**Retroactive Data Structures and Operations**

A Project Report

***Submitted by***

CH.EN.U4AIE20017 GUDIVADA DHYANASAI

CH.EN.U4AIE20021 HARISH B

CH.EN.U4AIE20024 IPPATAPU VENKATA SRISURYA

CH.EN.U4AIE20025 JAWAHAR C

CH.EN.U4AIE20039 MUKESH K

***in partial fulfillment for the award of the degree***

***of***

**BACHELOR OF TECHNOLOGY**

IN

ARTIFICIAL INTELLIGENCE



AMRITA SCHOOL OF ENGINEERING, CHENNAI

AMRITA VISHWA VIDYAPEETHAM

**CHENNAI 601 103**

January 2022

**AMRITA VISHWA VIDYAPEETHAM**

**AMRITA SCHOOL OF ENGINEERING, CHENNAI, 601 103**



**BONAFIDE CERTIFICATE**

This is to certify that the major project report entitled **“Retroactive Structures and Operations”** submitted by

CH.EN.U4AIE20017 GUDIVADA DHYANASAI

CH.EN.U4AIE20021 HARISH B

CH.EN.U4AIE20024 IPPATAPU VENKATA SRISURYA

CH.EN.U4AIE20025 JAWAHAR C

CH.EN.U4AIE20039 MUKESH K

in partial fulfillment of the requirements for the award of the **Degree of Bachelor of Technology** in **ARTIFICIAL INTELLIGENCE** is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering, Chennai.

This project report was submitted by us on 09th January 2022

Signature

Dr. Maheshwari A

Department of Computer Science

**ACKNOWLEDGEMENT**

We would like to offer our sincere pranams at the lotus feet of Universal guru, MATA AMRITANANDAMAYI DEVI who blessed us with her grace to make this a successful major project. We are grateful to our guide Dr. Maheshwari A,Department of Computer Science, ASE, Chennaifor her invaluable support and guidance during the major project work. We express our deep sense of gratitude to Dr. Prasanna Kumar R,Chairperson, for his constant help, suggestions, and inspiring guidance.

We would also like to extend our gratitude to our director Shri. Manikandan, Principal Dr. Shankar who has always encouraged us. We are also thankful to all our classmates who have always been a source of strength, for always being there and extending their valuable bits of help to the successful completion of this work.

**ABSTRACT**

Retroactive data structures are a type of data structure that allows actions to be done on it in the past. The data structure permits arbitrary operation insertion and deletion at arbitrary times, subject only to consistency constraints. The study of retroactive data structures begins with a formal definition of the model and its variants. It is shown that the efficient retroactivity, unlike persistence, is not always possible. In this project we have assessed their performance and demonstrated their utility through real-world use-cases and applications, in addition to going through the data structures employed in their implementation.

Dynamic shortest path algorithms are used to accommodate a whole range of sequence of update operations to the underlying graph topology, as well as to make subsequent query operations more efficient. For the many variations of this problem, there are numerous solutions that all identify the set of vertices whose shortest paths may be affected by the modifications and then update their shortest paths according to the update sequence. In this project we have dynamized the Dijkstra algorithm, which will be helpful for the efficient solution of the dynamic single source shortest path problem. The retroactive priority queue data structure is used to achieve dynamism in Dijkstra algorithm. Retroactive data structures identify the set of modified vertices one by one, reducing the number of calculations required to accept the changes. So, using an appropriate dynamic graph representation and a retroactive priority queue, we tried to dynamize Dijkstra algorithm that solves the dynamic single source shortest path problem with an efficient update time.

**Keywords:** Retroactive data structures, Persistence, Dynamic Dijkstra Algorithm, Retroactive priority queue, shortest path problem.

**Table of Contents**

1. Introduction…………………………………………………………………………………6

1.1 General………………………………………………………………………………….6

1.2 Persistent Retroactive data structures….………………………………………….…….6

1.3 Use of retroactive data structures….……….…………………………………….……..7

1.4. Challenges in Retroactive Data Structure……………………………………….……..8

2. Types of Retroactivity and Transformations………………………………………….…….9

2.1. Partial Retroactivity………………………………………….…………………………9

2.2. Full Retroactivity………………………...………………….………………………...10

2.3. Commutative Operations……………………...…………….………………………...11

3. Retroactive Priority Queues……………………………………………………………….11

4. Dynamic Dijkstra Algorithm...…………………………………………………………….12

5. Conclusion…………………………………………………………………………………15

Appendix……………………………………………………………………………………..17

References……………………………………………………………………………………18

**1. Introduction**

**1.1 General**

Suppose we've just found that an operation we conducted before in a database was incorrect, and we need to alter it. The only way to accommodate these changes in most existing systems is to roll back the system's state to before the time in question, then re-execute all of the operations from the adjustments to the present. Such processing is inefficient, wasteful, and frequently unneeded. We present and extend the concept of retroactive data structures in this project, which are data structures that efficiently support changes to the structure's previous sequence of operations.

