Connected Cluster Based Node Reduction and Traversing for the Shortest Path on Digital Map

Dhaval Kadia(✉)

The Maharaja Sayajirao University of Baroda, Vadodara 390001, Gujarat, India
dhavalkadiamsu@gmail.com

Abstract. This paper elaborates the algorithm and methods applied on a digital map and explains how the digital map is reduced into its nodal form for obtaining the shortest path along with navigation, directions and the individual distances. The algorithm uses different types of data structures appropriate for different approaches and interacts with the digital map or its nodal form. Auxiliary data structures along with the respective methods are added in the algorithm to decrease the time and space complexity. The time-memory tradeoff is observed during the implementation through the two memory allocation schemes. In the beginning, static memory allocation is used which is followed by the dynamic memory allocation. Both the schemes have their corresponding consequences. Precision in navigation is varied on the basis of peculiar requirements so that the overall complexity of the algorithm can be reduced. The algorithm first processes the digital map and identifies roads and junctions which will be the input for further processing. Cluster based approaches are applied for simplification and efficiency purposes. The paper compares the various algorithms for obtaining the shortest path. Comparison of the diverse experiment results show the improvement in the execution time and memory consumption.

Keywords: Shortest path · Algorithm · Analysis · Data structure
Cluster · Node representation · Node reduction · Optimization
Digital map · Navigation

1 Introduction

The shortest path algorithm is conceptualized on an appropriate data structure. Digital map, an image having numeric values, represents roads. The road is considered to be present where the connected pixels have 1 s. Digital map is a network that is taken as an input for finding junctions within it. Program memory is allocated on static or dynamic basis and as per starting and destination points, the algorithm is then executed, data structures are manipulated and decisions are taken. Initially, the paper explains the basic algorithm and its interaction with the data structure. Then, it describes how optimization is applied on the data structure as well as on the algorithm. Subsequently, it demonstrates how auxiliary data structures are introduced. These three phenomena are elaborated in their respective sections. The entire algorithm, along with its further

R. Sharma et al. (Eds.): ICAN 2017, CCIS 805, pp. 83–96, 2018.
https://doi.org/10.1007/978-981-13-0755-3_7

optimization, is implemented in Language C. Sufficient care is taken on memory deallocation by applying the methodologies present in [1, 4].

2 Initialization

Throughout the paper, the two-dimensional array *(MAP)* refers to the digital map. Different colors in figures are associated with their values of pixels and behaviors. L_s and L_d symbolizes starting location and destination location respectively. They are represented in terms of the Cartesian coordinate system i.e. (x_{start}, y_{start}) and ($x_{destination}$, $y_{destination}$). Digital map represents the interconnected paths from which the algorithm gets reference of available paths and locations from where there may be more than one way to go further i.e. a junction. Initially, a digital map is a binary-valued *MAP*.

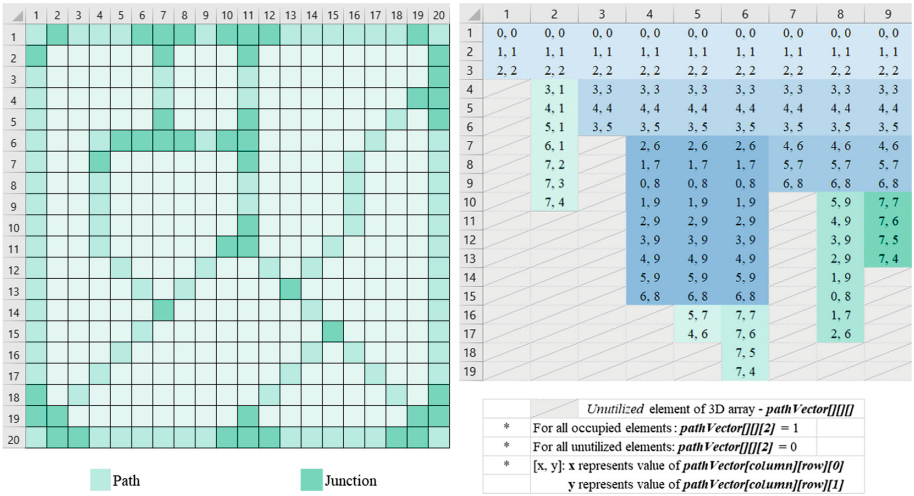

Fig. 1. Junction identification in digital map; 2D representation of 3D array *pathVector[][][]* having values assigned for (x_{start}, y_{start}) = (1, 1) and ($x_{destination}$, $y_{destination}$) = (8, 5). (Color figure online) (Figure is generated by retrieving values from data structures residing in the memory while execution).

Figure 1 shows pixels representing paths and junctions having values 1 and 2 respectively. Procedure for junction identification is now explained. $X - axis$ is vertical and $Y - axis$ is horizontal. That is, row and column numbers represent values on $X - axis$ and $Y - axis$ respectively. We as humans can identify junctions by looking directly to the digital map but for the algorithm to proceed further, it is necessary to determine locations where there are junctions. The function which is designed to find junctions, is now explained. As explained earlier, all coordinates having path will have value 1. The junction is a coordinate where the present path splits into two or more paths. So, current coordinate will be a junction if it is connected with more than two coordinates. It will have more than two adjacent pixels having value 1. The function *identifyJunction()*

performs the required task as it periodically checks all the coordinates and its adjacent coordinates. As a result, if it has more than two adjacent coordinates having values 1 or 2, then it will be assigned value 2. Thus, the algorithm will make its decision whether to split or not during the execution.

