Priority Queue Classes with Priority Update

Matti Rintala[1] and Antti Valmari[2]

Department of Pervasive Systems[1], Department of Mathematics[2]
Tampere University of Technology
Tampere, Finland
{Matti.Rintala,Antti.Valmari}@tut.fi

Abstract. A limitation in the design of the interface of C++ standard priority queues is addressed. The use of priority queues in Dijkstra's shortest path search algorithm is used as an example. Priority queues are often implemented using heaps. There is a problem, as it may be necessary to change the priority of an element while it is in the queue, but finding the element from within a heap is costly. The problem may be solved by keeping track, in a variable that is outside the heap, of the position of the element in the heap. Unfortunately, this is impossible with the template class interface used by the C++ standard library priority queue. In this research, the problem is analysed in detail. Four interface designs and corresponding implementations are suggested. They are compared experimentally to each other and the C++ design.

Keywords: data structure, interface, C++ standard library, priority queue, priority update

1 Introduction

Some modern programming languages such as C++ contain high-quality implementations of fundamental data structures in the form of *containers*. This has obviously reduced the need for programmers to implement fundamental data structures on their own.

Unfortunately, the shift from self-made data structures to ready-made containers prevents the exploitation of the data structures to their full potential. This is because the interfaces of containers reflect some idea of how the data structures will be used, making some other uses difficult or impossible. An obvious example is the dynamic order statistics data structure (DOSDS) [5, Ch. 14.1], which consists of a red-black tree with an additional size attribute in each node. Although the C++ standard map is typically implemented as a red-black tree [8, p. 315] and it seems easy to extend it to the DOSDS at the source code level, it seems impossible to implement the DOSDS only using the features offered by the interface of the C++ standard map.

This study focuses on the priority queue. The C++ standard priority queue is implemented as a heap. It has operations for pushing an element to the queue, and finding and removing an element with the highest priority from the queue.

It does not have an operation for changing the priority of an element that is in the queue, although the heap facilitates an efficient implementation of it. The problem is that the efficient implementation requires such co-operation between the queue and its user that is difficult to support in an interface. To solve the problem, we introduce four designs and test them experimentally.

As a use case of the priority queue, Dijkstra's shortest path algorithm is used. Because it is simple and widely known, it serves well for illustrating the problem and the interface designs. We emphasize that the goal of this study is not to make Dijkstra's algorithm as fast as possible (for that purpose, please see [4]). The purpose of the measurements in this study is to demonstrate that our alternative priority queue interface designs are not unrealistically slow.

There are C++ libraries that provide priority queues which are not based on binary heaps, and some of these also allow changing the priorities of queue elements. One such library is Boost.Heap [2], which contains priority queues based on d-ary, binomial, Fibonacci, pairing, and skew heaps. Comparing the performance of these queues to the ones presented in this study is a potential subject for further study. In this study, "heap" always refers to a binary heap, unless stated otherwise.

The use of priority queues in Dijkstra's algorithm is discussed in Section 2. Section 3 presents the alternative designs, and the experiments are reported in Section 4. Section 5 concludes the presentation.

2   Dijkstra's algorithm

Dijkstra's algorithm finds shortest paths in a directed multigraph whose edge weights are non-negative. It uses one vertex as the start vertex. It finds shortest paths from the start vertex to each vertex in increasing order of the length of the path. It may be terminated when a designated target vertex is found, or it may be continued until all vertices that are reachable from the start vertex have been found. We use the latter termination criterion.

Two versions of Dijkstra's algorithm are shown in Figure 1 in pseudocode. Each vertex v has an attribute edges that lists the outgoing edges of v; dist that contains the shortest distance so far known from *start* to v; and prev that tells the previous vertex in a shortest path so far known. Initially each dist attribute contains a special value ∞ that indicates that the vertex has not yet been reached. The edges attributes represent the multigraph and thus, together with *start*, contain the input to the algorithm. The initial values of prev do not matter.