It is shown that there is no general efficient transition from non-retroactive structures to retroactive structures after defining the concept. The development of specific retroactive structures followed. The general transformations that make data structures efficiently retroactive for specific classes of data structures like commutative and invertible data structures, and data structures for decomposable search problems have been discussed in this report. In other data structures, where activities are more interdependent, efficient retroactivity necessitates the creation of new approaches for dynamization.

**1.2 Persistent Retroactive data structures**

The concept of retroactive data structures is similar to the standard concept of persistent data structures in that they both consider the concept of time, but they differ entirely in other ways, a persistent data structure keeps track of many versions of a data structure, and operations on one version can be used to create a new version. Modifications can only be performed to the last structure in its simplest form, resulting in a linear relationship between the versions. In full persistence, any previous version can be used to build a new version, resulting in a tree structure of versions.

Persistent data structures enable for the creation of a new version by performing merge-like operations on many existing structures, forming a directed acyclic graph. The data structuring strategies for persistence save a lot of money compared to the traditional practise of keeping separate copies of all versions. The main difference between persistent and retroactive data structures is that each version is viewed as an unchangeable archive in persistent data structures. Each new version of the structure is dependent on the condition of previous versions. The dependency relationship between two versions, on the other hand, never changes because existing versions are never updated.

The user can inspect a previous state of the structure, but modifications can only be made by forking off a new version from a previous state. As a result, while the persistence approach is beneficial for keeping archival copies of a structure, it is ineffective when changes to the structure's prior state must be made directly. The retroactive methodology permits changes to be made directly to prior versions. Because of the inter-dependence of versions, a change like this can have a significant impact on the contents of all subsequent versions. In effect, there will be a break in connection between time as viewed by a data structure and time as perceived by a data structure's user.

**1.3 Use of retroactive data structures**

Complex systems that process a large number of transactions are common in the real world. Many instances arise in such systems when it is necessary to change the historical sequence of operations previously conducted on the system. We now present some scenarios in which a backwards-looking approach to data structures would be beneficial:

a) Security Breaches: Let's say an unauthorised user was found to have conducted some activities maliciously. It is critical in the context of computer security to not only erase harmful transactions, but also to act as if the malicious operation never happened. For example, if the intruder changed the password file, we should not only restore it, but also undo any logins that were made possible by the change.

b) Version Control Systems: Document version trees can be maintained using software like Microsoft Word and GitHub. The software allows the user to examine previous versions and develop new versions based on them. When a mistake is identified, the papers involved can be rolled back to a previous version and started again from there. In some cases, though, it would be beneficial if we could make changes to a previous version and then propagate those changes to subsequent versions. Consider the case where there are multiple versions of documentation corresponding to multiple versions of a product, all of which are available online for users of the various versions. If an error is discovered in the documentation, we'd like to be able to fix it in the version where it was introduced, and have the change propagate to all of the documents that contain the error, even though some later versions may have changed the erroneous text for other reasons and thus do not need to be changed. Although brute-force methods may be used to propagate such changes, a retroactive approach would be able to make such changes quickly, resulting in a new generation of universal document life-cycle management tools.

c) Dynamization: Some static algorithms or data structures are built by applying a sequence of operations dictated by the input to a dynamic data structure. In specifically, the search tree's evolution is tracked by the intersection of the input with a vertical line that sweeps continuously from left to right; hence, queries at a given period in the persistent structure correspond to queries at a certain horizontal position. We might update the search tree at any time if we used full retroactivity instead of persistence, which corresponds to dynamically changing the input that specified the sequence of operations done on the data structure. Retroactive data structures, in general, can help in the dynamism of static algorithms that employ dynamic data structures.

