Maze Router: Collected Techniques

Nashat Salih Abdulkarim Alsandi

Submitted to the
Institute of Graduate Studies and Research
in Partial Fulfillment of the Requirements for the Degree of

Master of Science
in
Computer Engineering

Eastern Mediterranean University
February 2015
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Serhan Çiftçioğlu
Acting Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Işık Aybay
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Asst. Prof. Dr. Ahmet Ünveren
Supervisor

Examining Committee

1. Asst. Prof. Dr. Adnan Acan ________________________________
2. Asst. Prof. Dr. Yiltan Bitirim ________________________________
3. Asst. Prof. Dr. Ahmet Ünveren ________________________________

ABSTRACT

In this thesis, Maze Router Problem (MRP) solved by using Connected Component Labeling, Depth First Search, Lee and A* Algorithms. The main goal of this research is to find a complete set of path which directs an agent to move from the Source node in the Maze towards a Target node in a Single-layer routing environment.

In experiments different sized Maze Router Problem (MRP) instances are solved by using different Algorithms. From the results obtained we can conclude that Lee and A* Algorithms finds the shortest path for all the problem instances. It can also be concluded that A* Algorithm is the fastest Algorithm that finds the shortest path.

Keywords: Maze**,** Maze Router, Maze Router Problem, Connected Component Labeling Algorithm, Depth First Search Algorithm, Lee Algorithm, A* Algorithm.

ÖZ

Bu tezde, Labirent Yönlendirici Problemi (LYP) Bağlı Bileşen Etiketleme, Derin Öncelikli Arama, Lee ve A^* Algoritmaları kullanılarak çözülmüştür. Bu araştırmanın temel amacı, tek katmanlı Labirent yönlendirme ortamında başlangıç düğümünden hedef düğümüne ulaşılabilecek tam bir yolun bulunmasıdır.

Farklı boyutlarda bulunan Labirent Yönlendirme problemlemi örnekleri farklı algoritmalar kullanılarak çözüldü. Elde edilen sonuçlara göre Lee ve A^* algoritmaları kullanılan örnekler için en kısa youlu veren algoritmalar olmuşlardır. Aynı zamanda A^* algoritmasının en kısa youlu en hızlı zamanda bulduğu tesbit edilmiştir.

Anahtar Kelimeler: Labirent, Labirent Yönlendirme, Labirent Yönlendirme Problemi, Bağlı Bileşen Etiketleme Algoritması, Derin Öncelikli Arama Algoritması, Lee Algoritması, A* Algoritması.

ACKNOWLEDGEMENT

I would like to thank my supervisor Asst. Prof. Dr. Ahmet Ünveren for his excellent supervision, good advices and proof reading of my thesis. His instruction and guidance made it possible to transform this project from a sketchy concept into a presentable thesis. And for his valuable explanations, advices, suggestions and time he spent helping me with this work.

Finally, I would like to thank my parents and my wife Berivan Chalo for all their support and encouragement. I am very fortunate to have a family with the understanding and willingness to take care of almost everything so I could spend all my time with my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The topic of Maze is very old, the earliest Mazes that known were of the Egyptian Labyrinth [1]. But many people consider the labyrinth is synonymous with Maze. Yet contemporary scholars note that there's a difference between a Maze and Labyrinth. Maze is a very complicated puzzle because it has many routes and directions, but the labyrinth has a single path, non-branching route, which leads to the purpose. A Labyrinth in this sense has an unambiguous route to the center and returning, and it isn't designed to be difficult to navigate. But it developed throughout the centuries. As a global education and wisdom level increased, Mazes became very popular as a fun and entertainment tool and also as a very interesting domain from the mathematical point of view. Mazes have been applied more and more often with the reality of life, it grew up in parallel with the personal computers [2].

In this thesis, collected techniques have been used for the solution of Maze Router Problem (MRP). Connected Component Labeling (CCL) Algorithm uses connected pixel by pixel search from the top to bottom and left to right to find the best solution for the given Maze Router Problem (MRP). Every contiguous pixel that shares the same set of intensity values identifies connected pixel regions. The definition of the Connected Component Labeling relies on pixels' neighbor. The current pixel has two systems Connected-Component if use 4-neighbours, CCL is said to be 4-connected or use 8-neighbours, CCL is said to be 8-connected [3]. When it is given the data to the CCL Algorithm, the new intended outcome will be provided.

Depth First Search (DFS) Algorithm is a systematic way to find out the solution of Maze Router Problem (MRP). In the First Depth of searching Algorithm, nodes are explored if connected with previous node and wall between nodes was opened. If any wall is closed, “Backtracks" Algorithm and go to another node, and the search will be repeated [4]. The process goes until has discovered all the nodes that are reachable from the *‘S’* node to *‘T’* node. The strategy of the DFS Algorithm is to search “deeper” in the Maze whenever possible until finding the first path to be discovered [5].

Lee Algorithm is one of the possible solutions for Maze Routing Problem (MRP) and was firstly defined by the Chaster Lee in 1961 [6]. This Algorithm represents the routing Single-Layer as a grid map, where each grid map point can contain connections to adjacent grid map points and finding a way between two terminals. This Algorithm also guarantees to find the shortest path between the Source node and Target node. Starting from the Source node the adjacent grid map nodes are progressively labeled one by one according to the node “neighborhood” from the Source node until to the Target node. A good property of Lee’s Algorithm is that it guarantees to find the shortest path between two points in Maze Router Problem (MRP) if such a path does exist [7].

Finally, A* Algorithm can be defined as starts out by looking at a neighbor, to estimate the cost of each of adjacent. The neighbor with the lowest cost is chosen to be the next node. This process continues until it gets to the Target node. And heuristics Equation are used to make smarter choices when determining which direction to search in a Maze Router Problem (MRP), if the heuristic is consistent. The function finds the fitness of the A* Algorithm given in equation (1.1).

$$F = G + H \qquad \text{Eq. (1.1)}$$

Where value ***F*** consisted of the collection of values of ***G*** and ***H*** for all neighbors. ***G*** is the cost of movement from the Source node to the current node and ***H*** is an estimation of the cost from the labeled grid point to the Target. Then nodes that are closer to the Target node have lower costs than nodes that are further from the Target node, this biases the search in the direction of the Target [8].

The proposed Algorithms for solving Maze Router Problem (MRP) are Connected Component Labeling, Depth First Search, Lee and A* Algorithms. The Maze Router Problem (MRP) is a complex problem as it contains many routes. In Maze Router Problem (MRP), the agent moves from the Source node and passes within each node until to the Target node is reached. The goal of the Agent is to reach the purpose.

The arrangements of the remaining chapters are as follows: in Chapter 2, Maze Router Problem will be explained. In Chapter 3, CCL, DFS, Lee and A* Algorithms are explained in detail. In Chapter 4, the experimental results that are discovered by using the described Algorithms are shown. Finally, conclusion is discussed in Chapter 5.

Chapter 2

MAZE ROUTER PROBLEM

2.1 Maze Router Problem

Maze Router Problem (MRP) is a connection routing problem that represents the entire network as a grid map; some routes on the grid map are blocked. Maze Router Problem (MRP) is guaranteed to find the path if the connection exists. However, in practice, it is found that many Maze Routing Algorithms are slow and needs large memory requirements [26].

The Maze Router Problem (MRP) was firstly defined by Chester Lee in 1961[6]. In 1974, Frank Rubin came up with a new method that solves Maze Router Problem (MRP) in a fast way [9]. Maze Router Problem (MRP) was initially intended for routing on a Single-Layer route, even though later in this chapter we will see that it has the extension to be used on multiple layer routes. The computational complexity of Maze Router Problem (MRP) depends on the sizes and existing paths. The main goal of Maze Router Problem (MR) is to find the shortest path between *'S'* node and *'T'* node and guarantees to find a path between two terminals of the connection exists.

A sample Maze Router Problem (MRP) is given in Figure 1.1. The Maze Router Problem (MRP) has two terminal nodes such as Source node '*S'* and Target node '*T'*. And some Parts of this grid map are blocked by components. The agent expands in the form of a wave from starting node to other nodes. The first wave that reaches the Target node determines the connecting path which is the feasible path between terminals in the Maze Router Problem (MRP).