The algorithm on the left in Figure 1 uses a priority queue Q that stores pointers, indices, iterators, or other kind of handles to vertices. For simplicity, we will talk about pushing, popping, and the presence of vertices in Q, although in reality handles to vertices are in question. The operation push(v) pushes v (that is, a handle to v) to Q, and is_empty returns a truth value with the obvious meaning. The operation pop returns a vertex whose dist value is the smallest among the vertices in Q. It also removes the vertex from Q. The operation

```
start.prev := nil; start.dist := 0        start.prev := nil; start.dist := 0
Q.push(start)                             Q.push(start, 0)
while ¬Q.is_empty do                      while ¬Q.is_empty do
  u := Q.pop                                (u, d) := Q.pop
  for e ∈ u.edges do                        if u.dist = d then
    v := e.head; d := u.dist + e.length        for e ∈ u.edges do
    if d < v.dist then                           v := e.head; d := u.dist + e.length
      v.prev := u                                if d < v.dist then
      if v.dist = ∞ then                            v.prev := u; v.dist := d
        v.dist := d; Q.push(v)                      Q.push(v, d)
      else
        v.dist := d; Q.decrease(v)
```

Fig. 1. Dijkstra's algorithm assuming that the priority queue decrease operation is (left) and is not (right) available

decrease(v) informs Q that the dist value of v has become smaller. This causes Q to re-organize its internal data structure.

The queue does not store (copies of) the dist values. The queue operations get the dist values from the vertex data structure. This means either that the code for Q depends on the vertex data structure or that the interface of Q contains features via which Q can access the vertex data that it needs. This issue is discussed extensively in the next sections.

Let us say that the algorithm *processes* a vertex when it is the u of the **for**-loop. The algorithm processes the reachable vertices in the order of their true shortest distance from *start*. It goes through the outgoing edges of u, to see whether a shortest path to u followed by the edge would yield a shorter path to the head state v of the edge than the shortest so far known path to v. If yes, the path and distance information on v is updated accordingly. Furthermore, v is either pushed to Q or its entry in Q is updated, depending on whether v is in Q already.

An example is shown in Figure 2. In it, the outgoing edges of vertices are investigated from top to bottom. Vertex A is used as *start*. Initially, the algorithm marks A found with distance 0 and no predecessor vertex, and pushes it to Q. Then it pops A from Q and processes it. It finds B with distance $0 + 1 = 1$ and predecessor A, and pushes it to Q. Then it does the same to E with distance $0 + 6 = 6$, and then to C with distance $0 + 2 = 2$.
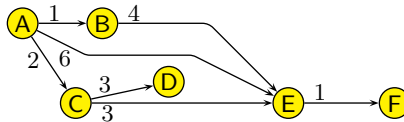
Fig. 2. An example graph for Dijkstra's algorithm

Next it picks B, because its distance 1 is the smallest among those currently in Q. It finds E with distance $1 + 4 = 5$ and predecessor B. It updates the entry for E in Q because of the distance decreasing from 6 to 5. Next the algorithm processes C, finding D with distance 5 which it pushes to Q. Then it finds E anew, but does nothing to it, because the distance does not improve. Now Q contains D and E. Both have distance 5. The algorithm processes them one after the other in an order that depends on implementation details. When processing E, it finds F. The predecessor of F is E, its predecessor is B, and then A, listing a shortest path from A to F backwards.

The C++ standard priority queue is based on a heap. A heap is an array A such that for $1 \leq i < k$, the priority of $A[\lfloor \frac{i-1}{2} \rfloor]$ is at least as high as the priority of $A[i]$, where k is the number of elements in the array. Thus $A[0]$ has the highest priority. We say that $A[\lfloor \frac{i-1}{2} \rfloor]$ is the *parent* of $A[i]$ and $A[i]$ is a *child* of $A[\lfloor \frac{i-1}{2} \rfloor]$. An element has typically and at most two children and one parent. A new element is added by assigning it to $A[k]$, incrementing k, and then swapping it with its parent, the parent of the parent, and so on until it is in the right place. The pop operation returns $A[0]$, moves $A[k-1]$ to its place, decrements k, and then swaps the new $A[0]$ with its higher-priority child until it is in the right place. These require $O(\log k)$ operations.

The fact that the priority queue operations obtain the dist values from the vertex data structure does not prevent the use of the C++ standard priority queue as Q. The dist values are only needed when comparing queue elements to each other. The C++ standard priority queue can be given a special comparison operator that fetches the values that it compares from the vertex data structure. Alternatively, one can define a special type for the queue elements such that its data consists of the handle, and its comparison operator works as described above. These are common practice.

On the other hand, the C++ standard heap and priority queue do not have any decrease(v) operation. Furthermore, decrease(v) cannot be implemented efficiently using the features that they offer (except with exorbitant trickery, see Section 3.5). The main problem is that there is no fast way of finding the handle to v from within A. The handle is not necessarily where push(v) left it, because later push and pop operations may have moved it. As a consequence, in practice, the C++ standard priority queue cannot be used in the algorithm in Figure 1 left.

This problem can be solved by using a self-made heap and by adding an attribute called loc to each vertex. The value v.loc tells the location of the handle to v in A, that is, $A[v.\text{loc}]$ contains a handle to v. When swapping handles in the heap, the heap operations also update the loc values of the vertices to which the handles lead.

