Acceleration of Thorup Shortest Paths Algorithm by Modifying Component Tree Demonstrated with Real Datasets

January 2014

Wei Yusi

Interdisciplinary Graduate School of Science and Engineering

Shimane University

Table of contents

Figure list

Table list

1. Introduction

The single source shortest path is the problem (SSSP) of finding the shortest path from a source vertex to every other vertex. It has been applied in many fields such as navigation [En12], keyword searching [Bhal02], computer network [Siva99], and it is widely used for finding the optimal route in a road network. SSSP is described as the following throughout this research, given a graph $G = (V, E)$ and a source vertex $s \in V$, supposes *s* can reach each vertex of the graph, then find the shortest path from s to every vertex $v \in V$, in which V and E represent the vertices and edges of *G* [Sedg03]. The *m* and *n* mentioned in the rest of this paper represent $|E|$ and $|V|$, respectively. Use $\mathrm{D}(v)$ to represent the tentative distance from the source vertex to *v*, and use $\mathrm{d}(v)$ to represent the ensured shortest distance from the source vertex to *v*, let L(*v*, *w*) represent the positive integer weights of edge(*v*, *w*). At the beginning, $\mathrm{D}(v) = \infty$ for every vertex except the source vertex, $\mathrm{d}(s) = 0$. The length of a shortest path should be the sum of the weights of each edge of the shortest path.

In the example shown in Figure 1.1, a graph is constructed by five vertices, the weights of each edge is also marked nearby. The target is to calculate the shortest path from *v1* to other vertices, after the problem is solved, we will get some basic information like Table 1.1, including the predecessor vertex and the shortest distance of each vertex. The predecessor is used to record the shortest path from the source vertex to current vertex, for example, the predecessor of *v5* is *v3*, and the predecessor of *v3* is *v1*, which is the source vertex. These records show the shortest path from *v1* to *v5* is *v1->v3->v5*, and the distance is 2, which is recorded in the column named distance. It is calculated by summing up the weights of edge (*v1*, *v3*) and (*v3*, *v5*).

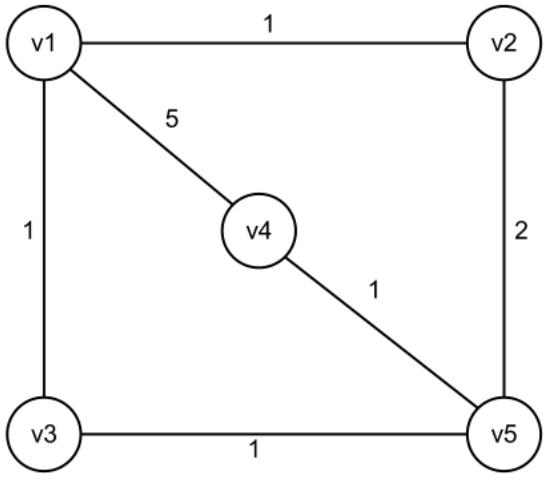


Figure 1.1. An undirected graph.

Table 1.1. Information we got when a problem is solved.

Vertex	Predecessor	Distance
v1	-	0
v2	*v1*	1
v3	*v1*	1
v4	*v5*	3
v5	*v3*	2

The Dijkstra algorithm and its variants will be introduced in Section 1.1. Section 1.2 and Section 1.3 introduce the priority queue and buckets based implementations of the Dijkstra algorithm, respectively. A linear time algorithm called Thorup will be introduced in Section 1.4 and later will be introduced in detail in Chapter 2. An improved Thorup algorithm is proposed in Chapter 3. Chapter 4 includes the introduction of the experiment and the result. Analysis is discussed in Chapter 5.

1.1. Dijkstra and the variants

One of the most influential algorithms is Dijkstra [Dijk59][Sedg03][Erik08] which is proposed in 1959. The basic idea of Dijkstra is described as follows,

At first, the distance of the source vertex will be set to 0, the distance of all the other vertices will be set to infinite. And then get the vertices adjacent to the source vertex one by one, update their tentative distance by an operation called relaxation, that is, if the distance of a vertex *u* will be reduced through another vertex *v*, then the distance of *u* will be set to *d*(*v*)+*w*(*v*, *u*), in which *d*(*v*) represents the tentative distance of *v* and *w*(*v*, *u*) represents the weights of the edge which connects *v* and *u*. After the distance of every vertex adjacent to the source vertex is relaxed (no matter updated or not), the source vertex will be marked as visited. Then do the operations as follows recursively till all vertices are marked as visited. Pick the vertex from the graph which is not marked as visited and has the shortest tentative distance, then update the distance of the vertices adjacent to this vertex again and mark it as visited.

The algorithm runs based on a matrix $M(n, n, V), n = |V|$ initially. [Yen70] is considered to be the first paper which implements Dijkstra with an array. In detail, an array is used to record the tentative distance of the vertices which adjacent to visited vertices. Every time when a vertex is visited, the distance of the vertices which adjacent to the vertex will be recorded in an array, and then, the vertex which has the shortest distance will be taken to try to relax the vertices which adjacent to this vertex. It takes $O(1)$ to insert a relaxed vertex and $O(n)$ to delete a vertex from an array. To relax all vertices, Dijkstra algorithm runs in $O(m)$ plus the time of maintaining the array, overall, it takes $O(m + n^2)$.

1.2. Dijkstra with Priority queues

There are several implementations of the Dijkstra algorithm, some of them use priority queues to reduce the cost of query vertex which has the shortest distance, such as using binary trees, Fibonacci heaps and so on. Table 1.2 lists the complexity of applying different structures to Dijkstra algorithm. The algorithm for implementing Dijkstra with a priority queue is as follows [Sedg03][Corm09],

Table 1.2. A comparison of time complexity of applying different structures to Dijkstra.

Structure	Time complexity
Array [Dijk59]	$O(n^2 + m)$
Fibonacci heaps [Fred87]	$O(m + n \log n)$
Relaxed heaps [Dris88]	$O(m + n \log n)$
Fusion trees [Fred93]	$O(m\sqrt{\log n})$
Atomic heaps [Fred94]	$O(m + n \log n / \log \log n)$
Monotone min PQs [Rama96]	$O(m + n\sqrt{\log n \ \log \log n})$

```
Dijkstra's algorithm
for each vertex v ∈ V
    if (v = source) v.key = 0
    else v.key = MAX
    INSERT (priority_queue, v)
while (priority_queue != empty)
    u = EXTRACT_MIN(priority_queue)
    for each v adjacent u
        if (v.key > u.key + v.weight)
            v.key = u.key + v.weight
            DECREASE_KEY(priority_queue, v)
```

Figure 1.2. Dijkstra's algorithm [Corm09].

According to the algorithm shown in Figure 1.2 [Corm09], initially, the tentative distance of source vertex will be set to 0, other vertices will be set to ∞, and then insert them to a priority queue. While the priority queue is not empty, extract the vertex which has the smallest tentative distance. Then try to use the extracted vertex to relax the vertices adjacent, the adjacent vertices can be found from an adjacent list which is pre-prepared. Once any vertex is relaxed, update the tentative distance of this vertex and update its position in the priority queue. If the distance from the source vertex through *u* to *v* is smaller than the tentative distance of *v*, then make the tentative distance of *v* equal to the sum of the distance of *u* and the weights of *v*, in which the shortest distance from the source to *u* must be ensured.

The priority queue used in the algorithm is not restricted. It decides the time complexity of the algorithm. An array based priority queue provides $O(n^2 + m)$ complexity, since

1. It takes $O(n)$ to extract the vertex with the shortest distance, for *n* vertices, it takes $O(n^2)$.

2. The algorithm tries to relax an endpoint of each edge with the other endpoint. For *m* edges, it takes $O(m)$.

Since the complexity of the algorithms mentioned below is analyzed by amortized analysis [Corm09]. We start from introducing amortized analysis in the beginning of this section. Amortized analysis shows the average cost of every operation of a sequence of operations in the worst case, which is different from an average-case analysis. For example, assume there is a MULTIPOP operation which is used to pop *n* objects from the top of the stack by calling POP operation *n* times. It takes $O(n)$ to operate a MULTIPOP in the worst case. Therefore, to operate MULTIPOP *n* times, it costs $O(n^2)$. The analysis is not very accurate, though it is correct. An amortized analysis can provide a tighter upper bound, that is $O(n)$. Because the number of objects popped by MULTIPOP cannot exceed the number of objects pushed into the stack. So the MULTIPOP operation pops object *n* times at most, it is equal to the number of PUSH operation. Since each POP operation cost is $O(1)$, *n* times cost is $O(n)$.

In 1987, Fredman and Tarjan [Fred87] proposed a priority queue named Fibonacci heaps. The amortized time of operations supported by Fibonacci heap is listed in Table 1.3 . It improved the time complexity of the SSSP problem to $O(m + n \log n)$, and also improved the time complexity of all pairs shortest path problem to $O(nm + n^2 \log n)$.

Table 1.3. Time complexity of operations supported by Fibonacci heap.

Operation	Time complexity (amortized)
Make-heap	$O(1)$
Insert	$O(1)$
Minimum	$O(1)$
Extract-Min	$O(\lg n)$
Union	$O(1)$
Decrease-Key	$O(1)$
Delete	$O(\lg n)$

The Fibonacci heap is created by several min-heaps, there may have more than one root in its root list, the key value of every root is smaller than the objects below. The heap needs to be consolidated after each call of Extract-Min operation. When consolidation is finished, the direct children (or say degree) of each root in the root list should be unique. Each node of Fibonacci heap has a key value, a mark sign, a degree counter and four pointers. A mark sign is used to tag the node which has lost a sub-node, a degree counter is used to record the number of children contained by a node. The four pointers *p*, *left*, *right*, *child* point to parent, left-sibling, right-sibling and any one of a node's children, respectively. If a node has only one child, then the child's *left* and *right* pointer point to the child itself. There are two other attributes in a Fibonacci heap,

1. *min* points to the minimum key value of the heap.

2. *n* is used to record the number of nodes in the heap.

The following descriptions of Fibonacci heap's operations are based on the algorithm proposed in [Corm09]. Following is the algorithm for inserting an object *x* to a Fibonacci heap *H*.

```
    INSERT (H, x)
1   x.degree = 0
2   x.p = NULL
3   x.child = NULL
4   x.mark = FALSE
5   if H.min == NULL
6           create a root list for H to contain x
7           H.min = x
8   else
9           insert x into H's root list
10          if x.key < H.min.key
11                  H.min = x
12  H.n = H.n + 1
```

Figure 1.3. Fibonacci heap's algorithm: Insertion.

When inserting a node, a root list will be created for the first inserted node, and then set *H.min* pointing to the only one in the root list. Otherwise, insert the node to the root list directly. If the key-value of the new node is smaller than *H.min*, then set the new node as the minimum node. Finally, increase the number of nodes in the root list by one. New nodes will be inserted to the root list directly without any locate operation, thus the running time is only $O(1)$. And the operation of consolidating nodes will be postponed to after the first time when extracts the minimum node.

Because *H.min* always points to the object which has the minimum key value, it cost only $O(1)$ to get the minimum key value. When remove the minimum key value, all of its children should be added to root list, then link the roots which have the same degree to make their degrees distinguishable. The followings are the algorithms for extracting the minimum key value [Corm09].

```
    EXTRACT_MIN(H)
1   z = H.min
2   if H.min != NULL
3       for each child x of z
4           add x to the root list of H
5           x.p = NULL
6       remove z from the root list of H
7       if z == z.right
8           H.min = NULL
9       else
10          H.min = z.right
11          CONSOLIDATE(H)
12      H.n = H.n – 1
13  return z
```

Figure 1.4. Fibonacci heap's algorithm: Extract-Min.

As the algorithm mentioned above, first pointing *z* to *H.min,* the object has the minimum key value. In the case that *z* is not null, all of its children will be added to the root list and then remove *z* from the root list. If *z* is also its right sibling, it means *z* is the only one in the heap, the heap will be turned to empty. Otherwise, set *H.min* points to another node in the root list, in the algorithm, *H.min* repoints to its right sibling, actually, any node in the root list will be all right. Because later when operate CONSOLIDATE to reform the heap, a new *H.min* will be calculated. At last, return *z*.

The processing of CONSOLIDATE includes two parts,

1. Link roots in root list, which have the same degree

2. Find the minimum node.

The algorithm of CONSOLIDATE is as follows,

```
    CONSOLIDATE(H)
1   let N equal to the maximum degree of any root in root list
2   let A[0.. N] be a new array
3   for i = 0 to N
4       A[i] = NULL
5   for each node w in the root list of H
6       x = w
7       d = x.degree
8       while A[d] != NULL
9           y = A[d]
10          if x.key > y.key
11              exchange x with y
12          LINK(H, y, x)
13          A[d] = NULL
14          d = d + 1
15      A[d] = x
16  H.min = NULL
17  for i = 0 to N
18      if A[i] != NULL
19          if H.min == NULL
20              Create a new root list for H containing just A[i]
21              H.min = A[i]
22          else
23              insert A[i] into H's root list
24              if A[i].key < H.min.key
25                  H.min = A[i]
```

Figure 1.5. Fibonacci heap's algorithm: Consolidate.

```
LINK(H, y, x)
1  remove y from the root list of H
2  make y a child of x, incrementing x.degree
3  y.mark = FALSE
```

Figure 1.6. Fibonacci heap's algorithm: Link.

In the first part, an array *A* is created initially for collecting the roots which have the same degree. The size of *A* is equal to the maximum degree of any root in the root list. Each root in the root list will be removed and then stored in *A*. The position of the index they should be inserted to *A* is equal to the number of their degree. When inserting a root, if its position is not vacant, means there is a root which has the same degree with this one. Let the one which has a larger key-value to be the other one's child by calling LINK. Empty the original position and store the new root to one position backward of *A.* In the next part of this algorithm, A will be traversed in order to store its objects to root list and find the minimum node.

DECRESE-KEY operation is used to update a node with a new key-value. The new key-value should be smaller than the node's current key-value. Its algorithm is as follows,

```
DECREASE_KEY(H, x, k)
1  if k < x.key
2        x.key = k
3        y = x.p
4        if y != NULL and x.key < y.key
5              CUT(H, x, y)
6              CASCADING_CUT(H, y)
7        if x.key < H.min.key
8              H.min = x
```

Figure 1.7. Fibonacci heap's algorithm: Decrease-Key.

```
CUT (H, x, y)
1   remove x from the child list of y, decrementing y.degree
2   add x to the root list of H
3   x.p = NULL
4   x.mark = FALSE
```

Figure 1.8. Fibonacci heap's algorithm: Cut.

```
CASCADING_CUT(H, y)
1   z = y.p
2   if z != NULL
3       if y.mark == FALSE
4           y.mark = TRUE
5       else
6           CUT(H, y, z)
7           CASCADING_CUT(H, z)
```

Figure 1.9. Fibonacci heap's algorithm: Cascading-Cut.

The DECRESE-KEY operation updates a key as follows, first of all, the new key-value has to be smaller than the *x*'s current key-value. After the update, if the *x*'s new key-value is smaller than its parent, *y*, then it should be removed from *y*'s children list. After that, the same as INSERT, *x* will not be located and inserted to any other node, but be turned into a root and then inserted in the root list directly. At the same time, the number of *y*'s degree is reduced by one. Since *y* just lost a child, *x*. If *y* is not in the root list and the removal of *x* causes y's children list empty, then *y* should be cut to root list too. Otherwise, mark *y* if it is not a root, since it has lost a child. Cut y's ancestors to the root list recursively if they match the requirements.

In 1988, a structure named "Relaxed heaps" is proposed by [Dris88]. It provides the same time complexity as [Fred87]. The paper also proposed a theoretical improvement over Fibonacci heaps and a parallel algorithm for calculating shortest paths on directed graphs with non-negative weights, the algorithm accommodates both of Fibonacci heaps and Relaxed heaps.

Fusion trees [Fred90] [Fred93] can answer predecessor/successor queries in $O(\log_w n)$ with *n* non-negative *w*-bits integers, each of which should not exceed 2^w in a *w*-bits machine. The structure of fusion tree is like a B-tree, but each node of which has $w^{1/5}$ sub-nodes. Figure 1.10 shows an example of Fusion trees, in which set *k* represents the keys in a node. $k =$

$\left\{k_0,\ k_1,\ \ldots,\ k_{w^{\frac{1}{5}}-1}\right\}$ and $k_0 < k_1 < \ldots < k_{w^{\frac{1}{5}}-1}$. In general, when performing a query, to decide the branch of a node to go, to visit all keys in a node is necessary. That is, k keys are needed to be read for making the decision of each node, and each of them has w-bits. Fusion trees compress the length of all the keys of a node within w bits though a technique named Sketch, thus makes it possible to let the CPU to process in parallel.

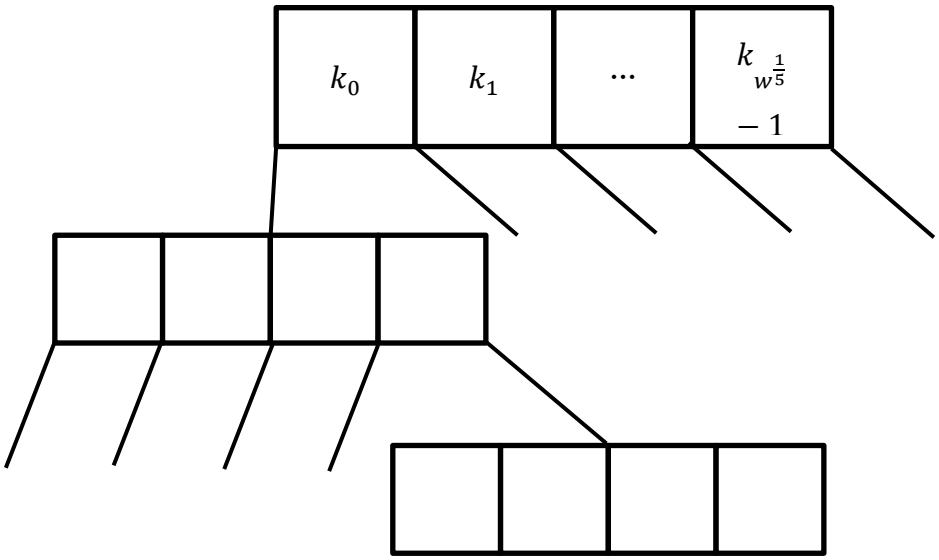


Figure 1.10. An example of Fusion trees, set *k* represents the keys of a node.

The basic idea of Sketch is to calculate a number of bits (say *s*) for each key in order to distinguish the keys in a node. Each *s* can be looked as a path from the root of a binary tree to each of its leaf node. The length of *s* is *k*-1 bits at most. An example is shown in Figure 1.11, black nodes are important nodes, which used to distinguish keys. First, the nodes which have branches should be important nodes, and second, the nodes in the same row with important nodes should be changed to important nodes, such as node2, node8 and node9. The left branch of an important node represent 0, and the right branch represent 1. In this example, three bits are used to distinguish four keys of a node. Thus the size of all the keys of a node is compressed to twelve in total. A Perfect Sketch is the whole path from the root of the binary tree to its leaf-node, it includes every bit in the node on the path. The "Perfect Sketch" can be used to calculate predecessor/successor of a search key if it doesn't fall in the Sketch set of keys of a node. After query in each node, the Perfect Sketch of both of the result leaf-node and search-key will be compared, in order to calculate the Perfect Sketch of predecessor/successor of the result, say *e*. It is calculated by,

1. Find the first different bit between the two Sketches.
2. Copy the Sketch of the result from the first bit to the first different bit to *e*, the first different bit is included.
3. Fill the rest space with inverse bits of the first different bit.

Then calculate the Sketch key of *e* to perform another query. The result should be predecessor/successor of the search-key. For example, if we cut off the leaf-node node13, it means that the key corresponds to node13 will not be pre-Sketched. Thus the Sketches of the example in Figure 1.12 have only two bits. Also, node2 and node3 are no longer important nodes, since there is only one way out of node3. Now we perform a query with a binary-key 1111, and its Sketch should be 11, which also equal to the Sketch of the leaf-node node12. We find that node12 is not the correct result by comparing its Perfect Sketch with the Perfect Sketch of search-key. The first different bit is node3, so *e* should be equal to 1011, and its Sketch is 11, query the binary tree again with *e*, then we find node12 is the predecessor of search-key and keep on perform query in the sub-node of node12 follow the branch just got.

