Fifth DIMACS Challenge

C. C. McGeoch

March 19, 1997

1 Priority Queue Core Tests

Two of the three research groups implemented monotone priority queues, which restrict the input so that the sequence of priorities obtained by *delete_mins* must be monotonically increasing. The third group has implemented a generalization (dynamic sortedness) and a relaxation (approximate-min) of basic priority queues.

In order to make the core tests work for all groups, the input sets are restricted to monotonic instances. These tests are for callibration and "baseline benchmark" purposes, and we expect that participants will perform much more extensive tests in pursuit of their own research questions.

1.1 What Files You'll Need

The test suite for monotone priority queues involves the following files and programs, all obtainable via anonymous ftp through the web page

`http://www.cs.amherst.edu/~ccm/challenge5/p\_queue/`

.

Programs for performing the tests. You will need the following programs to run the tests.

- Your own implementation(s), and others that you may wish to compare. You may of course test different variations.

- Test Driver. The test driver reads input files and calls the priority queue routines. The **new improved driver** runs 2-3 times faster than the old one.
- Demonstration PQ. This is a basic heap-style priority queue. The bug in the first version has been fixed, and this implementation runs slightly faster than before.
- Dummy PQ. The Dummy PQ does not perform priority queue operations. Run this with the driver to obtain timing statistics for the driver alone. Subtract from total runtime to get net cost of the data structure implementation.

Input Test Sets You will need the following programs (each available as a .tar.Z file) to generate the test instances. Although the generators are small, some of the generated files are huge: you will probably have to do the tests in stages, deleting old files as you go.

- (27 instances) Dijkstra tests. The Dijkstra inputs represent traces of Dijkstra's algorithm, run on different types of graphs. For these instances, *problem size* refers to the size of the graph, not the number of operations in the trace file. The number of inserts and of deletes equals the total number of nodes in the graph, but the number of decrease-keys varies.
  - (9) Dijkstra_rand. Dijkstra on random graphs; relatively small queues for the problem size. Run 3 random trials at each of 3 problem sizes (containing $2^18, 2^19, 2^20$ nodes).
  - (9) Dijkstra_grid. Dijkstra on grid-graphs; relatively large queues for the problem size. Run 3 random trials at each of 3 problem sizes (containing $2^17, 2^18, 2^19$ nodes).
  - (9) Dijkstra_spbad. Dijkstra on spbad graphs; lots of decrease-keys for problem size. Run 3 random trials at each of 3 problem sizes ($2^9, 2^10, 2^11$ nodes).
- (18 instances) Sorting. Three random sorting problems at each of three problem sizes (1K, 10K, 100K), and at two ranges of priority values. ($1 \ldots 10^4$ and $1 \ldots 10^8$). Priorities are of type *double*. These are N inserts of random priorities followed by

N deletes. (These inputs will be used twice in the tests, as described in the next section.)

- **NEW**. Goldberg and Silverstein have developed a generator of random monotone sequences, and a discrete event simulation generator. See the `.tex` file in their `.tar` file for suggested problem parameters. You can get the `.tar` file via the website mentioned above.

1.2 What to Measure

The Core tests involve a combination of runtime tests and counts of basic operations, as described below.

1. First, for each run, record the *total user running time* (on Unix, use *time qdriver ¡in* for each instance and each implementation you are testing (including the dummy item). For timing, the *#define TRACE* option should NOT be used in the driver.

2. For the Dijkstra tests, use *integer* priorities, and *info* types set to *str16* (strings of 16 characters). These declarations appear in *pq_instantiate.h.*

3. For the Sorting tests, use *double* priorities. Run these tests twice, first with the info type set to *str16*, and second with the info type set to *str64*. (You can skip the second set if you are sure that your runtime will not be affected by the size of the info part).

4. You might want to run separate tests with different compilers or compiler versions or with implementation variations.

5. Second, for each run, identify a small number of *key operation counts* according to the guidelines in the next section. Insert code to count these operations in your implementation, and insert code (perhaps inside *destruct_pq*) to report counts at the end of the run. Re-run the above tests, this time without the timer.

1.3 How to Perform Operation Counts

It is important that your runtime statistics, which can be very noisy and environment-dependant, be supported by and compared to some alternative measure of performance.

Some suggestions are discussed below. Choose *at least one*, and preferably more, of the following performance measures to report.

Count Representative Operations. Ahuja, Magnanti, and Orlin [1] describe how to identify a small set (perhaps 3 or 4) of *representative operations* to be counted. Candidates for representative operations counts might be: (1) a most-frequently-executed line of code, such as one appearing inside several nested loops; (2) a count of loop iterations that are sensitive to some property of the input; (3) a frequently-executed data structure operation, such as a comparison between two nodes (however, node comparisons during, say, *insert* should be counted separately from node comparisons during *delete*).