**1.4. Challenges in Retroactive Data Structure**

One might believe that the problem of retroactive data structures can be solved by simply adding another dimension to the structure in consideration. For instance, in the case of a min-heap in Fig 1.1, it would appear at first glance that we might design a two-dimensional variation of a heap and solve the problem. The concept is to utilise the y axis to show key values of items in the heap and the x axis to indicate time. Each item in the heap is represented by a horizontal line segment in this representation. The time when an item is entered into the heap is represented by the left endpoint of this segment, whereas the time when the item is deleted is represented by the right endpoint.

If the heap's only operations are insert() and delete-min(), we get the additional property that there are no points below any segment's right endpoint since only the smallest things are eliminated from the heap. While this appears to be a simple two-dimensional representation of a heap over time, adding and removing operations in the heap retrospectively cannot be accomplished by simply adding or removing a line segment. In fact, as shown in Figure 1.1, all of the segments ends might be altered by inserting a single operation. While time can be represented as a spatial dimension, it is unique in that it may have complex dependencies, requiring adjustments to the remainder of the diagram when minor changes are made to one component.

As a result, most retroactive data-structure problems cannot be solved directly using classic geometric and high-dimensional data structures. To develop retroactive data structures without explicitly recording every state of the structure, new strategies must be used.

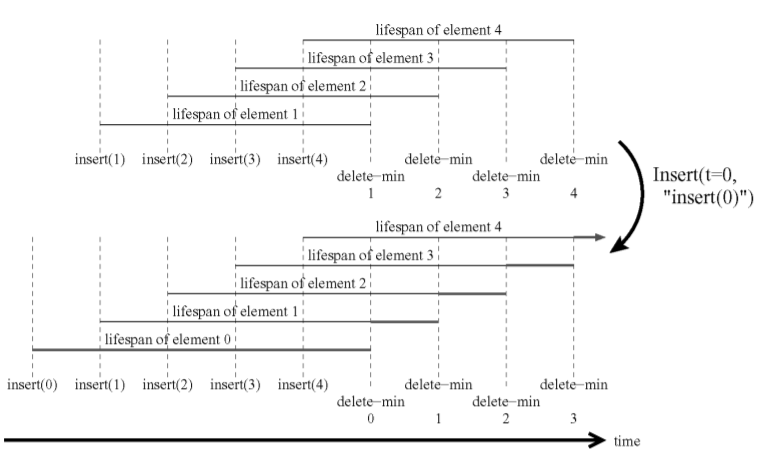


Fig 1.1 An example of retroactive heap data structure of inserting “0” at time 0 seconds.

**2. Types of Retroactivity and Transformations**

**2.1. Partial Retroactivity**

In the retroactive framework, any data structural challenge can be reformulated as partial or full. The data structure is made up of a series of updates and queries that are made over time. Let the list define the data structure changes, where is the operation done at time, and are the timestamps. At any given moment, we assume that only one operation is being done. The data structure is partially retroactive if, in addition to updates and queries on the data structure's "current state", it also allows for the insertion and deletion of updates from the past. In other terms, there are two incentive operations:

a) Insert (t, u): At time t, insert a new update operation u into U (assuming no operation existing at that time).

b) Delete (t): Remove the previous update operation from the update sequence U, assuming such an operation exists.

Insert (t, "insert(x)"), Insert (t, "delete(x)"), and Delete (t) are the retroactive versions of regular insert(x) and delete(x) procedures, where t indicates a point in time. If for example, Insert(t, "insert(x)") produces a new operation u = insert (x), which adds a specified element x, and updates history to make it appear that operation u occurred between and in the past. Informally, we're going back in time to a previous state of the data structure, introducing or preventing a change, and then returning to the present.

All such retroactive modifications to the data structure's operational history may have an impact on all existing operations between the time of modification and the present. The frequent instance in which local modifications propagate in effect to produce fundamentally different apparent states of the data structure is especially interesting. The problem is to use implicit representations to efficiently realise these observed variations.

**2.2. Full Retroactivity**

The descriptions previously given merely defines the possible operations of retroactivity like the capacity to insert or delete update actions in the past and see the results in real time. We can formally travel back in time to change the past, but we cannot see the past directly. A data structure is fully retroactive if it can answer inquiries about the past in addition to permitting updates in the past. This can be viewed as creating a partially retroactive version of a permanent version of the original structure in some ways. As a result, the conventional search(x) operation, which locates an element x in a data structure, becomes Query (t, "search(x)"), which locates an element x in the data structure's state at time t.