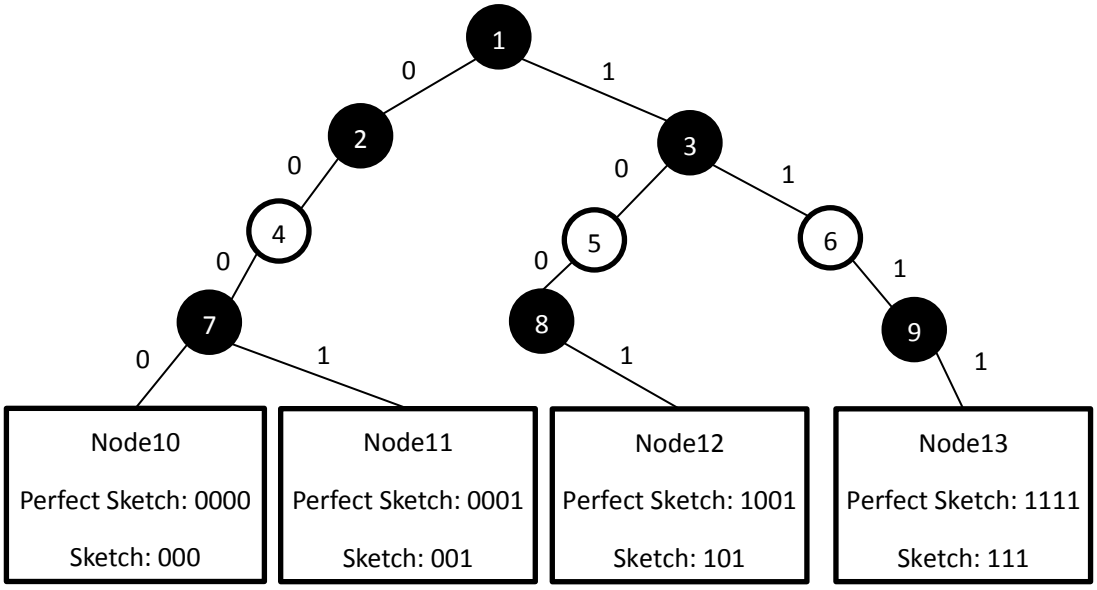


Figure 1.11. Important nodes (black nodes) of keys in a node.

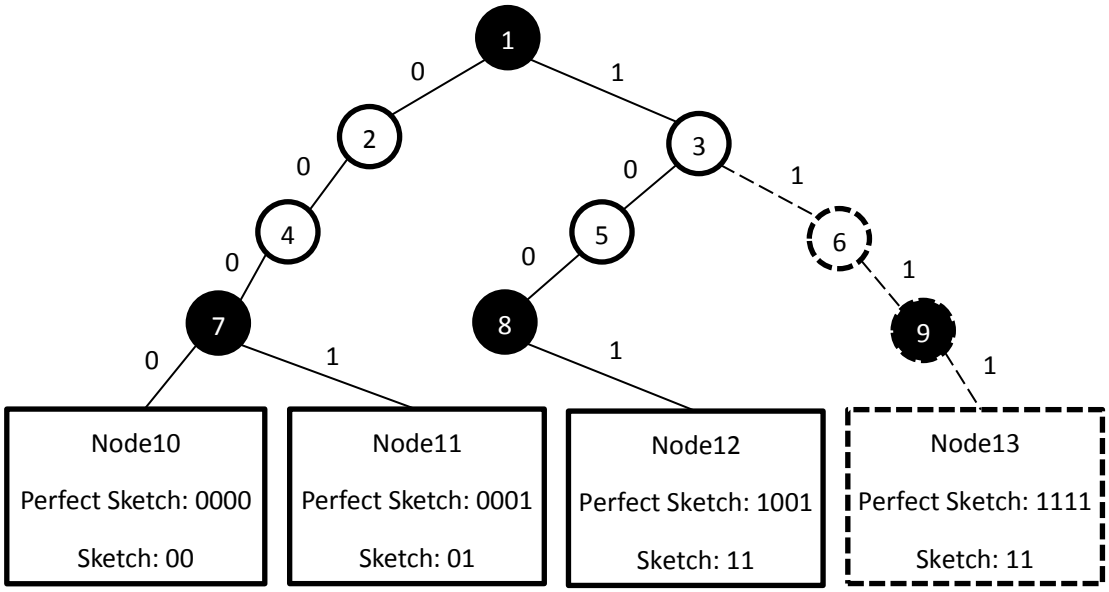


Figure 1.12. Find predecessor/successor of a search key.

1.3. Buckets based implementations

The complexity of sorting weights becomes a bottleneck of Dijkstra [Thor97]. The bucket is an alternative which has been widely used. A bucket is an array which can be used to map the tentative distance of vertices with index $\lfloor D(v)/\Delta \rfloor$. In which Δ represents the width of a bucket. For example, number 4, 6, 9 should be mapped to bucket 2, 3, 4, respectively when the bucket's width is 2. A heap could be added to each bucket when there are more than one element are mapped in the same bucket. [Hitc68][Dial69][Gils73] implemented buckets in networks which have positive integer edges [Dena79].

[Dini78] and [Dena79] proved that when set Δ equal to the smallest weights of a graph, with Dijkstra algorithm, we can visit vertices in the order of they lied in buckets [Meye01]. That is, the vertex in the lowest-numbered non-empty bucket should always be the vertex needed to be visited. Thus the time cost of visiting vertices is accelerated by overcoming the bottleneck of sorting. In the situation that the smallest weights is 1, the bucket width should be 2 but not 1. In addition, for those graphs which has zero-length weights, a heap should be added in each bucket. When adding a vertex to a bucket, it should be added to the heap of the bucket in ascending order, that is, the smallest on the top of the heap. Following is the basic idea of this implementation. The bucket's width is set to the smallest weights of a graph.

1. Set $D(V_{source}) = 0$ and $D(V_i) = \infty$ for other vertices. Insert V_{source} in the 0^{th} bucket, and keep other buckets empty.
2. Stop if all buckets are empty. Else set p^* point to the lowest-numbered nonempty bucket.
3. Get a vertex *v* from p^*, and then try to relax the vertices adjacent *v*. After that, delete *v*

from p^* and then map the relaxed vertex to buckets, respectively.

4. Go to step 2 if p^* is empty, otherwise, go to step 3.

[Dena79] also proposed a bucket based multi-level structure, in which buckets have different widths. For those sparse graphs which have few large weights, it might cost too much time to find the next non-empty bucket, thus the efficiency will be reduced.

1.4. Linear time algorithm in theory

Thorup [Thor97] (see also [Asan00] and [Prue09]) is an algorithm which has been theoretically proved that it can solve the SSSP problem in linear time with pre-processed index. The paper proposed a hierarchy and buckets based algorithm to preprocessing indices for performing queries in undirected graphs with non-negative weights. It consists of two phases, construction and visiting. In the first phase, it constructs a component tree in linear time, each node of the tree is a component that includes vertex(s). Leaf-nodes always contain only one vertex. A component is created by the connected vertices of which their weights are smaller than 2^i, where i represents an integer which increases from 0 till $2^i >$ the largest weights of *G*. In the second phase, the tentative distance of reached components will be mapped to its parent's buckets to decide the order of visiting vertices in different components in a hierarchy structure, in order to overcome the sorting problem. That is, when trying to get the vertex which has the smallest tentative distance, there is no method to sort the distances in linear time currently. This problem reduces the performance of Dijkstra every time when get the vertex from a priority queue which has the shortest distance from which to source vertex. Because of Thorup does not construct indices based on source vertex, the constructed indices can be used to calculate with different source vertex without reconstruction. In practice, due to the difficulty of implementation, Thorup algorithm occasionally does not perform as expected, according to the experimental result provided by Asano and Imai [Asan00], and Pruehs [Prue09]. This algorithm is introduced in Chapter 2 in detail.

2. The Thorup Algorithm

The Thorup algorithm [Thor97] has been theoretically proved that it can solve the SSSP problem in linear time with pre-processed index. The paper [Thor97] proposed a hierarchy- and buckets-based algorithm to preprocess indices for performing queries in undirected graphs with non-negative weights. Specifically, a bucket based component tree accumulates vertices to many components, and decides the order of visiting vertices, and an unvisited structure which looks like an interval tree is used to maintain the tentative distance of each vertex. Theoretically, this algorithm constructs the minimum spanning tree in O(m), constructs the component tree in O(n), construct unvisited data structure in O(n) and calculate distance of all vertices based on constructing structures in O(m + n). Due to the difficulty of the implementation, two alternatives are given by [Thor97], the methods will be introduced in the rest part of this chapter. The alternatives raise the time complexity, according to the experimental result provided by Asano, Imai [Asan00] and Pruehs [Prue09], the Thorup algorithm did not perform as expected.

2.1. Differences and similarities between Dijkstra and Thorup

A bucket based Dijkstra algorithm has been introduced in Section 1.3. The method decides the order of visiting vertices by mapping them in buckets according to their tentative distances. The interval of buckets is equal to the minimum weights of a graph. This method is very efficient, but not practicable. Since the tentative distance of a vertex in a big graph can be very long, and the minimum weights can be very small, an extremely huge number of buckets which cannot be created by today's computer may be needed. The problem is that, the interval of the buckets has to be invariable. For making the interval suit all tentative distances of a problem, it has to satisfy the minimum tentative distance, otherwise the vertex has the minimum tentative distance cannot be mapped. But for those extremely long tentative distances, the interval is too small. Many unused empty buckets may exist between mapped vertices.

Thorup algorithm solves this problem by using hierarchical buckets. The intervals of buckets in different levels are different. It accumulates connected vertices by the weights between them. The higher the level they are included, the larger the weights between them. The vertices in the the higher levels will be mapped in the buckets with bigger interval; on the contrary, the vertices in lower levels will be mapped in the buckets with smaller interval. Thus the bucket number is reduced.

Same as the Dijkstra algorithm, the Thorup algorithm also uses the relaxation operations, but the order of visiting vertices may be different.

2.2. Minimum spanning tree

The algorithms of minimum spanning tree are used to find the edges which connect all the vertices of a graph with the smallest weights, and also make sure vertices are connected with

each other through the shortest edge. In the Thorup algorithm, a component tree will be created based on a generated minimum spanning tree, so that each edge between two components also has the smallest weights. Theoretically, the Thorup algorithm uses the algorithm proposed in [Fred94] to construct the Minimum spanning tree in linear time, an alternative of this algorithm is [Krus56]. Here we first study the Kruskal's minimum spanning tree algorithm and then study Fredman and Willard's union-find algorithm [Fred94].

The Kruskal's algorithm first makes a set for each vertex in the graph, and sorts the edges by their weights in ascending order. Pick edges in ascending order, if the vertices of both sides of the picked edge are not in the same set, merge them into one set by the union-find algorithm which is introduced in Chapter 4. And then, this edge will the added to a list which all the edges in it forms a minimum spanning tree. The following algorithm is given by [Corm09]. Set *A* is used to store the edges which form a minimum spanning tree.

```
    Kruskal's minimum spanning tree
1   A = null
2   for each vertex v of the graph
3       MAKE-SET(v)
4   Sort the edges of the graph into ascending order by weight w
5   for each edge(u, v) of the graph, taken in ascending order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7           A = A ∪ {(u, v)}
8           UNION(u, v)
9   return A
```

Figure 2.1. The minimum spanning tree algorithm of Kruskal.

The algorithm used to generate a minimum spanning tree in the Thorup algorithm is proposed by [Fred94], in Section 2.1. The algorithm generates a minimum spanning tree by two passes with the usage of atomic heaps and Fibonacci heaps. The basic idea is that to unite a number of connected verties newly to create a super-vertex. The edges which have the smallest weights will be selected to connect vertices of the super-vertex. And start the second pass to unite the all of the super-vertices based on the same rule. After the second pass, a minimum spanning tree is generated. To begin with, an arbitrary vertex will be added to a priority queue based on atomic heaps, the weights of the vertex will be set to 0, set the weights of other vertices as infinite. And then we are going to the first pass, remove the first vertex from the heap and then mark it, update the weights of all the vertices connect it, and then add them into the heap. Then go back to the beginning of the first step. If the number of vertices over the size of the heap or a vertex removed from the heap connects to any marked vertex, the current process will be stopped. Another unmarked vertex will be selected arbitrarily, and then go back to the start of this algorithm. After every vertex is marked, the graph is condensed with several

super-vertices. To unite them again with the same method in the second pass. The difference is that in the second pass, a Fibonacci heaps based priority queue is used.

2.3. Tarjan's union-find algorithm

Union-find set is the problem of union given elements to sets. Elements in the same set have the same relation. Here the union-find set algorithm is used to construct the component hierarchy (Section 2.5).

Tarjan's union-find algorithm [Corm09] uses tree to arrange the elements. It makes a representative element of a set have multiple successors. Each tree in this algorithm represents a set, each node of a tree represents an element of a set, and the root of a tree includes the representation of a set which includes elements with the same relation. Each node points to its parent, and root's parent is the root itself. Sets are united by uniting trees. An example of unite two sets with Tarjan's algorithm is shown in Figure 2.2, (a) includes two sets, node A and C include two representative elements of the two sets, respectively. And (b) shows the structure after uniting the two sets in (a). This algorithm does not originally perform better than a list based algorithm without two heuristics,

1. Union by rank.

   A rank is used to compare the number of direct sub-nodes between two trees, when union two trees, the root of the tree which has a larger rank should be the other's parent. The rank may not record the exact direct sub-nodes' number of a tree. When unite two sets, only if the two trees of them have the same rank, then plus one to the rank of the new tree's root after uniting finished. This heuristic will reduce the time cost when unite two sets which has a different number of members, if a tree has a larger number of member is connected to a tree has smaller number of members, then more pointers of members have to be set to root.

2. Path compression.

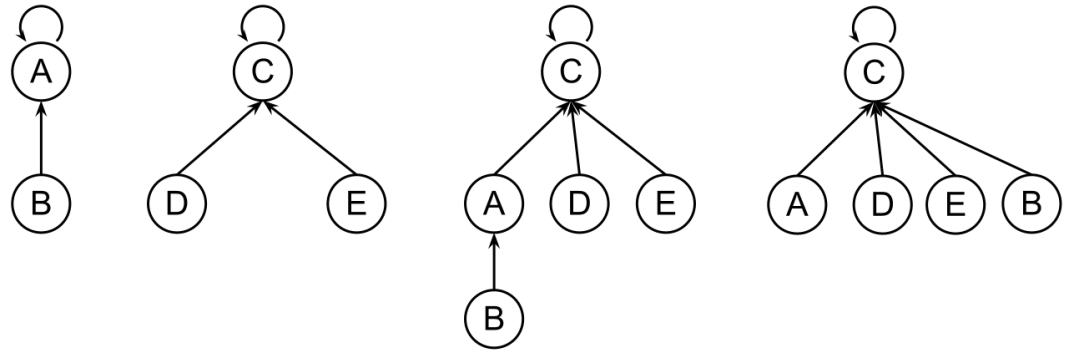
   Path compression is realized by setting the pointer of each node on the path of query to the root directly. In detail, when running the FIND-SET function, the ancestors of the nodes will be found recursively till get to the root. And then the root will be returned to all the nodes on the path of query, and then set each pointer of these nodes points to the root.

The algorithms of make set, union, and find set are as follows, based on [Corm09]. Initially, as the first member, *x* is appointed as the root of a set. A root does not have a parent, so the pointer of *x*'s parent point *x* itself, and *x* does not have any sub-nodes currently, so the rank of *x* is zero.

In FIND-SET function, the set of a member *x* belongs to is found by keeping on finding the parent of *x*, till when find an ancestor of *x*, say *r*, that *r*'s parent equal to *r* itself. After that, for accelerating query, all the pointers of nodes between *x* and *r* will be set to point *r*, it is called path compression.

When uniting two members, first the root of each of the two members will be found through FIND-SET operation, and then the root which has a larger rank will be the parent of the other. If their ranks are equal, pick one of them to be the parent of the other optional, the

algorithm listed following chooses the secondary parameter as the parent always.



(a) (b) (c)

Figure 2.2. An example of unite two sets with Tarjan's algorithm. (a) includes two sets and (b) shows the structure of unite the two sets. (c) shows the result obtained by applying path compression on (b).

```
MAKE-SET(x)
x.p = x
x.rank = 0
```

Figure 2.3. Algorithm of make set.

```
UNION(x, y)
x = FIND-SET(x)
y = FIND-SET(y)
if x.rank > y.rank
    y.p = x
else x.p = y
    if x.rank == y.rank
        y.rank = y.rank +1
```

Figure 2.4. Algorithm of unite two sets.

```
FIND-SET(x)
if x!=x.p
    x.p = FIND-SET(x.p)
return x.p
```

Figure 2.5. Algorithm of unite two sets.

2.4. Gabow's split-findmin algorithm

The split-findmin structure [Gabo85a] maintains the smallest key of each of a number of disjoint sequences. Just like a priority queue. Split-findmin structure has operations as follows,

1. *split*(x)

   Split the sequence which contains *x* into two sequences, one sequence contains the first element of the original sequence to *x*, the other sequence contains the rest elements.

2. *decrease-key*(*x*)

   Decrease the key of element *x*, and update the key of the sequence which contains *x*. The key of a sequence should be the smallest among all the elements it contains.

3. *findmin*(*x*)

   Find and return the element which has the smallest key in the sequence which contains *x*.

The split-findmin structure is used in [Thor97] [Prue09] to instead of the atomic heaps, for maintaining the tentative distance of each component in a component tree (the component tree will be introduced in Section 2.5). Every four sub-nodes have a father node which stores the minimum value of its sub-nodes.

2.5. Component hierarchy and Component tree

Component hierarchy decides the order of visiting vertices. It is the kernel of the Thorup algorithm. With such a structure, the algorithm does not need to do any sort before visiting vertices. But just visit vertices in buckets from the first to the last. Now, let's study its mechanism. At first, it groups vertices as different components in the hierarchy. The vertices included by a component are different at each level. It depends on the weights of edges between each pair of vertices. In each level, the vertices will be grouped to the same component if the weights of the edges between them are smaller than $2^i, i = 0, 1, 2, \dots, x$. *Here x* equals to the root's level of a component hierarchy. Components at each level contain the vertices which the edges between them are smaller than 2^{level}. Accordingly, 2^x should be just larger than the largest weights of the graph. Also, the root of a component hierarchy should contain the vertices of the whole graph.

In the example shown in Figure 2.6, a graph is constructed by 5 vertices and 6 edges. The largest weights of this graph is 7, so a component hierarchy constructed for this graph should have 4 levels, level 0 ~ level 3. The plot of a component hierarchy constructed based on the graph is shown in Figure 2.7. Since 2^3 equals to 8, it is just larger than 7. So the root on level 3 can contain all vertices which the edges between them are smaller than 8. Means it contains all vertices in the graph. Red frame in Figure 2.8 indicates the vertices contained by the root. And C13 represents the ID of the component.

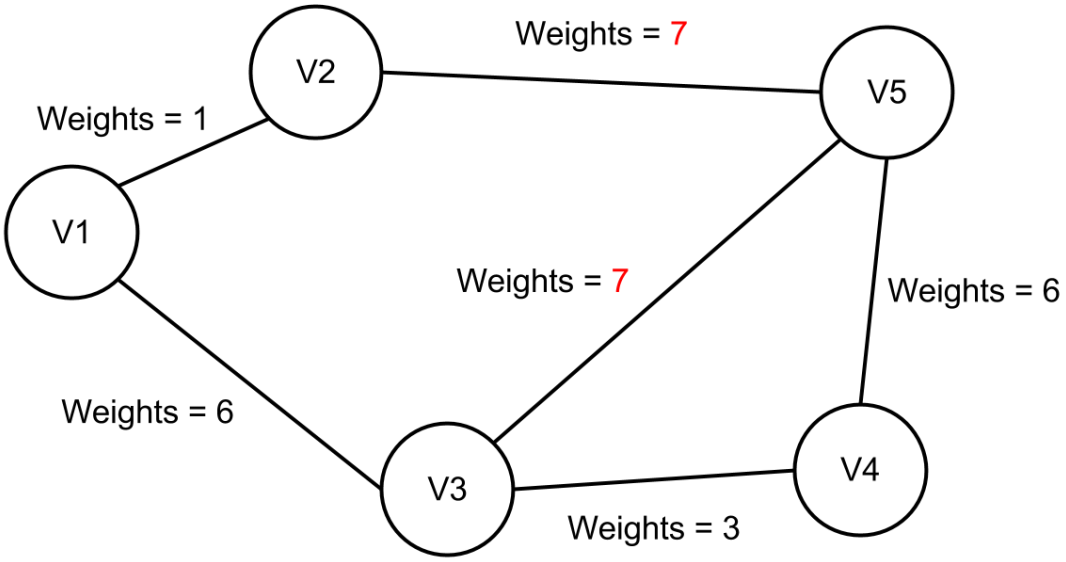


Figure 2.6. *x* should be equal to 3, since 2^3=8, and 8 is just larger than the largest weights in the graph, that is, 7.