Let n denote the number of vertices and m the number of edges in the multigraph. The running time of this implementation is $O(m \log n)$. If also the initialization of the dist attributes to ∞ is counted, then the running time is $O(n + m \log n)$. This formula is slightly different from the corresponding formula in [5, Ch. 24.3], because there all vertices are put initially to the queue, while only

reachable vertices ever enter Q in Figure 1. The edges, head, length, prev, and dist attributes contain the input data and the answer, so they are considered part of the problem and can be left out when comparing the memory consumption of alternative solutions. In addition to them, this solution uses n loc-attributes and at most n handles in the queue, which is $\Theta(n)$ bytes of memory. Assuming that each elementary data item consumes the same amount of memory, the increase in memory consumption is at most 40 %. This occurs if there is an edge from *start* to each other vertex, no other edges, and no other information in vertices and edges than mentioned above. If $m \gg n$, the increase is approximately 0 %.

With the C++ standard priority queue, Dijkstra's algorithm can be implemented as is shown in Figure 1 right. The elements of Q now consist of two components: a handle to the vertex and the dist value of the vertex at the time when it was pushed to Q. A vertex is pushed to Q each time it is found with a shorter distance than before. Therefore, it may be many times in Q. However, each instance has a different distance. The test $u.\mathsf{dist} = d$ recognizes the first time when u is popped from Q. It prevents the processing of the same vertex more than once.

In the case of Figure 2, the algorithm pushes $(\mathsf{A}, 0)$; pops $(\mathsf{A}, 0)$; pushes $(\mathsf{B}, 1)$, $(\mathsf{E}, 6)$ and $(\mathsf{C}, 2)$; pops $(\mathsf{B}, 1)$; pushes $(\mathsf{E}, 5)$; pops $(\mathsf{C}, 2)$; and pushes $(\mathsf{D}, 5)$. It does not push $(\mathsf{E}, 5)$ again, because 5 is not smaller than the dist value that E already had. At this point Q contains $(\mathsf{E}, 6)$, $(\mathsf{E}, 5)$, and $(\mathsf{D}, 5)$. The value of $E.\mathsf{dist}$ is 5. Next the algorithm pops $(\mathsf{D}, 5)$ and $(\mathsf{E}, 5)$ in this or the opposite order, processing D and E. Then it pops $(\mathsf{E}, 6)$. This time it does not process E, because the popped value 6 is different from $E.\mathsf{dist}$ which is 5.

With this implementation, the running time is $O(m \log m)$. This is asymptotically worse than $O(m \log n)$. (With a multigraph, it is not necessarily the case that $m \leq n^2$.) However, with such applications as road maps the difference is insignificant. This is because very few roadcrosses have more than 6 outgoing roads. So $m \leq 6n$, and the number of swappings of heap elements per each push or pop (or decrease) operation is at most approximately $\log_2 6n = \log_2 6 + \log_2 n \approx 2.6 + \log_2 n$, while the corresponding number with Figure 1 left is at most approximately $\log_2 n$. At the level of constant factors, Figure 1 right wins in simpler access of the distances but loses in two data items being exchanged in each swapping instead of one. As a consequence, which one is faster in practice is likely to depend on implementation details and the properties of the input multigraphs.

The additional memory consumption consists now of at most m handles and at most m distances in the queue, which is $\Omega(1)$ and $O(m)$ bytes of memory. If $m \gg n$, this may almost double the memory consumption.

3 Priority queue implementations discussed in this study

Based on the need for a priority queue with updateable priorities, several solutions were designed by the authors, each providing an increased level of encapsulation, modularity, and genericity. These solutions were implemented and

performance tested against each other. Figure 3 shows the structures of the five solutions used in this study (of which the first lacks updateable priorities). Sections 3.1–3.5 discuss these in detail.

3.1 Using STL priority queue with duplicated queue elements

Figure 3(a) shows the structure of a priority queue using STL priority queue. In this approach the lack of priority updates is circumvented by adding the same vertex into the queue several times, if necessary (as described earlier). Each queue element consists of a handle (a pointer, for example) to the graph vertex, as well as the shortest distance to the vertex at the time it was pushed to the queue. The priority queue uses this distance as the priority of the element. In addition to other graph data, each graph vertex contains the calculated shortest distance (originally initialized to infinity or other special value).