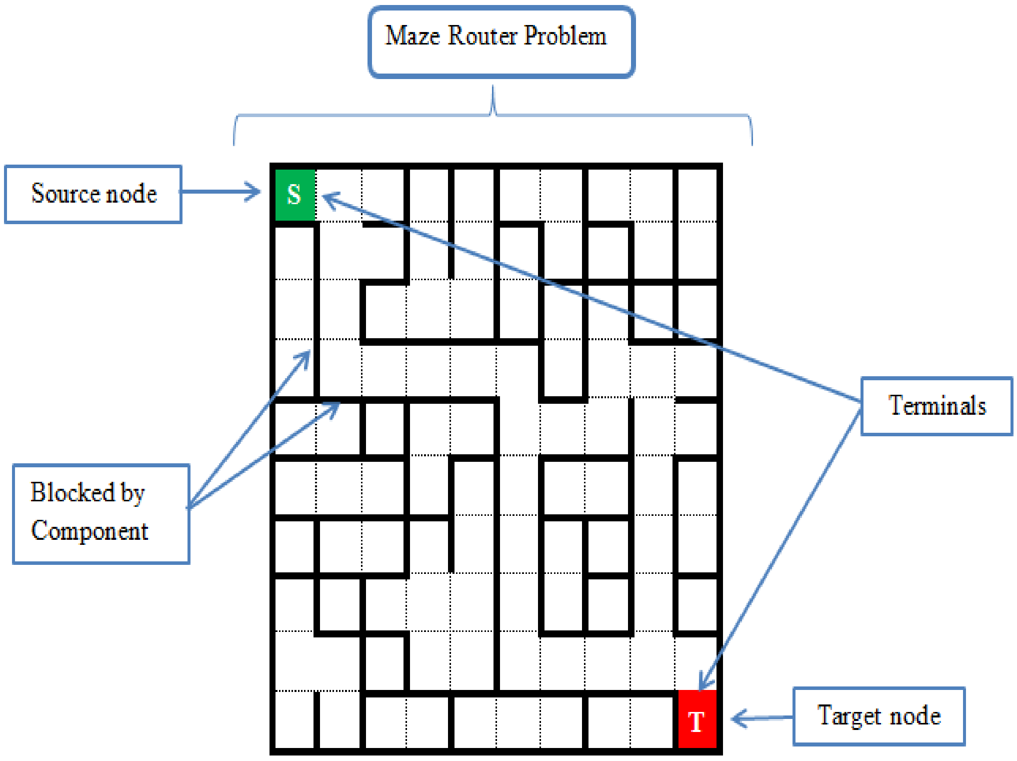


Figure 1.1: The Sample Maze Router Problem

The Maze Router (MR) Problem first searches to find the nodes that are closest to the Source node. And then proceeds in a Breadth-first search by diagnosing the nodes switching to other nodes that have been already diagnosed as it is shown in the Figure (1.2) from (B) to (C). The line is made when the Target node has been discovered, and the diagnosing is quitted. It should find out the shortest path to go, Backtrace the Source node from the intended node. And determine the path which has been already used by marking the path as shown in the Figure (1.2) from (D) to (E). It should Cleaned up all distance marks from other grid nodes accept the selected path as shown in (F) of the Figure (1.2). In short, the Maze Router Problem will always find a path if one exists, and the path found is guaranteed to be of the shortest path.

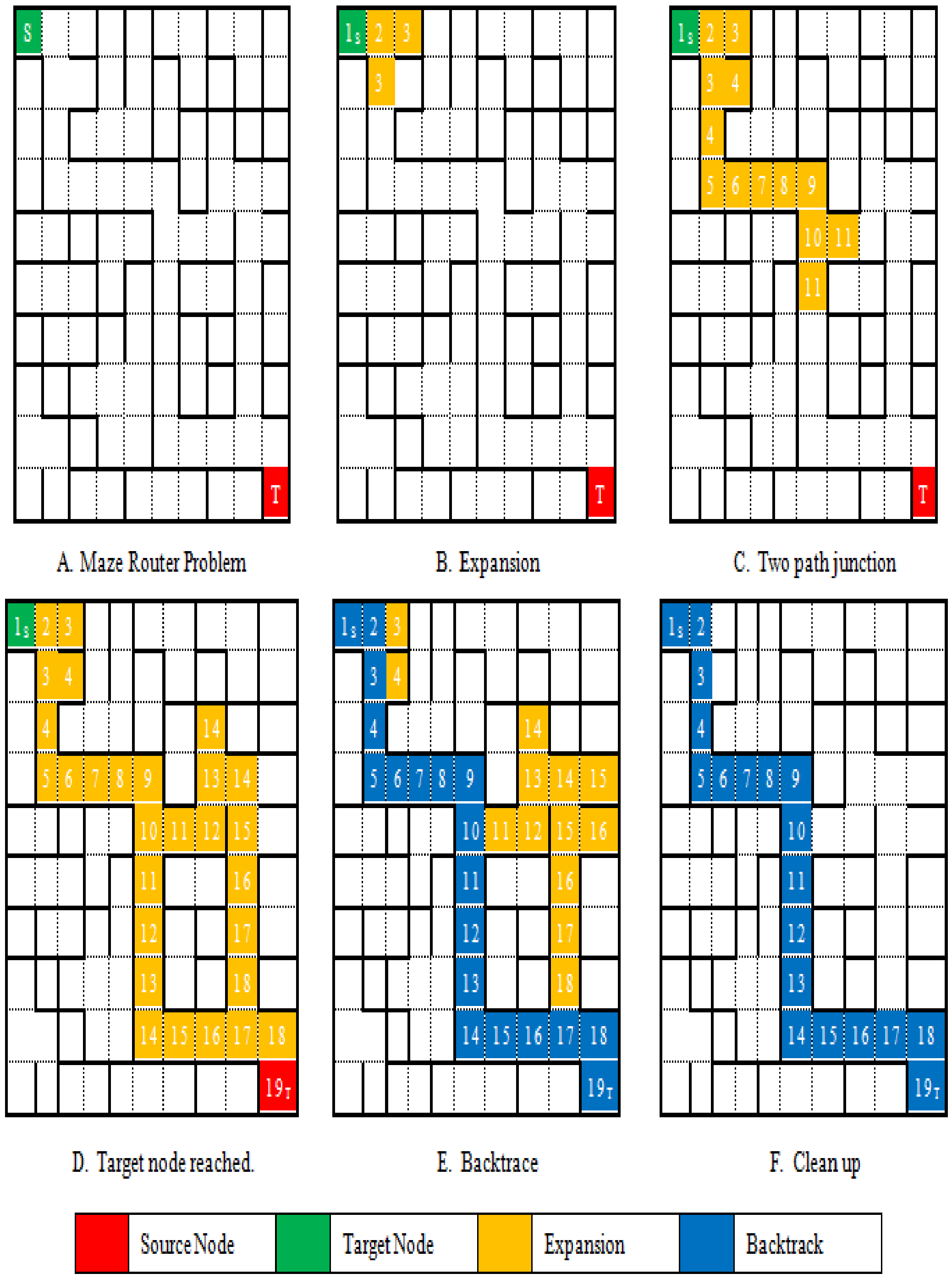
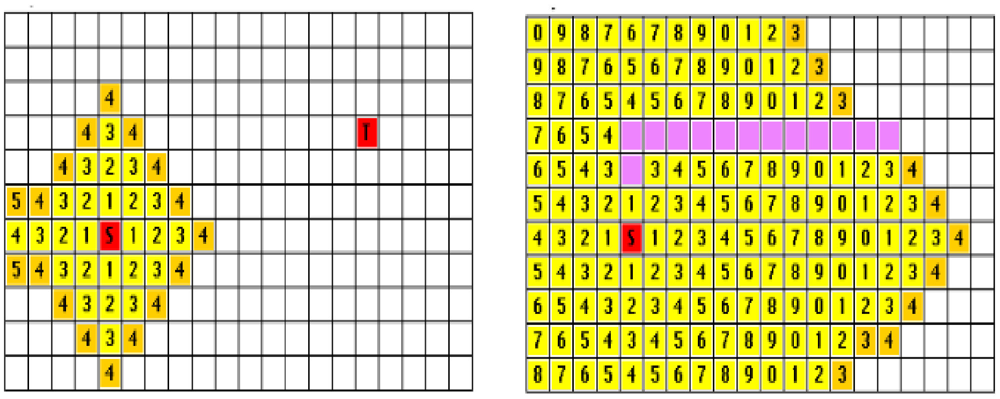


Figure 1.2: Possible Solution for Maze Router Problem by (Expansion, Backtrace and Clean up)

2.1.1 Different type of Maze Router Problems

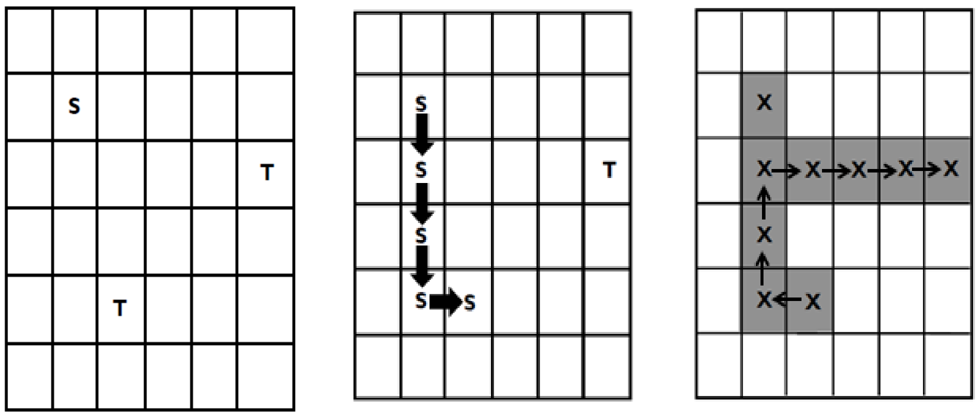
- **Two terminals in Single-Layer Routing:** The basic mechanics of Maze Router Problem (MRP) to have one Source and one Target nodes. In this mechanics, the shortest path to the Target node from the Source node is found after the expansion phase, and when the path has been found it is Backtrace to the Source node ‘S’. Then the Clean-up phase is executed as depicted in Figure 1.3. However this thesis is focuses on this type of mechanism [7].



A. Expansion Phase B. Backtrace

Figure 1.3: Single-Layer Routing with (Expansion Phase and Backtrace) [7]

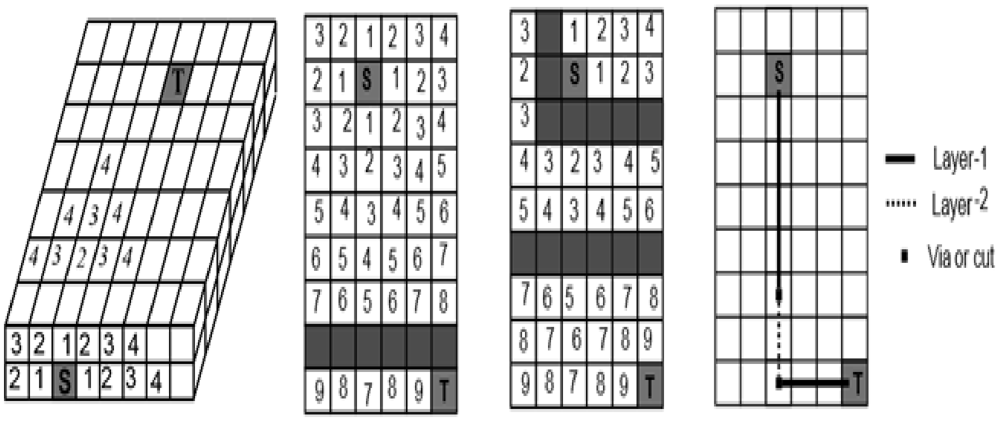
- **Multi Terminals in Single-Layer Routing:** It has many Target ‘T’ nodes with a single Source ‘S’ in a single layer. Its function is to find out a path that starts from the Source to the closest Target node [19]. As shown in the Figure 1.4.



