Dynamic Algorithms Takehome Exam

Magnus Erik Hvass Pedersen (971055)
May 2005, Daimi, University of Aarhus

Introduction

The purpose of this document is to verify attendance of the author to the *Dynamic Algorithms* course, at Department of Computer Science, University of Aarhus. The reader is assumed to be familiar with the course litterature and the problem description for this exam in particular.

The exam has two parts, the first regarding the design and performance analysis of *offline* and *dynamic* versions of algorithms for solving the *Minimum Spanning Forest* problem, and the second and last part, being about the implementation of these algorithms, as well as experimentation.

Although the paper is structured somewhat differently than the project-description, all questions in the exam are believed to have been solved to the expected degree of completion, with the exception of both the *last* 5% questions, which are left unanswered.

Minimum Spanning Forest

Let $G = (V, E)$ be an undirected weighted graph, with vertices V and edges E. We do not require G to be acyclic, but in the present context, having multiple edges between two vertices, is superfluous. Unless otherwise noted, we shall say that G has n vertices (denoted 1 through n), and m edges, each being designated by a triple (i, j, w) with i and j being the nodes that are connected, and w the weight.[1]

A spanning tree of G, is a set of edges $F \subseteq E$, so that whenever two vertices of G are connected by the edges in E, then they are also connected by the edges in F, and furthermore, the spanning tree must be acyclic. A Minimum Spanning Tree is such a spanning tree, that minimizes the sum of the edge-weights in F. When the vertices in G are not fully connected, then we speak of a Minimum Spanning Forest (MSF) instead of Minimum Spanning Tree (MST), but the definitions of spanning and minimalism remain the same. Obviously, an MST is just an MSF for the case where G is fully connected. We shall denote by $MSF(G)$ the MSF for the graph G, and note that this is not necessarily unique for G. Also note that since the MSF must be acyclic, it can at most have $n - 1$ edges, when the entire graph has n vertices.

[1] The problem-description states that the weight w is an integer, but this is not a requirement for the algorithms to work. All we need, is a socalled *metric* on the weight-space.

Kruskal's Algorithm

Assume we are given some graph $G = (V, E)$ and two disjoint partitions of the vertices V into V_1 and V_2. Then from [1, Proposition 10.5, p. 412] we know that if e is an edge between vertices in V_1 and V_2, minimizing the weight of all the edges connecting vertices in V_1 and V_2, then e belongs to some MST of G.

This proposition also holds for MSF's instead of just MST's. To see this, simply make a third set of vertices V_3, which are not connected to any of the vertices in V_1 or V_2. Clearly, having this third set does not change anything, as an MSF for the entire graph, must include the MST over the connected graph having vertices $V_1 \cup V_2$, for which the proposition holds.

An offline algorithm for solving the MST (and MSF) problem using this proposition, is known as *Kruskal's Algorithm* and can also be found in [1, Section 10.2.1]. It works as follows:

- Add all edges to a priority queue, prioritized by the edge-weights.
- Initialize n clusters, and put a different vertex in each.
- Until the priority queue is empty, repeat the following:
  - Remove the front element from the priority queue, that is, some edge (i, j, w).
  - If nodes i and j belong to different clusters, then the edge is part of the MSF, and we merge the two clusters that nodes i and j belong to, and output the edge. Otherwise, the edge is just ignored.

As can be seen, Kruskal's algorithm requires two auxiliary data-structures: A priority queue, and a data-structure that supports efficient cluster-merging. Let us start with the latter.

Union-Find

A general data-structure that supports the creation and merging of clusters or sets, is known as *Union-Find*. It may be implemented in different ways, and one efficient realization can be found in [1, Section 12.1.5], and works by representing a singular set as node, and the union of sets, as a tree of such nodes. That is, when forming the union of two sets, we attach one tree to the other, and in this way, we may identify the compound set, by the root of the resulting tree.

Clearly, a query for set equality, can be made by traversing the trees of the two nodes in question, up to their respective roots. And if the two nodes share the same root, then the two sets are equal.

If we can keep these trees balanced, we can ensure an amortized $O(\log^* n)$ performance on such a query,[2] this can be done by always hooking the tree with fewest nodes up to the tree with more nodes. But with a small trick, this can be improved even further: Whenever we traverse a tree upwards to find the

[2] $\log^*$ is the iterated logarithm, for all practical purposes it is ≤ 4.

root, we unhook the traversed nodes from their parents, and hook them up to the root instead. This is claimed to have time-complexity $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the socalled *Ackermann* function, which essentially, is just a function that grows asymptotically slower than the logarithm.

Priority Queue

The other data-structure needed in Kruskal's algorithm, is a priority queue. However, it is a particular kind of priority-queue, because it only needs to support a sequence of insertions, followed by a sequence of removals. This could just as well be implemented with sorting of the edges according to their weights. There are many efficient sorting algorithms available, and one that is also easy to implement, is *Merge-Sort*, which splits the sequence to be sorted into two, recursively sorts those, and finally merges them back into a single sequence. This obviously runs in time $O(m \log m)$. But let us see if we can do better.

Heap

A full-fledged and efficient priority-queue, is found in the *Heap* data-structure [1, Section 6.3], and takes as it input, pairs of keys and data-elements, and returns as its output, the data-elements according to the sorted order of their keys. The heap is implemented as a balanced binary tree that satisfies the socalled *heap property*, which is, that the key at a node, is less than or equal the keys of the node's children.[3] Thus, the root of the tree has the lowest key, and the root's element is therefore output first. After this, the last element of the tree, that is, the right-most element at the deepest level of the tree, is inserted in the root, and it is *bubbled* downwards in the tree, until the heap property is again satisfied. Similarly for insertions, a new key-element pair is inserted so that the binary tree is kept balanced, and is then bubbled upwards in the tree, until the heap property is satisfied. Both of the insert and removal operations can be done in $O(\log m)$ time, when the heap contains m elements.