3 Static Memory Based Algorithm (SMA)

Static Memory based Algorithm *(SMA)* runs on the allocated memory prior to runtime. All variables are of type *Integer*. It finds all the possible paths and saves all the coordinates sequentially in 3D data structure while execution. These procedures are described in the upcoming subsections in detail. Working of *SMA* is elaborated using the basic example. Figure 2 represents an example to demonstrate the outcome of function *identifyJunction(),* through which *SMA* will execute. Input parameters to the algorithm are L_s and L_d. Algorithm performs functions shown in Fig. 3 on each path and junctions and assigns values demonstrated in Fig. 1 into a data structure. At the end, it analyzes them and finalize the shortest path shown in Fig. 2.

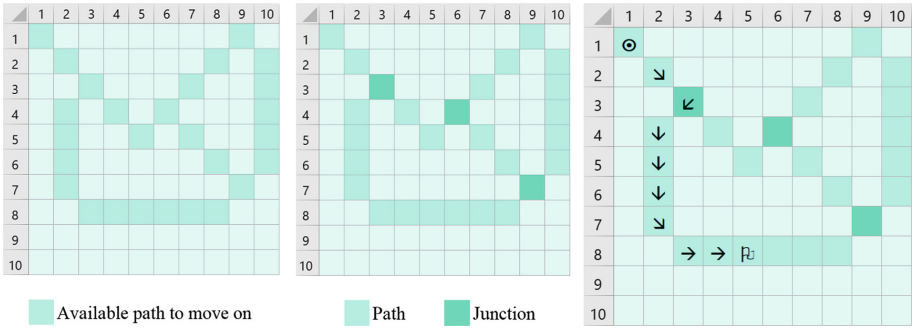

Fig. 2. Digital map to elaborate the *SMA* approach; Output of the shortest path between L_s = (1, 1) to D_s = (8, 5); Generated data structure: *pathVector[2][][]* in Fig. 1.

3.1 Data Structure

Data structure *pathVector[][][]* is a 3D array that stores all the possible paths from L_s to L_d. Its significance is discussed now. *pathVector[index][state][]*: *index* and *state* as column and row respectively in Fig. 1, *pathVector[][][0]* = x coordinate of any location, *pathVector[][][1]* = y coordinate of any location, *pathVector[][][2]* = 1 if (x, y) is a path else 0. Initially, L_s is inserted into *pathVector[1][1][]* i.e. x_{start}, y_{start} into *pathVector[1][1][0]* and *pathVector[1][1][1]* respectively.[1] Data structure *found[]* is an integer array to store all the indexes of *pathVector[index][][]* in which the last *state*

[1] Index 0 is shifted into index 1 only for first two dimensions so that representation can be user friendly i.e. *pathVector[0][][]* and *pathVector[][0][]* are shown as *pathVector[1][][]* and *pathVector[][1][]* only in figures. Index starts from 0 in actual practical implementation in Language C.

has L_d. That is $pathVector[index][state_{last}][0] = x_{destination}$ and $pathVector[index][state_{last}][1] = y_{destination}$. Till $state = state_{last}$, $pathVector[index][state_{last}][2]$ is 1. For next state = $state_{last}$+ 1, $pathVector[index][state_{last}+1][2]$ is 0, indicating that the ongoing path is now terminated. The *foundIndex* is a variable pointing to the current index of array *found[]* where the upcoming index value will be stored and the terminating location is L_d. This data structure is modified further for faster execution.

3.2 Algorithm Description

The algorithm starts its execution from L_s and terminates at L_d. Since L_s is inserted into *pathVector[][][]* as stated earlier, it is the location from where the algorithm explores all the possible paths through its adjacent locations towards L_d. This procedure is repeated continuously. Recursion is responsible for finding the shortest path automatically but for this, current paths must be saved so that they can be compared with upcoming paths and hence, the shortest distance can be computed. Algorithm saves coordinates of the path into *pathVector[][][]* on which it is currently moving. Now, for each next adjacent location, there are two possibilities i.e. whether it will be a path or a junction. If it is a path then the algorithm simply keeps moving on it, which indicates that there is only one adjacent location (except the previous location) that is available for the next step. If it is a junction then the traversed path till that junction will be saved following which all adjacent locations will be traversed. This procedure will be executed in a loop for all the adjacent locations (except the previous location), where the path till the current junction is accessible in the scope. In this case, for each adjacent location, a new index of *pathVector[][][]* will be generated for each new path traversed through that adjacent location. The next free index of *pathVector[][][]* is saved in a variable called *pathIndex*. After having junction and the new adjacent location, *pathIndex* will be incremented. Execution of the algorithm is elaborated with parts of the code.