Data that is only needed for priority changes is marked in red. In this version that data consists of the queue's internal priority field. In the 64-bit test setup this caused the size of graph vertices to be 24 bytes. The size of the distance field was 4 bytes, the remaining 16 bytes were used for pointers needed by the graph itself and the resulting shortest path, with 4 bytes added by the compiler for memory alignment purposes [1, Ch. 6]. The size of queue elements is 16 bytes (4 bytes for the internal priority field, 8 bytes for the vertex pointer, and additional 4 bytes again for memory alignment).

3.2 Allowing priority updates in a heap using additional location data

Figure 3(b) depicts a solution using a self-made priority queue based on a heap and a separate array for tracking the location of graph vertices in the heap (the location information is drawn as an arrow in the figure, but it can be implemented as an integer index). The additional location array has a corresponding element for each graph vertex in the graph array. This implementation requires that graph vertices are stored in an array so that indices can be used.

Each time a graph vertex is added to the heap, its location index in the heap is stored in the location array. Similarly each time an element is moved in the heap, the heap algorithms update the location index in the array. Moving an element may be caused by adding or removing an element or changing its priority. When the priority of a graph vertex has to be changed, its location in the heap can be found from the location array, so that only approriate elements of the heap have to be updated.

Handling of the location tracking is encapsulated inside the priority queue, which is an improvement to the previous version, where the user of the queue had to take care of duplicate queue elements and the extra distance fields.

In this implementation the only extra data needed for priority changes consists of the location array. In the test setup this adds extra 4 bytes for each graph vertex. The size of the graph vertex itself was again 24 bytes, including 4 bytes for memory alignment. The size of the queue elements in the heap was 8 bytes.
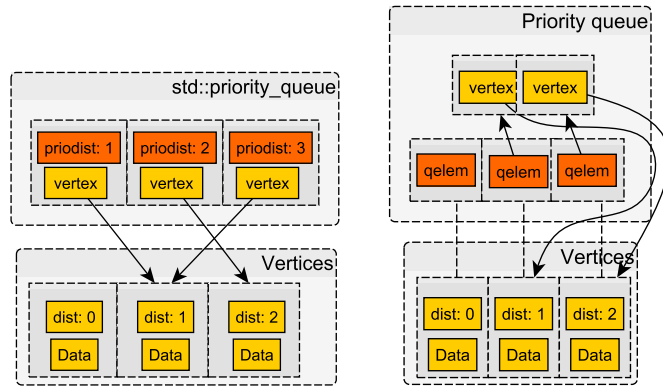
(a) `std::priority_queue` and duplicates

(b) separate index array
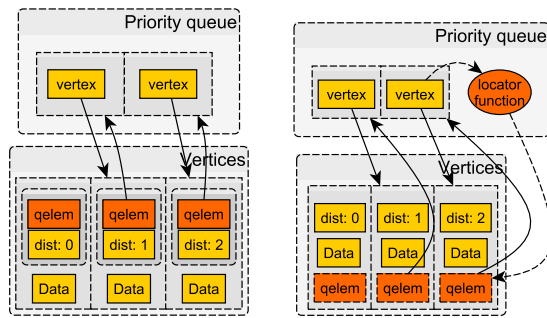
(c) inheritance-based

(d) locator function -based
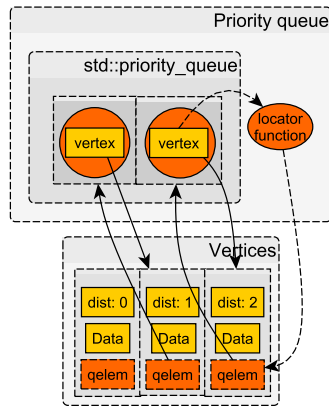
(e) `std::priority_queue` and locator -based

Fig. 3. Priority queue variations used in this study

3.3 Using inheritance to store location information in the graph vertices

The previous algorithm used an internal array for tracking the location of queue elements. This forces the actual graph vertices to be stored in an indexable data structure as well (so that the index of the vertex in the queue can be used to find the location in the location array). It also means that the number of graph vertices has to be known beforehand, so that a location array of suitable size can be allocated in the priority queue.

This situation can be improved by moving the location index inside the graph vertices themselves. However, to preserve modularity it should still be the responsibility of the priority queue to take care of this information. One solution is shown in Figure 3(c). The priority queue provides a base class from which graph vertices should be derived (using object-oriented inheritance). The base class contains both the distance field used as a priority and the location index in the heap. This way the priority queue does not have to know the exact type of the graph vertices, but it can access the base class data to provide a priority queue with updateable priorities. Similarly, the implementor of the graph vertex does not have to know the internal implementation of the queue, just inherit the vertex from the given base class. The priority of a graph vertex is set and changed using the interface of the priority queue, and the value of the priority (i.e., distance) can be queried directly from the base class.