Heap Performance in Kruskal's Algorithm

The heap can be initialized with m elements in linear time $O(m)$, and it takes $O(\log m)$ worst-case time to perform an insert or remove on a heap with m elements. Clearly, emptying a heap with m elements, without ever inserting any new elements, would mean a time-complexity sequence of:[4]

$$O(\log m)\ O(\log(m-1))\ O(\log(m-2))\ \cdots\ O(\log 2)\ O(\log 1)$$

or equivalently, a total time-complexity of:

$$\sum_{i=1}^{m} O(\log i)$$

[3] Keys being greater than or equal, reverses the ordering of the heap.
[4] Note that $\log 1 = 0$.

which can be re-written as:

$$\sum_{i=1}^{m} O(\log(i)) = O\left(\sum_{i=1}^{m} \log(i)\right)$$

And from the identities of logarithms, as well as recalling the factorial function $m!$, we find that:

$$\sum_{i=1}^{m} \log(i) = \log\left(\prod_{i=1}^{m} i\right) = \log(m!)$$

Here, Stirling's approximation [1, p. 711] to the factorial function, states that:

$$m! = \sqrt{2\pi m}\left(\frac{m}{e}\right)^m \left(1 + \frac{1}{12m} + \epsilon(m)\right)$$

where $\epsilon(m)$ is the error, and is $O(1/m^2)$. So inserting Stirling's approximation back into the above, gives us:

$$\sum_{i=1}^{m} O(\log(i)) = O(\log(m!)) = O(m\log(m))$$

So even though the heap allows initialization in time $O(m)$, and it is *monotonically* emptied, this emptying process still requires a total time-complexity of $O(m \log m)$, which means that the advantage of the linear-time initialization is lost, and time-complexity-wise, we could just as well have used a sorting of the edges in the Kruskal algorithm, which would have been easier to implement.

Note that the above result holds in general, that is, for any data-structure that is *emptied* or *filled*, and with the corresponding operation of removing or inserting an element having a worst-case time-complexity of $O(\log m)$, the total worst-case time-usage of such an emptying or filling, is still $O(m \log m)$.

Time Complexity of Kruskal's Algorithm

Assuming Kruskal's algorithm is implemented with Union-Find and a heap, and keeping the above analysis of emptying a heap in mind, we may initialize the priority queue (i.e. the heap) by inserting the edges one at a time, which can be done in $O(m \log m)$ time. Initializing the n Union-Find clusters, one for each vertice in G, takes $O(n)$ time in total.

Then concerning the loop, there is one iteration per edge in the priority queue, that is, there are m iterations of the loop, in which we first remove an edge from the heap in $O(\log m)$ time, then we look up the two Union-Find sets in time $O(\alpha(n))$, and if the sets are different, we output the edge in time $O(1)$, and perform the union-operation on the two Union-Find sets, again in time $O(\alpha(n))$.

This means that we have a total time complexity of $O(m \log m + n + m(\log m + \alpha(n)))$ which is $O(n + m(\log m + \alpha(n)))$. Now, if we do not allow multiple edges

between two vertices in G, then we can at most have:

$$m \leq \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

edges in G.[5] Furthermore, we may assume that there are at least $m \geq n-1$ edges, because otherwise there will be vertices that are not connected. We can preprocess the graph in time $O(n+m)$ to weed out any such unconnected vertices: First flag all vertices as unused, this takes $O(n)$ time, then run through all the edges and flag the vertices that the edges connect, as being used. This takes time $O(m)$, so the total time-complexity of this weeding is $O(n+m)$, and therefore does not alter the time-complexity of Kruskal's algorithm. To recap, we may therefore assume:

$$n-1 \leq m \leq \frac{n(n+1)}{2}$$

Inserting the worst-case $O(n^2)$ into the time-complexity of Kruskal's algorithm, we obtain $O(n+m(\log n^2+\alpha(n)))$, and noting that $O(\log n^2) = O(\log n)$ we have $O(n+m(\log n+\alpha(n)))$. Because we can assume that m dominates n (or at least $m = n-1$, which means that we at least have $O(m) = O(n)$), and we know that $O(\log n)$ dominates $\alpha(n)$, then Kruskal's algorithm has time-complexity $O(m \log n)$.

Sparsification

To understand *sparsification*, let us first describe a similar concept from number theory.

Repeated Squaring

First let $[x]_d$ define the unique remainder that results from dividing $x \in \mathbb{Z}$ with $d > 0$. If d is implicitly given, we shall just write $[x]$, and one can then prove that:

$$[xy] = [[x][y]]$$

which may be employed recursively, in an algorithmic scheme known as *repeated squaring* (see for example [2, p. 7]), for easily and efficiently computing the remainder of very large products, typically on the form x^y (hence the name repeated *squaring*, as y is divided by 2 when even, thus squaring x), but more generally, the remainder of products of a number of factors $x_i \in \mathbb{Z}$ can also be computed by employing this rule.

Indeed, one may transform this directly into a dynamic algorithm, which allows for the dynamic insertion and removal of factors x_i, without recomputing the remainder over the entire product. Such an algorithm would simply

[5] Note that we allow an edge to connect a vertex to itself.

store the intermediate values of applying the above rule in the nodes of a binary tree, and whenever a factor was added, a new leaf would be added to the tree, and each internal node on the way to the root, would have to be recomputed. Keeping the tree balanced, would mean logarithmic execution time of such an insert-operation. Similarly, removal of a factor, would mean that the leaf with that factor was removed, the tree manipulated to re-balance it, and each affected internal node being recomputed. Querying the dynamic data-structure for obtaining the remainder for the entire product, could then be done in constant time, as it would simply be the value stored in the root of the binary tree.

Whether this dynamic version would be useful for anything, is not of any concern here, but it illustrates well, the algorithmic concept behind sparsification.[6]

Sparsification of MSF

For convenience, let $MSF(E)$ denote an MSF for the graph with edges E and the implicitly given vertices V. Sparsification of MSF works by maintaining a balanced binary tree T, where each node contains the edges in the MSF of the edges in the node's children. More specifically, if we associate with a node a set of edges E, and partition this set into two disjoint sub-sets E_1 and E_2, then computing $MSF(E_1)$ in the left child-node and $MSF(E_2)$ in the right child-node, we will briefly argue below, that there exist an MSF over E, so that:

$$MSF(E) = MSF\left(MSF(E_1) \cup MSF(E_2)\right) \tag{1}$$

For the leafs in the binary tree, we have sets E_i that contain only a single edge each, for which the MSF is trivially given by:

$$MSF(E_i) = E_i, \text{ when } |E_i| = 1$$

So inserting a new edge in the graph G, we also insert a new leaf in the binary tree T, while ensuring the tree is kept balanced, and then re-compute the MSF for all affected nodes. Similarly when removing an edge, we remove the leaf in question, re-balance T, and re-compute the MSF for all affected nodes. This is described in more detail below.