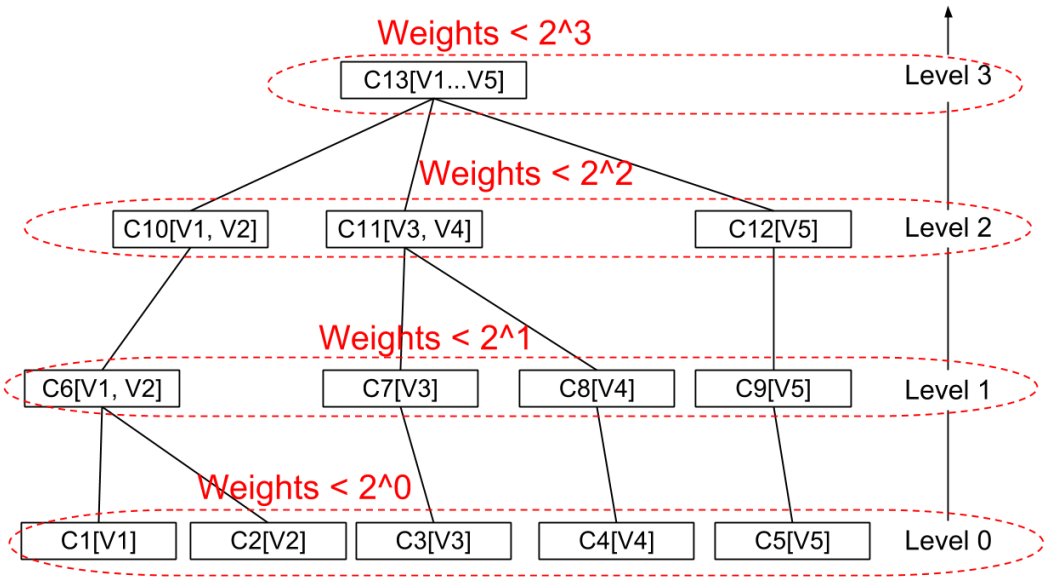


Figure 2.7. A component hierarchy constructed for the graph in Figure 2.6.

Under the root, there are 3 components at level 2. The weights of the edges between vertices in each component at this level are smaller than 2^2 equals 4. Since the weights of the edges e(v1, v3), e(v2, v5), e(v3, v5) and e(v4, v5) are not smaller than 4, they cannot be assembled to the same component at this level. So at this level, the graph is separated into 3 parts: C10, C11 and C12. According to the same reason, there are 4 components at level 1, all edges in each component are smaller than 2^1=2.

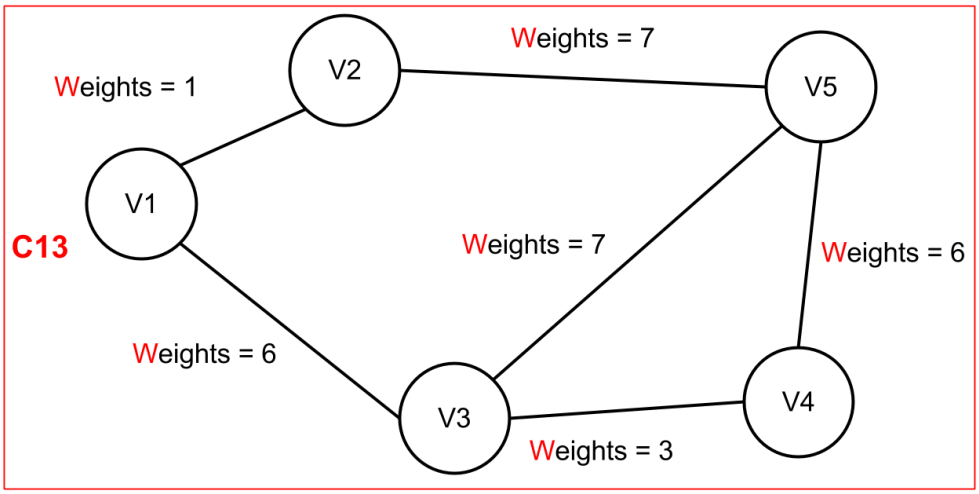


Figure 2.8. The root at level 3 contains all vertices of the graph.

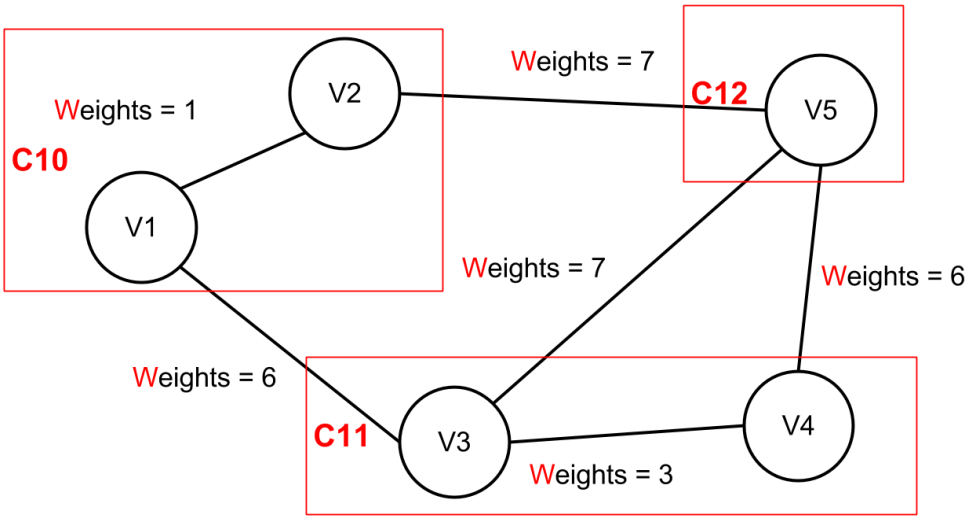


Figure 2.9. Level 2 has 3 components, vertices will be assembled if the edges between them is smaller than 4.

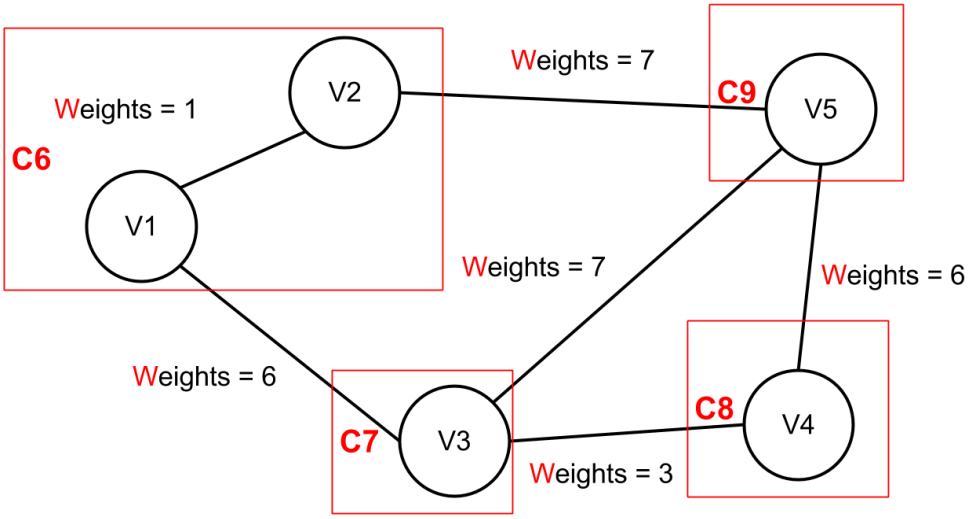


Figure 2.10. Level 1 has 4 components, vertices will be assembled if the edges between them is smaller than 2.

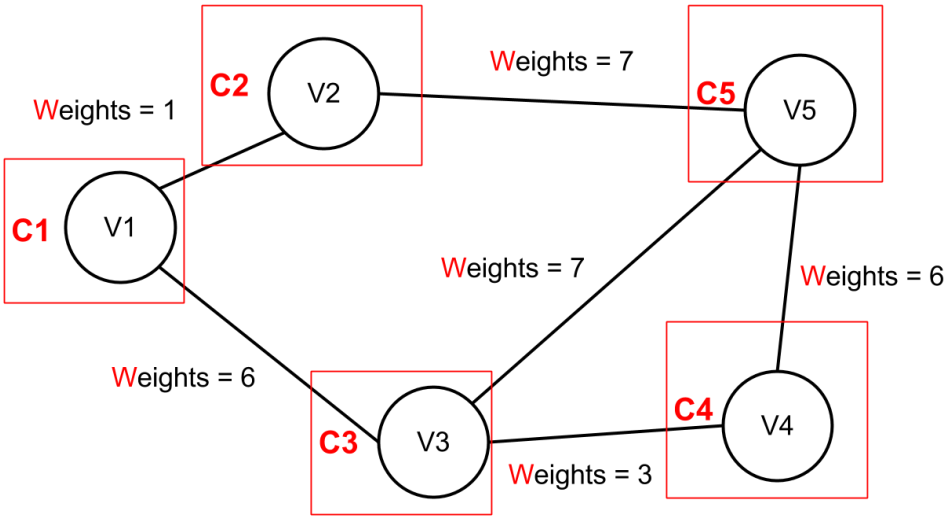


Figure 2.11. Level 0 has 5 singleton components; each of them includes a vertex.

At level 0, the requirement of weights for constructing components is smaller than 1. Since none of the edges matches the rule, components are constructed by each single vertex. A component hierarchy is completely constructed with 13 components at 4 different levels.

For deciding the order of visiting vertices, except the root, each component should be mapped to its father component's buckets, depending on its minimum tentative distance. A minimum tentative distance is the smallest tentative distance among all the vertices in the component. The unvisited structure which is used to retrieve the smallest tentative distance of a component is introduced in Section 2.5. To map a sub-component, say *c*, to its father, we need to get the index of the bucket which stores component *c with c*'s tentative distance (D(*c*)). The formula is as follows,

$$D(c) >> (c.\text{level} - 1)$$

An instance is shown in Figure 2.12, since component C10 includes the source vertex, so the tentative of C10 would be 0. And the shortest tentative distance of the vertex contained by C11 is 6, so the tentative distance of C11 is 6. Assume we only know the tentative distances of both of the two components, but do not know C12. Then we calculate the Bucket IDs for them by the formula mentioned above. The Bucket ID of C10 and C11 are 0>>1=0 and 6>>1=3, respectively. C10 and C11 will be mapped to the 0^{th} and the 11^{th} bucket of their father accordingly. Since we do not know the tentative distance of C12, yet, so it will not be mapped. During visiting process, C10 will be firstly visited. And then visit C11, if the visiting of C10 does not decrease the tentative distances of any brothers of C11 to smaller than C11. Because that will cause another component to be added in its father's buckets before C11. After all sub-components of C10 and C11 are visited, they will be removed from C13's bucket, .

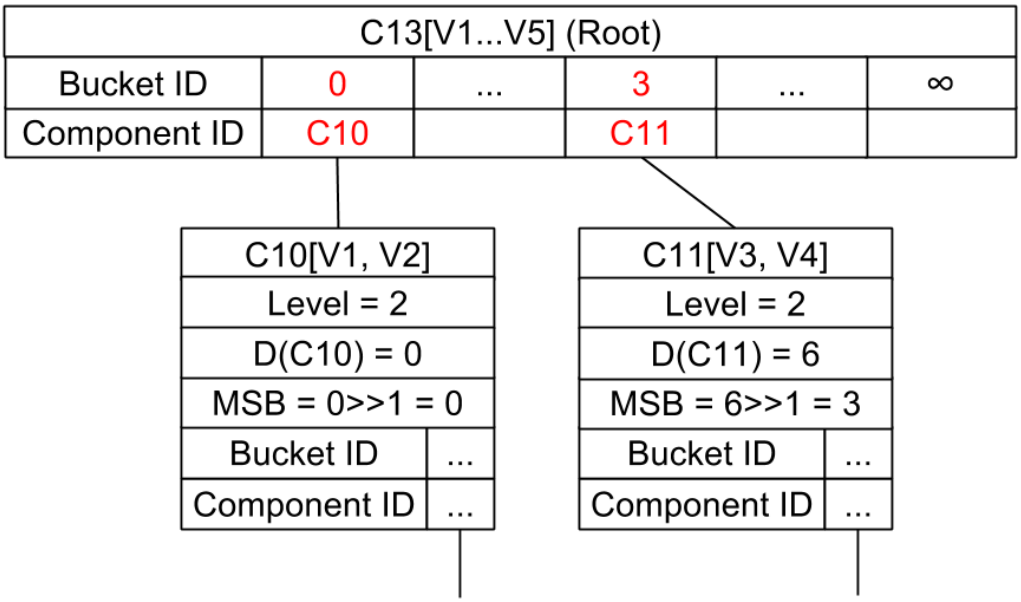


Figure 2.12. An example of bucketing components.

Looking back to the component hierarchy in Figure 2.7, you may find that component C10 and C6 are containing the same vertices. Similarly, C7 and C3; C8 and C4; C12, C9 and C5 also contain the same vertices. In visiting process, visit the same crowd of vertices repeatedly will become a burden. For reducing the burden, a component hierarchy should be compressed to a component tree. If there are some components that contain exactly the same vertices as their sub-components, then they should be removed from the component hierarchy, and their sub-components would be their fathers' direct sub-components. As the instance shown in Figure 2.13, component C10, C7, C8, C12 and C9 will be removed from the hierarchy.

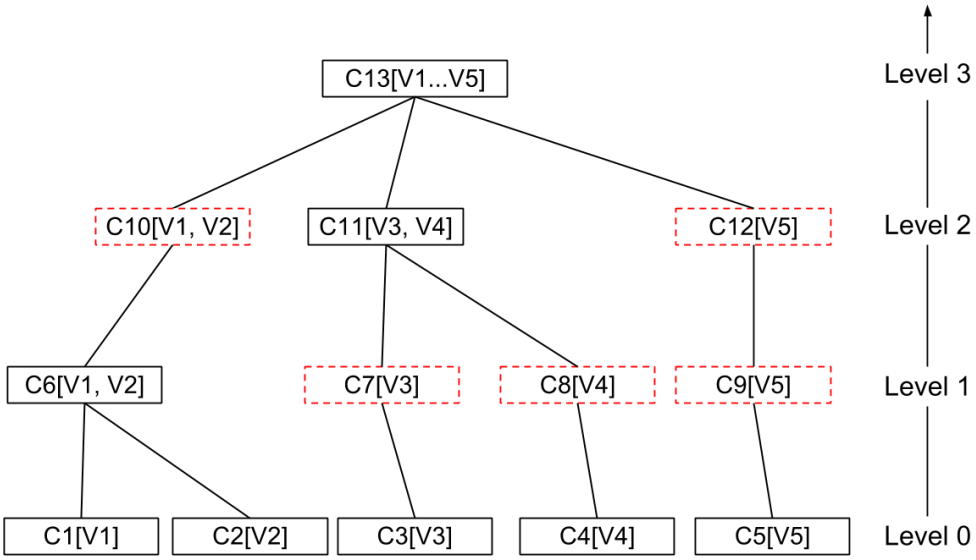


Figure 2.13. To compress the component hierarchy, C10, C3, C8, C12 and C9 will be removed.

2.6. Unvisited structure

An unvisited data structure is used to maintain the tentative distance of each vertex and component of a component tree. Theoretically, the Thorup algorithm maintains distances in linear time by using atomic heaps to construct the unvisited structure, but atomic heaps are defined for the situation that the number of vertices is more than $2^{12^{20}}$, it cannot be used in common problems. Thorup [Thor97] gives an alternative to avoid using atomic heaps by using split-findmin algorithm [Gabo85a]. This change increases the algorithm's cost to $O(\log C + \alpha(m,n)m)$, where C represents the maximum edge weights, and α is an inverse function of Ackermann which grows very slowly. If the number of vertices is less than 10^{80}, $\alpha(m,n)$ should be equal to, or less than 4 [Corm09].

The split-findmin separates sequences of elements to different nodes. The key of each node indicates the minimum key of the elements inside. Thorup algorithm uses split-findmin to maintain tentative distances of components by mapping all the vertices of a component tree on level 0 to split-findmin structure. As the example shown in Figure 2.14, there is a sequence S which includes all elements (①) need to be maintained. The keys of these elements are set to infinite. To construct the split-findmin structure, firstly, from left to right, every four elements will be collected as a new node, called *super element*. ③ of Figure 2.14 includes four *super elements* which are created by elements below. Every four *super elements* on each level compose a father *super element.* Till there is not enough *super elements* can be used. The rest element(s) of each level which are not enough to create a *super element* will be inserted to a *singleton element*(②), a *singleton element* cannot be used to create any *super element*. The keys of *singleton elements*

and *super elements* are equal to the minimum keys of elements maintained by them.

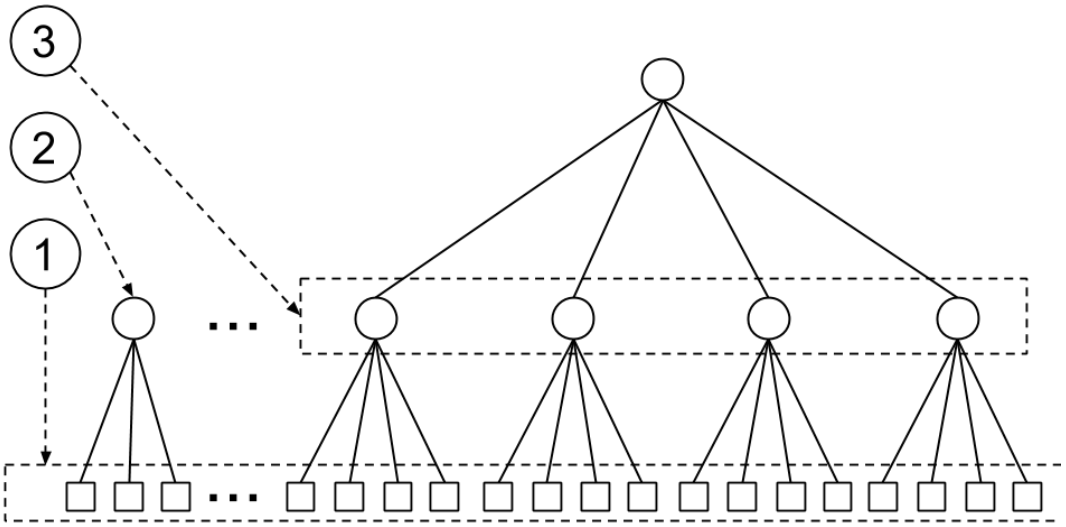


Figure 2.14. The data structure of split-findmin.①: A sequence of vertices, say S.②: Singleton element.③: Super elements.

The split-findmin supports three operations as follows,

1. Split(*x*), separate the sequence *S* which contains element *x* to two new sequences, *s1* and *s2*. *s*1 includes the elements of *S* from the first element to *x*. *s*2 includes the rest elements of *S*.
2. Findmin(*x*), return the minimum key of the sequence which contains *x*.
3. Decrease(*x*, *key*), if *key* is smaller than the key of *x,* set *key* as the new key of *x*.

2.7. Thorup algorithm in practice

In practice, due to the difficulty of implementation, [Asan00] and [Prue09] proposed their modification of the structures used in the Thorup algorithm. In the experiments executed by [Asan00] and [Prue09], priority queue based Dijkstra algorithm performance better than Thorup. Meanwhile, [Saku10] obtained an inverse result. In detail, Table 2.1 shows an experimental result from [Asan00], which compared their modified Thorup algorithm with the original Dijkstra algorithm and Dijkstra with Fibonacci heap. In which, rows MST and Data structures show the time cost of constructing the minimum spanning tree, data structures including component tree, buckets and interval tree, respectively. Row visit shows the time cost of calculating the shortest path, which is the second phase of the Thorup algorithm. From their result, we see that the time cost of constructing MST takes most of the time of total time, and reduced the efficiency of

modified Thorup. They also indicate that,

1. In grid graphs, the visiting part of modified Thorup will perform better than Binary heap based Dijkstra algorithm when $n > 2^{34}$.