Simple Multi-Terminals in Grid — Find the first Target — Backtrace and go to find second Target

Figure 1.4: Multi-terminal in single layer (One Source and Two Target nodes) [19]

- **Multi-layers Routing:** It has Three-dimensional grids. It is a model which is used to take the property of multi-layer routing. This model can take the varying parasites from the first layer to second layer routing [7]. In the Figure 1.5 below illustrates two layer-routing using two arrays.



A. 3D array — B. Layer-1 — C. Layer-2 — D. Retrace the path.

Figure 1.5: Illustrates two layer-routing using two arrays and Retrace the path [7]

- **Weighted cost:** It is started with spreading out of the identified node wave to find out the shortest path length. Discovering the neighbor of identified node will be maximized when the searching is allowed in the identified node. They will be stopped if there are not any waves to spread out [7]. Figure 1.6 illustrates how Weighted Grid defined to find the shortest path.

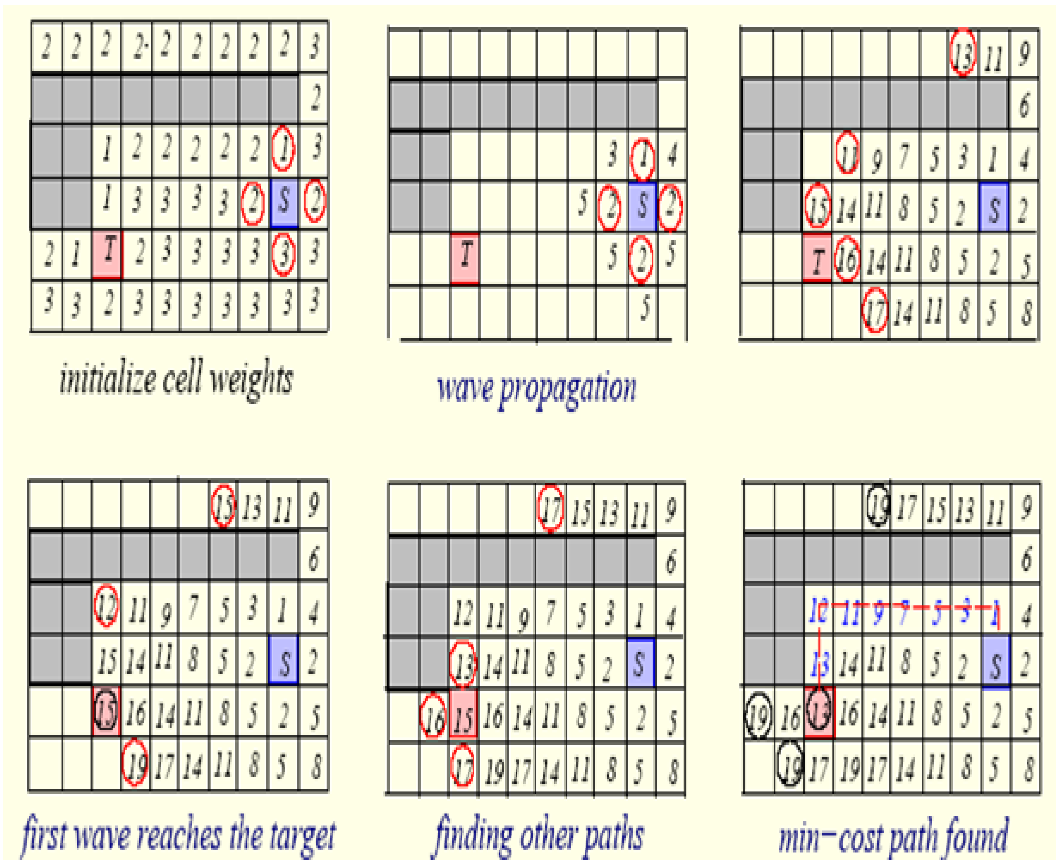


Figure 1.6: Illustrates Weighted Grid to find the shortest path [7].

2.1.2 Related work on Maze Router Problem

The Maze Router Problem (MRP) is one of the important problems which are solved using many different Algorithms, which work on two Terminals in Single-Layer routing.

H. C. Lee and Fikret Ercal [11], have showed a way of applying time-efficient Algorithm to solve the Maze Router Problem on a Reconfigurable mesh (RMESH) architecture. Where Reconfigurable mesh (RMESH) architectures are excellent candidates to solve the Maze Router Problem efficiently. They used fast Algorithm for Maze Routing on an RMESH. The result indicates that a large percentage of the shortest path that exists between two randomly selected terminals fall into one of the categories studied. This confirmed the author's practical significance of our Algorithms.

Y. Wu, M. Tsai and T. Wang [12], came up with two practical Optical Proximity Correction (OPC) Maze Router Problems and solved the two problems by modifying the Lee Algorithm. The Optical Proximity Correction (OPC) is employed to correct the process variation of the diffraction effect. Both Algorithms solved the problems in an optimal way, and they have both been implemented. The results from the Algorithms demonstrate their effectiveness. However, the effort, to solve the issues in Maze Router Problem, has only been implemented on two-layer routing models. Although they can be applied to Multiple-Layer including both reserved and unreserved layer models. The self-interactive effect will be considered in the future for more accurate calculation.

S. Hur, A. Jagannathan, and J. Lillis [20], have proposed an Algorithm that speeds up the techniques for time-driven Maze Router Problem by using a multigraph model. The multigraph model is generally and naturally captures the optimization techniques like the wire sizing through alternate edges. The basic Algorithm is a straightforward labeling Algorithm which uses a pseudo-polynomial runtime and has been found to be impractical if implemented naively.

2.1.3 Description and Formulation of MRP

The constraint for the problem must be considered to obtain a feasible solution so as to have a complete solution for the MRP. For this reason, the objective, feasibility and formulation of the Maze Router Problem are illustrated below.

The objective of Maze Router Problem is to find the shortest path. As mentioned before, at the end of a tour, the result is the path length.

A feasible solution of a MRP is when it visits or goes to all the nodes ones or more times till it reaches the Target node.

In the formulation we can calculate the distance between the nodes using the Euclidean Distance since our objective is to find the shortest path. The Euclidean Distance between Current node and Target node can be given in equation (2.1).

$$\text{Path}(X, Y) = \sqrt{\left(\sum_{i=1}^{n}(X_i - Y_i)^2\right)} \qquad \text{Eq. (2.1)}$$

$X_i = \{x_1 \dots x_n\}$ and $Y_i = \{y_1 \dots y_n\}$,

Where Path(X, Y) is the distance between Source node *'S'* with coordinates (x_1, y_1) and Target node *'T'* with coordinates (x_t , y_t) and *Path* represents the distance from *'S'* to *'T'*.

Chapter 3

COLLECTED TECHNIQUES FOR THE SOLUTION OF MAZE ROUTER

3.1 Introduction

In this chapter, four Algorithms Connected Component Labeling, Depth First Search, Lee and A* will be used for the solution of the Maze Router Problem. Figure 2.1 shows the Collected Methods for the Solution of Maze Router Problem.

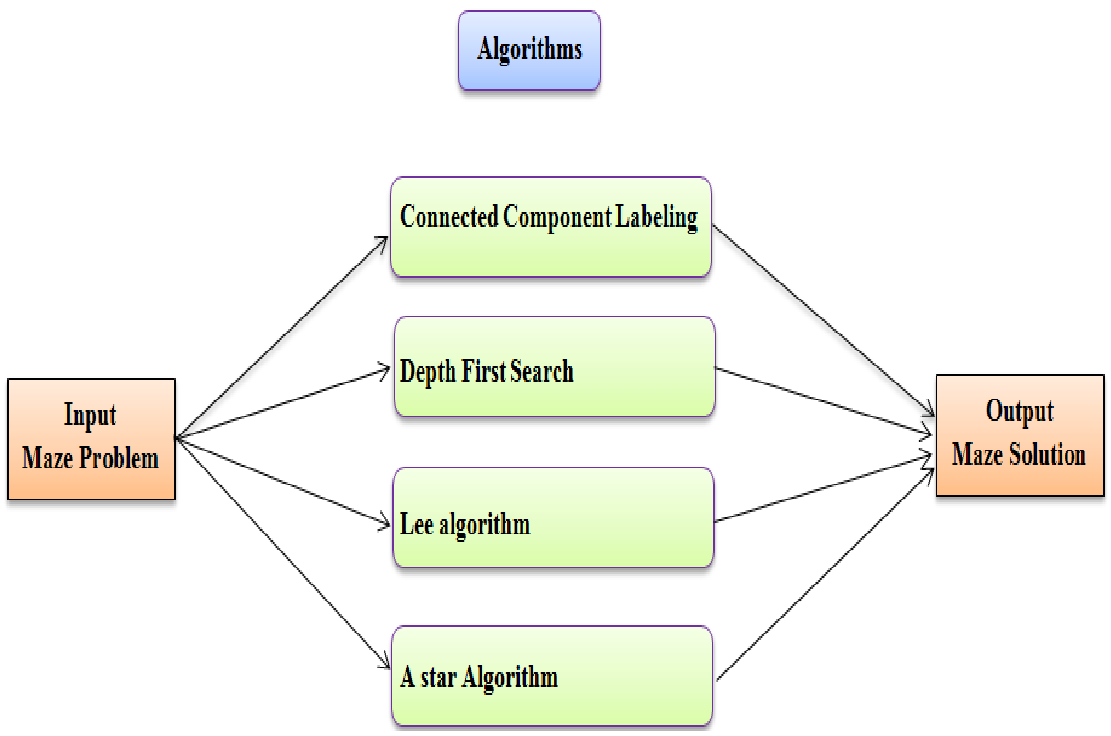


Figure 2.1: Collected Methods for the Solution of Maze Router Problem

3.2 Connected Component Labeling Algorithm

Connected Component Labeling (CCL) Algorithm is an Arithmetic application of graph theory. It is used in computer vision to discover connected regions in binary images [3][10].