3.3 Implementation

Functions *trace()*, *split()*, *getEndpoint()* and *replicateIndex()* are explained below using the definition and the body of each. The code is having pointers of Language C. Pointers are used so that they can be parameterized and passed into the function representing the address of the particular 2D array.

```
void trace(int x, int y, int **p, int vect)
{
    if (x, y) is Destination location
        found[foundIndex] = vect
        foundIndex++
    else
        if (x, y) is Junction
            split(x, y, p, vect)
        else if (x, y) is Path
            for each adjacent location (m, n) of (x, y)
                if getEndpoint(m, n, p) is false
                    if MAP[m][n] is Path
                        insert (m, n) at p[branchEnd][]
                        trace(m, n, p, vect)
                    else if MAP[m][n] is Junction
                        trace(m, n, p, vect)
}
```

```
void split(int x, int y, int **p, int vect)
{
    insert (x, Y) at p[branchEnd][]
    for each adjacent location (m, n) of (x, y) do
        if(MAP[m][n]> 0 && !getEndpoint(m, n, p))
            newIndex = replicateIndex(vect)
            insert (m, n) at pathVector[newIndex][branchEnd][]
            trace(m, n, pathVector[newIndex], newIndex)
}
```

Fig. 3. Definitions and bodies of functions *trace()* and *split()*. (End of *if* and *for* is not shown in figures because their scopes are indented by their relative positions).

Function Trace(). Algorithm executes the function shown in Fig. 3 each time it moves on to the next adjacent location. If the current location is L_d then, the currently used index of *pathVector[index][][]* will be saved as *found[foundIndex] = index* where *foundIndex* will be incremented. Thus, the list of paths having L_d as a terminating location can be acquired from *found[]* for further processing. If the current location is a junction then it will call the function *split()*. Else, if it is a path then it will find the next adjacent location. If the adjacent location is a path then it will insert that adjacent location into the end location of the current branch i.e. at the location pointed by *branchEnd*, which gives the array index that is not filled by the location value earlier and comes next to that array index having a location value. Here, ***p* is a pointer of a 2D array to the *pathVector[][][]* which comes from parameters of the function. Function *trace()* is called by inserting that adjacent location as a parameter, ***p* being same as that of the parameters. Parameter *vect* points the *pathIndex* of ***p*. If an adjacent location is a junction then the function *trace()* will be called without any insertion and control will be transferred to the function *split()*.

Function Split(). The function shown in Fig. 3 is applied when the current location is a junction. Since it is a junction, there will be more than one possibility where the algorithm will spread itself. Those new possible paths will have paths from start till the current location in common. After the initialization of these common locations, their independent path locations will be filled up in arrays pointed by their respective indexes. Here, the function *replicateIndex(int)* returns an *Integer* value of the array index that is available for replicating common location coordinates. Current *pathIndex* is pointed by *vect* and the common locations are to be copied from the index *vect* into the newly generated index *newIndex* for each new possible path passing from an adjacent location. The function *getEndpoint(int, int, int**)* returns the *Integer* value 1 if the locations represented by first two arguments are present in the array and 0 if they are not. It changes the global variable *branchEnd* to the index where the current location coordinate will be inserted and the function *trace()* will be executed on it along with that *newIndex* pointing it for being further copied if any junction comes.

4 Dynamic Memory Based Algorithm (DMA)

4.1 Solved Issues of SMA

Dynamic Memory based Algorithm *(DMA)* uses the *Linked List* because it is a memory efficient data structure. *SMA* was based on the static memory that was allocated before the actual execution of the algorithm. *DMA* shows a significant improvement in the space complexity. In the data structure *pathVector[][][]* shown in Fig. 1, there are unutilized memory blocks which remains unutilized throughout the execution. This issue of wastage of useful memory under *SMA* needs to be solved. *DMA* allocates only necessary chunks of memory while execution for computation. As a specific part of the execution ends, it deallocates the memory used during such execution so that the same memory can be allocated again. The vital benefit of *DMA* is that, the memory parts of all the *pathVector[][][2]* which were used to show whether the location coordinates are saved or not, making a distinction, is not needed in *DMA* because *Linked List* will

allocate and append only the necessary memory chunks and when *NULL* comes, it will be the end of the *Linked List*.

4.2 Data Structure, Converted Methods and Results

The entire data structure in *DMA* is dynamically allocated, there is no static allocation. The *Structure* of Language C is used, variables of the *Structure* are passed by reference into the function so that the function can manipulate them. In Fig. 4, *Structure panel* has a link with the *Structure gateway* that contains coordinates and its own pointer, which will point the next structure representing the next location. Here, the map size is variable and the algorithm is executed on different patterns of the map. Coordinates of the map starts from 0 which can be calibrated to 1 as well (Fig. 5).

```
void trace(short int x, short int y, struct panel *p)
{
    if (x, y) is DestinationIocation
        calculate currentPathLength
        if minimum > currentPathLength
            minimum = currentPathLength
            copyPanel(p, shortestPanel)
    else
        if (x, y) is Junction
            split(x, y, p);
        else if (x, y) is Path
            for each adjacent location (m, n) of (x, y)
                if getEndpoint(m, n, p) is false
                    if MAP[m][n] is Path
                        bottom->down=(struct gateway*)malloc(sizeof(struct gateway*))
                        bottom = bottom->down
                        bottom->X = m
                        bottom->Y = n
                        bottom->down=NULL
                        trace(m, n, p)
                    else if MAP[m][n] is Junction
                        trace(m, n, p)
}
```

```
void split(short int x, short int y, struct panel *p)
{
    if getEndpoint(x, y, p) is false
        bottom->down=(struct gateway*)malloc(sizeof(struct gateway*))
        bottom=bottom->down
        bottom->X = x
        bottom->Y = y
        bottom->down = NULL

    for each adjacent location (m, n) of (x, y) do
        if MAP[m][n] > 0 and getEndpoint(m, n, p) is false
            struct panel *newIndex = (struct panel*)malloc(sizeof(struct panel))
            newIndex = replicateIndex(p)
            if getEndpoint(m,n,newIndex) is false
                bottom->down = (struct gateway*)malloc(sizeof(struct gateway*))
                bottom = bottom->down
                bottom->X = m
                bottom->Y = n
                bottom->down = NULL
            trace(m,n,newIndex)
            freePanel(newIndex)
}
```

```
void freeGateway(struct gateway *gat)
{
    if gat is NULL
        return
    else
        freeGateway(gat->down)
        free(gat)
}
```

```
void freePanel(struct panel *vect)
{
    freeGateway(vect->gate)
    free(vect)
}
```

```
struct gateway{
    short int X;
    short int Y;
    struct gateway *down;
}*gatewayVariable;
```

```
struct panel{
    struct gateway *gate;
}*panelVariable;
```

