__ ":" S2(__LINE__) ": "
#define BAIL(...)                           \
  do { fprintf(stderr, COORDS __VA_ARGS__); \
       exit(EXIT_FAILURE); } while (0)
#define CAL(p, n, s)                                    \
  do { if (NULL == ((p) = (int *)calloc((n), (s))))     \
         BAIL("calloc(%lu, %lu): %s\n", (n), (s), ERRSTR);\
     } while (0)
```

Readers may recall from the previous Programming Workbench column a function-like macro called "DIE()" that differs from BAIL() above but serves a similar purpose. The contrast between the two stems from differences in how they are used and from differences in the programs they inhabit. DIE() is used exclusively to handle failed library calls, and thus it is adequate for DIE() to report only the name of the failed call via perror(). By contrast, BAIL() is sometimes used to check user input, so it supports flexible printf()-like formatting of more informative diagnostics, such as the input line number where a parse error occurs. DIE() is used in multithreaded code where failed library calls may arise from Heisenbugs, so it aborts with a core dump to facilitate debugging. BAIL() serves a simple single-threaded program and is used in situations where a core dump would not be very helpful, so it merely calls exit(). DIE() expands to an expression because it is used in contexts that demand expressions, but BAIL()'s simpler role allows it to expand into a statement block, which is easier to understand.

The following struct will eventually contain a CSR representation of a graph. The roles of V, E, N, and F are as described in the previous section.

```
static struct {
  int V,  // max vertexID; valid vertexIDs are [1..V]
      E,  // total number of edges
     *N,  // indexed by "from" ID; outdeg(v) == N[1+v]-N[v]
     *F;  // "to" vertexIDs accessed via N[]
} CSR;
```

For brevity we consider only unweighted graphs, but it's easy to handle edge weights: add to the struct CSR above an *E*-long dynamically allocated array of weights—one weight for every edge in array F. Note that such edge weights can be updated efficiently; they need not be completely static.

One way to understand CSR format is to study the function below, which prints a text representation of the graph in the struct above. The outer for loop iterates over all vertexIDs a. The inner for loop iterates over all vertexIDs b such that a directed edge exists from a to b. Pointers begin and end delimit the part of array F containing a's adjacent vertexIDs.

```
static void print_adjacencies(void) {
  printf("per-vertex adjacencies:\n");
  for (int a = 1; a <= CSR.V; a++) {
    int *begin = CSR.F + CSR.N[  a],
        *end   = CSR.F + CSR.N[1+a];
    printf("%d:", a);
    for (int *b = begin; b < end; b++)
      printf(" %d", *b);
    printf("\n");
  }
}
```

Our `struct CSR` contains ordinary `int` variables, which are 32 bits long on many computers. In practice we may encounter graphs with many billions of vertices and edges, so when the user enters graph size parameters *V* and *E* on our program's command line, we verify that they both fit in an `int`—with room to spare, because we index into array `N` using values up to *V*+1. Function `s2i()` below performs string-to-`int` conversions carefully and gripes if it encounters weirdness of any kind. The C11 `static_assert` feature confirms at compile time our assumption that the largest integer type is larger than an `int`.

```
static_assert(sizeof(intmax_t) > sizeof(int), "int sizes");
static int s2i(const char *s) {
  char *p;  intmax_t r;
  errno = 0;
  r = strtoimax(s, &p, 10);
  if (0 != errno || '\0' != *p || 0 >= r || INT_MAX <= r)
    BAIL("s2i(\"%s\") -> %" PRIdMAX ", errno => %s\n",
         s, r, ERRSTR);
  return (int)r;
}
```

The `main()` function begins by declaring a few variables and checking user-supplied command-line arguments, then opening the file containing the edge list representation of the input graph. Reading *V* from the command line, as opposed to inferring it from the largest vertexID on the edge list, accommodates zero-degree vertices with IDs greater than any on the edge list, like vertex 9 in our example graph.

```
int main(int argc, char *argv[]) {
  int a, b, line = 0, t = 0;
  FILE *fp;

  if (4 != argc)
    BAIL("usage:  %s  V  E  edgelistfile\n", argv[0]);
  CSR.V = s2i(argv[1]);
  CSR.E = s2i(argv[2]);
  if (NULL == (fp = fopen(argv[3], "r")))
    BAIL("fopen(\"%s\"): %s\n", argv[3], ERRSTR);
```

Next, we allocate memory for the `N` and `F` arrays using the `CAL()` macro, which calls `calloc()`. As explained above, array `N` is of size *V*+2 because it is indexed with integers up to *V*+1.

```
  CAL(CSR.N, 2 + (size_t)CSR.V, sizeof *CSR.N);
  CAL(CSR.F,     (size_t)CSR.E, sizeof *CSR.F);
```

We make *two* passes over the input file to construct CSR format. The first pass, below, verifies the sanity of each vertexID pair and stores the out-degree of each vertex in array `N`; later `N` will be altered to play its role in CSR format as described in the previous section. We check for flagrant parse errors and verify that the *E* given on the command line matches the length of the input file.