2. In random graphs, set $m = 6n$, the execution time of visiting part of modified Thorup should less than the binary heap based Dijkstra algorithm when $n > 2^{25}$.

Table 2.1. A result of running time (sec) comparison from [Asan00], $n = 50{,}000$.

m	175065	299914	424396
Dijkstra (original)	327.53	326.63	326.97
Dijkstra (Fibonacci heap)	2.33	2.83	3.26
Thorup (total)	25.89	40.74	57.43
MST	23.33	35.24	56.60
Data structures	1.25	1.30	1.23
Visit	1.96	2.46	2.91

With another variant of Thorup algorithm, a suit of more comprehensive experiments are proposed by [Prue09], in which graphs with varied vertex number, edge number per vertex and edges' weights are generated randomly. Besides that, an experiment on accumulate running times is also included, which uses road network of New York City as the dataset. The experimental result is shown from Table 1.5 to Table 1.8. The paper indicates that the results were contrary to the theory, it may be caused by the inefficient implementation or the word length of today's computer is still not large enough to realize the Thorup algorithm.

[Saku10] obtained a different result in their experiment of a large-scale network simulation. Their result indicated that Thorup's algorithm is slightly faster that Dijkstra's with approximately 30% larger consumption of memory.

Table 2.2. Running time (ms) comparison with varied vertex number [Prue09], $m = 5n$.

n	Dijkstra		Thorup		
	Array heap	Fibonacci heap	MST	Data structures	Visit
2000	9	12	6	12	21
4000	37	25	28	25	49
6000	78	43	37	28	96
8000	132	62	46	48	131
10000	203	82	62	53	181
16000	-	140	112	100	302
24000	-	229	159	150	575
32000	-	319	191	193	866
40000	-	419	254	291	1228

Table 2.3. Running time (ms) comparison with varied edge numbers per vertex [Prue09], n = 20000, $3n \leqslant m \leqslant 24n$.

Edges per vertex	Dijkstra	Thorup		
	Fibonacci heap	MST	Data structures	Visit
6	190	165	119	453
12	218	247	126	515
18	273	337	97	534
24	269	451	131	581

Table 2.4. Running time (ms) comparison with varied edges' weights length [Prue09], n=20000, m=5n.

Maximum edges' weights	Dijkstra	Thorup		
	Fibonacci heap	MST	Data structures	Visit
256	212	148	153	461
1024	209	153	171	423
16384	212	145	156	430
262144	219	168	140	403

Table 2.5. Accumulated running times (sec) for ten queries on road network of New York City [Prue09].

Query times	2	4	6	8	10
Dijkstra (Fibonacci heap)	2768	5536	8367	11089	13779
Thorup	20160	39928	58118	76979	95756

3. The Improved Thorup Algorithm

Although theoretically Thorup algorithm is very efficient, according to the experimental result provided by Asano and Imai [Asan00], and Pruehs [Prue09], it is still slower than array and Fibonacci based Dijkstra algorithm in practice. One of the reasons is that the data structures given by Thorup are complicated and having complex operations. They are difficult to be realized in practice. Specially, the split-findmin needs to rearrange elements to create new *super elements* when initializing new component. This may highly increase the time cost of calculating shortest paths.

In the rest of this Chapter, a performance enhanced Thorup algorithm is proposed. There are two modifications,

1. Make the component tree able to maintain the tentative distances of vertices.
2. Reduce the depth of the component tree by extending the weights' limitation when creating components.

3.1. Mechanism

About the first modification, the basic idea is that, the component tree could be an efficient structure for maintaining tentative distances, constructing another structure to answer queries from the component tree may not necessary. Our algorithm maintains the tentative distance of each vertex with the component tree, so as to avoid creating and using any unvisited data structures. This change has two benefits as follows.

1. Save the time cost of constructing unvisited structures, which is in the first phase of Thorup algorithm.
2. Accelerate the process of calculating shortest paths, which is in the second phase of Thorup algorithm.

Two variables should be added to each node of a component tree,

1. *distance*, which is used to record the tentative distance of each component.
2. *deleted*, which is used to mark that whether the tentative distance of a component is not needed to be updated. It happens when we start to bucket the component's children.

The second modification intends to reduce the depth of a component tree. In Section2, we have studied that the original Thorup algorithm creates components at different levels depends on the weights of the edges. The edges will be included in the same component if their weights are greater than 2^i and smaller than $2^{i+1}, (0 \leq \mathrm{i})$. In the real world, the information of roads could be enormous and various. This method might create a component tree with high depth. This will reduce the advantage provided by using buckets. To visit components frequently through such a structure is inefficient. Accordingly, we try to reduce the depth of the component tree by

extending the limitation for edges' weights of components in different levels. That is, edges which their weights are greater than $base^i$ and smaller than $base^{i+1}$ will be accumulated in the same component, $base$ should be a number which is power of 2, such as 4, 8, 16 and so forth. It is set to 16 in our experiment which will be introduced later in Section 3.3. The *base* should not be restricted, but changes along the sizes of different graphs. The bucket size of each component *c* is then increased to,

$$\left\lceil \sum_{edge \in c} w(edge)/base^{\text{c.level}-1} \right\rceil$$

And we also need to right-shift $\log_2 base$ times to calculate the positions of components in their father components. The formula used to calculate the index of the bucket which stores component *c* with *c*'s tentative distance (D(*c*)) is accordingly changed as follows,

$$D(c) >> (c.\text{level} - 1) * \log_2 base$$

3.2. Algorithms

Figure 3.1 to Figure 3.4 show the improved algorithms starting from the one named Visit. In function Visit, from the line 11 to line 17, sub-components are repeatedly visited until a component does not have any child, or when *v.ix* >> (*j-i)**$\log_2 base$*)* is increased. If *v.ix* >> (*j-i)**$\log_2 base$*)* is increased, means the tentative distance of *v* is no longer the minimum distance of its parent. It is demonstrated by Thorup in Lemma 22. Since we should always visit the component has the minimum tentative distance, so the repeat will be stopped. Comparing to the original algorithm, since the base is changed to a number larger than 2, when manipulating buckets, the right-shift times is also increased in our algorithm. When decreasing keys of components, because our algorithm does not use unvisited structure, new keys should be updated in each undeleted father component of the current component.

```
1.  Visit(v)
2.  if v is a leaf node of component tree then
3.      VisitLeaf(v)
4.      Remove v from the bucket of v's parent
5.      return
6.  end if
7.  if v has not been visited previously then
8.      Expand(v)
9.      v.ix = v.ix0 //The lowest bucket index of this component
10. end if
11. repeat until v has no child or v.ix >> ((j-i)*log2 base) is increased
12.     while the bucket B[v.ix] is not empty
13.         let wh equal to the node in bucket B[v.ix]
14.         Visit(wh)
15.     end while
16.     v.ix = v.ix+1
17. end repeat
18. if v has any child then
19.     move v to bucket B[v.ix>>((j-i)*log2 base)] of v's parent
20. end if
21. if v does not has child and v is not the root of the component tree then
22.     remove v from the bucket of v's parent
23. end if
```

Figure 3.1. Algorithm of visit.

```
1.  Expand (v)
2.  v.ix0 = v.distance >>((i-1)*log_2 base)
3.  v.deleted = TRUE
4.  for each child wh of v
5.      store wh in bucket B[wh.distance >>((i-1)*log_2 base)]
6.  end for
```

Figure 3.2. Algorithm of Expand.

```
1.  VisitLeaf(v)
2.  for each vertex w connected with v, if v.distance + L(v, w) < w.distance
3.      Let wh be the unvisited root of leaf w
4.      Let wi be the unvisited parent of wh
5.      Decrease(w, v.distance +L(v,w))
6.      if this decreases wh.distance >> ((i-1)*log_2 base) then
7.          Move wh to bucket B[wh.distance >> ((i-1)*log_2 base)] of wi
8.      end if
9.  end for
```

Figure 3.3. Algorithm of VisitLeaf.

```
1.  Decrease (v, newValue)
2.  if v.distance > newValue and v.deleted !=TRUE then
3.      v.distance = newValue
4.      let n to be the parent of v
5.      while n.deleted !=TRUE and n.distance >newValue
6.          n.distance = newValue
7.          let n to be the parent of n
8.      end while
9.  end if
```

Figure 3.4. Algorithm of Decrease.

3.3. Examples

Here is an instance for explaining the mechanism of our algorithm when calculating shortest paths. The *base* of this instance is set as 2. Let *v*1 be the source vertex of this problem. Figure 3.5 shows the graph of component arrangement on level 0. Since *v*1 is contained by component C1, the tentative distance of C1 is set to 0. Components on level 0 are indexed by an ArrayList according to their IDs, so they can be visited directly. Currently, we do not know the tentative distances of other components, set their tentative distances to infinite.

Figure 3.6 shows the component hierarchy after decreasing the tentative distances of component C1 and its father components. To avoid using unvisited structure, a new variable used to record the tentative distance is added to each component. It makes the component hierarchy the same as component tree able to maintain tentative distances. When the tentative distance of a leaf-component is updated, the tentative distance of its undeleted father components should also be updated one by one. Therefore, after decrease the tentative distance of C1 from infinite to 0, the tentative distances of its fathers C6, C10 and C13 are also decreased. For getting the information of connected vertices, *adjacent list* will be visited. *v*2 and *v*3 are two vertices connected to *v*1, the components which contain them are C2 and C3, respectively. As the figures shown in Figure 3.7 and Figure 3.8, their tentative distances are decreased from infinite to 1 and 6, respectively. The tentative distances of the father components of both of C1 and C2 will not be updated, since they are smaller than the tentative distance of C2. Then tentative distances of C6 and its father components C7 and C11 will be decreased. C13 is not included, since its tentative distance is smaller.

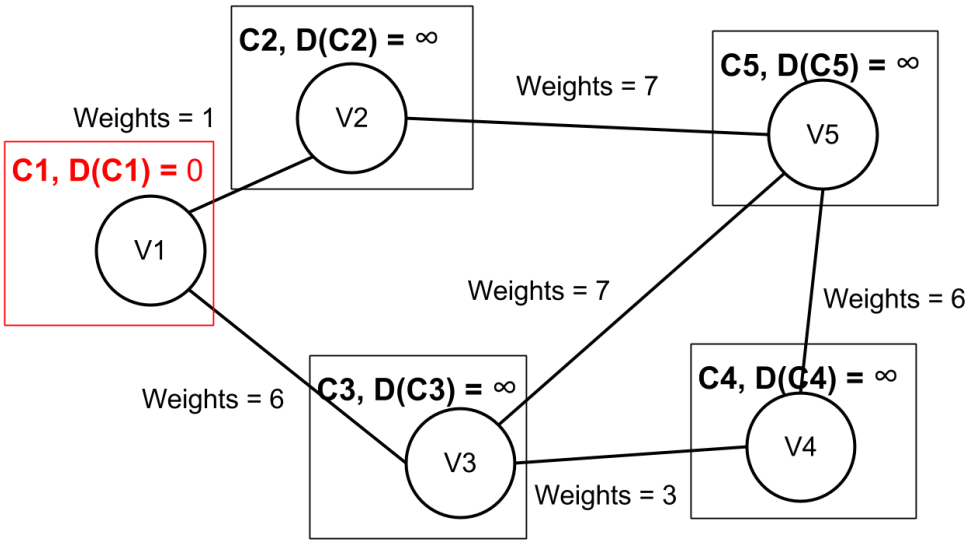


Figure 3.5. Graph of the components arrangement on level 1. Initially, set v1 as the source vertex. Since v1 is contained by component C1. The tentative distance of C1 is set to 1.

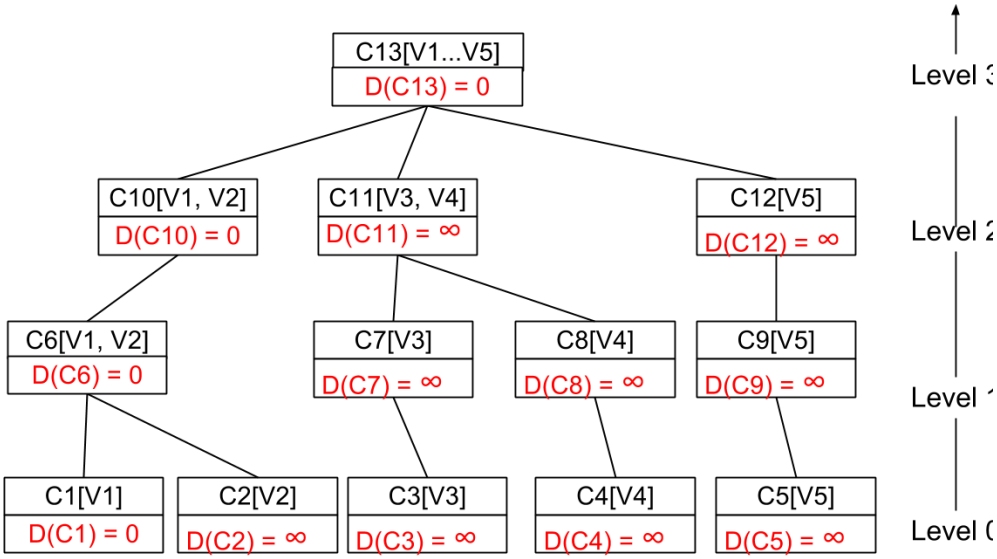


Figure 3.6. Plot of the component hierarchy when finished decreasing the tentative distance of C1 and its father components.

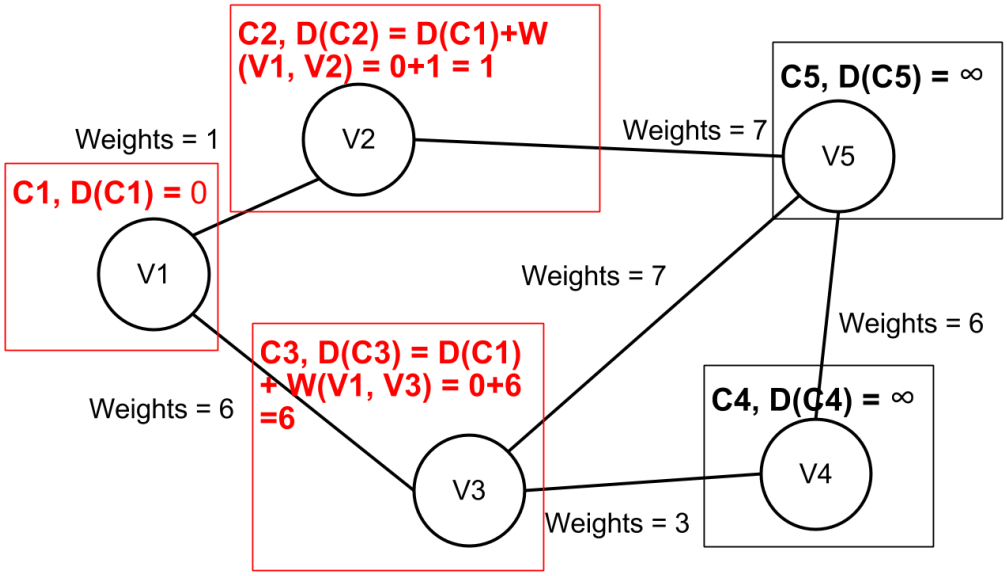


Figure 3.7. Decrease the tentative distance of v2 and v3 to 1 and 6, respectively.

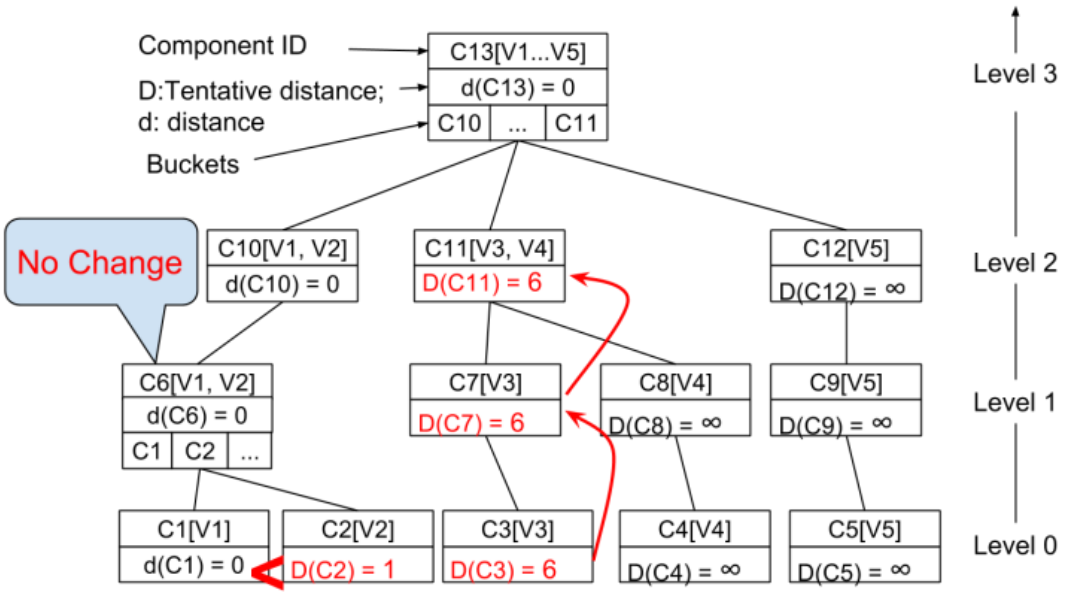


Figure 3.8. The tentative distances of C1 and C2's father components will not be updated, since they are smaller than the tentative distance of C2.

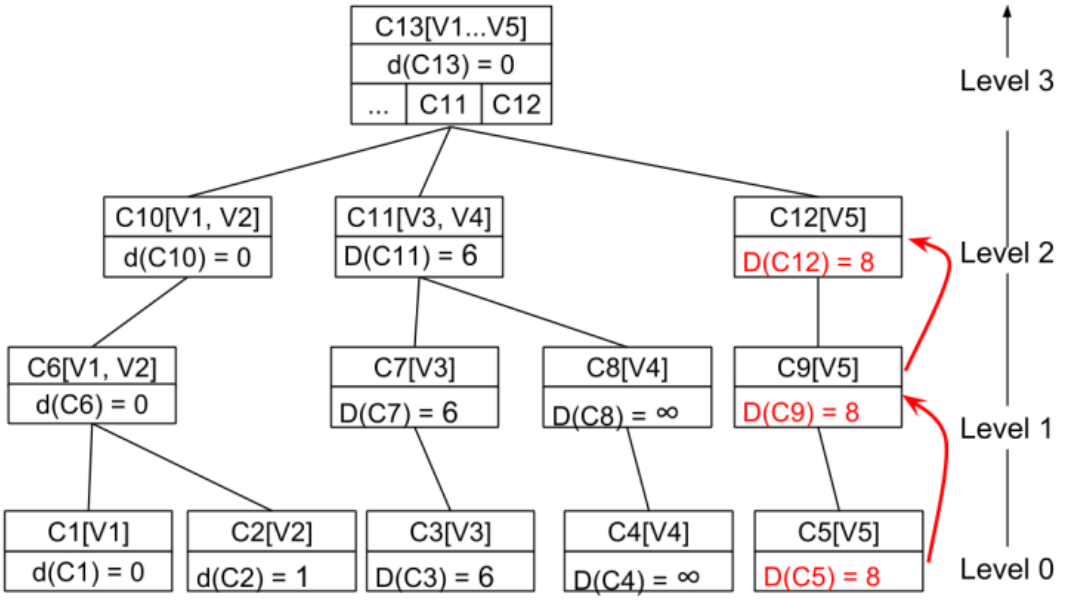


Figure 3.9. The tentative distances of C5 and its father components C9 and C12 will be decreased from infinite to 8.