Only valid retroactive procedures are assumed to be done. For example, in a retroactive dictionary, a delete(k) action for a key k must always come after a matching insert(k) in the list U; and in a retroactive stack, the number of push() operations for each prefix of U is always greater than the number of pop() operations. Although the retroactive data structures described in this project report do not validate the validity of retroactive updates, it is typically simple to design a data structure that does.

**2.3. Commutative Operations**

The concept of commutative operations is used to emphasise the challenging case of non-local effects. If the state of the data structure resulting from a sequence of operations is independent of the order of those operations, then the collection of operation types is commutative. If a data structure has a commutative set of operations: executing an operation at any moment in the past has the same impact as performing it in the present. For a data structure that has a commutative set of operations and follows retroactivity, then it satisfies the below statements.

(a) At no additional statistical cost, any data structure that supports a commutative set of operations permits the retroactive insertion of operations from the past and queries from the present.

(b) At no additional statistical cost, every data structure that supports a commutative and invertible set of operations can be rendered partially retroactive.

(c) At no additional statistical cost, every data structure for a searching issue can be made partially retroactive.

A set of operations is said to be invertible if, for each operation u, another operation u’ negates the effects of operation u, i.e., the sequence of operations [u, u’] does not modify the state of the data structure. For searching issues, commutative data structures are an important class. The purpose is to keep a set S of objects under insertion and deletion operations so that we can quickly respond to queries Q(x, S) that ask about a new item x relationship to the set S. The set of operations for a searching issue is commutative because a set S is by definition unordered, and the subsequence of operations involving the same object always starts with an insertion and alternates between insertions and deletions. Dictionary structures, for example, can be expressed as searching problems, as can dynamic convex hull or planar-width data structures, and are therefore automatically partially retroactive.

**3. Retroactive Priority Queues**

The priority queue, which is more advanced than queues, stacks, and deques, supports operations such as insert(k), which adds an element with key value k, delete-min(), which deletes the element with smallest key, and the query find-min(), which reports the current minimum-key element. Because of its dependence on all previous activities, the delete-min() procedure is particularly important in this context, which element is eliminated is determined by the set of elements present at the time the action is performed. The non-commutative nature of the set of actions is due to delete-min() operation. Priority queues appear to be significantly more difficult to manage than queues and stacks. Even due to a single Insert (t, "insert (k)") operation, the lifespan of all items can change in the data structure. Such cascading effects must be stated simply in order to avoid the cost of explicit element lifetime maintenance.

We assume that all key values inserted in the structure are distinct without affecting generality. Let be the time it took to insert key k, and be the time it took to delete it. Let represent the set of elements in the priority queue at time t, and represent the set of elements in the queue right now. Let be the set of keys that were inserted after time t, and be the set of keys that were deleted after time t. We need to learn more about the problem's structure before we can build a retroactive priority queue. A sequence of updates is represented as a planar graphic with the x axis representing time and the y axis representing key values.

The number of elements || is an obvious invariant of a priority queue data structure. The number of elements in the queue is always equal to the number of inserts minus the number of deletes. Delete-min operations are the number of delete operations. As a result, when we add the operation u = "insert (k)" to the equation, in, one element will have to be inserted at time t in the past. If the element k is not erased between time t and the present, there are two possibilities: It's as simple as adding it to.

Otherwise, some operation u = "delete-min()" deletes the element k, while the element that was supposed to be eliminated by u stays in the structure a bit longer until deleted by another delete-min() operation, and so on. As a result, the insertion of operation u triggers a series of changes, as shown in Figure 4.2.

“After an operation insert (t, “insert (k)”), the element to be inserted in is ”