The binary image, containing nodes and connecting edges, where is constructed from relevant input data. The connectivity is determined for a node in the binary image with 4-connected or 8-connected nodes [13]. In this section, the information is presented in the labeling stage. After the first node when the connected component was found, all the connected nodes of that connected component were labeled before going onto the next node in the binary image. Moreover, the binary image was divided into sub-groups and each group has different values, so after that, the original information can be recovered and processed in the way that CCL Algorithm makes two passes over the binary image. Firstly, the pass should be made to assign temporary labels and record equivalences. Secondly, another pass should be made to replace each temporary label by the smallest label of its equivalence class. The labeling process scans the image, node-by-node from Northwest to Southeast, in order to identify the connected node regions, i.e. areas of the neighboring node that share the same set of intensity values. In this thesis, 4-connected methods are used to find the path.

3.2.1 Finding Connected Components

Two nodes in the image are 'connected' if a path can be discovered for which the value of the image function is the same all along in the path. Figure 2.2 shows how to Find a path between two nodes.

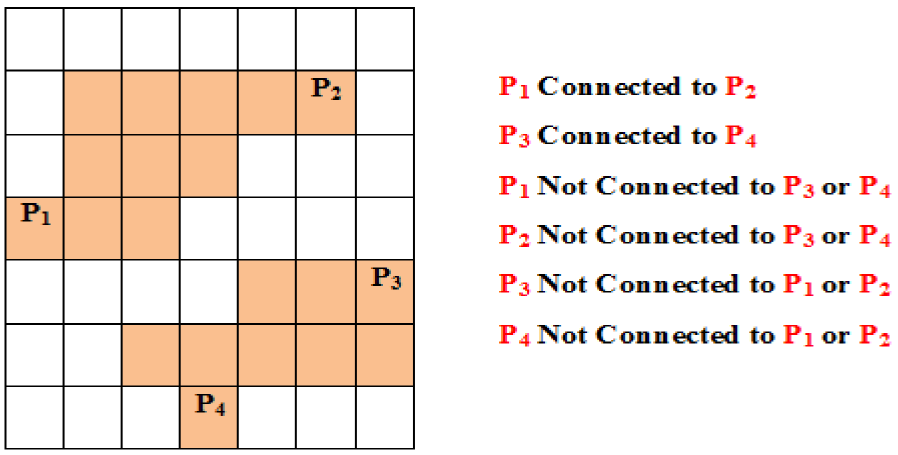

P1
P2
P3
P4
P1 Connected to P2
P3 Connected to P4
P1 Not Connected to P3 or P4
P2 Not Connected to P3 or P4
P3 Not Connected to P1 or P2
P4 Not Connected to P1 or P2

Figure 2.2: A path between two nodes (P_1—P_2) or (P_3—P_4)

3.2.2 Neighbors

Consider the definition of the term 'neighbor'. Two common definitions:

1. 4-neighbors (4-connected) { [$i+1,j$], [$i-1,j$], [i ,$j-1$], [i ,$j+1$] }

	X($i-1, j$)	
X($i, j-1$)	X(i, j)	X($i, j+1$)
	X($i+1, j$)	

2. 8-neighbors (8-connected)

{[$i+1, j$], [$i-1, j$], [$i, j-1$], [$i, j+1$], [$i+1, j+1$], [$i+1, j-1$], [$i-1, j+1$], [$i-1, j-1$]}

X($i-1, j-1$)	X($i-1, j$)	X($i-1, j+1$)
X($i, j-1$)	X(i, j)	X($i, j+1$)
X($i+1, j-1$)	X($i+1, j$)	X($i+1, j+1$)

A binary image is an array of two-dimensional that has two possible values for each node *"0"* or *"1"*. In this thesis, two colors are used which are black and white. The white represent number *"1"* and black represent number *"0"* as shown in the Figure 2.3.

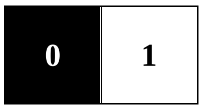


Figure 2.3: A binary image has two possible values

CCL Algorithm starts by selecting the first point, then scan it as the current point. It then check if the current point is a foreground point, if yes then it checks if any reference neighbors pixel is a foreground point. If yes, then it assign old label to the current point and also check for another point, after checking for another point it checks if the point is the last point, if it's the last point it stop searching and the final path is found, if the point checked is not the last one the algorithm will move to the next point on the right of the current then scan the point and repeat the steps all over until the final path is found as shown in the flowchart given Figure (2.4).

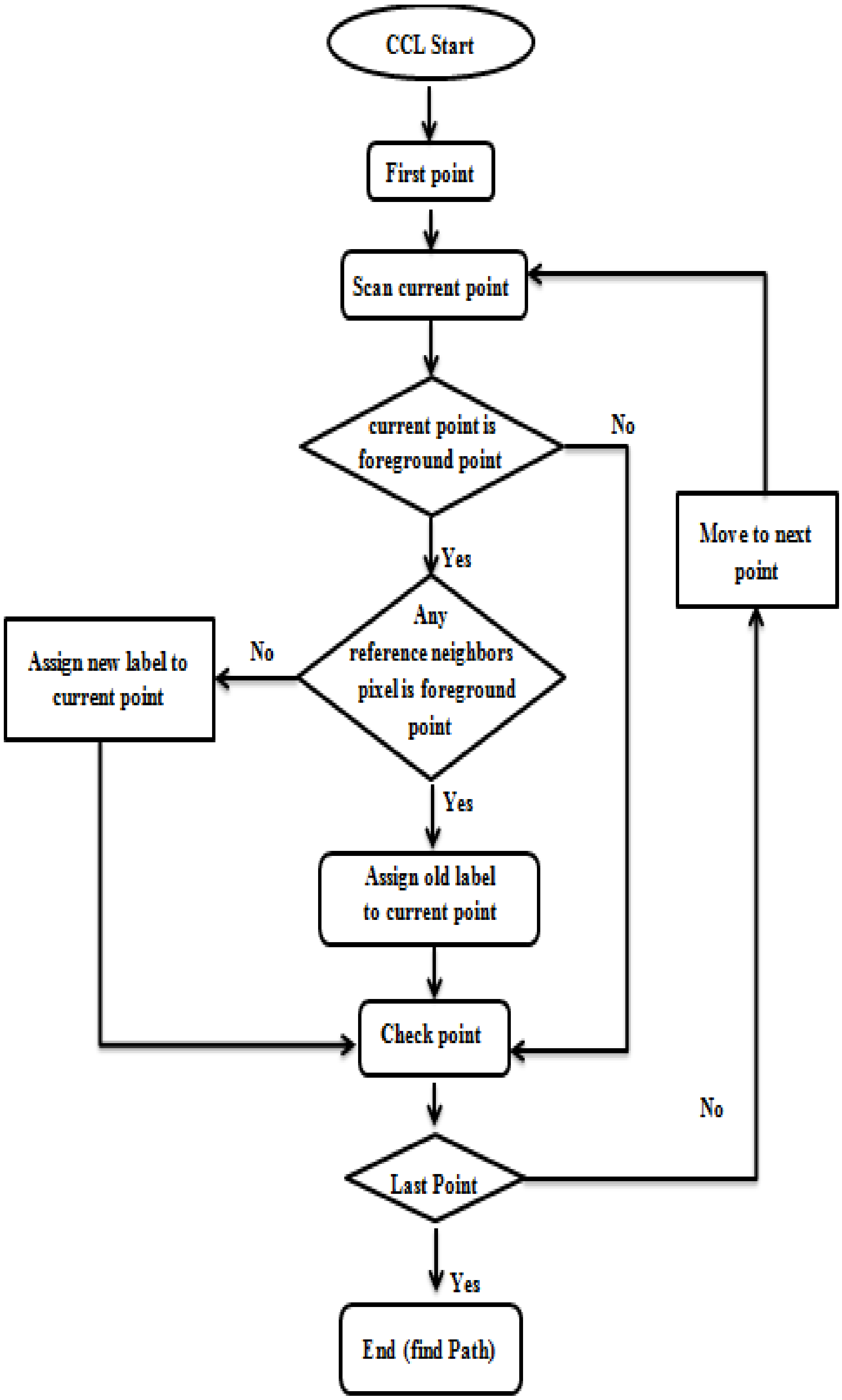
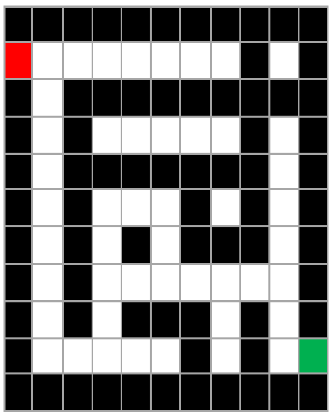