The size of the base class became 8 bytes in the test setup (4 bytes for the distance, 4 bytes for location index), making the size of the graph vertex again 24 bytes. Queue elements were 8 bytes each.

3.4 Adding modularity: externalizing priority and location data

The inheritance-based implementation abstracts the priority and location index into a separate base class, allowing vertex data structures to be created without having to pay attention to data needed by the priority queue. This adds modularity and helps in separation of concerns.

However, even the inheritance-based implementation requires that the programmer implementing the vertex data structure is aware of the priority queue and its special needs. The vertices have to be inherited from the base class provided by the priority queue, and this means that the choice to use the priority queue described in this study has to be made when the vertex data structure is defined. This rules out using the modifiable priority queue with third-party data structures, whose definition cannot be changed. The fact that priority changes have to be made using the base class's interface also forces the code that changes the priority to be aware of the chosen priority queue algorithm.

In order to solve those problems, a fourth implementation of the priority queue is shown in Figure 3(d). It uses additional functions to calculate the location of the priority data and storage needed for the location index. When a priority queue is created, it is given three parameters: the type of data used as queue elements and two functions, one to compare the priorities of two vertices,

and one to calculate the location of the vertex's location index. The priority queue implementation uses these functions each time it has to compare priorities or access the location index.

This approach removes the dependency between the graph vertex data type and the priority queue algorithm. Any data type can be used as a graph vertex as long as it is possible to calculate the priority and the location of the location index based on a handle to the vertex. This makes it possible to use a third-party data type as a graph vertex, and code the name and the location of the priority field into the priority function. Similarly the storage needed for the location index can be embedded inside the graph vertex or in an external data structure.

In fact, this approach is general enough to be used to produce implementations described in Sections 3.2 and 3.3. The first one can be achieved by providing a location index function that calculates the index of a graph vertex and uses that to access an external location index array. Similarly for the inheritance-based approach, the location index function would return a handle to the location index stored inside the base class of the queue element.

The downside of this approach is that using external functions causes additional performance overhead everywhere where the priority or location index is used. Also, using this approach is a little bit more complicated for programmers, since they have to write two small extra functions in addition to the queue element data type.

In this approach, the size of the graph vertex is 24 bytes (4 bytes are again needed for the priority, 4 bytes for the location index). Queue elements are 8 bytes long.

3.5 Test of reuse: Using STL priority queue to create a modifiable queue

The standard C++ already provides a priority queue (`std::priority_queue`). Therefore, it is logical to find out if that queue could be used for creating a priority queue capable of modifying priorities of its elements. As mentioned earlier, the semantics of the standard C++ priority queue assumes that the priority of each element stays constant after the element has been added to the queue.

The benefits of this approach are that the standard C++ priority queue is likely to have been implemented quite optimally (considering the compiler), and this way further development of the standard priority queue benefits the modifiable queue also. Since this implementation is based on the standard C++ priority queue just like the one using duplicate elements, this version can also be used to get an estimate of how much updating the position of each element in the queue costs compared to keeping duplicate queue elements.

The C++ standard guarantees that its priority queue is on a heap, and uses library heap algorithms `std::make_heap`, `std::push_heap`, and `std::pop_heap` to maintain the heap [8, Ch. 12.3, Ch. 11.9.4]. The standard priority queue allows the programmer to specify the underlying data structure for the heap, but by default a vector `std::vector` is used. To make element priorities modifiable,

there have to be mechanisms to check whether an element already exists in the queue, find the element to be updated in the heap, and to update the heap structure based on the new priority.

The first obstacle in this approach is that standard C++ heap algorithms only allow removal of the top element (`pop_heap`) and addition of a new element (`push_heap`). No standard algorithm for updating the position of an existing heap element exists. Fortunately, the implementation provided by the GCC compiler has such operations internally, called `__push_heap` and `__adjust_heap`. The former moves an element up in the heap to its correct position, and the latter moves an element down, if necessary. Using these GCC-specific functions it is possible to readjust the heap after the priority of an element has been changed.

Figure 3(e) shows the structure of this approach. Just like the previous one, this approach is based on a locator function which calculates the location of the location index based on a handle to a graph vertex (making it possible to either embed the location index inside the vertex itself, use a separate location vector, or something else). However, updating the location index when the heap is modified is more difficult. The standard C++ heap algorithms used by the C++ priority queue do not provide means for knowing when the location of a heap element is changed.