Fig. 4. *Structure gateway* and *panel* (implemented in Language C); modified functions *trace()* and *split()*; *freePanel()* and *freeGateway()* are functions for the memory deallocation.

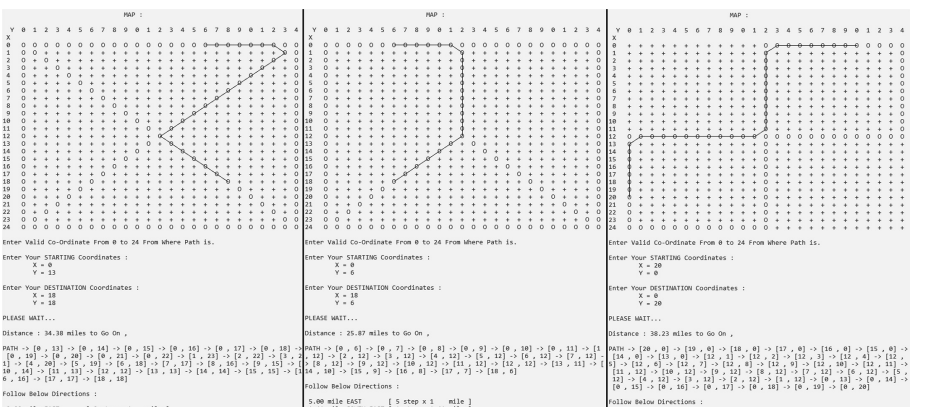

Fig. 5. The shortest path with navigation is generated in (Windows 10) terminal for different patterns of the map. In the map, 'O' indicates presence of the path on that coordinate and '+' represents an empty area.

5 Cluster Based Node Reduction Algorithm (CNRA)

5.1 Solved Issues of SMA and DMA

This approach gives a major breakthrough to *DMA* in dealing with much more complex maps particularly when there are cluster of coordinates close-by, representing the same nearby location. *DMA* is unessentially accurate which is not required in the real-world. By referring [2, 3], multiple interconnected junctions can be represented by one junction representing all of them. Hence, there is only one *split()* instead of multiple splits which implies a reduction in the time and space complexity.

5.2 Algorithm with Auxiliary Functions

Cluster based Node Reduction Algorithm *(CNRA)* creates cluster of multiple junctions connected to each other representing at least 3 locations which are connected to their respective paths. Additional data structures in terms of 3D and 4D arrays are kept as arrays because they are accessed many times while execution. *CNRA* has both static and dynamic allocation schemes. Before the actual execution, the function *analyzeLayer()* is called for each coordinate of the *MAP*. It calls other functions *traceLayer()*, *exactCenter ()*, *assignCenter()* & *allocateBranch()* once for each cluster sequentially. For each cluster, *traceLayer()* starts from one of its junctions and spreads to each of the junction to calculate how many junctions are residing in the cluster. It sums up each coordinate of the junctions as a weight to find out the center of the cluster. *(centerX, centerY)* is assumed to be the center of the cluster which is further confirmed for being the proper center and if the calculated center is not on the junction of the cluster, then *exactCenter()* assigns it its nearby location i.e. a junction (Fig. 6).

```
void traceLayer(int x, int y, int &sumX, int &sumY, int &count)
{
    mark MAP[x][y]
    sumX = sumX + x + 1
    sumY = sumY + y + 1
    increament count

    for each adjacent location (m, n) of (x, y) do
        if MAP[m][n] is unmarked junction
            traceLayer(m, n, sumX, sumY, count)
}
void assignCenter(int x, int y, int centerX, int centerY, int count)
{
    mark MAP[x][y]
    centerMap[x][y][0] = centerX
    centerMap[x][y][1] = centerY
    centerMap[x][y][2] = count

    for each adjacent location (m, n) of (x, y) do
        if MAP[m][n] is unmarked junction
            assignCenter(m, n, centerX, centerY, count)
}
void getBranch(int x, int y, int &branchX, int &branchY)
{
    for each adjacent location (m, n) of (x, y) do
        if MAP[m][n] is path
            branchX = m
            branchY = n
            return
}
void analyzeMap()
{
    for each coordinate of MAP[i][j] do
        if MAP[i][j] is 2
                analyzeLayer(i, j)
}
```

```
void allocateBranch(int x, int y, int centerX, int centerY, int count)
{
    mark MAP[x][y]
    branchX = branchY = -1

    getBranch(x, y, branchX, branchY)

    linkMap[centerX][centerY][count][0] = branchX
    linkMap[centerX][centerY][count][1] = branchY
    increament count

    for each adjacent location (m, n) of (x, y) do
        if MAP[m][n] is unmarked junction
            allocateBranch(m, n, centerX, centerY, count)
}
void analyzeLayer(int x, int y)
{
    sumX = sumY = count = 0
    traceLayer(x, y, sumX, sumY, count)

    centerX = accuCX = (float)sumX / (float)count - 1
    centerY = accuCY = (float)sumY / (float)count - 1

    if accuCX - (int)accuCX > 0.5
        increament centerX
    if accuCY - (int)accuCY > 0.5
        increament centerY
    if MAP[centerX][centerY] is not junction
        nearestX = accuCX, nearestY = accuCY, leastD = infinity
        exactCenter(x, y, nearestX, nearestY, centerX, centerY, leastD)

    assignCenter(x, y, centerX, centerY, count)
    unmark all marked junction
    allocateBranch(x, y, centerX, centerY, 0)
    unmark all marked junction
}
```