CCL Start
First point
Scan current point
current point is foreground point
No
Yes
Any reference neighbors pixel is foreground point
No
Assign new label to current point
Move to next point
Yes
Assign old label to current point
Check point
No
Last Point
Yes
End (find Path)

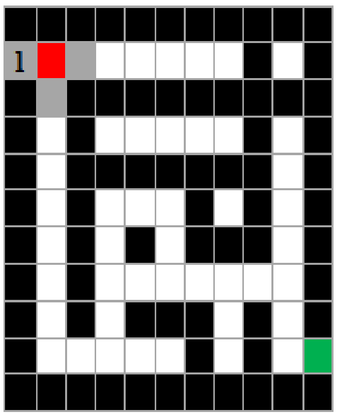
Figure 2.4: CCL Algorithm Flowchart

3.2.3 How Connected Component Labeling work

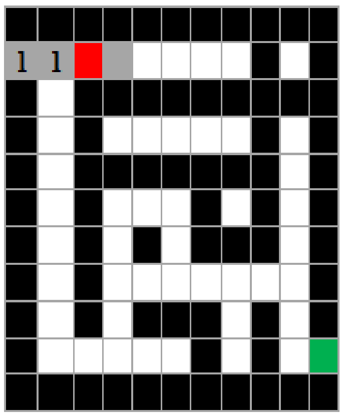
A Connected Component Labeling Algorithm is the process for analyzing binary images includes the following steps. Firstly, a forward scan is executed from Northwest to Southeast, to assign a label to each current node in a simple binary image. When found, non-background current node will have a background as shown in (a) of the Figure (2.5). Secondly, when the connected component is found the value one is given to "*currentLabelCount*". Where "*currentLabelCount*" is an Interface holds one function that takes a Bitmap as an input and returns a collection of discovered objects, where the initial value of "*currentLabelCount*" is *1*. And all the connected nodes of that connected component are determined before moving to the next node in the image as shown in (b - d) of the Figure (2.5). Thirdly, if a new node is found which is not connected with the previous node then increment "*currentLabelCount*" by *1* and assign label to that node. That node has a different label as shown in (e) of the Figure (2.5). When new node connected with previous node and If this node is a forward node and not already labeled, then give it a "*currentLabelCount*" and add it to the first component in the queue as shown in (f) of the Figure (2.5). All neighbors for the current node are non-connected with parent label value then increment value for "*currentLabelCount*" by 1 and assign label to the current node as shown in (g - h) of the Figure (2.5). Finally, after completing the scan the CCL Algorithm gives labels to all the nodes in the image. Finally, same labeled nodes will give all the connected paths in the binary image as shown in (i) of the Figure (2.5). The flowchart of the CCL Algorithm is given in Figure 2.4.



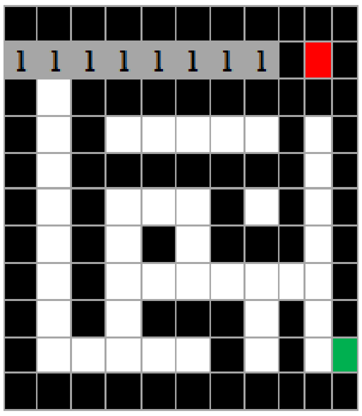
a. A simple binary Image.



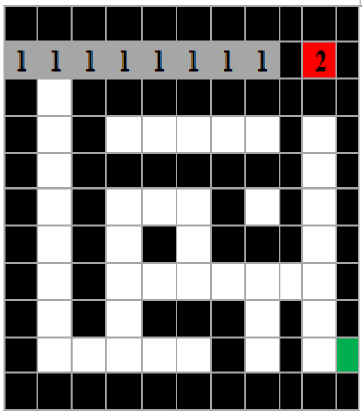
b. Current point checks three Neighbors.



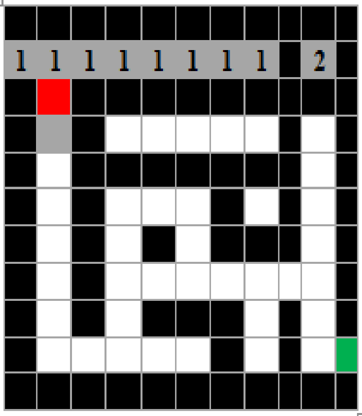
c. Current point check two neighbors one of them is already labeled.



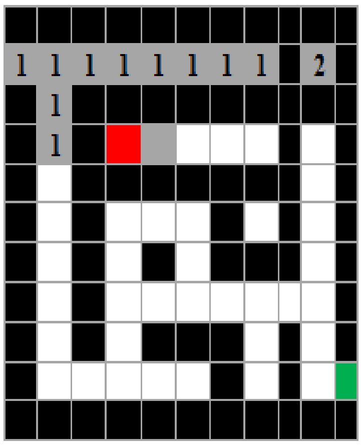
d. Continue assign the current point label to the neighbors.



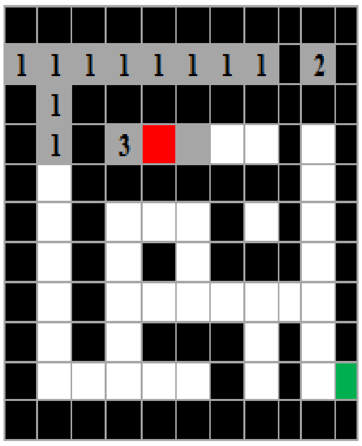
e. None connected node with new label.



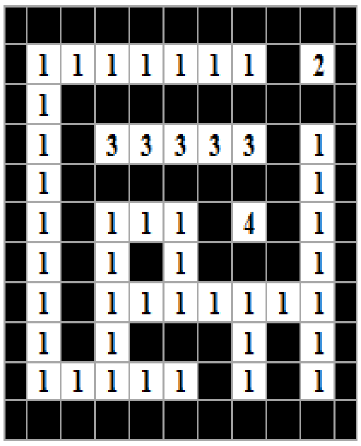
f. Scan and find new node.



g.



h. Current point has a different label.



i. All the nodes are labeled the image.

Figure 2.5: Connected Component Labeling Algorithm steps

3.3 Depth First Search Algorithm

A Depth First Search Algorithm (DFS) is a systematic method to find all the nodes reachable from the Source node 'S' to the Target node 'T' in the Maze. The Depth First Search method was published in 19th century by French mathematician Charles P. [14], as a strategy for solving Mazes [15][16]. This approach is one of the simplest methods for solving a Maze. Consider the area of the Maze being a large grid map of nodes. Where each node has four neighbors, starting from the Source node and keep digging paths in one of the four-directions, North, East, South and West, until you can’t go any further. When a blocked way is found, the agent is Backtrack to find a previous node with an unvisited neighbor. This process is repeated until path is discovered. The “**Backtracking**” is a general Algorithm for finding solutions to some computational problems. And is an important tool for solving constraint satisfaction problems, such as Maze, and many other puzzles [5][17][18]. This technique guarantees to find a path in the Maze.

This method will find a path and is fast for all types of Mazes. With Depth First Search (DFS) Algorithm move can be made in four directions. Writing '*' symbol when it tries a new direction, and delete a '*' symbol when it fail to find a path. It writes 'x' symbol when using Backtrace Algorithm, and a single solution will be printed out when it find the Target node. This Algorithm will always find a solution if it exists, but will not necessarily be the shortest solution.