In order to keep track of element locations in the heap, this approach uses an auxiliary HeapElement class, and a C++ priority queue is created to store these HeapElements instead of regular data elements. Each HeapElement object contains a handle to a graph vertex. In addition to performing the assignment itself, the assignment operators of HeapElement update the location of the element using the locator function. This way, when a HeapElement object is moved inside the heap by C++ heap algorithms, the HeapElement assignment operators update the location index of each element. To be specific, C++11 now provides two kinds of assignment, *copy assignment* and *move assignment* [7, Ch. 12.8]. Copy assignment copies data and keeps the original intact, whereas move assignment may transfer parts of the original data to its target and reset the original data to an empty state. All C++11 library algorithms prefer move assignment, if available. The HeapElement class only provides move assignment, making sure that multiple copies of the same element cannot be temporarily created during heap algorithms.

After this, implementation of a `std::priority_queue`-based modifiable priority queue is quite straightforward. Standard priority queue operations can be used to add and remove elements from the queue. New operations are provided for modifying priorities of existing elements. In order to change the priority of an element, first the priority data inside the element is modified by the programmer. Then the programmer informs the queue about the priority update, and the queue updates the heap. This of course introduces the risk that the programmer updates the priority but forgets to inform the queue, causing the invariants of the heap to be violated with unpredictable results.

The sizes of the graph vertices and queue elements are the same as in the previous version, 24 bytes and 8 bytes, respectively.

4 Testing the algorithms

In order to test the algorithms, they were used to implement Dijkstra's shortest path algorithm and run through several graphs. This section presents the test results and discussion.

4.1 The test graphs

Figure 4 shows the graph types used for testing the algorithms.

(a) levels graph

(b) matrix graph

(c) San Francisco road network
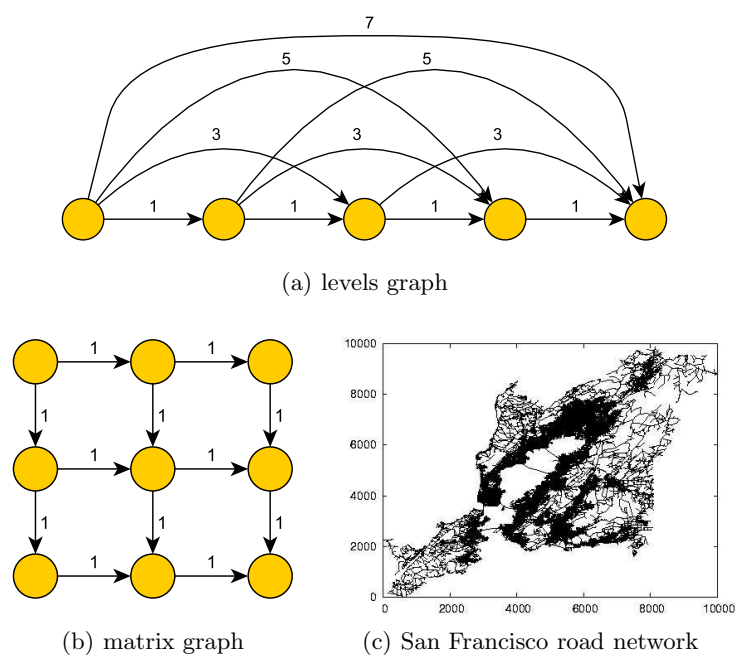
Fig. 4. Graphs used for testing

Graph (a) in Figure 4 depicts a family of graphs that was designed to be increasingly beneficial to the mutable priority queue. In the simplest case (level 1), the graph is simply a series of vertices connected to each other with an edge of length 1 (edges marked "1" in the figure). In level 2 new edges are added. They skip over one vertex and have length 3 (edges marked "3"). Similarly, a level k graph contains for all $i = 1, \ldots, k$ edges of length $2i - 1$ that leave from every possible vertex skipping $i - 1$ vertices.

In Dijkstra's algorithm, this type of graph causes the priority of most vertices to be changed $k - 1$ times for a level k graph. For example, when $k = 2$, each vertex (other than the first two) is first pushed to the priority queue with distance

$d + 3$ (where d is the shortest distance to the previous vertex). Then in the next step a shorter path of length $d+1+1$ is found and the priority must be changed. Similarly, for $k = 3$, most vertices get distances $d+5$, $d+1+3$, and $d+1+1+1$, respectively. Thus, the higher the level, the more the basic STL priority queue -based algorithm has to pay for extraneous elements in the queue. However, the total size of the priority queue stays small (at most k vertices at any time) and does not depend on the total number of vertices.