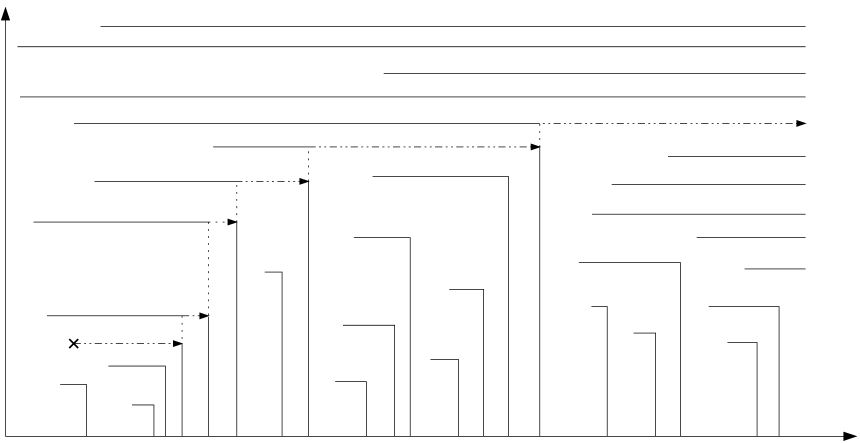


Fig 3.1 The Insert (t, “insert (k)”) operation causes a cascade of changes of deletion times, and one insertion in.

It's worth noting that undoing a delete-min() action has the same impact as instantly reinserting the element that was being deleted. As a result, we can deduce the following conclusion.

“The element to be added in after an operation Delete (t), where the operation at time t is "delete-min ()," is”

It would be difficult to maintain explicitly because it can change for many different values of t each time an operation is done. We can avoid this task by using the next statement. If, we say there is a bridge at time t. In Figure 4.2, bridges are depicted as dotted vertical lines.

“Let's say t’ is the last bridge before t. Then ”

After an operation Delete(t) where the operation at time t is = “insert (k)”, the element to be removed from is k if k ∈; otherwise, it is where t’ is the first bridge after time t.

“After an operation Insert(t, “delete-min()”), the element to be removed form Qnow is ”

“After an operation Delete(t) where the operation is insert(k), the element to be removed from Qnow is ”

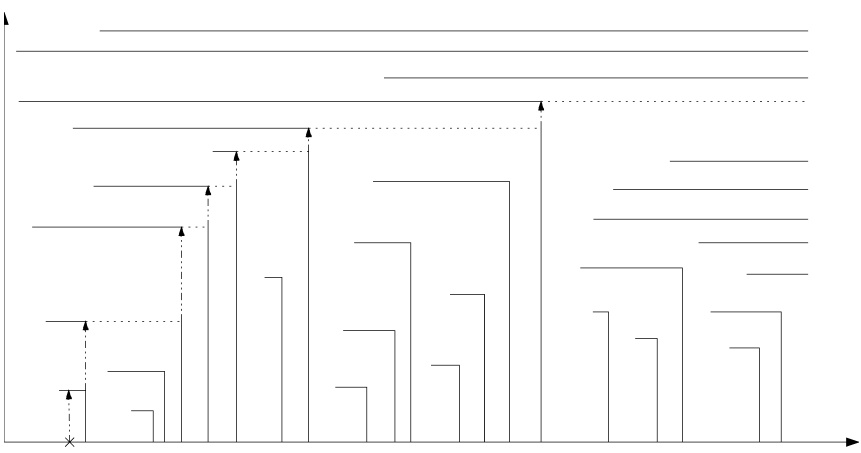


Fig 3.2 The Insert (t, “delete-min ()”) operation causes a cascade of changes of deletion times, and one deletion in.

There exists a partially retroactive priority queue data structure supporting retroactive updates in O (log m) time and supporting present-time queries in O (1) time. The data structure maintains the history of all update operations in a doubly linked list, and explicitly maintains the set in a binary search tree, associating with each key a pointer to its insert operation in the linked list. After each retroactive update, an element will be inserted or deleted in according to the rules described in the preceding lemmas. In order to decide which element to insert or delete, we need to be able to perform two types of operations:

(a) Find the last bridge before t or the first bridge after t; and

(b) Find the maximum key in or the minimum key in.

If we maintain the list of updates, assigning a weight of 0 to insert (k) operations with k ∈ , +1 to insert(k) with k , and −1 to delete-min() operations, every bridge corresponds to a prefix with sum 0. So, using the data structure, we can answer queries of type A in O (log m) time. Because every retroactive update adds or deletes at most one element from, only one weight change has to be performed in the structure, which also takes O (log m) time. If we maintain the list of insertions augmented by the modified (a, b)-tree, and store in each internal node the maximum of all keys in its subtree which are absent in, we can easily find the maximum key in in O (log m) time by walking down the tree. The minimum key in can also be maintained if we store in every internal node of the tree the minimum of all keys in its subtree which are in. Those values can be maintained in O (log m) time per retroactive update because each update changes at most one element of .