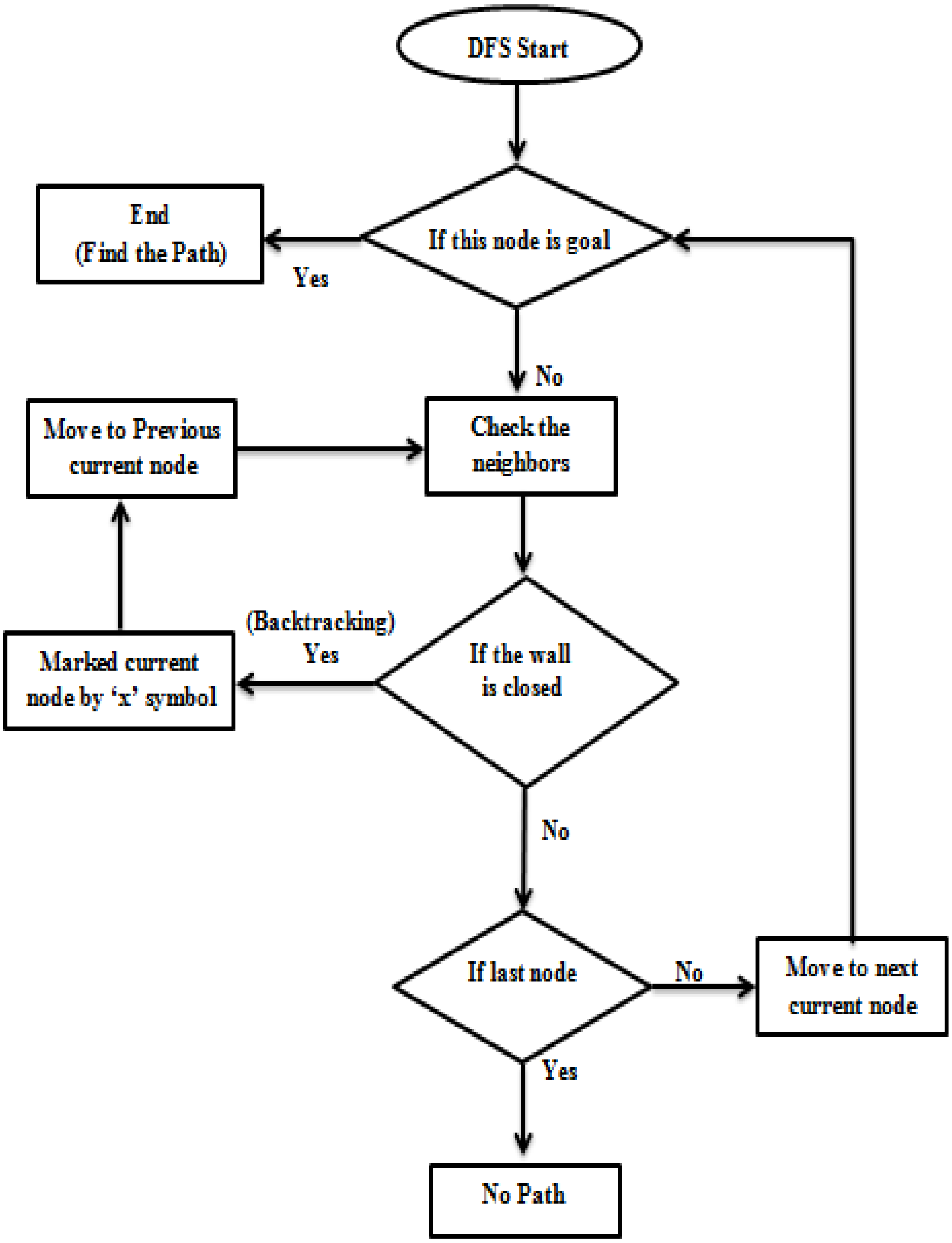


Figure 2.6: The Flowchart for Depth-First Search Algorithm

From the Figure (2.6) the DFS Algorithm starts by checking if the current node is the goal node, if yes path have been found, if not it move to the next neighbor then check if the wall is closed, if it is closed then it mark current node with 'X' and move to previous current node then check if the wall is closed, if not it then checks if the current node is the last node, if it is then there is no path and if the current node is not the last node the algorithm will move to the next current node and repeat the whole process again.

3.3.1 Depth First Search Algorithm has two cases

Recursion case and the Base case.

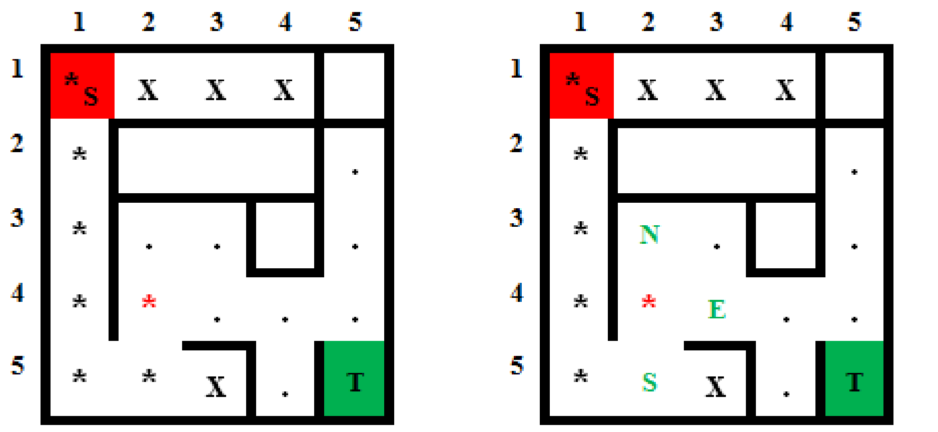
- **Recursion case:**

  The Recursive is a process return, where allows the function to be returned several times, since it calls itself during its execution. Functions that incorporate Recursion are the named Recursive functions. In order to have our Algorithm recursive, we need to view the problem in terms of related to subproblems. That means we need to find the path they start from the Source node 'S' to Target node 'T' in a Maze, where each node has four neighbors, and keep digging paths in one of the four-directions, North, East, South and West. However, recursion must be incorporated accurately, since it can lead to an infinite circle if no condition is met that will stop the function.

  Assume that the Source node *'S'* has the position *i*=*a*, *j*=*b* in the Maze Router (MR) Problem. What we now want to know is whether it has a path from the position in coordinates (*a*, *b*) to the Target node *'T'* in coordinates (*e*, *d*). If there is a path to the Target node from *i*=*a*, *j*=*b*, then there is a path from the source node *'S'* to the Target node *'T'* as shown in (a) of the Figure (2.7).

  To find a path from position *i*=2, *j*=4 to the target node *'T'*, we can use function *FINDPATH(i, j)* to try to find a path from the North, East, and South of *i*=*a*, *j*=*b*. *FINDPATH(i, j)* function pseudo code is given in Figure 2.8.

Note that can't use *FINDPATH(i, j)* function from the West direction because the wall is closed as shown in (b) of the figure (2.7).



a. Agent in the position (x=2, y=4). b. An agent with three open directions

Figure 2.7: The *FINDPATH*(*i*, *j*) function using Recursion to find Target node

```
FINDPATH (i, j)
    beginning
        if (i, j outside maze) return false;
        if (i, j is find Target node) return true;
        if (i, j is close wall) return false;
        mark x, y as part of solution path;
        if (FINDPATH (i, j-1) ~= true) return false;  → North
        if (FINDPATH (i+1, j) ~= true) return false; → East
        if (FINDPATH (i, j+1) ~= true) return false; → South
        if (FINDPATH (i-1, j) ~= true) return false;  → West
        unmark i, j as not part of solution path;
        return false;
    end
```

Figure 2.8: Pseudo Code for Recursion and Base case.

- **Base case:**

It is not enough to know how to use *FINDPATH(i, j)* function recursively to advance through the Maze. They also need to discover when *FINDPATH(i, j)* function must end. One such base case is to stop when it approaches the Target node *'T'*. The other base cases have to know what to do with invalid nodes. For example, they have mentioned how to search North of the current node, but disregarded whether the North node is legal.

All these steps together complete Depth First Search Algorithm that discovers and marks a path from Source node *'S'* to the Target node *'T'*, if the connection exists. The path will be called at least once for each node in the Maze that is tried as part of the path.

Path marking will be done with the '*' symbol and unmarking with the 'x' symbol.

3.3.2 Solving Maze Router by Depth First Search Algorithm

Assume that The Simple Maze Router Problem has two terminal nodes. It determines the Source node *'S'* in North West in coordinates (i=1, j=1) and the Target node *'T'* in the South East in coordinates (i=5, j=5). Nodes in the Maze will either be open or blocked with the wall, the mark '.' Symbol is the open wall the agent can move, and mark '|' symbol is used as closed wall the agent can't move to this way then need to choose another direction as shown in the Figure 2.9.

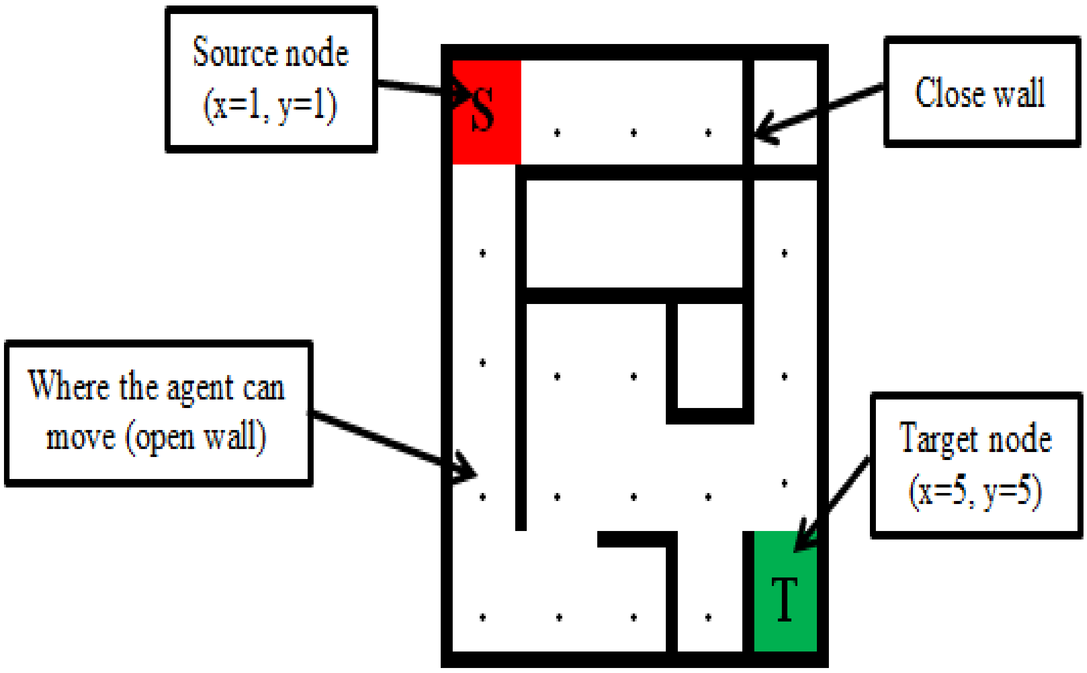


Figure 2.9: A Simple Maze Router Problem

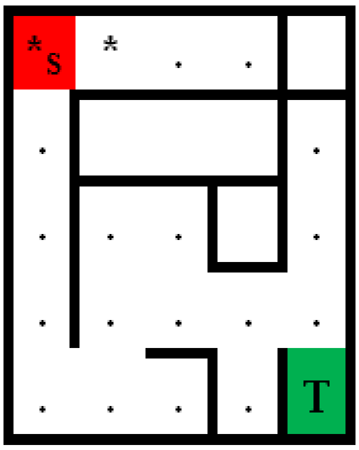
In any given moment, the agent can move only one step to one selected direction.