The project-description does not call for a proof of Eq.(1), and since a formal proof appears to be rather lengthy, we shall suffice with an intuitive argument of its validity. The difficulty in proving this formally, comes from the MSF not being unique. If we informally note that any partitioning of the graph's vertices into three sets, the first two only containing vertices that mutually connected, and the third set containing the rest of the vertices, then this partitioning contributes a single edge to the graph's MSF (that is, if any edges connect vertices

[6] Another similar but perhaps more interesting dynamic algorithm for solving a mathematical problem, is the computation of the maximum or minimum value of a sequence of values, where we may then insert and remove values, and efficiently compute the new max or min values.

between the first two partitions). This edge is unique up until its weight being minimal over the set of edges connecting these two sets of vertices. This is essentially shown in [1, Proposition 10.5]. Clearly, for every edge e in any $MSF(E)$, it must still be unique in this regard for any subset of edges $E' \subseteq E$, with $e \in E'$. To see this, consider the same partitioning of vertices as before, only now with the edges E' instead of E. The set of edges connecting vertices between the first two vertex-sets, is at most the same as before, but it may have some of its edges removed. The edge e of course still has minimal-weight over the vertex-connecting subset of edges, and hence e is unique in this regard, and therefore it is in (some) $MSF(E')$ also. So, the union $MSF(E_1) \cup MSF(E_2)$ may contain more edges than $MSF(E)$, but it contains at least the edges of some $MSF(E)$. Then taking the MSF over this union, weeds out the additional edges, and we end up with an MSF over E.

Note that if we can prove a similar relationship for another problem than MSF, then we may sparsify that problem in the same manner as was done with MSF.

Edge Insertion & Removal in MSF Sparsification

Let us give a bit more details about the insertion and removal of edges in the graph G, and how we maintain the balanced binary tree T.

Recall that a tree is balanced, if each leaf has depth $d-1$ or d, with d being the maximum depth of the tree, and each leaf of depth $d-1$ occurs to the right in the tree, and leafs of depth d occur to the left in the tree. If we store a pointer to the rightmost leaf of depth d, then we may find the next position in which to insert a leaf that satisfies this depth-criterion, in time $O(d)$, by using an algorithm which will be given in the description of the implementation below. Similarly, if we remove the rightmost leaf of depth d, then we may update the pointer to the new rightmost leaf in time $O(d)$.

Upon insertion of an edge e' in G, we add the set $E' = \{e'\}$ as a leaf in T, at the next position that will satisfy the above depth-criterion. After this leaf-insertion, we need to update the MSF's stored at the internal nodes of T. Specifically, we recompute the MSF's for the nodes that are on the path from the new leaf and to the root of the tree, starting at the leaf.

For the removal-operation, we first unhook the bottom- and right-most node of the tree, then we delete the leaf to be removed, and put the bottom- and right-most node we just unhooked, where the leaf was. This requires two paths in the binary tree to have their MSF's updated. A few special cases are when unhooking and deleting the last node and the leaf, but these cases will be covered below, in the description of the implementation.

The algorithms for keeping the binary tree balanced are similar to those used in the heap, with the small exception that we always ensure that leafs in T remain leafs, even after a new leaf is inserted, or one is removed. Fortunately, this is not too difficult, if we observe that each internal node in the tree, has nodes connected in both its children. Then upon inserting a new leaf in a non-empty tree, we need to create a new internal node, store as its left child

the appropriate and existing leaf from T, replace the new internal node with that leaf, and store the new leaf in as internal node's right child. Similar when removing a leaf, we know that it is either the root or the right-child of an internal node, so in the latter case, we simply replace its parent (which is an internal node), with the leaf's sibling, which is either an internal node or a leaf itself.

MSF Query

Querying the sparsification data-structure to obtain the MSF of the entire graph G, is then simply a matter of outputting the edges stored at the root of the binary tree T, as it is re-computed on every insertion and removal of an edge.

Time Complexity of MSF Sparsification Operations

Recall that since the MSF is acyclic, it has at the most, $n - 1$ edges. When computing the MSF associated with a node in the binary tree T, it will be computed from the two sets of MSF-edges stored in that node's children-nodes. That is, in computing the MSF for any node in the binary tree T, it will be done from $2(n - 1)$ edges at the most. Using Kruskal's algorithm to compute the MSF over $2(n - 1)$ edges, therefore takes time $O(n \log n)$.

The binary tree T holding the sparsification data-structure, is balanced and therefore has logarithmic height, in the number of leafs, of which there is one for each edge in G, meaning there are m leafs in T. But in the analysis of the time-complexity of Kruskal's algorithm, we saw that we can assume m to be bounded by $n - 1$ and $n(n + 1)/2$. So the height of the tree T is at most $O(\log m) = O(\log n^2) = O(\log n)$.

When inserting or removing an edge we need to re-compute the MSF's for each node on the way to the path, for at most two such paths. These paths can be no longer than the height of the tree, that is, $O(\log n)$ nodes. Since computing the MSF for each node takes time $O(n \log n)$, and we must perform $O(\log n)$ such re-computations, the time-complexity for an insert or remove operation, is $O(n \log^2 n)$.

As described above, querying the data-structure to obtain the MSF of the entire graph G, is simply a matter of outputting each of the edges stored in the root of the binary tree T. Again, there are at most $O(n)$ edges in any MSF, so outputting the edges stored in the root of T, can naturally also be done in $O(n)$ time.

Problem 1

This section briefly summarizes the solutions to the subproblems, as well as solves the remaining.

Subproblem 1.1

Kruskal's algorithm was described, including its auxiliary data-structures, their performance, and the overall time-complexity for Kruskal's algorithm. Although not explicitly stated, it was assumed that weights can be compared in constant time. It should be noted that for most practical purposes, this is by no means a restriction, and the assumption is more of theoretical importance than practical. Under this assumption, the time-complexity for computing the MSF using Kruskal's algorithm, was found to be $O(m \log n)$.