Graph (b) is designed to work in the opposite direction. The graph is a matrix where each vertex (other than the rightmost and bottommost) has edges of length 1 to its neighbour vertices on the right and below. This means that during Dijkstra's algorithm, the shortest distance to a vertex never has to be updated. On the other hand, the size of the priority queue can grow to k for an $k \times k$ graph. This type of graph forces the mutable priority queue algorithms to pay for updating the location index in the queue, but without benefiting from priority changes.

Finally, graph (c) represents a "real world" example without any intentional bias towards any algorithm. It is the road network of San Francisco with 174,956 vertices and 446,002 edges (each road in the map data was assumed to be two-way). The map data and graph image were obtained from [6].

4.2 The test setup

All tests were run under 64-bit OpenSuse 13.2 Linux with Intel Core i5-3570K processor. GCC 5.1.0 compiler was used with -O3 optimization. Each test was compiled as one compilation unit so that the compiler was able to fully optimize the code.

For increasing the accuracy of the time measurements the operating system was in single-user mode with networking disabled. In each test, the test graph was first created and a dry-run of Dijkstra's algorithm was run to fill CPU caches, etc. Then calculated distances in the graph were zeroed out, and a high-precision time-stamp was acquired from the operating system (using `clock_gettime()` and a raw hardware-based system clock with a claimed resolution of 1 ns). After this, Dijkstra's algorithm was run again, and a new timestamp was acquired at the end of it. The difference between these two timestamps was recorded as the duration of the algorithm. Then distances in the graph were zeroed out again as preparation for repeating the test.

Hardware interrupts and operating system services can slow down the execution of the program, even in the single-user mode. To eliminate extra overhead, each test run of the algorithm was executed at least 10 times or the test was repeated for 30 seconds, whichever took longer. For tests with small graphs, this meant running the tests for millions of times, whereas with large graphs even the 10 test runs could take several minutes to run. The algorithms themselves are fully deterministic, so any fluctuation in the results is caused by outside overhead, which can only increase the measured time durations. Because of this, only the *minimum* running time for each test was chosen, because it has to contain

the least overhead [3]. In practice, the average and minimum running times only differed on average 1.8 % in the tests.

4.3 Test results

For the levels graphs, tests were run for graph sizes of 10^m, for $m = 1, \ldots, 8$ and for levels $k = 1, \ldots, 5$. The graph size had no measured effect on the relative results, which is logical since the length of the priority queue does not depend on the graph size in this test. Therefore, only the results for the largest graph of size 100,000,000 vertices are shown. Figure 5 contains the results of the tests. For each algorithm, the table shows both the absolute time for running Dijkstra's algorithm, as well as the difference percentage compared to the basic algorithm using STL priority queue and duplicate queue elements.

k	(a) `std::p_q`		(b) sep. index		(c) inheritance		(d) locator		(e) `std::p_q&loc`	
1	0.895s	0.00%	0.734s	-18.1%	0.827s	-7.60%	0.827s	-7.60%	0.693s	-22.6%
2	3.23s	0.00%	1.05s	-67.4%	1.40s	-56.7%	1.38s	-57.2%	1.52s	-53.1%
3	5.64s	0.00%	1.54s	-72.6%	2.01s	-64.3%	2.08s	-63.2%	2.13s	-62.3%
4	8.85s	0.00%	2.22s	-74.9%	2.51s	-71.7%	2.56s	-71.1%	2.51s	-71.6%
5	12.4s	0.00%	2.80s	-77.3%	3.06s	-75.2%	3.20s	-74.1%	3.02s	-75.6%

Fig. 5. Results of performance tests with levels graphs of size 100,000,000

The results show that mutable priority queue algorithms presented in this study are faster than the basic STL priority queue algorithm for all test graphs, and the speed difference grows for graphs with more levels. This is as expected, since the priority of a vertex in the queue is changed an increasing number of times when new levels are added to the graph, forcing STL priority queue to pay for duplicate elements in the queue.

It is interesting to notice that mutable queues are faster than the basic STL priority queue also for level 1, where the graph is a simple string of vertices and no priority changes occur. This also holds for the case where a mutable queue is implemented using STL's own priority queue, ruling out differences in the queue's internal implementation. A possible explanation for this difference is that the basic STL-based approach requires an extra distance field in the queue elements, making the queue elements larger.

Figure 6 shows the test results for matrix graphs. Tests were run for graphs of size 10×10, 100×100, 1000×1000, and 10000×10000. Again the table shows both the absolute time for running Dijkstra's algorithm, as well as the difference percentage compared to the algorithm using `std::priority_queue` and duplicate queue elements.