Fig. 6. Definitions and bodies of *CNRA* auxiliary functions.

The function *assignCenter()* traverses through all the junctions, assigns the center coordinate value and the number of junctions to the arrays, which are represented by the junctions of the cluster in *centerMap[][][][]*. Whenever the algorithm approaches any of the junction, it will move to the center for the junction and then it will spread towards each path connected to that cluster. The center of the cluster is associated with the paths connected by the junctions. Function *allocateBranch()* links those paths in 2D array represented by the location of the center for junction on 4D array *linkMap[][][][]*. Function *getBranch()* gives coordinates of path connected with a junction for assigning with *allocateBranch()*.

5.3 Proposed Data Structure

Data structures *centerMap[][][]* and *linkMap[][][][]* are added to *DMA* with their auxiliary functions. The *centerMap[][][]* is a 3D array whereas *linkMap[][][][]* is a 4D array. If (x, y) is a junction and the remaining junctions are connected to it then, for (a, b) representing junctions of the cluster, *centerMap[a][b][3]* will be allocated. Center of cluster (C_x, C_y) is inserted into the *centerMap[a][b][0]* and *centerMap[a][b][1]* respectively, and the number of junctions per cluster is inserted into the *centerMap[a][b][2]*. For *linkMap[][][][]*, the number of junctions $Junction_{total}$ per cluster is calculated and (C_x, C_y) on $linkMap[C_x][C_y][Junction_{total}][2]$ memory is allocated during the runtime. Coordinates of the junctions will be saved in it, if any junction is surrounded by junctions then (–1, –1) will be inserted. In Fig. 7, *MAP[11][11]* represents map with paths and junctions. There are 5 clusters, four of them have four junctions and one has 5 junctions; e.g. cluster of junctions J = {(4, 0), (5, 0), (6, 0), (5, 1)} have (C_x, C_y) = (5, 0). Now, $\forall$ (a, b) $\in$ J, *centerMap[a][b][0]* = 5, *centerMap[a][b][1]* = 0, *centerMap[a][b][2]* = 4 because there are four junctions in the cluster. Those junctions are connected with (3, 0), *NULL*, (7, 0), (5, 2) respectively. These coordinates represents the paths (branches) connected to the cluster and assigned to the $linkMap[C_x= 5][C_y= 0][i][2]$ one by one. For example, path (7, 0) is saved in *linkMap[5][0][0][0]* and *linkMap[5][0][0][1]* respectively. When algorithm executes *split()* from any junction, it will investigate $linkMap[C_x][C_y][][]$ and sequentially spreads through each paths i.e. if control is on coordinate (6, 0) then it will spread through each coordinate $(linkMap[C_x][C_y][i][0],\ linkMap[C_x][C_y][i][1])$ where $0 \leq i \leq 3$. If it finds (–1, –1) then it will not proceed due to absence of the path.

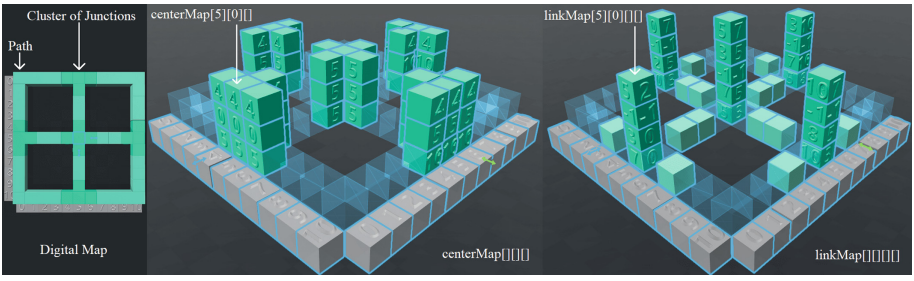

Fig. 7. 3D representation of *centerMap[][][]* and *linkMap[][][][]* (using Microsoft 3D Builder).

6 Connected Cluster Based Traversing Algorithm (CCTA)

6.1 Traversal Overhead in CNRA

CNRA is based on digital map traversal, where the algorithm finds the path towards the adjacent location. If there is a path instead of a junction then the algorithm will simply traverse through all the continuous locations representing the same path. On every traversal of each path from their adjacent junctions, many locations are traversed invariably and this is the overhead. The algorithm should therefore, jump onto one junction from another directly without traversing intermediate nodes.