Subproblem 1.2

For the graph with 8 vertices and the 7 edges:

$$E = \{(1,2,2),\ (1,5,4),\ (1,8,8),\ (2,5,3),\ (2,8,6),\ (4,6,1),\ (5,8,7)\}$$

the binary sparsification tree T is shown in figure 1. As the problem-description notes, the tree is not unique, and indeed, we could likely get even sparser MSF's stored at the internal nodes, if we had chosen the ordering of the edges into leafs more carefully.

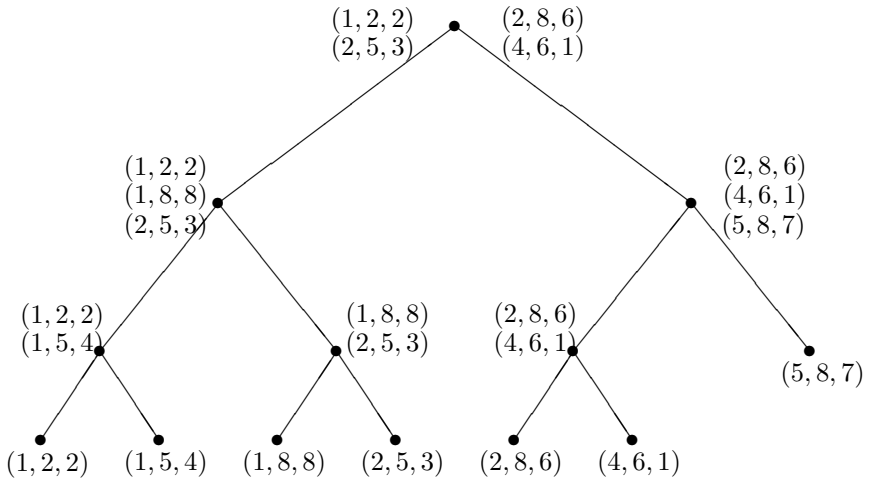

Figure 1: The binary sparsification tree T for the edges in subproblem 1.2.

Subproblem 1.3

This subproblem is expressed in a slightly different way in Eq.(1), and is dealt with in detail above.

Subproblem 1.4

Regarding the operations `insert`, `delete`, and `MSF?`, they are also detailed in the above, including their time complexities.

Implementation

Implementation is done in *MS Visual C++ .NET* and compiles to a *MS Windows* executable. The source-code makes heavy use of socalled *template*-classes, which allow datatypes to be defined as abstract types, so that classes may be instantiated for many different classes. Although we could have used the *Standard Template Library* (STL) for such things as the heap or sorting algorithms, it was unclear what the project's desired level of completion was, so the heap was also implemented, and STL has only been used for more trivial data-structures such as linked lists.

This section only describes essential points regarding the implementation and source-code, and the reader is referred to the actual source-code, for details on class-declarations, etc. Some of which are a bit intricate because of the extensive use of template programming.

Union-Find

The Union-Find data-structure is implemented in the `LUnionFind`-class which instantiates objects of the `LUnionFindSet`-class as needed, thus building trees from such objects. All members in the latter are declared *protected*, and `LUnionFind` is then declared as a friend-class, allowing it access to these protected members. This means that only functions in the `LUnionFind`-class may create and destroy objects of the `LUnionFindSet`-class. Anyone may still reference `LUnionFindSet`-objects however, thus providing an abstract handle for a cluster or set in the Union-Find data-structure. For example in Kruskal's algorithm, each node in the graph G, stores such an abstract reference to a `LUnionFindSet`-object, which it supplies when querying the Union-Find data-structure.

The functions of the `LUnionFind`-class are more or less trivial pointer manipulations, but we should single out the following function, which not only looks up the root of some `LUnionFindSet` instance, but during this recursive traversal, hooks up all intermediate nodes to the root-node:

```
LUnionFindSet*
LUnionFind::DoFind (LUnionFindSet* a)
{
    assert(a);

    LUnionFindSet* parent = a->GetParent();
    LUnionFindSet* root;

    if (parent)
```

```
    {
        // Unhook node 'a' from its parent.
        parent->SubNumChildren(a->GetNumChildren()+1);
        a->SetParent(0); // This is not necessary, but
                         // makes it easier to understand.

        // Recursively traverse upwards the tree.
        root = DoFind(parent);

        // Hook node 'a' up to the real root.
        a->SetParent(root);
        root->AddNumChildren(a->GetNumChildren()+1);
    }
    else
    {
        // This means 'a' is the root.
        root = a;
    }

    return root;
}
```

Note the use of the `assert()`-function, which is not compiled into the final program, and therefore does not consume any resources there. Assertions are not only beneficial for debugging, but also for letting the developer quickly knowing what is expected from the parameters, and in other cases, ensuring validity of invariants, states, etc.

Binary Tree Node

Both the heap and the dynamic MSF algorithm use a binary tree, and their algorithms for manipulating the tree are somewhat similar. For this reason the class `LBinaryTreeNode` has been made, containing pointers to its parent node, as well as its left and right children nodes. These pointers are typed by the template-type `TNode` which is supposed to be the sub-class of `LBinaryTreeNode` in question, e.g. `LHeapNode` for the heap. It requires a number of slightly intricate forward declarations and such to make it work, but it alleviates the need for type-casting, or having similar code several times, while providing appropriate scope-protection.

Some may find this style of template programming abusive, but it has its clear advantages. One should keep in mind, that C++ does not have the most graceful syntax to begin with, in particular when it comes to more sophisticated object-oriented hierarchies.

Most of the functions in `LBinaryTreeNode` are straight-forward pointer queries or manipulations, and the only function that deserves special mentioning here, is used for finding the bottom- and right-most node, after removal of the current

one (which is assumed to be the object for which the function is invoked on). This function is used both in the heap data-structure and in maintaining the sparsification tree, and is as follows:

```
template <class TNode>
TNode* LBinaryTreeNode<TNode>::FindLast ()
{
    TNode* w = static_cast<TNode*>(this);

    // Go up the tree until root or a right-child is found.
    while (!(w->IsRoot() || (w->IsRightChild())))
    {
        w = w->GetParent();
    }

    TNode* node;

    // Find start-node for down-traversal of the tree.
    if (w->IsRoot())
    {
        node = w;
    }
    else // w is right-child of some parent-node.
    {
        node = w->GetLeftSibling();
    }

    // Traverse down the tree until the last right-child is found.
    while (node->GetRightChild() != 0)
    {
        node = node->GetRightChild();
    }

    return node;
}
```

The algorithm is implemented from the description in [1] in regards to the heap.