Moves are:

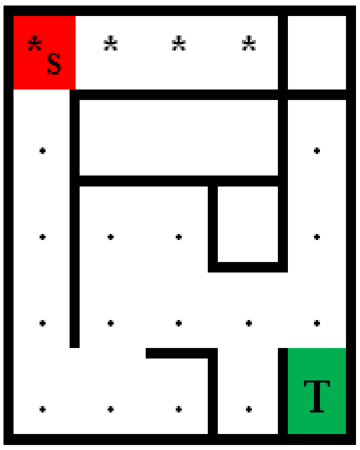
- move to North: (*i*, *j*) → (*i*, *j*-1)
- move to East: (*i*, *j*) → (*i*+1, *j*)
- move to South: (*i*, *j*) → (*i*, *j*+1)
- move to West: (*i*, *j*) → (*i*-1, *j*)

The agent can only move to nodes without wall blocked and must stay within the Maze. The agent should search for a path from the Source node 'S' to the Target node 'T' until it finds one or until it exhausted all possibilities. In addition, it should print the path it finds in the Maze.

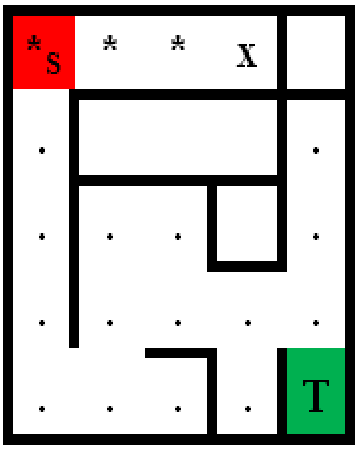
The agent started from NorthWest in coordinates (i=1 and j=1) move one by one step to East direction. The path can be marked by the '*' symbol. A path refers to either a partial path, marked while the agent is still searching as shown in the Figure 2.10 from (a) to (b). An important capability that the Depth First Search parts of the Algorithm will give us is the ability to backtracking. In the node that has coordinates (i=4 and j=1) the agent is stopped because the East wall for the current node was closed, and then the Algorithm is processed to find which direction is open. Firstly, it will try to find a path to the Target node *'T'* from the point North that has coordinates (*i*=4, *j*=0), by calling *FINDPATH*(*i*=4, *j*=0). Since the North position is not open, the *FINDPATH*(*i*=4, *j*=0) will return false. And then it will go backward to the previous node to *FINDPATH*(*i*=4, *j*=1) and resume at the step only after it went North. Secondly, it will go to check South wall for the current node, calling *FINDPATH*(*i*=4, *j*=2). This node is not open, so it will backtrack to the previous node and resume at the step just after it went South. Finally, it will go to the West direction that has coordinates (*i*=4, *j*=1), by calling *FINDPATH*(*i*=3, *j*=1). Since the West direction is the last direction to search from (i=4, j=1). It will unlabel coordinates (*i*=4, *j*=1) with the 'x' symbol. And then backtracks to the previous node, *FINDPATH*(*i*=3, *j*=1) as shown in the Figure (2.10) from (c) to (e). Will continued this mechanism In the Maze Router Problem until it find Target node 'T' that has coordinates (i=5, j=5) and then stopped it. And then print the path from Source node 'S' to Target node 'T' as shown in (f) of the Figure (2.10).



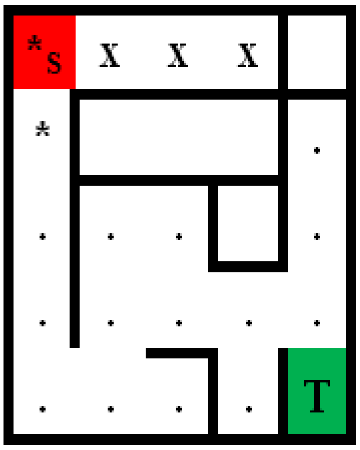
a. Start search to East direction



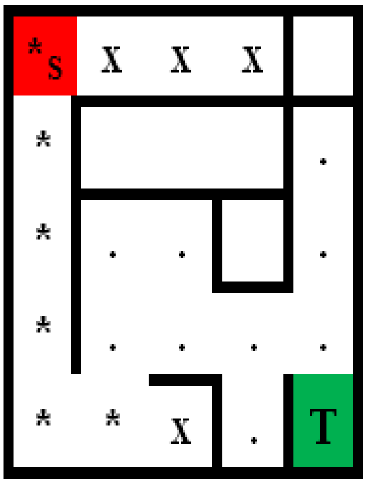
b. Stop Searching toward East direction



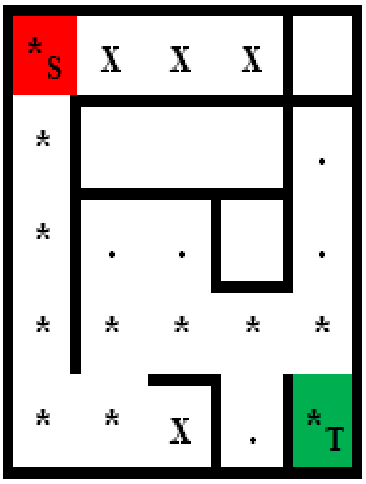
c. Backtracking is started



d. Continue Backtracking until a node has an unvisited neighbour



e. Move to new direction

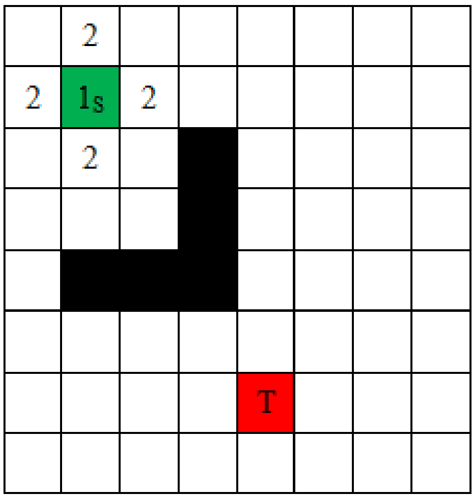


f. The agent finds the Target node

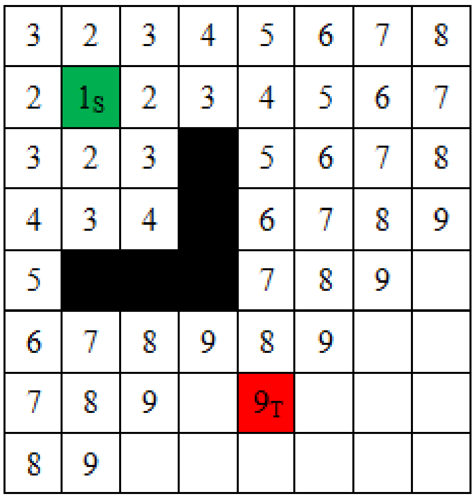
Figure 2.10: Solving Maze Router Problem by DFS Algorithm

3.4 Lee Algorithm

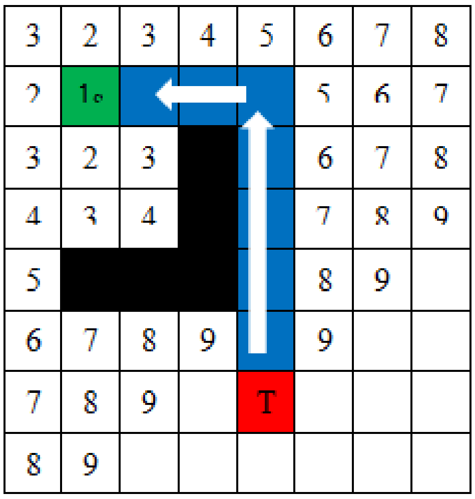
Lee Algorithm is another Algorithm that solves Maze Router Problem (MRP) [6]. The Maze Routing Problem is defined on the single layer as a grid map for this Algorithm. Each grid node may have at most four neighbors. It searches for the shortest path between two terminal nodes and guarantees to find a path from the Source node to the Target node if the connection exists. Lee router Algorithm connects a Source node to the Target node in three main steps: Expansion, Backtracking, and Clearance. Firstly, Expansion performs a breadth-first search from the Source node until the location of Target node, thus, all nodes have been visited [22]. During the search each unvisited node is checked and then numbered by its length from the Source node to the Target node. Occupied nodes cannot be crossed directly, and router must divert around them as it is shown in the Figure (2.11) from (a) to (b). Secondly, Backtracking runs if expansion locates the Target node. Backtracking starts from the Target node and iteratively finds its neighboring node, chooses the neighbor that has a lower value and occupies it until it reaches the Source node as it is shown in (c) of the Figure (2.11). Finally, Cleaning-up of all the distance marks from other grid nodes except the selected path should be used as it is shown in (d) of the Figure (2.11).