**4. Dynamic Dijkstra Algorithm**

Finally, we make use of the retroactive priority queues to implement dynamized Dijkstra algorithm in this section. Let us assume that, for some edge the weight has been changed, i.e., either increased or decreased. Then we will go back to time t in priority queue at which instant the head node of the edge whose weight changed will be visited in Dijkstra algorithm. The series of operation done from that time of visiting till present will be deleted, and the Dijkstra algorithm will continue calculating shortest path from time t. Thus, we don’t need to run the whole algorithm again from the start. This will save a lot of time if the graph in which Dijkstra algorithm is performed has a large number of nodes and edges.

Now let us take an example where the cost of travelling between 5 states will be calculated with dynamic Dijkstra algorithm. Let us first apply Dijkstra algorithm on it and find the path with low cost for each state from “Tamilnadu”. Now let us update the cost of edge connecting the states “Karnataka” and “Mumbai” from 2000 to 1000. Finally, we apply the dynamic Dijkstra algorithm on it. Since the head vertex of edge in “Karnataka”, we send it as an argument to the function. We have used both the dynamic Dijkstra algorithm and normal Dijkstra algorithm and calculated the time take to find the path with lowest costs. The dynamic Dijkstra took 0.19 milli seconds to run, and the normal Dijkstra took about 0.37 milli seconds to run. From this we can say that the dynamic Dijkstra has used the time efficiently and took less time when compared to normal Dijkstra algorithm.

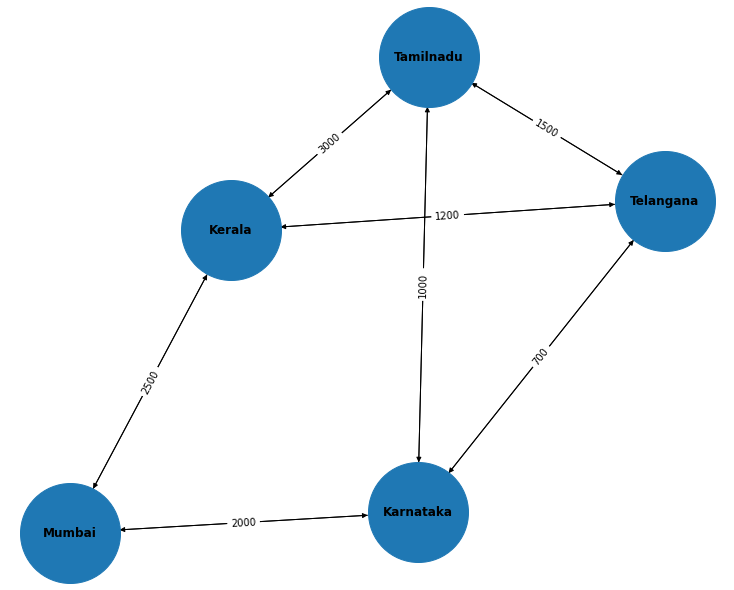
****

Fig 4.1. The graph consists on states as nodes and some costs as been assigned as weights to the edges.