A number of recursive functions for checking whether a binary tree is consistent and balanced, are also provided. These functions are called as assertions in the heap and dynamic MSF algorithms, after insertion and removal operations, and particularly in the latter, assisted a great deal in debugging.

Heap

The heap is implemented in the class `LHeap` with the auxiliary class `LHeapNode` deriving from `LBinaryTreeNode` as described above, and furthermore providing a key and an element.

The heap implementation allows for a comparison-class as a template-argument, with the default argument being `std::less_equal`, which means the heap is ordered with smallest key on the top. Instantiating the heap with another comparison-class, would give us a different ordering, for example greatest key on top.

The `LHeap`-class contains a number of functions, of varying difficulty. The functions for insertion and removal of elements in the heap, need to take special cases into account, for example when the heap is empty. The functions `DownHeapBubble()` and `UpHeapBubble()` do just that, bubble elements down or up in the binary tree, to restore the heap order. The most complex function is the one that finds the appropriate place for inserting a new node, so as to maintain a balanced tree. It is similar in flavor to `FindLast()` from the `LBinaryTreeNode`-class, albeit slightly more complex, as we need to keep a reference to the child-pointer that is to be updated:

```
template <typename TElm, typename TKey, class LT>
void LHeap<TElm,TKey,LT>::InsertLast (TNode* newLastNode)
{
    assert(mLastNode);
    TNode* w = mLastNode;

    // Go up the tree until root or a left-child is found.
    while (!(w->IsRoot() || (w->IsLeftChild())))
    {
        w = w->GetParent();
    }

    TNode* node;
    TNode** child;

    // Find start-node for down-traversal of the tree.
    if (w->IsRoot())
    {
        node = w;
        child = &(w->mLeftChild);
    }
    else // w is left-child of some parent-node.
    {
        node = w->GetParent();
        child = w->GetRightSiblingPtr();
    }

    // Traverse down the tree until an empty left-child is found.
    while (*child != 0)
    {
        node = *child;
```

```
        child = &node->mLeftChild;
    }

    // Insert the new last-node in the empty left-child that was
    // just found.
    *child = newLastNode;

    // Update the new last-node's parent.
    newLastNode->SetParent(node);
}
```

Kruskal's Algorithm

In the `LMinSpanForestKruskal`-class we find the implementation of Kruskal's algorithm. Again, the class is a template-class, allowing for different types of weights on the edges. The class maintains a list of all the edges, and in debug-mode, the insert operation checks to see if the edge is already contained in the graph. The function implementing Kruskal's algorithm is as follows:

```
template <typename T>
void LMinSpanForestKruskal<T>::MSF (TEdges& edges)
{
    // Ensure that the supplied list is empty.
    edges.clear();

    // Create the Union-Find data-structure. O(1)
    LUnionFind unionFind;

    // ... and create a Union-Find set for each vertice in
    // the graph. O(n)
    for (int i=0; i<kN; i++)
    {
        mVerticeSets[i] = unionFind.MakeSet();
    }

    // Create the heap that is to contain the edges. O(1)
    LHeap<TEdge, T> heap;

    // ... and fill the heap with the edges. O(m*log(m))
    TEdges::iterator itor;
    for (itor=mEdges.begin(); itor != mEdges.end(); itor++)
    {
        TEdge& edge = *itor;
        heap.Insert(edge, edge.GetWeight());
    }
```

```
    // Empty the heap of edges, thus finding the MSF.
    // O(m*(log(m)+a(n)))
    while (!heap.IsEmpty())
    {
        // Remove from the heap, the edge with the lowest weight.
        // O(log(m))
        TEdge edge = heap.Peek();
        heap.Remove();

        // The sets associated with the vertices connected by the
        // edge. O(1)
        LUnionFindSet* iSet = mVerticeSets[edge.GetI()];
        LUnionFindSet* jSet = mVerticeSets[edge.GetJ()];

        // If the vertices are in different sets, then edge is in
        // MSF. O(a(n))
        if (!unionFind.SameSet(iSet, jSet))
        {
            // Add edge to MSF. O(1)
            edges.push_back(edge);

            // Union or merge the sets for the two vertices.
            // O(a(n))
            unionFind.Union(iSet, jSet);
        }
    }
}
```

Where it should be noted, that `mVerticeSets` is a member-field holding an array of pointers to Union-Find sets, alleviating the need to allocate it on each execution of the `MSF()` function – this may seem redundant as the data-structure is most often used in an offline manner, so `MSF()` is only called once anyway.

Dynamic MSF Algorithm

The dynamic MSF algorithm uses a binary tree, so we sub-class `LBinaryTreeNode` into `LMSFDynNode`, and from this we sub-class into `LMSFDynNodeInternal` and `LMSFDynNodeLeaf`. Apart from being node's in a binary tree, these classes provide two functions of relevance to MSF, namely `GetMSF()` and `ComputeMSF()`, where the latter is used for recursively re-computing the MSF's in the binary tree, for each internal node on the way to the root, and `GetMSF()` simply returns (well, copies) the edges in the MSF that was just computed. The specialization of this function in `LMSFDynNodeInternal`, is as follows:

```
virtual void ComputeMSF (int const n)
{
    // Get MSF's for children.
```

```
    TNode* leftChild = GetLeftChild();
    TNode* rightChild = GetRightChild();
    TEdges edges;

    // ... add MSF edges from left child (if any).
    if (leftChild)
    {
        leftChild->GetMSF(edges);
    }

    // ... add MSF edges from right child (if any).
    if (rightChild)
    {
        rightChild->GetMSF(edges);
    }

    // Create and execute offline MSF-algorithm (Kruskal).
    LMinSpanForestKruskal<T> msfOffline(n);
    msfOffline.Insert(edges);
    mEdgesMSF.clear();
    msfOffline.MSF(mEdgesMSF);

    // Call overloaded function to recurse up through tree.
    LMSFDynNode<T>::ComputeMSF(n);
}
```

Note that before the call to Kruskal's algorithm, we clear the node's list of MSF-edges. This list however, does not contain more than $O(n)$ edges, and so does not alter the overall time-complexity.