6.2 Connected Clusters Approach and Its Significance

Connecting clusters over their path removes overhead produced by traversing the entire path. If each cluster knows its adjacent clusters connected to it then, each cluster will be identified as a node and will be traversed directly to its adjacent clusters. Functionalities of *DMA* and *CNRA* are preserved in Connected Cluster based Traversing Algorithm *(CCTA)*, only intermediate approach is modified. For applying this approach, the data structure of *CCTA* is modified and the required functions are added. Here, data structure *linkMap* is extended to its 4^{th} dimension to length 5 i.e. *linkMap[][][][5]*. First two values represent branch coordinates, 3^{rd} and 4^{th} values represent coordinates of adjacent cluster connected to those branch coordinates and 5^{th} value represents the length of the path between the current cluster and the adjacent cluster. These values are calculated before the actual execution of *CCTA*.

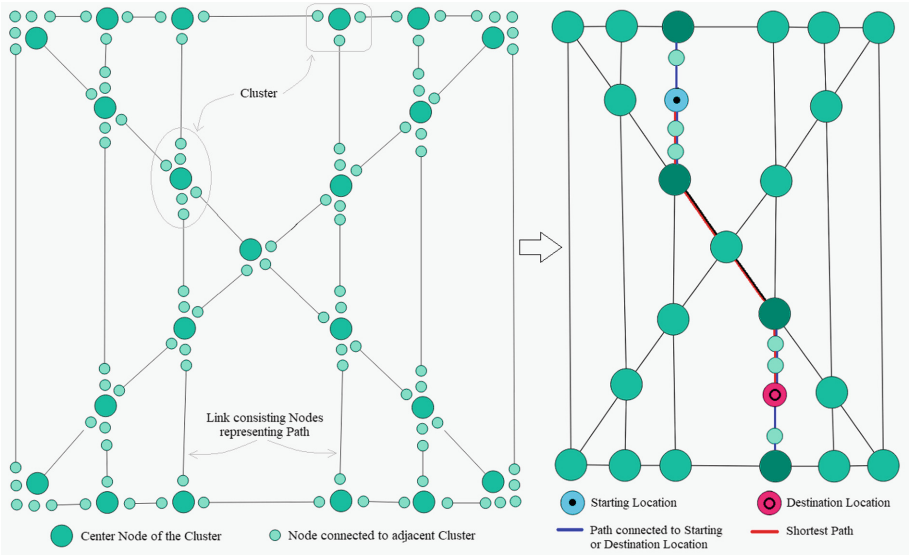

Fig. 8. Transformation of a digital map (Fig. 9) into the cluster based map and further conversion into the connected cluster (node) based network. (Color figure online)

The shortest path calculated by *CCTA* is illustrated in Fig. 9 on 50 × 50 digital map. The shortest path from (10, 16) to (40, 34) is generated and drawn in Fig. 9. *CCTA* traces one by one location when it is on the paths consisting of a starting location and a destination location i.e. paths (2, 6) to (13, 6) and (37, 34) to (47, 34) because there will be no option of the jump. Initially, starting location acts like a junction (if it is on the path) to traverse towards two ways. Once it reaches a cluster and if the destination location is not on any of the paths connected to that cluster, then it will jump to an adjacent node without traversing nodes of intermediate paths, else it will traverse all nodes of the only path that leads to the destination location. Cluster represented by dark-colored nodes are connected to either starting or destination location in Fig. 8, remaining clusters are connected to each other so that the traversal can jump directly while execution. Though *SMA* can be integrated with *CCTA*, the methodology for saving traversed locations is the same as that of the *CNRA* i.e. using *Linked List* so that, if in case the map is much complex, the segmentation fault does not occur. The map is reduced for faster execution and the original data is used.