```
  while (2 == fscanf(fp, "%d %d\n", &a, &b)) {
    line++;
    if (0 >= a || a > CSR.V || 0 >= b || b > CSR.V)
      BAIL("%d:  bad vertexID:  %d %d\n", line, a, b);
    if (a == b)
      fprintf(stderr, "%d:  warning:  self edge\n", line);
    CSR.N[a]++;
  }
  if (! feof(fp))
    BAIL("parse error after %d lines: %s\n", line, ERRSTR);
  if (line != CSR.E)
    BAIL("%d input lines != %d edges\n", line, CSR.E);
```

The standard `fscanf()` function used above silently performs incorrect conversions if the input vertexIDs are too large. For example, on my system `fscanf()` happily converts 4,294,967,299 to 3 without complaint. Performing conversions more carefully, e.g., with the `s2i()` function that we saw earlier, would substantially increase the overhead of parsing the input. Instead we warn users that it's their responsibility to ensure that vertexIDs on the input edge list must not exceed the *V* argument supplied on the command line, which is checked carefully by `s2i()`.

This next bit of code updates the contents of array `N` to contain *cumulative* out-degrees. After the code below executes, `N[a]` contains the *sum* of the out-degrees of vertices 1 through `a` inclusive. `N[V+1]` contains the sum over all vertices of their out-degrees, i.e., the number of edges *E*.

```
  for (a = 1; a <= CSR.V; a++) {
    t += CSR.N[a];
    CSR.N[a] = t;
  }
  CSR.N[a] = t;
  assert(CSR.N[1 + CSR.V] == CSR.E);
```

We're still not done with array `N`, because at this point each entry `N[a]` is too large by the out-degree of vertex `a`. Our second and final pass over the input fixes the problem. The second pass adds edges to `F` while walking the moral-equivalent-of-pointers in `N` back to their final correct CSR values.

```
  rewind(fp);
  while (2 == fscanf(fp, "%d %d\n", &a, &b))
    CSR.F[--CSR.N[a]] = b;  // add directed edge a -> b

  if (0 != fclose(fp))
    BAIL("fclose(): %s\n", ERRSTR);
```

Sorting the outgoing edges of each vertex isn't strictly necessary, but we'll do it anyway because it makes it easy to detect duplicate edges. Furthermore it allows us to perform a binary search on each vertex's adjacencies in *O(log D)* time, where *D* is the average out-degree. Would it be easier to sort the input edge list rather than sorting the adjacencies of each vertex? That might be conceptually simpler and easier to implement, but it would be asymptotically less efficient: sorting the edge list with a general method such as `qsort()` would would require *O(E log E)* time, whereas sorting the adjacencies of each vertex requires *O(V D log D)* time; the latter is typically smaller. The integer comparison function below, `icmp()`, seems prone to overflow in the subtraction operation—consider `INT_MAX` minus negative one—but overflow can't

happen in our program because all of the integers being sorted are non-negative.

```
static int icmp(const void *a, const void *b) {
  return *(const int *)a - *(const int *)b;
}
...
  for (a = 1; a <= CSR.V; a++)
    qsort(CSR.F + CSR.N[a],
          (size_t)(CSR.N[1+a] - CSR.N[a]),
          sizeof *CSR.F, icmp);
```

Now that we've constructed CSR format, we dump it for inspection and then print per-vertex adjacencies using the function we defined earlier:

```
  printf("dump CSR format:\n"
         "V = %d    E = %d\n"
         "N: ", CSR.V, CSR.E);
  for (a = 0; a <= 1 + CSR.V; a++)
    printf(" %d", CSR.N[a]);
  printf("\n"
         "F: ");
  for (a = 0; a < CSR.E; a++)
    printf(" %d", CSR.F[a]);
  printf("\n");

  print_adjacencies();
```

Our final chore before terminating is to deallocate arrays `N` and `F`:

```
  free(CSR.N);
  free(CSR.F);

  return 0;
}
```

Running `el2csr` on our example graph yields the expected results:

```
% ./el2csr 9 9 example_graph.txt
dump CSR format:
V = 9    E = 9
N:  0 0 0 3 6 6 6 6 8 9 9
F:  1 6 8 1 6 7 2 4 4
per-vertex adjacencies:
1:
2: 1 6 8
3: 1 6 7
4:
5:
6:
7: 2 4
8: 4
9:
```

The example code tarball contains a random graph generator and a test script in addition to `el2csr.c`. The test script compiles the random graph generator and compiles `el2csr` in a special test mode that dumps an *edge list* representation of the input graph to a file. The test script then feeds many random graphs to `el2csr` and verifies that in each case the edge list regurgitated by `el2csr` is byte-for-byte identical to the sorted input file.

Persistence

Converting an edge list to CSR format takes time—parsing textual input can be orders of magnitude slower than running a graph analysis algorithm—and it would be wasteful to perform the conversion more often than necessary. It's usually best to store the binary CSR representation of the graph in a file for future use. One way would be to `write()` variables `V` and `E` and arrays `N` and `F` to a file. An easier and more elegant approach is to employ "the persistent memory style of programming" [4, 5]: lay out the data structures in a file-backed memory mapping using `msync()` to persist the data after constructing a CSR representation and later using `mmap()` to load the file containing CSR back into memory as needed. This is convenient and is often the most efficient way to handle graphs in practical applications, because after the initial conversion to CSR format no further parsing or serializing is ever needed. The CSR file is in the compact in-memory format used by subsequent analyses, which access the data via LOAD instructions after `mmap()`-ing the file into memory.

Other Implementations

The Boost Graph Library [9] offers C++ implementations of many graph algorithms, and it supports several graph formats including adjacency lists and CSR. BGL emphasizes generic programming and is written in a different style from my example code; comparing the two may lead the reader to additional insights. Galois is a platform for parallel computation that includes substantial support for graphs [2]. Distributed/scale-out graph analysis platforms were blooming like mushrooms in the research community several years ago; many were so grotesquely inefficient that they are of tragicomic interest only [7].

Going Further

Extending my example code can be an informative exercise. You can avoid the time overhead of parsing the input edge list on the second pass by converting it to a temporary binary edge list on the first pass. Adding support for weighted edges is easy. To appreciate the benefits of CSR format over adjacency lists or adjacency `<vector>`s, compare their memory footprints on real or randomly generated graphs. Similarly, compare the runtimes of standard graph algorithms on the different formats.

Random graph generators are often used for testing and performance benchmarking because they make it easy to sweep key graph parameters such as size, average degree, and density. Storing large random graphs in short-lived files can be slow, awkward, and cluttery, but an easy trick avoids the need to create files, even when their consumer must make multiple passes over each: run multiple instances of the random graph generator as background jobs that spit identical byte streams into named pipes, one pipe for every pass needed by the consumer. For

example, if the `el2csr` program listed above is the consumer, it would be modified to read two identical byte streams from two named pipes supplied on the command line—an easy exercise. This approach preserves a clean separation of responsibilities between graph generator and graph consumer while avoiding the fuss of large temporary files.

Conclusion

Compressed sparse row is typically the best format for sparse graphs, provided that new edges aren't added and relatively few edges are deleted. CSR is compact, avoiding the memory waste of adjacency lists and `<vector>`s, and its memory footprint can be calculated directly from *V* and *E*. CSR is furthermore contiguous in memory, eliminating the time overhead of pointer chasing. It's easy to persist CSR in memory-mapped files, and CSR is convenient once you become accustomed to it. The two-pass construction approach implemented above is asymptotically faster than sorting an edge list.

Graphs are essentially simple, and coding graph algorithms can be positively pleasant. The next time you're faced with a problem involving graphs, consider solving it by writing your own code instead of using someone else's software; the result might well be superior overall. Please share your experiences and feedback with me!

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd edition (MIT Press, 2009).

[2] "Galois": https://iss.oden.utexas.edu/?p=projects/galois.

[3] T. Kelly, H. Kuno, M. Pickett, H. Boehm, A. Davis, W. Golab, G. Graefe, S. Harizopoulos, P. Joisha, A. Karp, N. Muralimanohar, F. Perner, G. Medeiros-Ribeiro, G. Seroussi, A. Simitsis, R. Tarjan, and S. Williams, "Sidestep: Co-Designed Shiftable Memory and Software," HP Labs Tech Report HPL-2012-235, November 2012: https://www.labs.hpe.com/techreports/2012/HPL-2012-235.pdf.

[4] T. Kelly, "Persistent Memory Programming on Conventional Hardware," *ACM Queue*, vol. 17, no. 4 (July/August 2019): https://queue.acm.org/detail.cfm?id=3358957.

[5] T. Kelly, "Good Old-Fashioned Persistent Memory," *;login:*, vol. 44, no. 4 (Winter 2019): https://www.usenix.org/publications/login/winter2019/kelly.

[6] T. Kelly, Example code to accompany this article: https://www.usenix.org/sites/default/files/kelly_csr_code.tar.gz.

[7] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST?" 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS '15): https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-mcsherry.pdf.

[8] R. Sedgewick and K. Wayne, *Algorithms*, 4th edition (Addison-Wesley, 2011).

[9] J. G. Siek, L. Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual* (Addison-Wesley, 2002).

[10] B. Stroustrup, *The C++ Programming Language*, 4th edition (Addison-Wesley, 2013). See p. 888 for the representation of `<vector>`s.