The tree of `LMSFDynNode`-instances is maintained by the `LMinSpanForestDynamic`-class. Because some special cases are needed when inserting and removing nodes, the functions are printed here. First is the main insert-function:

```
template <typename T>
LMSFDynNodeLeaf<T>* LMinSpanForestDynamic<T>::Insert (TEdge const& edge)
{
    TLeaf* leaf = new TLeaf(edge, 0);

    // Insert in binary tree.
    if (IsEmpty())
    {
        // Make leaf the only node in the tree.
        // No MSF-recomputation is necessary.
        mTopNode = leaf;
    }
    else // !IsEmpty()
    {
```

```
        // Insert the leaf as the bottom- and right-most leaf.
        // MSF-recomputation is done.
        InsertLast(leaf);
    }

    // Update pointer to the bottom- and right-most leaf.
    mLastNode = leaf;

    // Ensure tree is valid after leaf-insertion.
    AssertValid();

    return leaf;
}
```

Note the call to `AssertValid()` which checks various aspects of the binary tree. Also note that a pointer to the leaf is returned to the user, and serves as the handle when the user wants to remove it from the data-structure again. The user may not delete the object pointed to, because the class' destructor is scope-protected. In case the tree was non-empty, the `Insert()`-function calls `InsertLast()` with the new leaf, to insert it as the new bottom- and right-most node of the binary tree, so as to keep it balanced. The function is given by:

```
template <typename T>
void LMinSpanForestDynamic<T>::InsertLast (TLeaf* newLeaf)
{
    assert(mLastNode);
    TNode* w = mLastNode;

    // Go up the tree until root or a left-child is found.
    while (!(w->IsRoot() || w->IsLeftChild()))
    {
        w = w->GetParent();
    }

    TNode* node;

    // Find start-node for down-traversal of the tree.
    if (w->IsRoot())
    {
        node = w;
    }
    else // w is left-child of some parent-node.
    {
        node = w->GetRightSibling();
    }

    // Traverse down the tree until a leaf is found.
```

```
    while (node->GetLeftChild() != 0)
    {
        node = node->GetLeftChild();
    }

    // Find parent (if any), and create new inner node in
    // place of existing leaf.
    TNode* parent = node->GetParent();
    TNodeInternal* innerNode = new TNodeInternal(parent);

    // Hook previous leaf up as new inner-node's left child.
    node->SetParent(innerNode);
    innerNode->SetLeftChild(node);

    // Hook new leaf up as new inner-node's right child.
    newLeaf->SetParent(innerNode);
    innerNode->SetRightChild(newLeaf);

    if (parent)
    {
        parent->ReplaceChild(node, innerNode);
    }
    else
    {
        mTopNode = innerNode;
    }

    // Re-compute MSFs on the way to the binary tree's root.
    innerNode->ComputeMSF(kN);
}
```

Note the call to `ComputeMSF()`, whose parameter is the number of vertices n in the graph.

The main function for deletion of a leaf is more involved, as we need to take care of a few special cases. For example when the leaf to be removed, is the current bottom- and right-most node of the tree, or if the leaf is the next such node, after removal of the current. The function is as follows:

```
template <typename T>
void LMinSpanForestDynamic<T>::Delete (LMSFDynNodeLeaf<T>* leaf)
{
    assert(leaf);

    if (leaf->IsRoot())
    {
        // Only a single node exists in the tree,
        // it will be deleted below.
```

```
    mTopNode = mLastNode = 0;
}
else // More than one node exists in the tree.
{
    // Unhook and update mLastNode, but don't delete current one.
    TNode* oldLastNode = mLastNode;
    UnhookLast();

    // Only replace leaf with oldLastNode, if
    // leaf was not the previous last-node.
    if (oldLastNode != leaf)
    {
        // Get the leaf's parent (if any).
        TNode* parent = leaf->GetParent();

        // Hookup oldLastNode to leaf's parent (may be zero).
        oldLastNode->SetParent(parent);

        // Hookup oldLastNode instead of leaf.
        if (parent)
        {
            // Replace leaf in parent with oldLastNode.
            parent->ReplaceChild(leaf, oldLastNode);

            // Re-compute MSFs on the way to the binary tree's root.
            parent->ComputeMSF(kN);

            // Correct new last-node if it was the leaf which
            // we are deleting.
            if (mLastNode == leaf)
            {
                mLastNode = oldLastNode;
            }
        }
        else // no parent
        {
            // Leaf is the only remaining node in the tree,
            // so replace it with oldLastNode.
            mTopNode = mLastNode = oldLastNode;
        }
    }
}

// Delete the leaf.
delete leaf;
```

```
    // Ensure tree is valid after leaf-deletion.
    AssertValid();
}
```

And furthermore, we need a function used by `Remove()`, for unhooking the bottom- and right-most node of the tree, it is as follows:

```
template <typename T>
void LMinSpanForestDynamic<T>::UnhookLast ()
{
    assert(!IsEmpty());
    assert(!mLastNode->IsRoot() && mLastNode->IsRightChild());

    TNode* parent = mLastNode->GetParent();
    TNode* leftSibling = mLastNode->GetLeftSibling();
    TNode* parentParent = parent->GetParent();

    // Unhook the last-node and its sibling from their mutual parent.
    mLastNode->UnhookFromParent();
    leftSibling->UnhookFromParent();

    if (parentParent)
    {
        parentParent->ReplaceChild(parent, leftSibling);
    }

    // Hook up the remaining leaf to the parent's parent (which may
    // be zero).
    leftSibling->SetParent(parentParent);

    // Delete the inner-node which is no longer used.
    delete parent;

    // Find out which nodes are now the top- and last-nodes of the
    // binary tree.
    if (leftSibling->IsRoot())
    {
        // leftSibling is the only node left in tree; make it top-
        // and last-node. Since it is the only node left in tree,
        // it is therefore also a leaf.
        mTopNode = mLastNode = leftSibling;
        mLastNode = leftSibling;
    }
    else
    {
        // Traverse tree to find new last-node.
        TNode* newLastNode = leftSibling->FindLast();
```

```
        mLastNode = newLastNode;

        // Re-compute MSFs from the last-node just removed to root
        // of tree.
        leftSibling->ComputeMSF(kN);
    }

    // Ensure tree is valid after mLastNode unhooking.
    AssertValid();
}
```

Recall that the `MSF()`-function for this dynamic version, is just the copying of the elements stored in the tree's top-node:

```
virtual void MSF (TEdges& edges)
{
    assert(mTopNode);

    edges.clear();
    mTopNode->GetMSF(edges);
}
```

Problem 2

This section briefly summarizes the solutions to the subproblems, as well as solves the remaining.