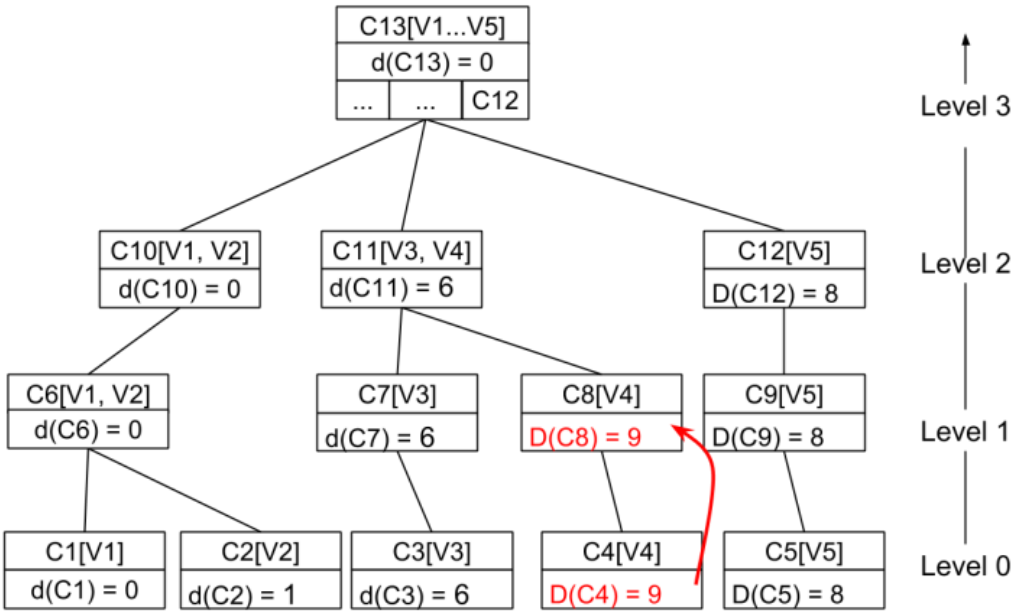


Figure 3.10. The tentative distances of C4 and its father component C8 will be decreased from infinite to 9.

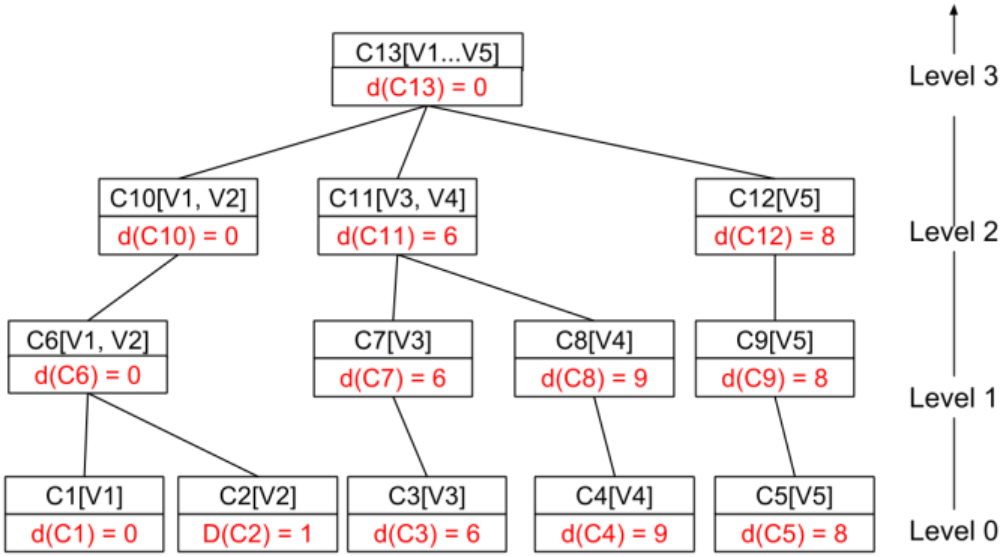


Figure 3.11. All children of C13 are visited, algorithm finished.

After decreasing the tentative distance of the components connected to source vertex, the component hierarchy will be visited from the root. At the first time when visiting a component, a sequence of buckets will be initialized for arranging the order of visiting its children. An ArrayList will be added in each of the buckets for storing components. In this situation, C10 is stay before C11 in the buckets, since its tentative distance is smaller. When visiting C6, C1 should be visited

firstly, and then C2, according to the same rule. Visited leaf-components will be removed from its father's buckets. A component which does not have any unvisited components will also be removed from its fathers' buckets. C1 and C2 will be removed from C6 after visiting. Since *v*2 connects with *v*5, as the structure shown in Figure 3.9, after visiting C2, the tentative distance of C5 and its father components C9 and C12 are decreased from infinite to 8. C12 will be stored to C13's buckets behind C6. All children of C6 are visited, remove C6 from C10's buckets, and remove C10 from C13.

Now we have C11 and C12 in C13's buckets, visit C11 first. C7 will be stored in C11's bucket. C8 will be ignored for the moment since we do not know its tentative distance yet. When visiting C3, we found v3 connects *v*4 and *v*5. The tentative distance of the component contains *v*4, and C4 and its father C8 will be decreased from infinite to 9. Keep C5 and its fathers' tentative distance. Remove C3 and C7 from C7 and C11, respectively. Go back to C11, storing C8 to the buckets of C11. Then visiting C4 through C8, *v*4 connects with *v*5, but will not decrease its tentative distance, so keep it as well. As the structure shown in Figure 3.10, after removing C4, C8 and C11 from the buckets of C8, C11 and C13, respectively, we have C12 only in C13's buckets. Visit C5 through C12 and C9, no vertex's tentative distance will be decreased. Then remove C5, C9 and C12 from C9, C12 and C13. As the structure shown in Figure 3.11, finally, all children of the root of this component hierarchy are visited, algorithm finishes.

4. Practical Experiment

The source code of experiment in this paper is modified based on [Prue09]. This experiment focuses on comparing the performance among Dijkstra result algorithm with priority queues including Fibonacci heaps and binary heap, original Thorup algorithm and the improved Thorup algorithm in Chapter 3. Since some algorithms mentioned in [Thor97] are difficult to implement, [Prue09] replaced some algorithms in [Thor97] with other algorithms. The detailed information is as follows,

1. Use Kruskal's algorithm [Krus56] to generate a minimum spanning tree so as to avoid using Fredman and Willard's union-find algorithm [Fred94], since their algorithm uses atomic heaps as a priority queue, which requires $n > 2^{12^{20}}$. It is the same as the solution given by Thorup [Thor97]. Moreover, in Kruskal's algorithm, Tarjan's union-find algorithm [Tarj75] is used to make sets. The time cost of constructing a minimum spanning tree is bounded in $O(\alpha(m,n)m)$. Here α is an inverse function of Ackermann, which grows very slowly. If the number of vertices is less than 10^{80}, $\alpha(m,n)$ should be equal to, or less than 4 [Corm09].
2. Use Tarjan's union-find algorithm [Tarj75] instead of the tabulation-based algorithm [Gabo85b] to construct components of the component tree.
3. Use Gabow's split-findmin data structure [Gabo85a] instead of atomic heaps to maintain the tentative distance of each vertex. It is called unvisited data structure in [Tarj75]. It is the same as the solution given by Thorup.

   The method of importing Shapfiles as datasets will be introduced in Section 4.1.

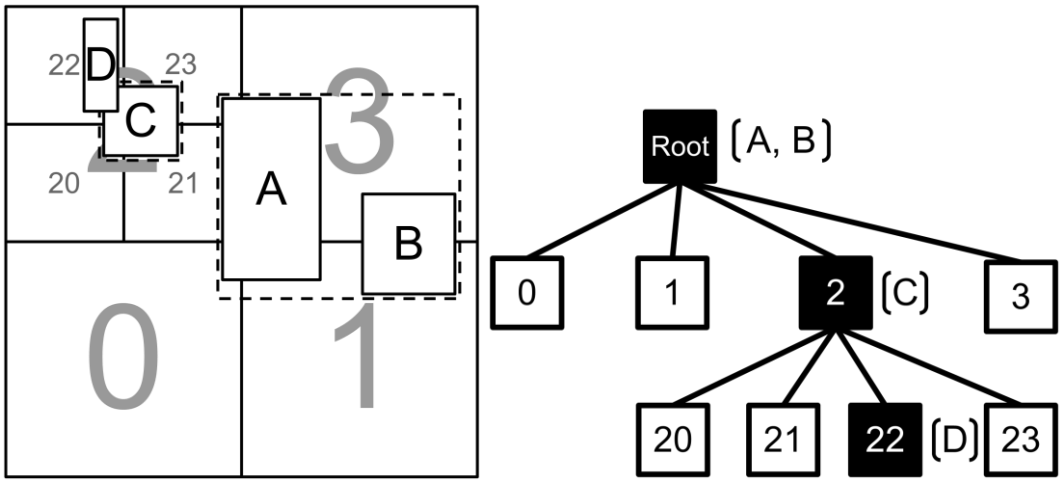
4.1. The improved MX-CIF quadtree

Generally, when performing a search in spatial index, all the objects' Minimum Bounding Rectangles (MBRs) which intersect the search-window will be taken as primary results, and then relations between the primary results and search-window are judged by a secondary query for an exact result. The amount of primary result is an important factor to affect time cost of the query. When creating experimental datasets from Shapefiles, an improved MX-CIF quadtree [Wei13] is used to prune the lines which do not intersect with any other lines. These lines will cause errors when calculating the minimum spanning tree, so they should not be included in experimental datasets. Every line will be queryed to find intersected lines, then delete the lines which do not intersect with any other lines. The improved MX-CIF quadtree reduces the time cost of query by decreasing primary results in number of objects.

Like other variants of the quadtree, in an improved MX-CIF quadtree, more than one object can be associated with both of leaf node and non-leaf node. Moreover, an MBR should be associated with only one node. Once an MBR is associated with a node, *N*, then the MBR will never be split to any sub-nodes of *N*. The associated objects will be inserted into a list of the node directly. The planar partition and structure is given by Figure 4.1. Four objects are indexed

with MX-CIF quadtree, *A* and *B* belong to the root, *C* belongs to node 2, that is, the northwestern sub-node of the root, and *D* belongs to node 22. An object should be inserted into the first node which its *x* or *y* axis intersects the object's MBR. So although *A* overlaps both of the axes of the root and the root's northwestern sub-node, it is still associated with the root only.

The specialty which accelerates the query of the improved MX-CIF quadtree is the element called Region-MBR. The Region-MBR is a rectangle used to present the minimum bounding of all the objects in the same node. It is created by the maximum *x* and *y* coordinate, and the minimum *x* and *y* coordinate of all the objects in a node. Comparing to the border of a node, Region-MBR provides a more precise frame for judging relations between object and search-window. When making queries to the original MX-CIF quadtree, the objects in the nodes which intersect the search-window will be taken as approximate results and reported to a secondary query for an exact judgment. According to the usage of the MX-CIF quadtree, assume *D* as a search-window to launch an overlap query, because both of the root and its northwestern sub-node intersect *D*, the primary results should include object *A*, *B* and *C*. And then, *A*, *B* and *C* will be reported to have an exact comparison with the search-window. But in the improved MX-CIF quadree, only in the situation that the nodes' Region-MBRs intersect the search-window, the objects associated with them can be reported to the secondary query. Appling this rule to the example mentioned before, the primary result will be *C* only. Because *D* does not intersect the Region-MBR of root, so the objects in the root, that is, *A* and *B*, should not be included. This method reduces the number of approximate results will be reported, so the query performance is enhanced.



(a) (b)

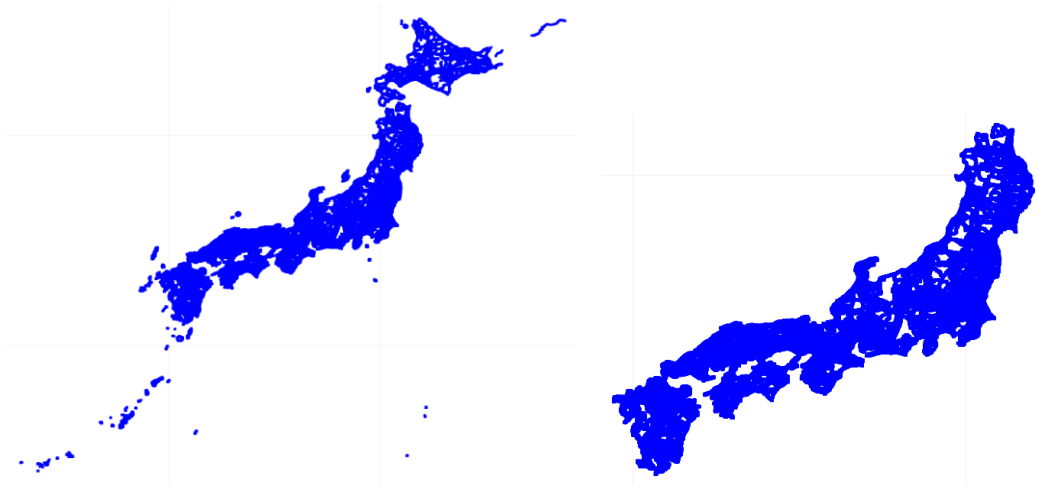
Figure 4.1. The planar partition (a) and structure (b) of an MX-CIF quadtree. Dash-line represents the Region-MBR of each node.

4.2. Importing real dataset

The experiment is originally supported running with generated datasets, in this paper, real dataset is also available. The result of [Prue09] is given in the introduction part. The dataset used in this experiment is the transportation of Japan, which is from the Geospatial Information Authority of Japan. The dataset can be found at:

www1.gsi.go.jp/geowww/globalmap-gsi/download/data/gm-japan/gm-jpn-trans_u_2.zip.

Since the vertices in the dataset are not connected with each other totally, a part of the dataset which is considered that all of its vertices are connected is extracted. The plot of the original dataset and extracted dataset are shown in Figure 4.1.



(a) (b)

Figure 4.2. Transportation of Japan (a) and dataset used in experiment (b).

For evaluating the performance of algorithms with different amount of vertices, five datasets are derived from the original dataset. The pruned datasets can be found at:

http://weiyusi.com/resource/gm-jpn-trans_u_2.7z

Their plots are shown in Figure 4.4. Their detailed information is included in Table 4.1.

The source code of this project can be found at:

http://weiyusi.com/resource/ShortestPaths.7z

Because [Prue09] uses adjacent list to present a graph, so the dataset should be transformed to adjacent list. For making the datasets easier to be edited, firstly, they are transformed from the format of Shapefile to WKT (Well-Known Text) by functions given in JTS [Davi03]. With the WKT format, a line which is created by three points $p1(x1, y1)$, $p2(x2, y2)$, $p3(x3, y3)$ will be presented as Linestring ($x1\ y1$, $x2\ y2$, $x3\ y3$).

Before the transformation, the dataset need to be pruned to delete single edges inside. Both sides of this kind of edge do not connect to any other edge. These edges should be cut to let the

dataset be an intact graph. An example of a single line is shown in Figure 4.2, and the algorithm is shown in Figure 4.3.

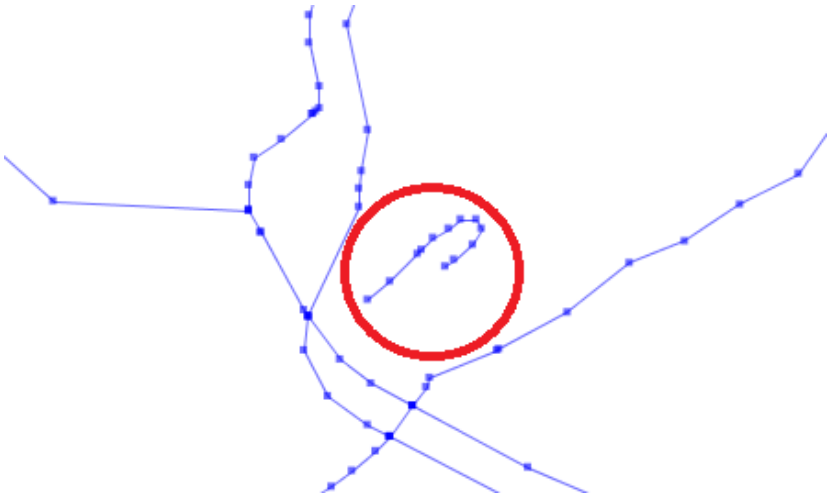


Figure 4.3. A single line (circled) in the graph of dataset.

```
Delete_single_line (dataset)
for each line x in dataset
    start = x.StartPoint
    end = x.EndPoint
    for each line y in dataset
        if (start.intersection(y) && x! = y) start_intersect = TRUE
        if (end.intersection(y) && x! = y) end_intersect = TRUE
        if (start_intersect == TRUE && end_intersect == TRUE)
            newdataset.add(x)
            break
return newdataset
```

Figure 4.4. Algorithm for deleting single lines in dataset.

Figure 4.5. Five datasets derived from original dataset.

Table 4.1. Information of datasets.

Info\Data	**1**	**2**	**3**	**4**	**5**
Vertices	1889	6913	11478	14295	16670
Edges	5920	21798	36262	44904	53318

Then the pruned datasets need to be transformed to an adjacent list. The list includes the information of the connection between vertices and their weights, so that the datasets can be used in [Prue09]. The algorithm of transforming dataset is shown in Figure 4.5. It uses hash tables to allocate a unique vertex ID to each endpoint of each line in a Shapfile, and gets the length of a line as the weights of an edge. So that it turns endpoints to vertices, and turns lines to the edges, which will be used in the experiment. In the situation that many lines have the same endpoints, the one has the shortest length will be reserved, and the others will be ignored. For making a distinction, let's call lines from Shapfile dataset as lines, and call lines in a graph as edges. Line 1 initializes two hash tables for storing vertices' IDs and weights, respectively. Line 2 initializes two lists. The list named *edges* is used to store the edges with allocated IDs. The list named *graph* is an adjacent list, it stores edges from the list *edges* in both directions. Lines 5 and 6 get the endpoints of each line as the vertices of each edge, and store them in variables *s* and *e* in the form of coordinate. Line 7 gets the length of a line as the weights of an edge. Lines 8 to 21 allocate a unique ID to each endpoint which was not allocated before. Each allocated endpoint will be stored in a hash table. Before allocating an ID, firstly, the program checks whether the endpoint is already in the hash table, which means it was allocated an ID before. This is used to make sure the spatial intersection point of two or more lines will not be allocated as two different IDs. In detail, Lines 8 and 9 query in the hash table *hb* with endpoints of each line. If the return is null, means the point have been queried is new and does not have an ID. Lines 11 and 17 allocate unique IDs for new points. Lines 12 and 18 store the ID of new points to two variables, respectively. It is for creating an edge later. If any of the endpoints has been allocated an ID, use the allocated ID to create the edge later. When there are more than one line having the same endpoints (an example is given in Figure 4.6), lines 22 to 30 make sure that only the edge which has the shortest length can be kept in the dataset. Line 22 creates a key with the two IDs got before. The key will be used in Line 23 to query the weights of the edge created by the two IDs. If the return is null, means no edge was created by these two IDs before. Line 25 inserts this edge to a hash table named *hbw* with the edge's weights. Line 26 adds the edge to the list *edges*, for creating adjacent list later. If the return is not null, means there are more than one lines have the same endpoints. Lines 27 to 30 compare the two weights and leave the smallest as the new weights. Lines 33 to 35 get all edges from *edges*, and get their weights from *hbw*. Lines 36 to 37 create adjacent list by storing each edge twice: positive and negative directions.

```
1.  Hashtable hb, hbw
2.  List edges, graph
3.  Int id = 0
4.  for each line l of the Shapfile do
5.      sp = l.StartPoint
6.      ep = l.EndPoint
7.      weight = l.Length
8.      s = hb.get(sp)
9.      e = hb.get(ep)
10.     if s ==null then
11.         hb.put(sp, id)
12.         sEdge = id
13.         id++
14.     else sEdge = s
15.     end if
16.     if e ==null then
17.         hb.put(ep, id)
18.         eEdge = id
19.         id++
20.     else eEdge = e
21.     end if
22.     we = sEdge + "-" + eEdge
23.     w = hbw.get(we)
24.     if w == null then
25.         hbw.put(we, weight)
26.         edges.add(WeightedEdge(sEdge, eEdge))
27.     else if w > weight then
28.         hbw.remove(we)
29.         hbw.put(we, weight)
30.     end if
31. end for
```

```
32. for each edge e in edges do
33.     sEdge = e.Source
34.     eEdge = e.Target
35.     weight = hbw.get(sEdge+ "-" + eEdge)
36.     graph.add(WeightedEdge(sEdge, eEdge, weight))
37.     graph.add(WeightedEdge(eEdge, sEdge, weight))
38. end for
```

Figure 4.6. Algorithm of transform dataset to adjacent list.