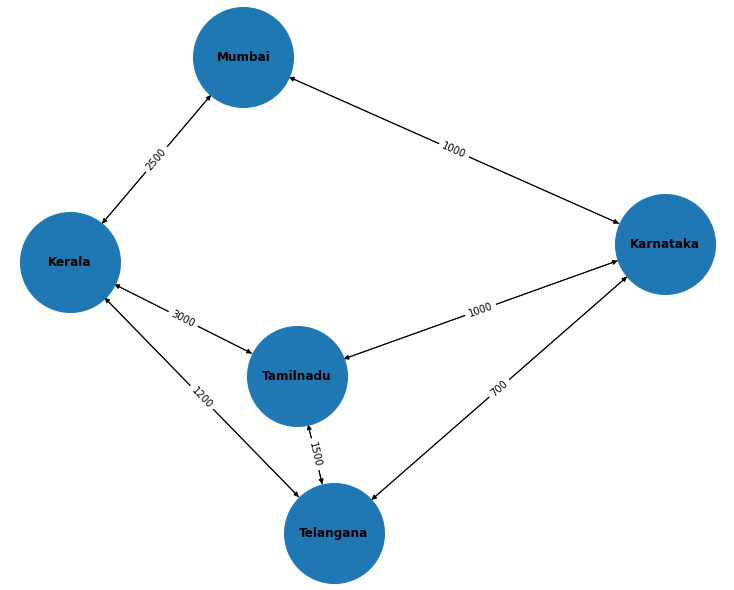
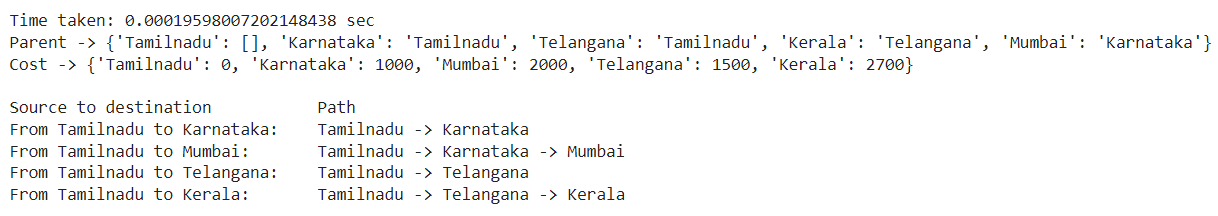
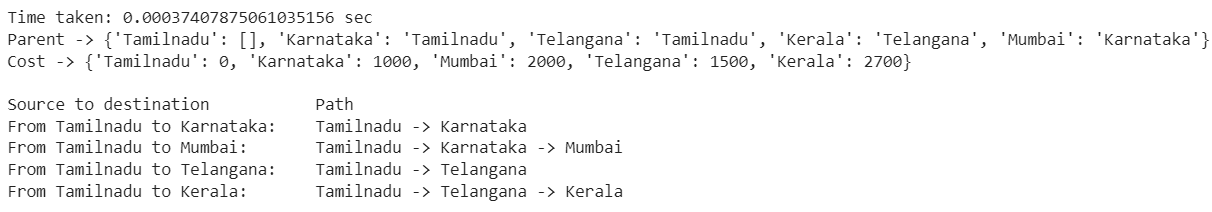


Fig 4.1. Updating the weight between nodes “Karnataka” and “Mumbai” with cost 1000



(a) Dynamic Dijkstra algorithm



(b) Normal Dijkstra algorithm

Fig 4.3. The shortest path calculated with (a) Dynamic Dijkstra algorithm. (b) Normal Dijkstra algorithm.

**5. Conclusion**

The dynamic shortest path problem is solved efficiently and dynamically using the retroactive priority queues in this project. Because we're utilising binary search trees to create the retroactive priority queue, the cost of all priority queue operations is determined by the height of the underlying tree data structure. Also, in the event of dynamic changes in the graph, the number of computations is reduced because we are travelling back in time to a point just before the time when the edge under update will be employed in any of the shortest pathways. We are selectively selecting the point that has been directly impacted by the change and applying the update to only that area. The use of retroactive data structures to solve the shortest path issue can easily be modified to relax the assumption we made in our approach. The dynamic all-pair shortest-path problem can also be solved using the dynamic Dijkstra algorithm. The algorithm's performance may fall short of the best resource bounds currently available. Additionally, increasing the efficiency of the underlying retroactive priority queue may increase the approach's performance. By treating a number of update activities as a single operation in the future, will further help to lower the problem's time constraints.

**References**

1. Erik D. Demaine, John Iacono, and Stefan Langerman. 2007. Retroactive data structures. ACM Trans. Algorithms 3, 2 (May 2007), 13–es.

2. Acar, U.A., Blelloch, G.E., Tangwongsan, K., 2007. Non-oblivious retroactive data structures. Technical Report CMU-CS-07-169.

3. Adelson-Velskii, G., Landis, E.M., 1962. An algorithm for the organization of information. Soviet Math. Doklady. 3, 1259-1263.

4. Ahuja, R.K., Orlin, J.B., Pallottino. S., Scutella, M.G., 2003. Dynamic shortest paths minimizing travel times and costs. Networks. 197-205.

5. Bergamini, E., Meyerhenke, H., 2015. Fully-dynamic approximation of betweenness centrality. In the proceedings of European Symposium on Algorithms.155-166.

6. Bergamini, E., Meyerhenke, H., Staudt, C., 2015. Approximating betweenness centrality in large evolving networks. In Proceedings of the Meeting on Algorithm Engineering & Expermiments. 133-146.

7. Bernstein, A., Roditty, L., 2011. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In proceedings of the ACM-SIAM Symposium on Discrete Algorithms. 1355-1365.