Subproblem 2.1

The implementation of Kruskal's algorithm is described in the above. Note that it is indeed possible to replace the `LMinSpanForestKruskal`-class with a single function. The purpose of the class was originally to provide a uniform interface to both the dynamic and Kruskal's algorithm for computing MSF. This was not fully achieved, due to time constraints, and would have required for a uniform way to address an edge in the internal data-structures of the `LMinSpanForest`-classes. It is provided in the dynamic version, but the offline version still references internal edges by their vertice-connectivity (i.e. a number (i, j)).

As a linked list is used to store the edges internally in the offline algorithm, we require $O(m)$ time to look them up again, for example as is done in the `HasEdge()` function, or the `Remove()` function. This however, does not alter the overall time comlexity for the implementation of Kruskal's algorithm itself.

Subproblem 2.2

The implementation of the dynamic algorithm for solving MSF, based on sparsification, is also detailed above.

Subproblem 2.3

Here we argue for the efficiency and correctness of the implementations.

Efficiency

In regards to efficiency of the implementation, the chosen programming language (C++) ensures that no unexpected non-constant overhead is incurred, as it would in e.g. a language based on runtime interpretation. C++ does have its quirks that we must be careful about however. In particular, one must pass parameters by reference instead of by value, whenever possible. In particular when passing larger data-structures, such as linked lists, passing by value would mean a copy had to be made, and unless this is specifically desired, it is a very costly overhead.

The implementation does have some unneeded overhead though. For example in the dynamic algorithm, we do not need to copy lists as much, and the same is the case with Kruskal's algorithm. However, it is important to note that the list-copying in question do not infer a larger time-complexity, and it is believed that it makes the source-code more modular this way and easier to maintain. Clever rewrites could probably retain (most of) the modularity and ease of maintainance, while getting rid of this copying.

Correctness

Regarding correctness of the implementations, we saw in the above that the abstract algorithms are correct, and the implementations are therefore also correct, if they are a true rendition of these abstract algorithms. We shall use a two-fold argument that this is so. First, we use assertions nearly whenever possible. In particular, we have assertions that check the binary treas of the heap and the dynamic MSF algorithm, after insertion and removal of nodes. It is true that these assertions may have bugs in them also, but since they are by far much simpler than the tree-manipulating algorithms, the chance of a bug going undiscovered is relatively low. The second part of the correctness-argument, is that we make a number of test-executions, until no more errors are uncovered. These executions are of course made in debug-mode, so the assertions will have effect.

We use the edges from subproblem 1.2 for our test-executions. We make 7 executions with Kruskal's algorithm (which is also implicitly tested in the dynamic MSF algorithm), where we start with the first edge $(1, 2, 2)$, and then add an edge in each query. For the first edge, Kruskal's algorithm reports that the edge $(1, 2, 2)$ is the MSF – as expected. All edges that are added are part of the MSF, until we reach the fourth edge $(2, 5, 3)$, in which case the MSF is reported to consist of edges $(1, 2)$, $(2, 5)$, and $(1, 8)$. Adding the fifth edge, Kruskal reports the edges $(1, 2)$, $(2, 5)$, and $(2, 8)$ to be an MSF. Adding the sixth edge $(4, 6, 1)$ also merely adds this to the MSF. And adding the seventh edge $(5, 8, 7)$ does not change this.

Checking these results by hand, we see that they are correct, and as no assertions failed, we shall take this as a good indication, that the implementation of Kruskal's algorithm is correct.

In the dynamic MSF algorithm, we execute the same tests, but we start by adding all edges, and then remove them one at a time, so as to test the leaf-removal while we are at it. If we remove the first edge $(1, 2, 2)$, then the MSF is reported to have the edges $(4, 6)$, $(2, 5)$, $(1, 5)$, and $(2, 8)$. Continuing to delete the second edge also, the MSF is reported to be $(4, 6)$, $(2, 5)$, $(2, 8)$, and $(1, 8)$. Removing the third edge gives us an MSF of $(4, 6)$, $(2, 5)$, and $(2, 8)$. Removing the fourth edge gives us an MSF with edges $(4, 6)$, $(2, 8)$, and $(5, 8)$. Removing the fifth edge we get the MSF with edges $(4, 6)$ and $(5, 8)$. Removing the sixth edge we get the single edge $(5, 8)$ as an MSF.

As with Kruskal's algorith, checking the results of the dynamic algorithm by hand, we see that they are correct, and again, since no assertions failed, we take it as a good indication that the implementation is correct.

Subproblem 2.4

Making a fair and useful comparison between the offline and dynamic versions of the MSF algorithm, is very difficult without a plausible usage scenario. At a glance, it appears that the dynamic version has much more overhead, and it may very well be, that the offline version is faster for many usage scenarios on graphs with few vertices, and regardless of the number of edges.

If we were to use the dynamic version in an offline manner, that is, first fill it up with m edges, and then make a single MSF-query, then it has worst-case performance $O(mn \log^2 n)$, which is much greater than Kruskal's $O(m \log n)$.

Where we would expect the dynamic version to excel, is when many MSF-queries are interleaved with updates to the graph, and when the number of vertices (and hence the number of edges) is very large.

Note that in scenarios where we only wish to add edges and never remove them again, we could simply use Eq.(1) once on the existing MSF and the new edge, and while having almost the same time-complexity, namely $O(n \log n)$ versus $O(n \log^2 n)$ for maintaining and querying the fully dynamic data-structure for this update, its overhead would be greatly reduced. At any rate, this usage-scenario is expected to be an improvement over the offline algorithm, in regards to time-usage.

Subproblem 2.5

Apart from testing the implementation in regards to correctness as was done in subproblem 2.3, it would be interesting to test it for performance also. However, lacking a real-world usage scenario (the only scenarios that could be thought up, were purely incremental, i.e. without any edges being removed), we can resort to testing whether the time-complexities for the two algorithms, are as expected. Furthermore, we can test for which graphs the Kruskal's algorithm is faster than the dynamic algorithm, when the latter is used as an offline algorithm also.