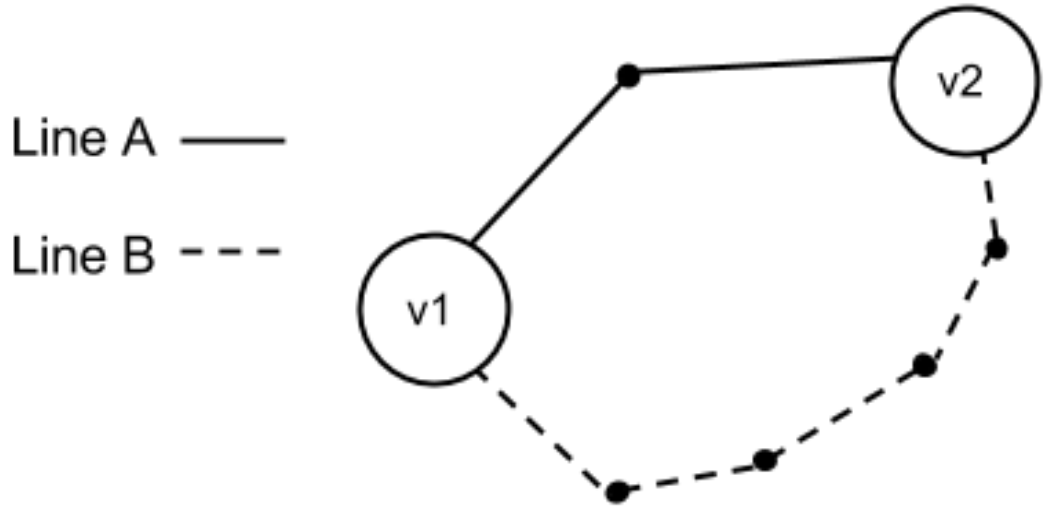


Figure 4.7. When there is more than one line having the same endpoints, the edge which has the shortest length can be kept in the dataset

After transforming lines to edges, all edges will be stored into a two-dimensional list two times, positive sequence and negative sequence as follows,

edge(*SourceVertexID*, *TargetVertexID*, *Weights*)

edge(*TargetVertexID* , *SourceVertexID*, *Weights*)

The list uses vertex ID as the index of the first dimension, and stores the information of edges in the second dimension, each item of information occupies a node. The data structure of the list is as follows,

list[*VertexID*][*Connections*]

For instance, the graph in Figure 4.8 will be transformed as follows,

List: [1]-[*edge*(1, 2, 1)]->[*edge*(1, 3, 6)]

[2]-[*edge*(2, 1, 1)]->[*edge*(2, 5, 7)]

[3]-[*edge*(3, 1, 6)]->[*edge*(3, 4, 3)]->[*edge*(3, 5, 7)]

[4]-[*edge*(4, 3, 3)]->[*edge*(4, 5, 6)]

[5]-[*edge*(5, 2, 7)]->[*edge*(5, 3, 7)]->[*edge*(5, 4, 6)]

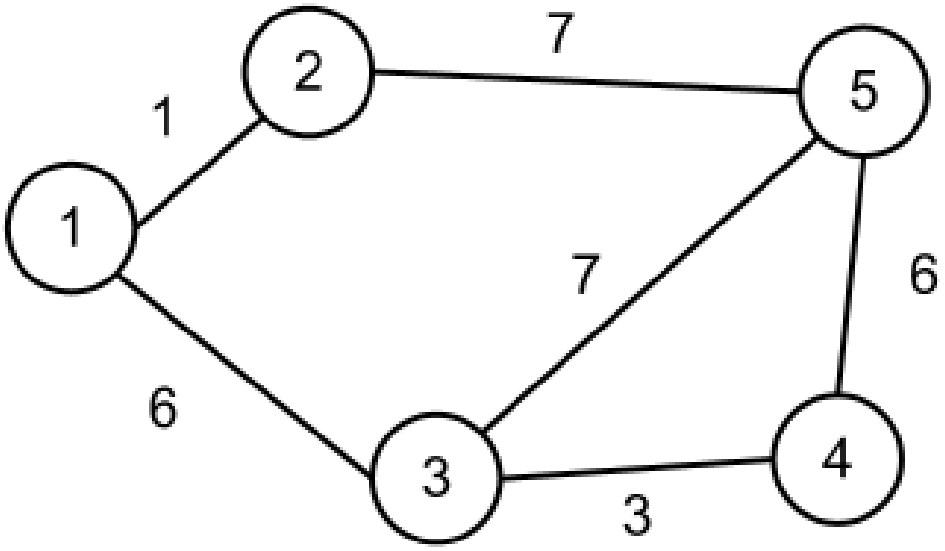


Figure 4.8. A graph with five vertices.

4.3. Experiment and result

The performance of the improved Thorup algorithm is evaluated through our modified experiment which originally provided by Pruehs [Prue09]. The experiment compared the performance of Dijkstra and Thorup algorithm with synthetic data sets. The time cost of finding the distance of every vertex to a given source vertex is compared among the following three: Dijkstra with array-based primary queue, Dijkstra with Fibonacci-based primary queue, and Thorup. Based on the modifications that has been introduced at the very beginning of this Chapter, we modified the experiment in two phases,

1. Added the code of the improved Thorup algorithm.
2. Made the experiment possible to run with real datasets.

The result of comparison between Dijkstra, Thorup and our algorithm is given in Table 4.2. Figure 4.9 and Figure 4.10 show the chart of results, respectively. Each of them is the average of 1,000 times calculation. Their standard deviations are in the Figures and are also shown in Table 4.3. Because of Thorup algorithm can respond in arbitrary time of shortest path query from any vertex by constructing an index only one time, here we focus on the comparison of visiting part.

Comparing to the Array-based Dijkstra, Fibonacci-based Dijkstra and the original Thorup algorithm, our algorithm reduced 17.3%, 78.7% and 87.7% of time cost for all the five datasets, respectively. Theoretically, since the amortized time complexities of the Fibonacci heap is better than array heap, it should help the Dijkstra algorithm to perform better. But the experiment indicates that the Fibonacci heap did not perform as well as expected. It is because the difficulty of implementation reduces its efficiency, which is also mentioned in [Corm09].

The value of *base* has been changed from 8 to 64 for testing the most suitable value of these datasets. The comparison result is shown in Table 4.4 and Figure 4.11. The chart in Figure 4.12 shows the trends of *base* increment and the total time cost. The results show that with these datasets, the total time costs are almost the same when setting *base* equal to 16 and 32. Using bigger *base* will reduce the number of buckets and then decrease memory usage. When setting *base* equal to 64, for these datasets, will let the component tree has only one component, the root, with a batch of buckets. All vertices will be mapped to it, directly ($2^{64} = 18,446,744,073,709,551,616$). Since the vertices are visited more efficiently when all vertices are mapped in the same level of the component tree, the total time cost is less than using smaller *base* value. In this case, the memory space is large enough to create a sufficient number of buckets for the experimental datasets, the buckets based Dijkstra algorithm should be more efficient, since it does not need pre-indexing and the number of created buckets may fewer.

Table 4.5 shows the comparison of memory usage among four algorithms. The result is obtained by using JConsole. Comparing to the original Thorup algorithm, since our algorithm does not use unvisited structure, the memory usage is greatly saved. But still takes about 200% more memory than the array based Dijkstra algorithm. In the four results about the improved Thorup algorithm, using base 8, 16 and 32 took almost the same memory usage. Since the component trees have the same depth with these values of base. It took more memory when using 64 as the base. It is because in this situation, the component tree has only on component with a batch of buckets for mapping all vertices in the graph. The buckets number is increased. The experimental environment is listed as follows, since the memory space of the experimental environment (16GB) is much larger than the most memory usage (about 50 Mbytes), so there should be no memory swapping.

CPU	Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz, 2394 MHz
Memory	PC3-12800 (800 MHz), 16GBytes
OS	Microsoft Windows 7 64-bit Service Pack 1

Table 4.2. Result of Experiment (milliseconds).

Algorithms\Datasets	**1**	**2**	**3**	**4**	**5**
Array heap based Dijkstra	0.2670	1.2448	2.1284	2.7046	3.2503
Fibonacci heap based Dijkstra	1.4217	4.6497	8.2387	10.5074	12.4469
Thorup Construct Structures	0.8900	3.2701	5.6787	7.3159	8.9413
Thorup Visiting	1.3443	7.1367	14.1758	20.2040	21.4530
Improved Thorup Constructing (*base* = 16)	0.8724	3.2929	5.7094	7.3788	8.9999
Improved Thorup Visiting (*base* = 16)	0.2151	0.9161	1.6719	2.2689	2.8642

Table 4.3. Standard Deviations (milliseconds).

Algorithms\Datasets	**1**	**2**	**3**	**4**	**5**
Array heap based Dijkstra	0.0028	0.0553	0.0492	0.0843	0.0939
Fibonacci heap based Dijkstra	0.0102	0.0626	0.0775	0.0725	0.0606
Thorup Visiting	0.0095	0.1230	0.1301	0.1241	0.1376
Improved Thorup Visiting (*base* = 16)	0.0023	0.0178	0.0789	0.0713	0.1045

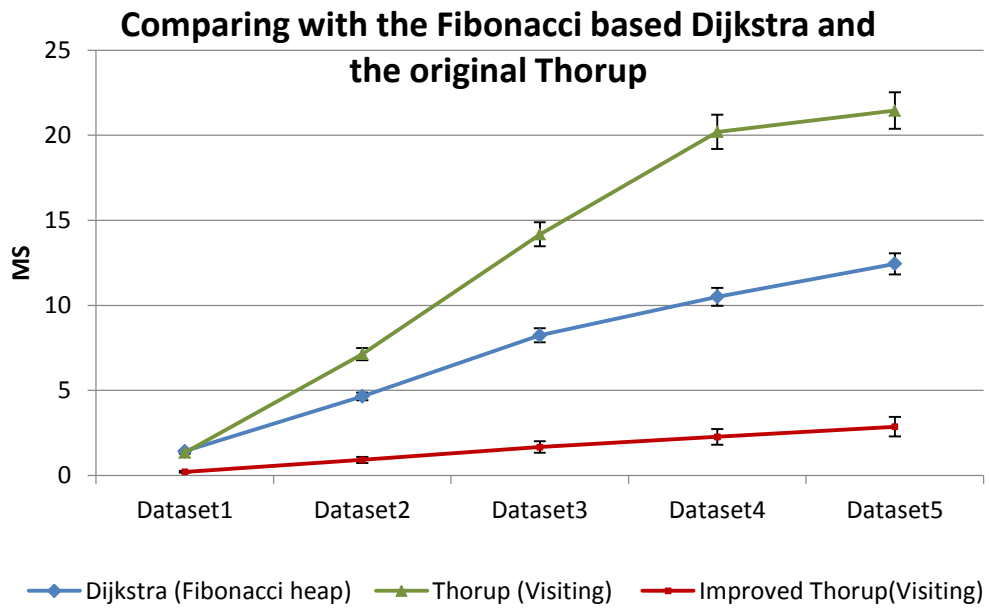


Figure 4.9. Chart of the results. Comparing with the Fibonacci based Dijkstra and the original Thorup (*base* = 16).

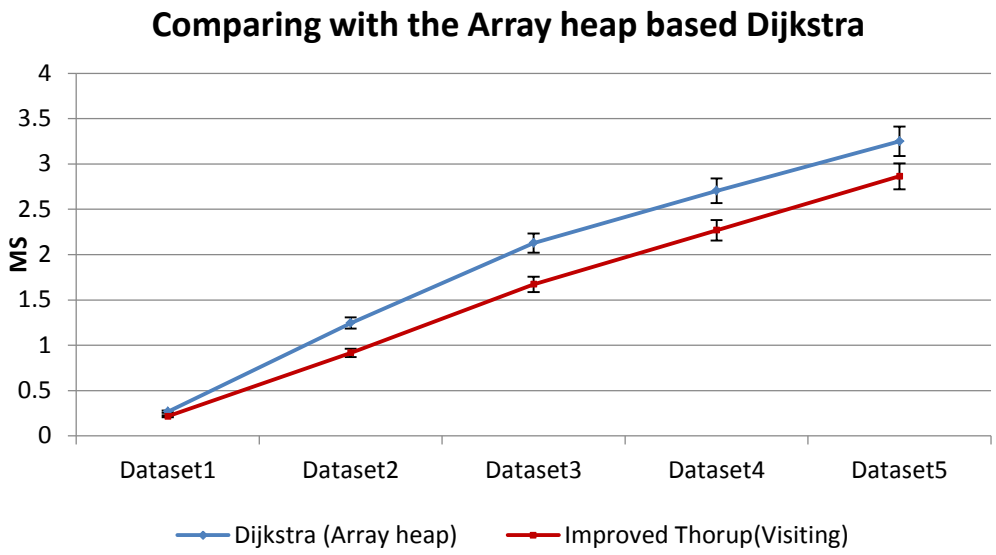


Figure 4.10. Chart of the results. Comparing with the Array heap based Dijkstra (*base* = 16).

Table 4.4. Experimental result of using different values of *base* (milliseconds).

	Base = 8	Base = 16	Base = 32	Base = 64
Dataset1	0.2408	0.2151	0.2095	0.1940
Dataset2	0.9577	0.9161	0.9157	0.8509
Dataset3	1.7266	1.6719	1.6364	1.5463
Dataset4	2.3574	2.2689	2.2514	2.1748
Dataset5	2.9962	2.8642	2.8679	2.7139
Total time cost	8.2787	7.9362	7.8809	7.4799

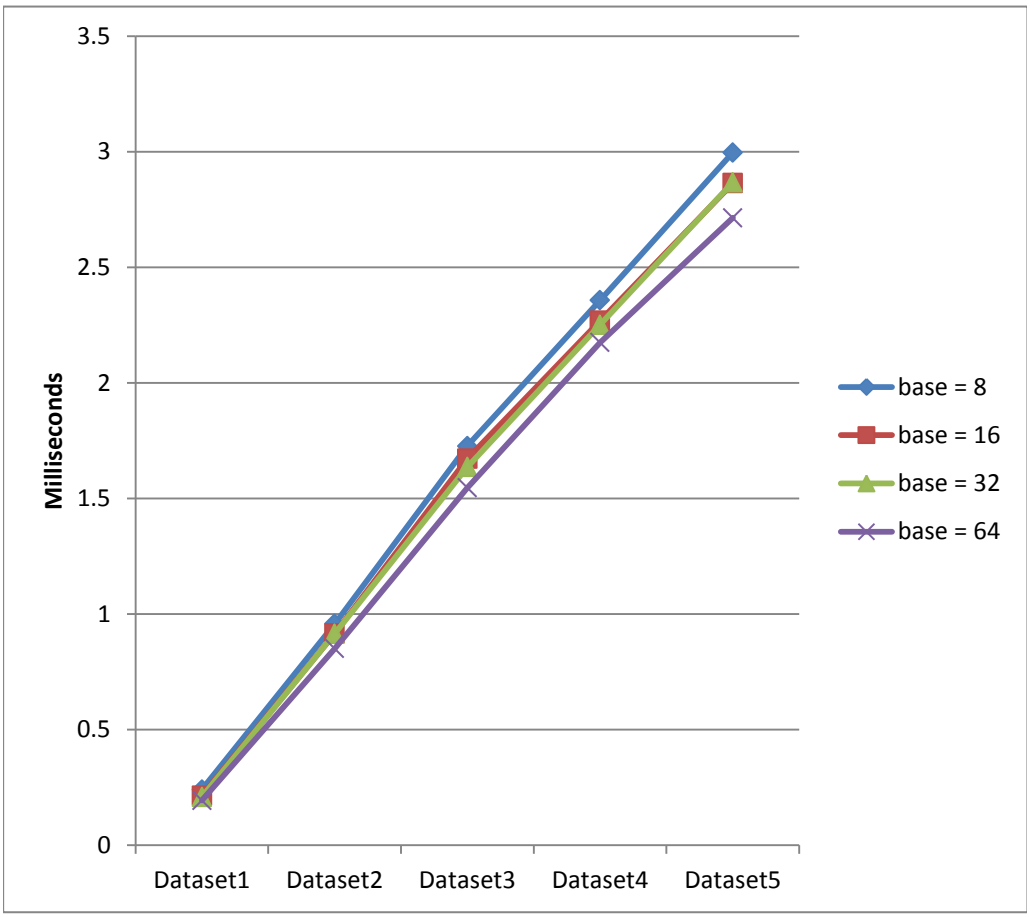


Figure 4.11. Time cost comparison among different values of *base*.

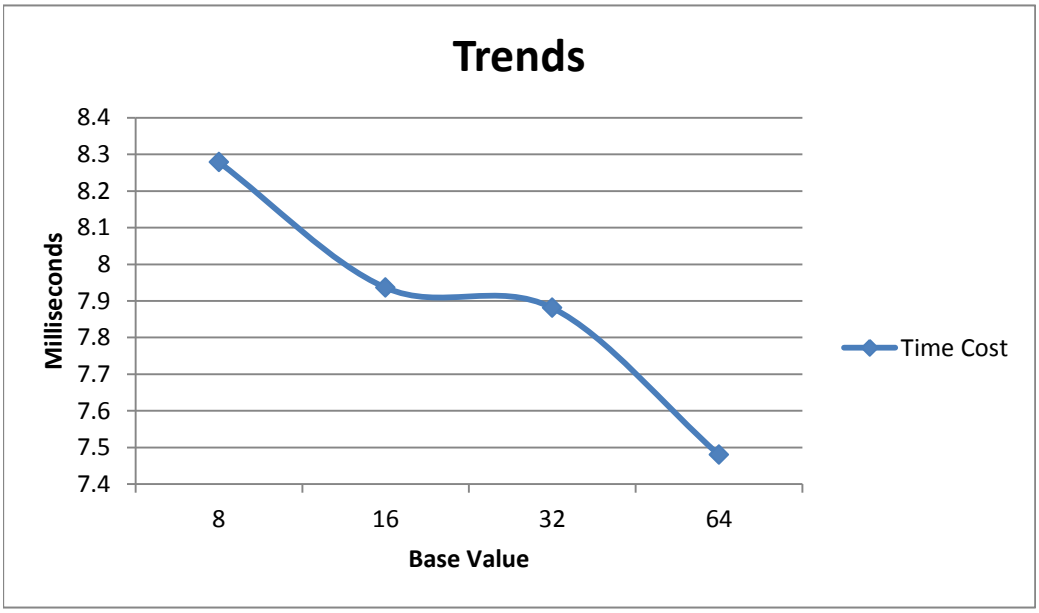


Figure 4.12. Trends of *base* increment and the total time cost.

Table 4.5. Comparison of memory usage.

Algorithms	**Memory Usage (bytes)**
Array heap based Dijkstra	6,429,896
Fibonacci heap based Dijkstra	12,865,056
Thorup	51,456,936
Improved Thorup (*base* = 8)	19,299,944
Improved Thorup (*base* = 16)	19,281,368
Improved Thorup (*base* = 32)	19,298,168
Improved Thorup (*base* = 64)	19,446,720

5. Conclusion

This paper introduced a single source shortest path algorithm named Thorup, and proposed its improvement. Comparing to the original Thorup algorithm, the improved algorithm reduces the time cost in both of constructing and visiting phases. This is realized by two improvements as follows,

1. Modifying the component tree to make it able to maintain the tentative distances instead of the unvisited structure,

2. Reduce the depth of the component tree in order to enhance the efficiency during visiting phases.

The experimental result indicates, comparing to the array-based Dijkstra, Fibonacci-based Dijkstra and the original Thorup algorithm, our algorithm reduced 17.3%, 78.7% and 87.7% of the time cost, respectively. About the memory usage, since we avoid using unvisited structure, our algorithm takes about only a half of the memory usage of the original Thorup algorithm. But still takes about 200% more memory than the array based Dijkstra algorithm. The original Thorup algorithm costs much more time to finish the query comparing to other algorithms. One of the reasons is that the structures and algorithms used in the Thorup algorithm are complex. It makes the implementation lose its efficiency. The other reason is, the atomic heaps [Fred94] used by Thorup as the unvisited structure, designed for running on the RAM which its word length has to be larger than log*n*, where *n* is the number of vertices. But the computers we used in common and also in the experiment mentioned above use byte addressing RAM. To my understanding, that is to say, to make the Thorup algorithm performance well in common computer, the length of a word has to be equal to the length of a byte, which is 8bit. Due to the requirement (word length > log*n*), *n* must be less than 256. For breaking this limitation, the split-findmin algorithm is used to instead. Its time complexity is $O(\alpha(m,n)m)$, where α is an inverse function of Ackermann which grows very slow. If the number of vertices is less than 10^{80}, $\alpha(m,n)$ should be equal to, or less than 4. [Corm09, Section 21.4] This change reduces the Thorup algorithm's cost to $O(\log C + \alpha(m,n)m)$, where *C* represents the maximum edge weights.