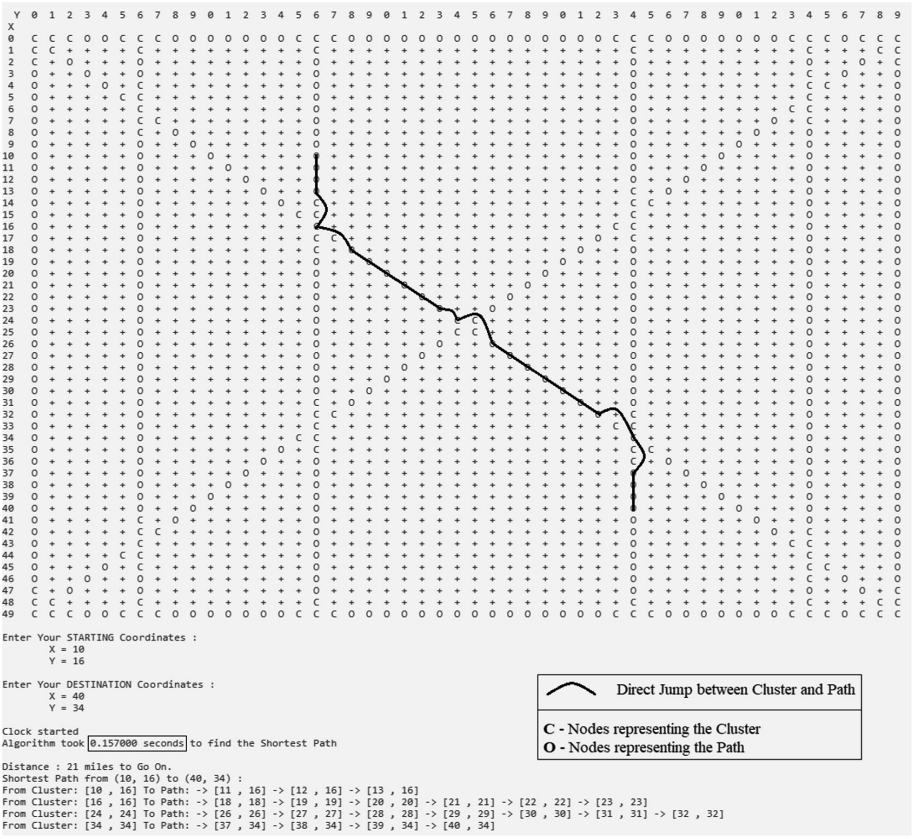

Fig. 9. The shortest path with *CCTA* on the comparatively complex map. Its connected cluster based node representation is shown in Fig. 8.

7 Comparison Among Approaches

An overview of the execution time and memory consumption are illustrated in Figs. 10, 11 and 12. Memory consumption of *SMA* is 19 GB.

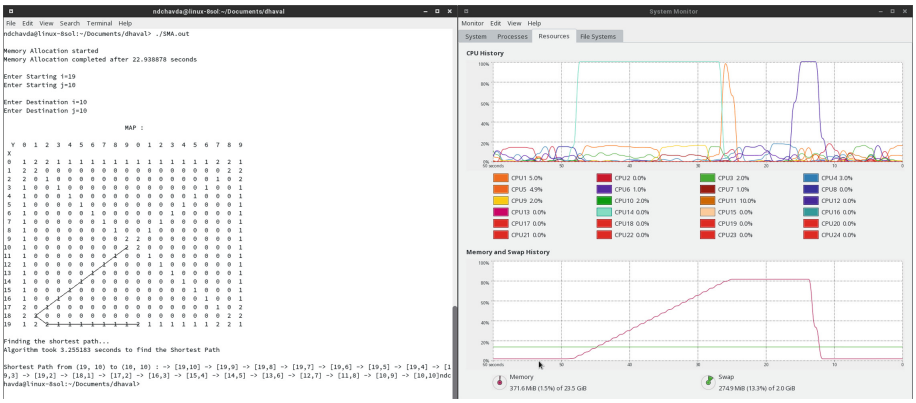

Fig. 10. *SMA* execution (Workstation, OpenSUSE, Intel Xeon, 24 Core CPU, 24 GB RAM).

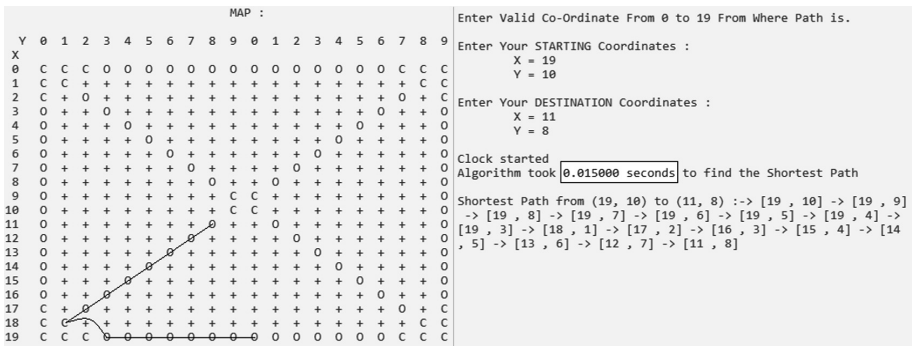

Fig. 11. *CNRA* execution time (PC, Windows 10, Intel Core i7, Quad Core CPU, 16 GB RAM).

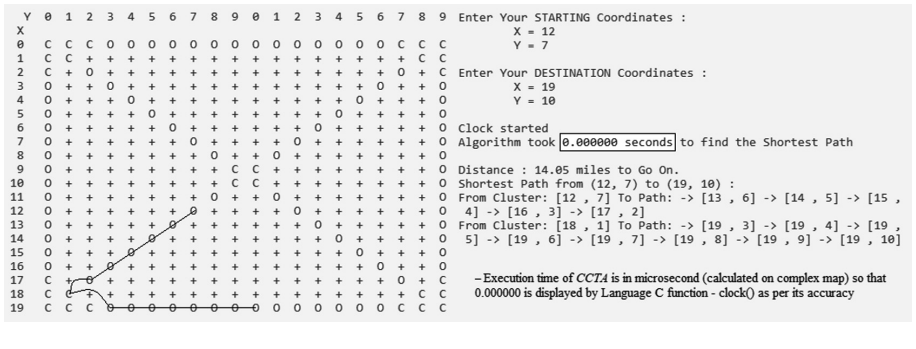

Fig. 12. *CCTA* execution time (PC, Windows 10, Intel Core i7, Quad Core CPU, 16 GB RAM).

CNRA decreases the execution time to the milliseconds, and *CCTA* further decreases it to the microseconds, memory usage being in KB in both the approaches (Figs. 13, 14 and Table 1).