As a side note, we could also *profile* the source-code by running a special piece of software alongside an execution, thus obtaining a measure of which parts of the code execute for the most time. This is one of the only sound ways of doing code optimization by hand, as we often mistake smaller bottlenecks for greater ones, and vice versa, when simply looking at the source-code.

Subproblem 2.6

We test both algorithms in an offline manner, for graphs with various numbers of vertices n and edges m, to see if the time complexities are right. Again, for more complete coverage, the dynamic algorithm should also be tested with a sequence of interleaved inserts, removes, and MSF-queries.

For testing the performance of Kruskal's algorithm, we generate 30 graphs with the number of vertices n going in uniform steps from 10 up to roughly 40000. For the dynamic MSF algorithm, we do the same, only with n going from 10 up to roughly 1000. The edges are created randomly, by running through each valid combination of vertices i and j (that is, so that $i < j$), and then according to a probability, an edge with a random weight is inserted. The weight is `double`-typed and picked uniformly, that is $\sim U(0, 1)$. We let the probability for any edge (i, j) to be added, be such that the graph has approximately $1.5 \cdot n$ edges. By the way, these are the experiments, that the `MS Windows` executable is set up to run.

Subproblems 2.7 & 2.8

What we wish to show with these experiments, is that our time complexities for the two algorithms are correct. For Kruskal's algorithm the dominating factor is m, and for the dynamic algorithm the dominating factor is $m \cdot n$. So we have depicted the time usage as a function of these entities, as can be seen in figures 2 and 3.

Since we only let the x-axis be the dominating factor of the asymptotic time complexities, we would expect a slight bend to the curve, which also seems to be the case for the dynamic algorithm, as can be seen in figure 3. For Kruskal's algorithm, the time usage is so low, that we are having trouble measuring it correctly. Nevertheless, the linear dominance of its time-complexity, still seems clear. We therefore conclude, that the time complexities of the two algorithms, appear to be as expected.

References

[1] Michael T. Goodrich and Roberto Tamassia: Data Structures and Algorithms in Java, John Wiley & Sons, 1998, ISBN: 0-471-19308-9

[2] Niels Lauritzen: Concrete Abstract Algebra (v4.0), Department of Mathematical Sciences, University of Aarhus, 2003.

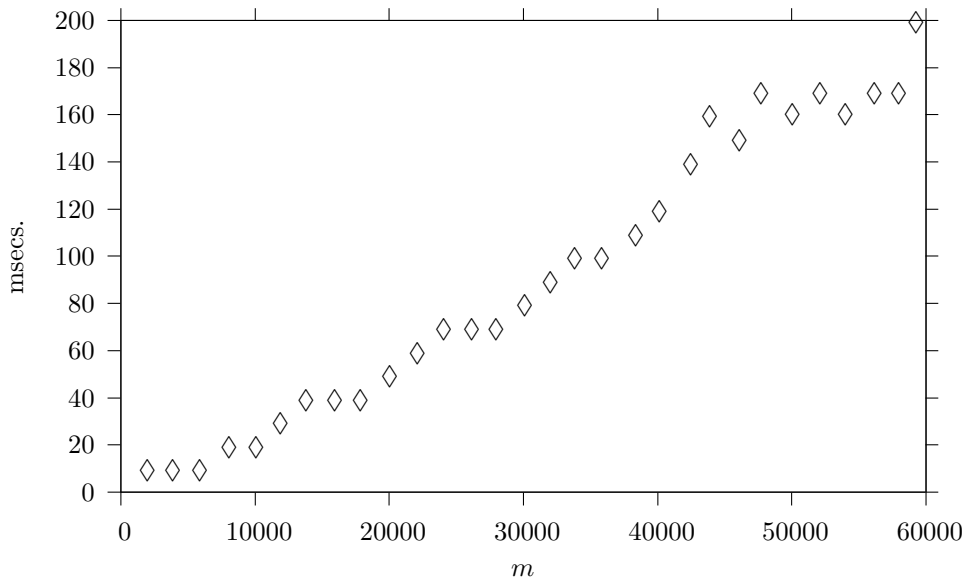

Figure 2: Time usage for Kruskal's algorithm, when used purely in an offline manner.

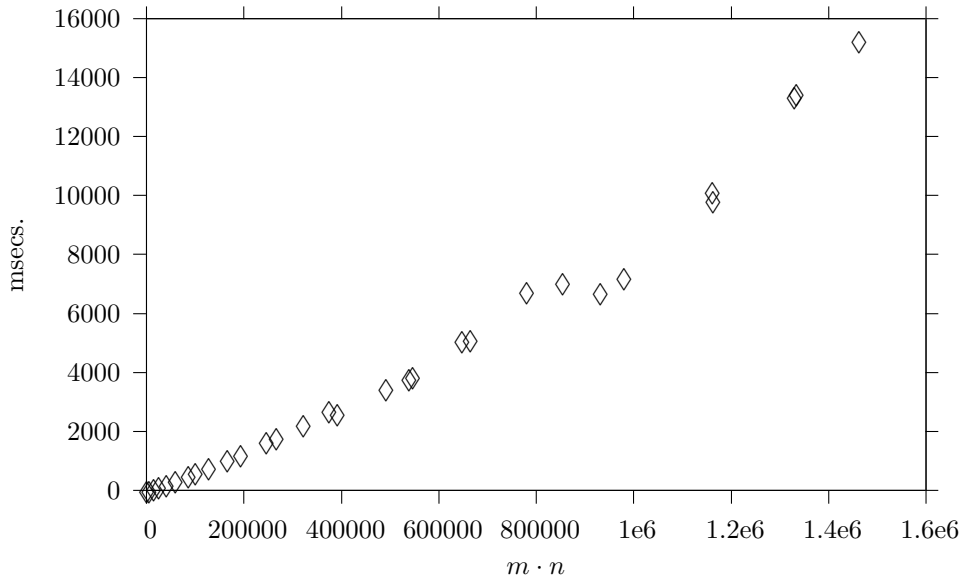

Figure 3: Time usage for the dynamic MSF algorithm, when used in an offline manner.