The improved Thorup algorithm uses a modified component tree which could afford all the work of unvisited structure. Since the unvisited structure is not used in the new algorithm, the time cost of constructing structures and the memory space is saved. During the period of visiting, instead of visiting both of the component tree and the unvisited structure to calculate the tentative distance of each component, we only need to visit the component tree, thus the time cost of visiting is saved too. In the worst case, the time cost of the improved Thorup algorithm including both of construction and visiting part is $O(\log C + \alpha(m,n)m + dn)$, where *C* represents the maximum edge weights, *n* represents the number of vertices, and *d* represents the depth of a component tree. It happens when all the tentative distances of the vertices are smaller than their parents' tentative distance. In this situation, all the vertices have to update its tentative distance to all its parents in each level of the component tree, cost $O(d)$ for each vertex. $O(\log C + \alpha(m,n)m)$ is the time complexity of construction, the same as the original

Thorup algorithm., though the new algorithm does not need to construct the unvisited structure. And $O(dn)$ is the time complexity of visiting part. Since generally, in the case of memory space is sufficient, the depth of a component tree should be controlled as small as possible, such as 3 or 4, so the time complexity should be almost the same with the original Thorup algorithm. On the other hand, theoretically, the Fibonacci heaps based Dijkstra algorithm should be faster than the array based Dijkstra algorithm. And the original Thorup algorithm should be the fastest. The reason of getting opposite experimental result is that the Fibonacci heaps and the split-findmin algorithm is difficult to implement. Using Fibonacci heaps as the priority queue or using split-findmin algorithm as the unvisited structure might not be efficient in practice. To invent more efficient algorithm and implementation used to construct the unvisited structure will be very likely to accelerate the Thorup algorithm again.

Acknowledgment

I would like to extend my sincere gratitude foremost to my supervisor, Professor ShojiroTanaka, for his continuous guidance of my Ph.D study, for his patience and encouragement. This dissertation could not have accomplished without his instruction. Second, I would like to express my gratitude to Professor Okamoto, Professor Hamaguchi and Professor Hirotomi, who have instructed and helped me a lot. Last my thanks would go to my beloved family for their loving considerations and encourage through these years.

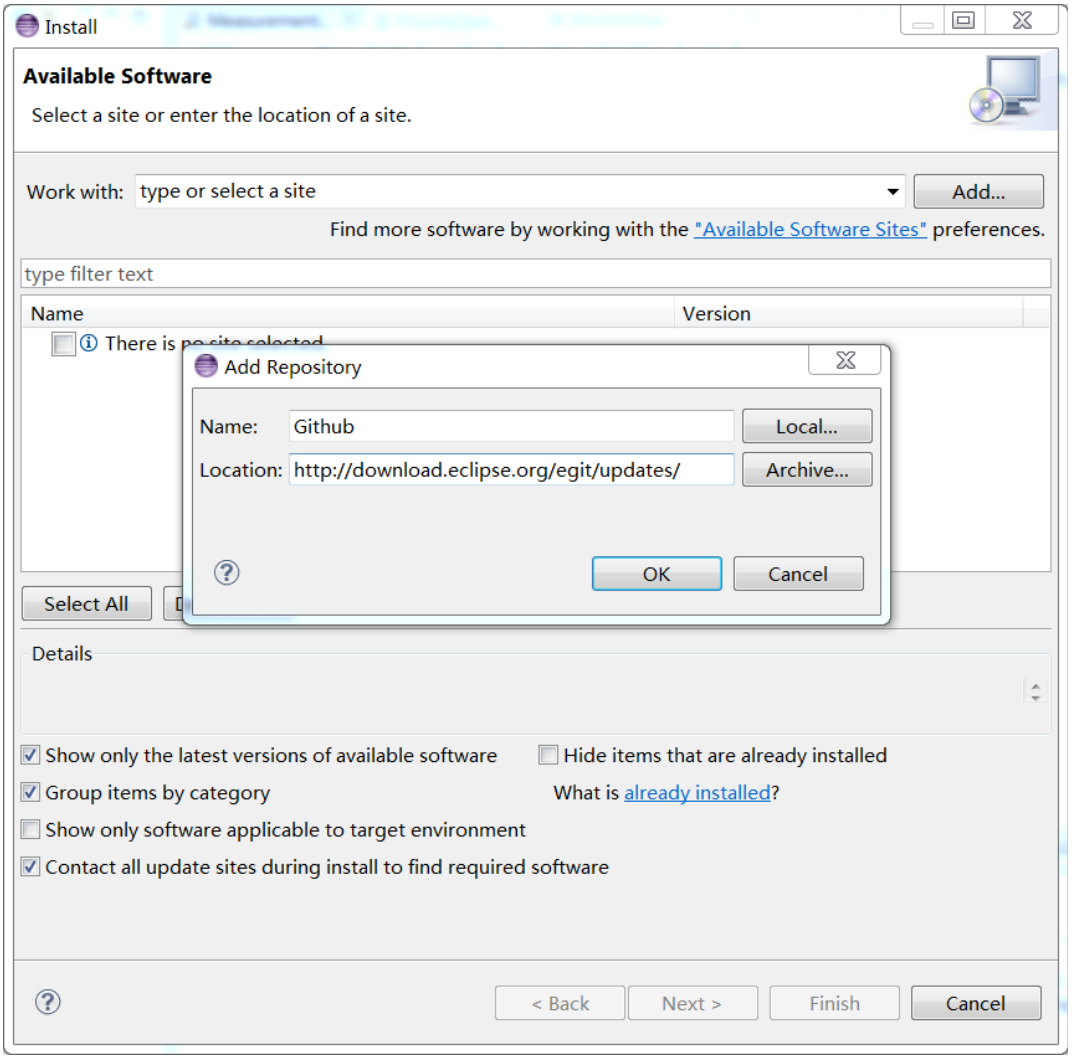
Appendix

Using GitHub

GitHub is a web-based revision control system. It offers free service for open source project. The code of the experiment mentioned in this Dissertation will be maintained through GitHub. The method of pulling code from GitHub will be introduced in this Section.

Here we use Eclipse as the local environment. Initially, the plugin of GitHub should be installed to Eclipse. The steps of installing are as follows,

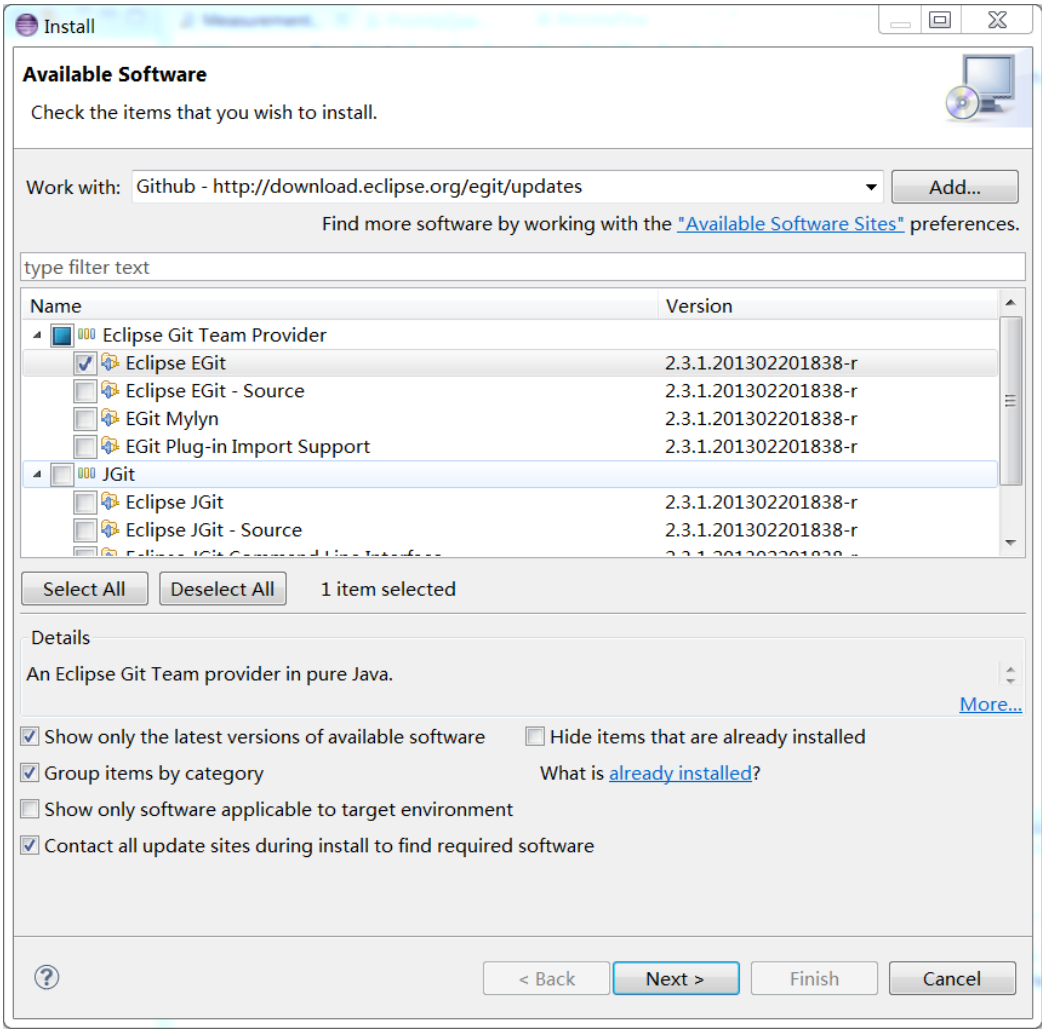
1. Click “Help” from the menu, and then select “Install New Software….”
2. Click the button “Add…” to add a source as the example shown in Appendix. Figure 1. Here we use http://download.eclipse.org/egit/updates. Fill the “Location” column with this address, and fill the “Name” column with any words you like.
3. Select the source just added from the Drop-down box “Work with”. Open up “Eclipse Git Team Provider”, and then check the item “Eclipse EGit” as the example shown in Appendix. Figure 2. When the installation is finished, a restart of Eclipse is required.
4. After restart, we are going to pull the project from GitHub. First right click in the area of “Project Explorer”, and then click “Import” as the example shown in Appendix. Figure 3. The “Project Explorer” is normally on the left of Eclipse’s interface. If it is not there, open it by clicking “Window” -> “Show View” -> “Project Explorer” on the menu.
5. Open the drop-down menu “Git”, and then select “Projects from Git” as the example shown in Appendix. Figure 4. Click “Next”, and then select “URI” for pulling project from GitHub’s server. Click “Next” again.
6. When come to “Source Git Repository”, fill “URI” column with the address https://github.com/weiyusi/ShortestPaths.git, then the rest columns will be filled by default. And then click “Next.”
7. Select a branch in Branch selection. “Master” branch will be selected by default. Click “Next.”
8. Set a local path to store the pulled project in “Local Destination.” Click “Next.”
9. Select “Import existing projects” to store pulled project to the place decided in the last step. Click “Next.”
10. Confirm the projects you would like to pull, and then finish the importing. The project “ShortestPaths” will be shown in the “Project Explorer” if successful.



Appendix. Figure 1. Adding the source of GitHub to Eclipse.

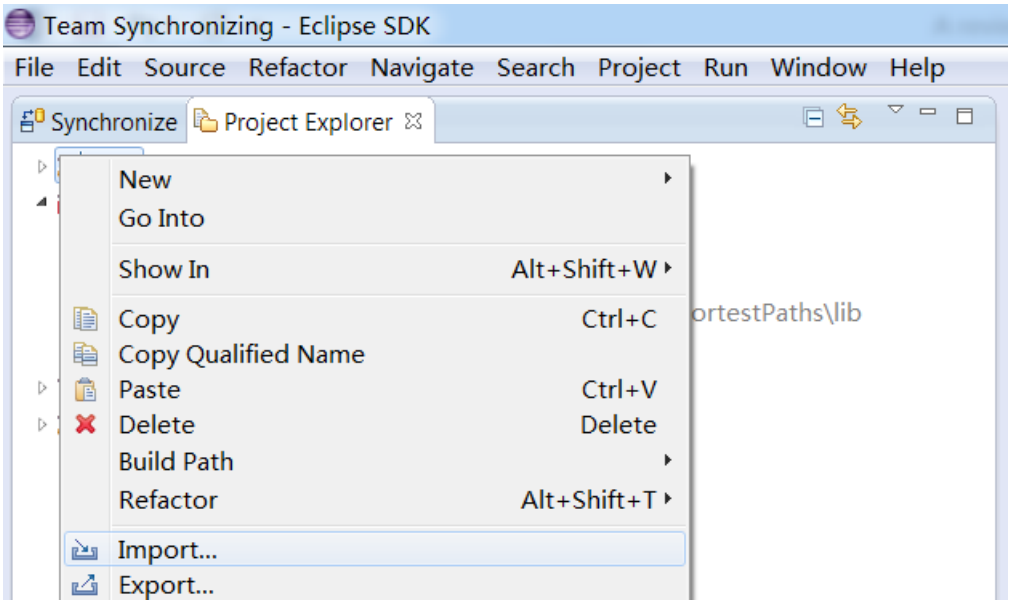
Click the button “Add…” to add a source.

Here we use http://download.eclipse.org/egit/updates. Fill the “Location” column with this address, and fill the “Name” column with any words you like.



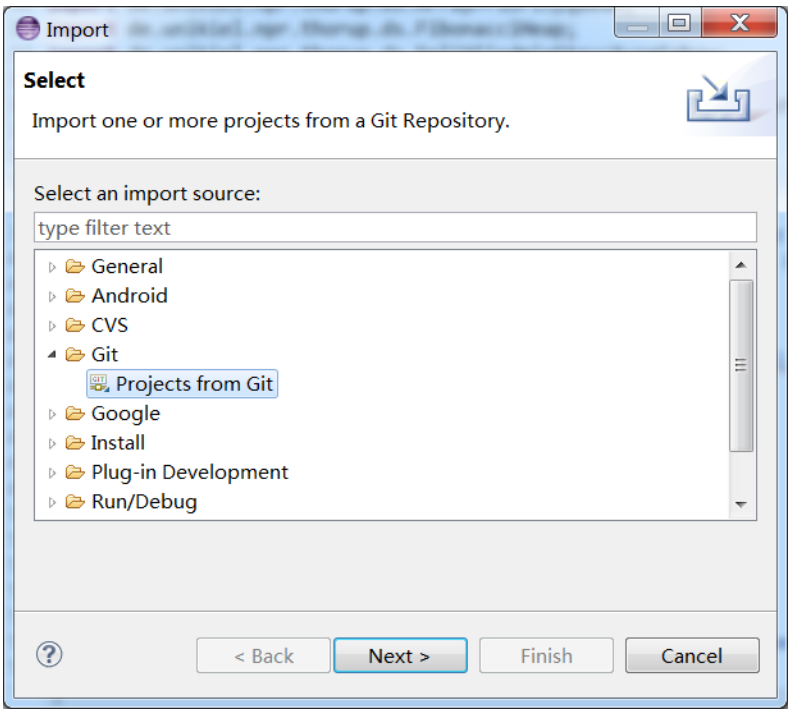
Appendix. Figure 2. Select Eclipse Egit from the source.

Select the source just added from the Drop-down box “Work with”. Open up “Eclipse Git Team Provider”, and then check the item “Eclipse EGit”. When the installation is finished, a restart of Eclipse is required.



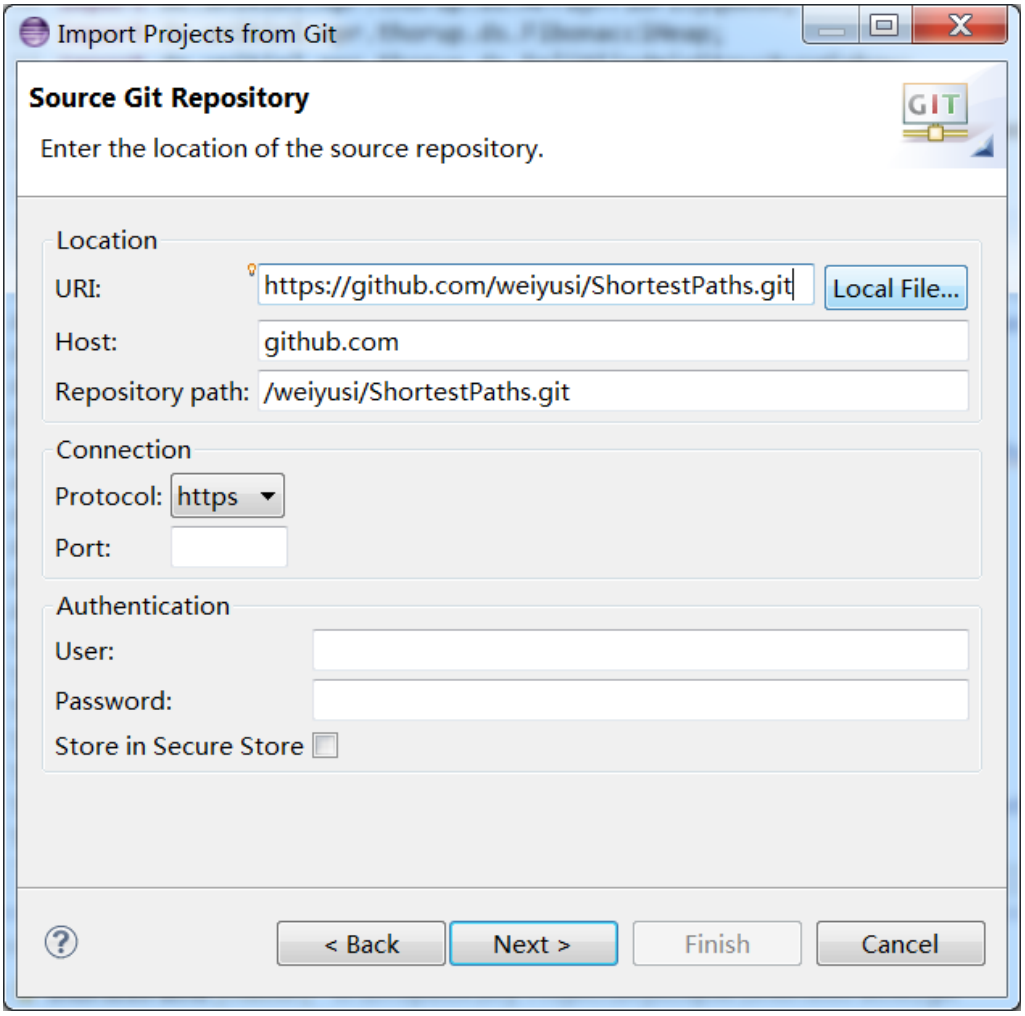
Appendix. Figure 3. Import project (1).

After restart, we are going to pull the project from GitHub. First right click in the area of "Project Explorer", and then click "Import". The "Project Explorer" is normally on the left of Eclipse's interface. If it is not there, open it by clicking "Window" -> "Show View" -> "Project Explorer" on the menu.