A more detailed discussion of representative operation counts, with several examples, may be found in [1] (Chapter 18). Some guidelines and suggestions for choosing appropriate operations, gleaned from that reference, appear below.

- A single "operation" should require constant (although perhaps some aggregate) time; the total number of operations performed in an execution are expected to progressively dominate the runtime as problem size increases.

- The operation should be fundamental to the algorithm or data structure, and the count should not vary much across machines, compilers, or details of program implementation.

- The goal is for CPU time to be expressible as a linear function of the representative operations. In this sense the identified operations may be similar to the "dominant operation" of traditional algorithm analysis, except there are more than one, and lower-order costs may be included in the function.

Count Memory References Knuth [2] argues that counts of memory references (called *mems*) provide much better measures of program performance than do runtime statistics. He points out that running times depend (among other things) on complicated interactions between pipelines, memory caches, and operating systems. Counts of memory references provide good approximations to running times (which but are likely to get better in the future), while giving less distortion in comparisons among different computing environments. A more detailed discussion of mem-counting may be found in [2], especially in Chapter 4, Section MILES_SPAN, and notes 10, 11, and 71.

To perform counts of memory references, Knuth describes a collection of easily-implemented C macros named *o, oo, ooo, ...,* as follows.

```
#define o mems++
#define oo mems += 2
#define ooo mems += 3
<Global variable>
long mems;
```

With these definitions, the appropriate macros can be inserted before expressions that perform references to memory. For example the following fragment of Kruskals algorithm appears in [2] (pg 466).

o, n= g→ n;

for *(l = 0; l ¡ 64, l++) oo, aucket[l] = bucket[l] =* Λ

for *(o, v= g→ vertices; v ¡ g→ vertices + n; v++)*

  for *(o, a = v → arcs; a ∧ (o , a→ tip ¿ v); o, a= a→ next)* {

    o, a→ from = v;

    o, l = a→ len & #3f

    oo, a → klink = aucket[l];

    o, aucket[l] = a;

}

(Recall that the C comma operator "," has lowest precedence of any binary operator, so parentheses are necessary in the second boolean test of the inner for loop.) For numerous examples of mem counting conventions, see [2], section MILES_SPAN notes 12 through 71. Here are some guidelines for counting mems, gleaned from Knuth's discussion.

- Values of significant variables (like a in the above) can be assumed to be in registors. Arithmetic operations on register values need not be counted as memory references. However there can only be a finite and fixed number of "register variables" in your implementation. (p. 464).

- Count mems for each memory reference before each assignment statement and boolean expression in the program.

- Count nothing for subroutine calls if the subroutine is not recursive; if the subroutine is recursive, charge twice the number of things that need to be saved on the runtime stack (this includes parameters and the return address.) (p. 470.)

- Count mems which are associated with the algorithm or data structure that you are studying; don't bother to to count mems for operations associated with, for example, user interfaces, system timer calls, data type conversions, or input-robustness tests. (p. 465).

- "Of course authors are expected to be reasonable and fair when they are competing for minimum-mem prizes. They must be ready so submit their programs to inspection by impartial judges. A good algorithm will not need to abuse the spirit of realistic mem-counting." ([2], p. 465.)

1.4 What to Report

For each implementation that you test, prepare a chart that resembles the one below, and fill out the information at the top of the chart. (The example numbers are taken from Andersson and Thorup's draft report.) It is OK to split this table by columns if it becomes too large; if you do split it, include (at least) the first two columns with each subsequent table, for ease of cross-reference.

The column headings have the following meanings:

- Input File Name.

- Total number of priority queue operations, counting all Inserts, Delete-mins, and Decrease-Keys. The next two columns show the number of Inserts/Delete-mins and

Authors:

Code Available From:

Test Machine, Operating System, Compiler, Optimization Level:

Implementation Notes:

Input	Tot Ops	Ins Del	Decr	M. Tot RT	M. Net RT	Range RT	Op Counts
D_rand	782539	262144	258251	2073.7	167.7	1	
D_grid							
D_spbad							
D_gb							
Sort							
Event							

Figure 1: The chart.

of Decrease-keys (usually the number of Delete-mins equals the number of Inserts). In some cases you will have to report an average over three trials. In these columns, print averages in underlined font.

- Total Runtime, in *microseconds*, of your implementation.

- Net Runtime, in *microseconds*. This is the total runtime for your implementation minus the runtime for the Dummy implementation.

- Range. The Maximum Net Runtime minus the Minimum Net Runtime observed across the three runs at this problem size.

- Operation Counts. There could be several columns here. Report mean and the range for relevant operation counts and/or mem counts. These would be most useful if you could identify and count similar operations in a variety of implementations.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] D. E. Knuth, *The Stanford GraphBase: A Platform for Combinatorial Computing*, ACM Press, New York, and Addison-Wesley Publishing Company, Reading Mass, 1993.