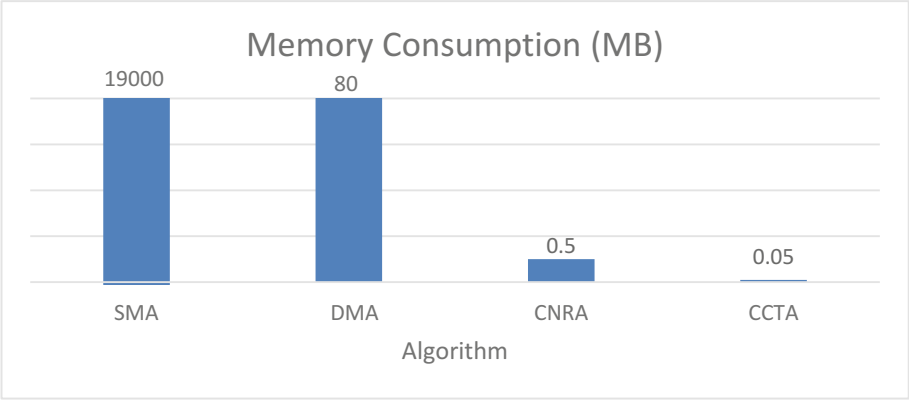

Fig. 13. Memory consumption for above approaches.

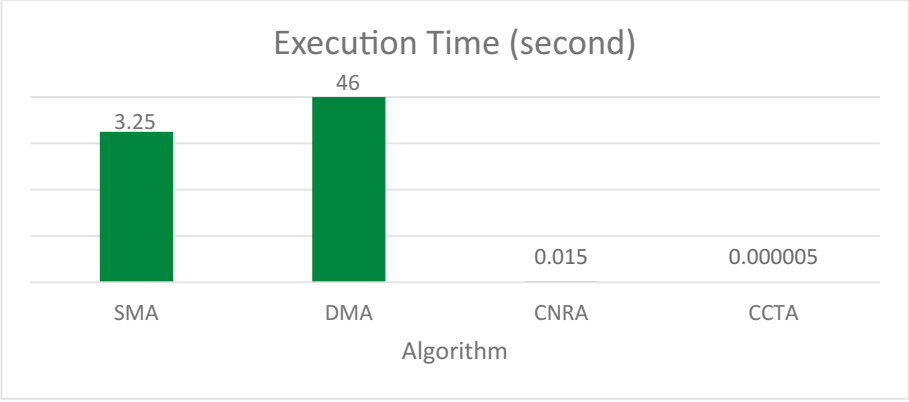

Fig. 14. Execution time for above approaches.

Table 1. Performance comparison among approaches.

	Execution time	Memory usage	Machine
SMA	3.25 s	19 GB	Workstation, Intel Xeon, 24 Core CPU, 24 GB RAM
DMA	46 s	80 MB	PC, Intel Core i7, Quad Core CPU, 16 GB RAM, Clock speed 3.4 GHz
CNRA	15 ms	500 KB	Same as *DMA*
CCTA	5 µs	50 KB	Same as *CNRA*

8 Conclusion

Thus, by the aforementioned analysis and the implementation of the algorithms, it is observed that *SMA* requires plenty of memory because of its static allocation. *DMA* on the contrary requires less memory but needs more execution time because it is entirely based on dynamic memory allocation – algorithm allocates and deallocates memory for further use which takes the CPU time (clock cycles). *CNRA* solves problems of both *SMA* and *DMA* by implementing auxiliary methods which simplify the execution. Clustering of junctions decreases the number of effective nodes and extends feasibility for much complex maps. *CCTA* bypasses the traversal of the path directly to the cluster so that it removes unnecessary intermediate traversal overhead. In doing so, it gives the same result as *CNRA*. In addition, it improves the execution time and memory consumption.

This paper illustrates how the digital map is represented and processed in its nodal form. *CCTA* can be elaborated like this – "Transform the given data into its compact form, process its compact form, generate an intermediate result and map it to the original data for an accurate result" i.e. transform the digital map into its cluster based node representation, process and map it to the digital map for obtaining the real-world shortest path.

These approaches can further be extended by compressing the nodal map of *CCTA* by some layers for simplifying it. Each layer can be mapped in a sequence. After having the shortest path by processing the compact map of the bottommost layer, its mapping with an above layered map can be used for obtaining an equivalent shortest path which is much apparent. Repeating the same procedure till the digital map i.e. till the topmost layer will give the real-world shortest path. Moreover, if the algorithm is modified to make it follow the direction towards the destination, it will eliminate the false positive directions further. In case, the algorithm is unable to find the destination, other possibilities of directions can be listed into the priority queue that depends upon the gradient of the current location to the destination location. The direction based approach is fast and intelligent while the earlier recursive approach is accurate, hence they exhibits the time-accuracy tradeoff.

References

1. Giorgio Gallo, F., Stefano Pallottino, S.: Shortest path algorithms. Ann. Oper. Res. **13**(1), 1–79 (1988). https://doi.org/10.1007/BF02288320
2. Krishna, P., Vaidya, N.H., Chatterjee, M., Pradhan, D.K.: A cluster-based approach for routing in dynamic networks. ACM SIGCOMM Comput. Commun. Rev. **27**(2), 49–64 (1997). https://doi.org/10.1145/263876.263885
3. Zhang, D., Yang, D., Wang, Y., Tan, K. L., Cao, J., Shen, H.T.: Distributed shortest path query processing on dynamic road networks. VLDB J. **26**(3), 399–419 (2017). https://doi.org/10.1007/s00778-017-0457-6
4. Kadia, D.: Binary search optimization: implementation and amortized analysis for splitting the binary tree. Int. J. Comput. Sci. Inf. Secur. **15**(3), 338–341 (2017)