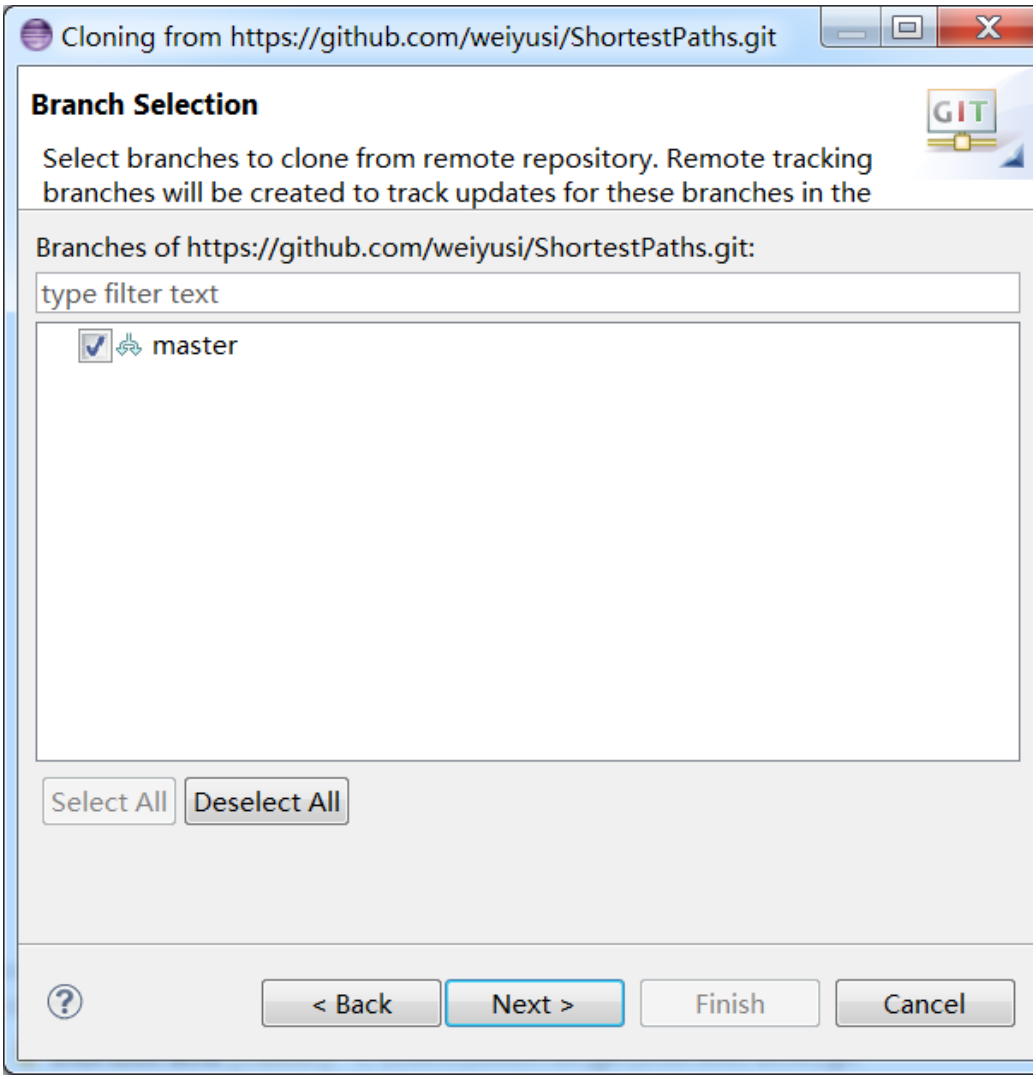
Appendix. Figure 4. Import project (2).

Open the drop-down menu “Git”, and then select “Projects from Git”. Click “Next”, and then select “URI” for pulling project from GitHub’s server. Click “Next” again.



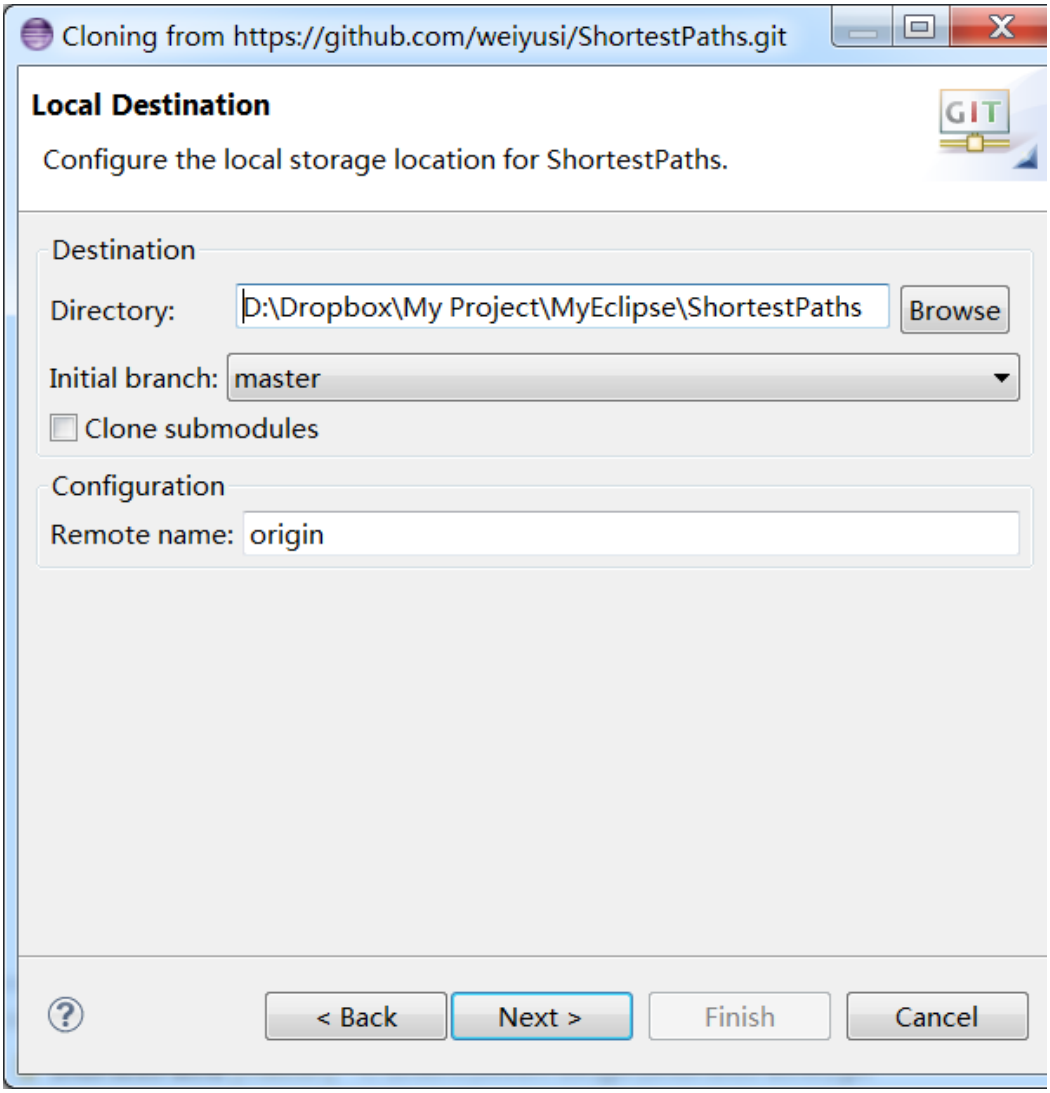
Appendix. Figure 5. Import project (3).

Fill “URI” column with the address https://github.com/weiyusi/ShortestPaths.git, then the rest columns will be filled by default. And then click “Next.”



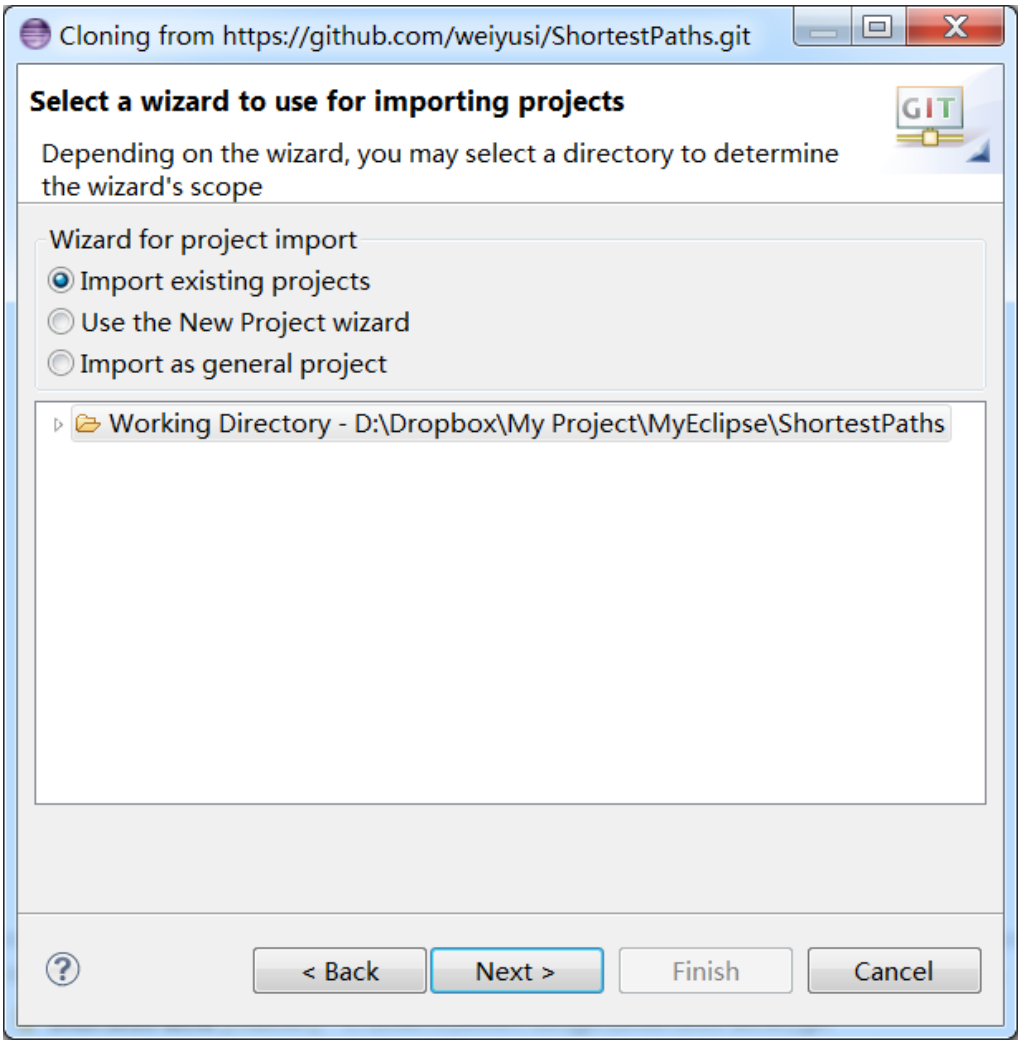
Appendix. Figure 6. Import project (4).

Select a branch in Branch selection. “Master” branch will be selected by default. Click “Next.”



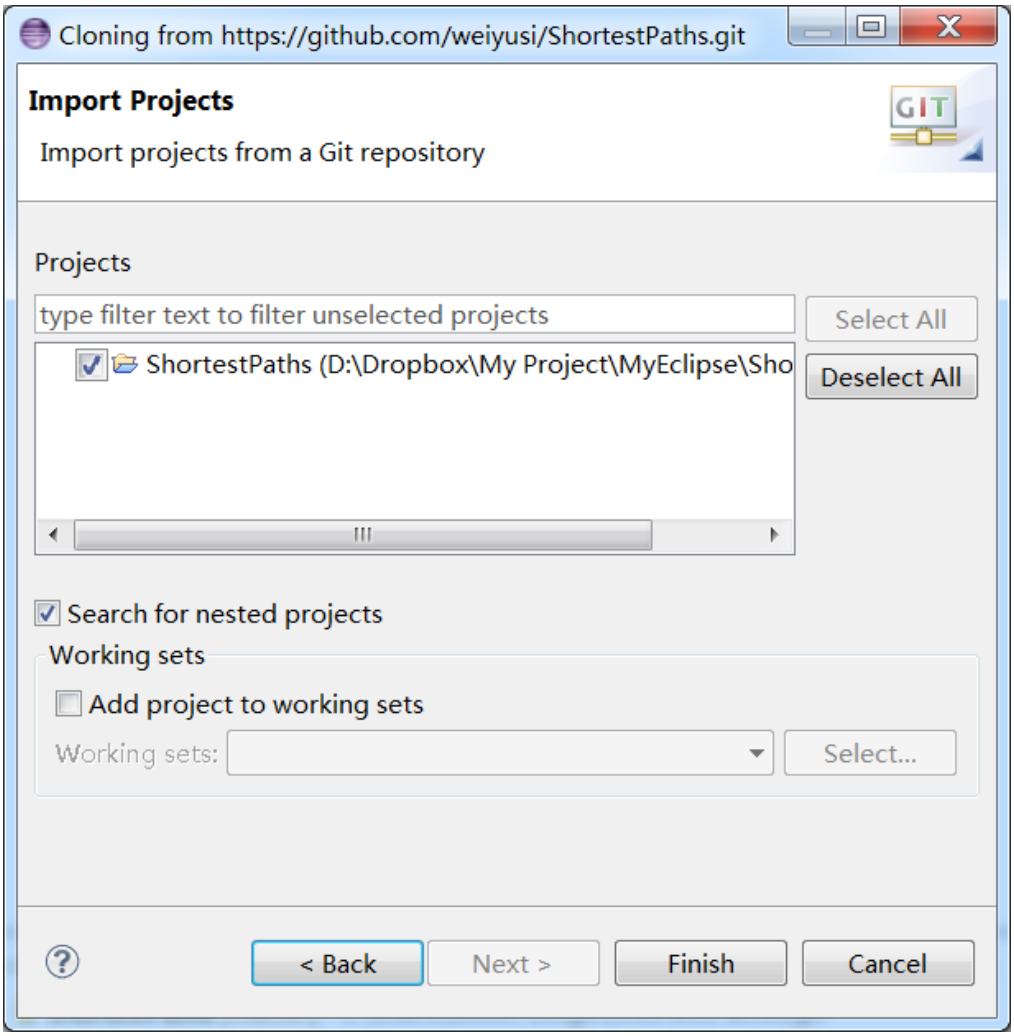
Appendix. Figure 7. Import project (5).

Set a local path to store the pulled project in “Local Destination.” Click “Next.”



Appendix. Figure 8. Import project (6).

Select “Import existing projects” to store pulled project to the place decided in the last step. Click “Next.”



Appendix. Figure 9. Import project (7).

Confirm the projects you would like to pull, and then finish the importing. The project "ShortestPaths" will be shown in the "Project Explorer" if success.

References

[Asan00] Asano, Y., Imai, H., 2000, Practical Efficiency of the Linear Time Algorithm for the Single Source Shortest Path problem, Journal of the Operations Research, Society of Japan, Vol. 43, No. 4, pp.431-447.

[Bhal02] Bhatia, G., et al., 2002, Keyword Searching and Browsing in Databases using BANKS, ICDE Conf, pp.431-440.

[Cher96] Cherkassky B V, Goldberg A V, Radzik T. Shortest paths algorithms: theory and experimental evaluation. Mathematical programming, 1996, 73(2), pp.129-174.

[Corm09]Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill, 1292pp.

[Davi03] Davis M, Aquino J. Jts topology suite technical specifications. The Jump Project, 2003.

[Dena79] Dennard, E. V., Fox, B. L., 1979, Shortest route methods: 1. reaching pruning and buckets, Operations Research, 27, pp.161-186.

[Dial69] Dial, R. B., 1969, Algorithm 360: Shortest Path Forest with Topological Ordering, Comm. ACM 12, pp.632-633.

[Dijk59] Dijkstra, E. W., 1959, A note on two problems in connexion with graphs, Numerische Mathematik 1, pp.269-271.

[Dini78] Dinic, E. A. 1978, Economical algorithms for finding shortest paths in a network. In Transportation Modeling Systems, Y.Popkov and B. Shmulyian, eds. Institute for System Studies, Moscow, CIS, pp. 36-44.

[Dris88] Driscoll, J. R., Gabow,H. N., Shrairman, R., Tarjan, R. E., 1988, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, Communications of the ACM, 31(11), pp.1343-1354.

[En12] En, D., Wei, H., Yang, J., Wei, N., Chen, X., Liu, Y., 2012, Analysis of the Shortest Path of GPS Vehicle Navigation System Based on Genetic Algorithm, Electrical, Information Engineering and Mechatronics 2011, Springer London, pp.413-418.

[Erik08] Erik, D., Rivest, R., Devadas, S., 2008, 6.006 Introduction to Algorithms, Massachusetts Institute of Technology: MIT OpenCourseWare.

[Fred87] Fredman, M. L., Tarjan, R. E., 1987, Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34, pp.596-615.

[Fred90] Fredman, M.L, Willard, D.E., 1990, Blasting through the Information Theoretic Barrier with Fusion Trees, Proc. ACM Symp. on Theory of Computing, pp.1-7.

[Fred93] Fredman, M. L., Andwillard, D. E., 1993, Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. 47, pp.424-436.

[Fred94] Fredman, M.L., Andwillard, D. E., 1994, Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. Syst. Sci. 48, pp.533-551.

[Gabo85a]Gabow, H. N. 1985. A scaling algorithm for weighted matching on general graphs. In Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science.IEEE Computer Society Press, Los Alamitos, Calif., pp.90-100.

[Gabo85b]Gabow, H. N., Tarjan, R. E., 1985, A linear-time algorithm for a special case of disjoint set union. J. Comput. Syst. Sci. 30, pp.209-221.

[Gils73] Gilsinn, J., Witzgall, C., 1973, A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees, NBS Technical Note 777, National Bureau of Standards, Washington, D.C, 87pp.

[Gold04] Goldberg, A., Harrelson, C., 2004, Computing the shortest path: A* search meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research.

[Hage00]Hagerup, T., 2000, Improved shortest paths on the word RAM. In: Proc. 27th Int'l Colloq. on Automata, Languages, and Program-ming(ICALP). LNCS vol. 1853, pp.61-72.

[Hitc68] Hitchner, L. E. 1968. A comparative investigation of the computational effciency of shortest path algorithms. Tech. Rep. ORC 68-17, University of California at Berkeley.

[Hopc83] Hopcroft J E. Data structures and algorithms. Addison-Weely, 1983, 427pp.

[Krus56] Kruskal, J. B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Am. Math. Soc. 7, pp.48-50.

[Meye01]Meyer, U., 2001, Single-source shortest-paths on arbitrary directed graphs in linear average-case time, Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, pp.797-806.

[Pett05a] Pettie, S., Ramachandran, V., 2005, A shortest path algorithm for real-weighted undirected graphs. SIAM J. Comput.34(6), pp.1398-1431.

[Pett05b] Pettie, S., 2005, Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time, In Proceedings 16th Int'l Symposium on Algorithms and Computation (ISAAC), pp.964-973.

[Prue09] Pruehs, N., 2009, Implementation of Thorup's Linear Time Algorithm for Undirected Single-Source Shortest Paths with Positive Integer Weights.

[Rama96] Raman, R., 1996, Priority queues: small monotone, and trans-dichotomous. In Proceedings of the 4th Annual European Symposium on Algorithms.Lecture Notes on Computer Science, vol. 1136, Springer-Verlag, New York, pp.121-137.

[Saku10] Sakumoto, Y., Ohsaki, H., Imase, M., 2010, On the effectiveness of thorup's shortest path algorithm for large-scale network simulation, 2010 10th Annual International Symposium on Applications and the Internet, pp.339-342.

[Sedg03] Sedgewick, R., 2003, Algorithms in java part 5, graph algorithms (3rd edition), Addison-Wesley Professional, 528pp.

[Siva99] Sivakumar, R., Sinha, P., Bharghavan, V., 1999, CEDAR: a core-extraction distributed ad hoc routing algorithm. IEEE Journal on Selected Areas in Communications, 17, pp.1454-1465.

[Tarj75] Tarjan, R. E. 1975. Efficiency of a good but not linear set union algorithm. J. ACM 22 , 2 (Apr.), pp.215-225.

[Thor97] Thorup, M., 1997, Undirected single source shortest paths in linear time. Proceedings of the 38th Symposium on Foundations of Computer Science, pp.12-21.

[Wei13] Wei, Y. and Tanaka, S., Improvement of MX-CIF Quadtree With Downloadable Source Code and Benchmark Datasets[M]. LAP LAMBERT Academic Publishing, 2013, 99pp.

[Yen70] Yen, J. Y., 1970, A Shortest Path Algorithm, Ph.D. dissertation, University of California, Berkeley.