The matrix graph tests show that when the size of the graph (and thus the size of the priority queue) grows, the mutable algorithms become slower compared to the basic STL priority queue. Again this is as expected because in

size	(a) `std::p_q`		(b) sep. index		(c) inheritance		(d) locator		(e) `std::p_q&loc`	
10^2	$2.05\mu s$	0.00%	$1.15\mu s$	-44.0%	$1.62\mu s$	-20.8%	$1.65\mu s$	-19.4%	$1.84\mu s$	-10.1%
100^2	$390\mu s$	0.00%	$324\mu s$	-17.0%	$342\mu s$	-12.3%	$350\mu s$	-10.4%	$401\mu s$	2.72%
1000^2	63.8ms	0.00%	70.3ms	10.3%	63.4ms	-0.52%	62.1ms	-2.62%	61.0ms	-4.26%
10000^2	10.0s	0.00%	18.7s	86.9%	11.9s	19.0%	11.6s	15.6%	15.0s	50.3%

Fig. 6. Results of performance tests with matrix graphs

this test priority changes do not occur, but the mutable queues have to pay for updating the location index of queue elements. It is interesting to see, however, that the mutable algorithms are still somewhat faster for matrix graphs smaller than 1000×1000. This is again probably caused by the additional distance field needed in the unmutable duplicate-based version. It can also be seen that the mutable version using a separate internal location array is the fastest for small graphs, but becomes the slowest for large ones. No obvious reason for this behaviour was found, but differences in memory locality are one possible explanation.

Finally Figure 7 shows the results when the algorithms were used on the San Francisco road network. The figures are averages of 10 test runs with random starting points (the same starting points were used for all algorithms). Individual runs produced results that differed at most 1.7 % from the average, so they are not shown.

size	(a) `std::p_q`		(b) sep. index		(c) inheritance		(d) locator		(e) `std::p_q&loc`	
174956	18.7ms	0.00%	20.0ms	7.34%	19.8ms	6.00%	19.7ms	5.47%	18.3ms	-2.14%

Fig. 7. Results of performance tests with SF road map graph

In this "real" test differences between the algorithms were relatively small. Most of the mutable algorithms were 5–7 % slower than the basic STL priority queue -based one. The results suggest that this graph did not contain enough priority changes for mutable versions to win, and it did not cause the priority queue to grow large enough for the basic version to be clearly faster. Perhaps surprisingly, the modified STL priority queue -based algorithm was 2 % faster than the others. This can probably be attributed to more optimized queue/heap algorithms in the STL priority queue, which can tip the balance a bit to the other direction.

5 Conclusion

The measurements in Section 4 demonstrate that either the simple implementation or our designs are better, depending on the nature of the input graph. With

graphs where the same vertex is found several times with decreasing distance, the performance benefit provided by our designs is significant. On the other hand, with graphs that resemble road maps (Figure 6 and 7), the new designs were slower, sometimes significantly.

Adding the priority change operation to the interface of the priority queue proved to be a trade-off between simplicity and generality. Fortunately, the most general of our designs, the locator function -based, was never so much slower in our experiments that it should be rejected on that basis. Therefore, the locator function -based design seems suitable for a general purpose library.

References

1. Alexander, R. & Bensley, G.: *C++ Footprint and Performance Optimization.* SAMS Publishing, 2000
2. Blechmann, T.: *Boost.Heap.* `http://www.boost.org/doc/libs/1_59_0/doc/html/heap.html`, page last modified 4th August 2015, contents checked 20th August 2015
3. Bryant, R. E. & O'Hallaron, D. R.: *Computer Systems: A Programmer's Perspective*, Chapter 9. Measuring Program Execution Time. Prentice Hall 2003
4. Chen, M. & Chowdhury, R. A. & Ramachandran, V. & Lan Roche, D. & Tong, L.: *Priority Queues and Dijkstra's Algorithm.* UTCS Technical Report TR-07-54, The University of Texas at Austin, Department of Computer Science, October 12, 2007
5. Cormen, T. H. & Leiserson, C. E. & Rivest, R. L. & Stein, C.: *Introduction to Algorithms, 3rd edition.* The MIT Press 2009
6. Feifei, L.: *Real Datasets for Spatial Databases: Road Networks and Points of Interest.* `http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm`, page last modified 9th September 2005, contents checked 11th August 2015
7. ISO/IEC: *Standard for Programming Language C++.* ISO/IEC 2011
8. Josuttis, N.: *The C++ Standard Library, 2nd. Edition.* Addison-Wesley 2012