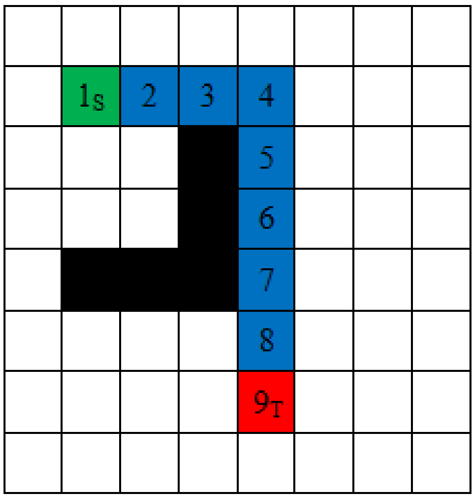
a. Expansion starting from the Source node



b. Expansion find the Target node



c. Backtrace from target node to source node



d. Clearance

Figure 2.11: Illustration of Expansion, Backtrace and Clean-up for Lee Algorithm

The Figure (2.12) shows the pseudocode for Lee Algorithm which is represented by Initialization, Wave expansion, Backtrace, and Clearance to find the shortest path from Source node *'S'* to the target node *'T'* from the Maze Router Problem.

```
Input: A grid map of two-dimensions (N x N) with Initial blockages, if any marked.
The Grid map has two-terminals to be routed.
Output Grid map with all nodes routed, if possible.
Starting
  For each node in the grid map do
    1. Initialization : CCS (current node set) = Source node;
                        NCS (neighbor node set) = Q;
                        WaveCounter = 1;
    2. Label all node in CCS with value of WaveCounter;
       If Target node has been added to NCS then
       Goto step 5;
    3. Wave Expansion: for each node in CCS, add all its unexpanded neighbors to
       NCS;
    4. If NCS is empty then
         this Grid map is not routable;
         Goto step 6;
       Else
          CCS = NCS;
          NCS = Q;
          WaveCounter = WaveCounter +1;
          Goto step 2;
    5. Backtrack: Retrace the shortest path to the source node, starting from the
       Target node, by considering neighboring nodes in the descending order of
       labels. It have more than one node with label L-1 that are adjacent to a node
       with label L, and then choose the one that causes no change in direction in the
       path traced out so far.
    6. Clearance: Reset all labeled nodes, except those used for the path just found,
       to be empty.
  End for
End
```

Figure 2.12: Pseudo code for Lee Algorithm [26]

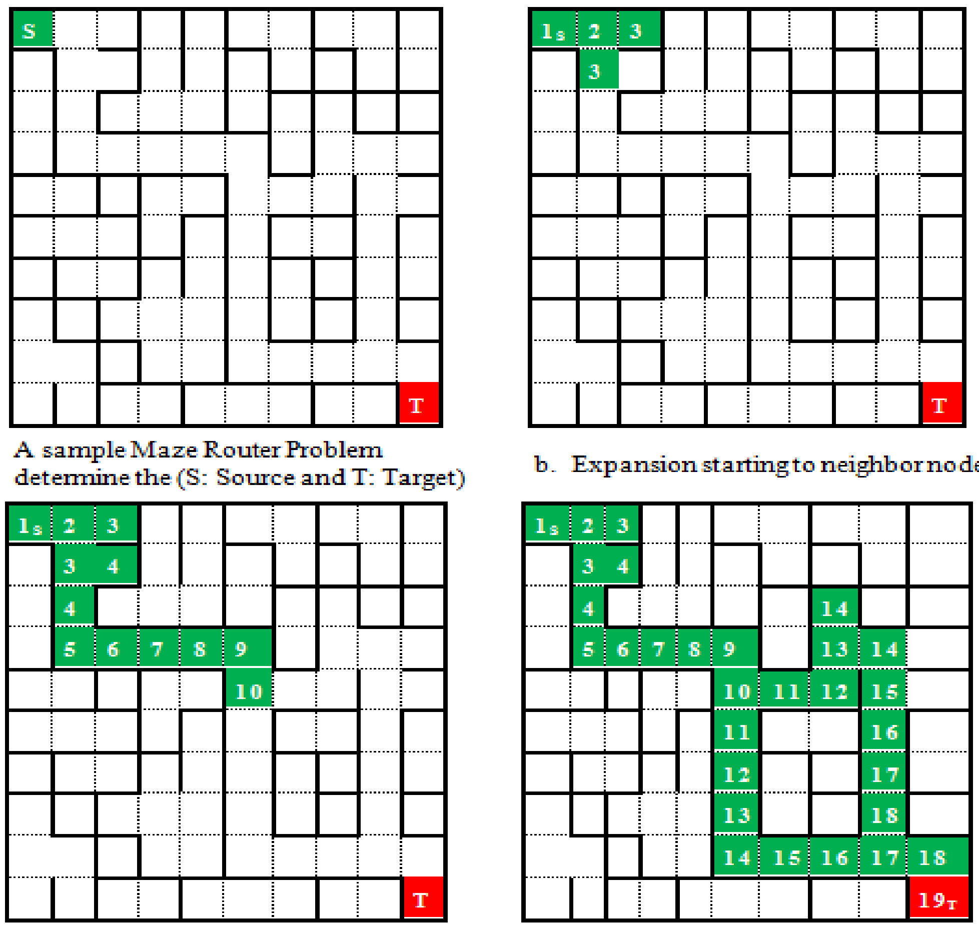
3.4.1 Advantages and Disadvantages

- **Advantages:**
    - i) Guarantee to find connection between two terminals if it exists.
    - ii) Guarantee minimum path.
- **Disadvantages:**
    - i) Requires large memory for dense layout.
    - ii) Slow. Because the wave expansion phase take many of the time.

3.4.2 There are three main steps to solve Maze Router Problem

- **Expansion**:

  It searches for a shortest-path connection between the Source and Target nodes, i.e. labeling each node with its distance from the Source node, the expansion phase will eventually reach the destination node if a connection is possible. The Expansion uses a breadth-first search Algorithm to find the Target node from the Source node, and all nodes have been visited if the wall is open shown the Figure 2.13 from (b) to (c). During the search each node is checked whether it is unvisited, and then numbered by its distance from the *'S'*. If it is a blocked node, it cannot be cross directed and routing must divert around them. Thus, the Target node can be found by using the shortest path shown in (d) of the Figure 2.13.


a. A sample Maze Router Problem determine the (S: Source and T: Target)

b. Expansion starting to neighbor nodes

a. Expansion continuo

b. Expansion find the target node

Figure 2.13: Expansion from the Source node to the Target node and find the goal

- **Backtrace**:

The Backtrace starts when Target is reached, and the Algorithm starts from the Target node and looks for nodes with the minimum number to add to the chosen pathway in order to obtain the shortest path between the terminals and the formed connection. Usually the routing is performed in ascending order of distance, i.e. lowest routes first. This guarantees the shortest pathways, which commonly have more alternatives. Not displacing the shorter ones from their original positions is shown in (a-b) of the Figure 2.14.

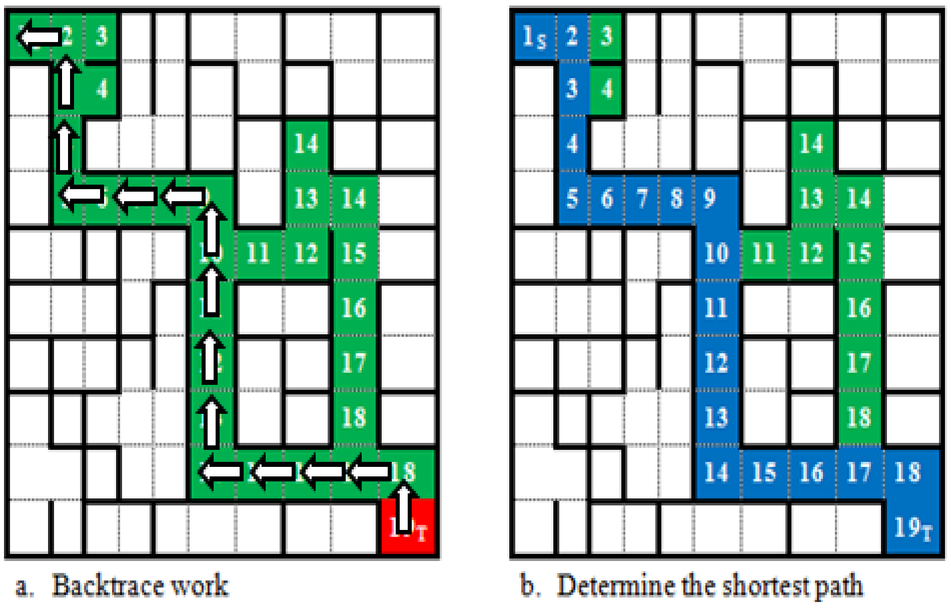


Figure 2.14: Illustration Backtrace work

- **Clean-up:**

  In this step it should Clean-up all distance marks from other nodes except the selected path shown in the Figure 2.15.

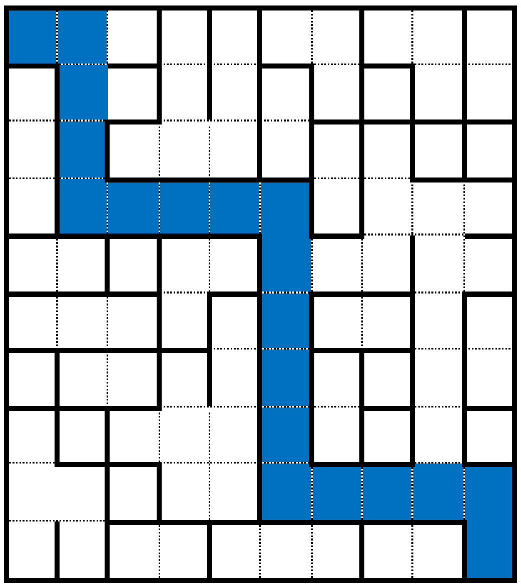


Figure 2.15: Illustration Clean-up on Grid map without Path mark

3.5 A* Algorithm

A* is a Pathfinding Algorithm, one of the faster way to make agent find its way in the Maze Router Problem. It is the process of plotting an efficiently traversable path between nodes. A* Algorithm combines features of uniform-cost search and pure heuristic search to compute efficiently the optimal solutions. It is generally outperformed by the Algorithms that can pre-process the grid map to attain better performance. For A* to be superior to another Algorithm it needs to be able to find the shortest path [23][24].

It was first described in 1968 by Peter H., Nils N. And Bertram R. of Stanford Research Institute (now SRI International)[25]. It is an extension of Dijkstra's Algorithm [8]. A* is a search Algorithm that finds the shortest path between two node, Source nodes and Target node in the grid map. A* Algorithm chooses the next node which should have least cost as a heuristic function from the current node. If the current node has more than one least cost nodes, we can choose the nearest one as the next node. Usually, a good heuristic function obtains solution in a short time. The fitness of the A* Algorithm given in equation (3.1).

$$F\ (n) = G\ (n) + H\ (n) \qquad \text{Eq. (3.1)}$$

Where the value of F consisted of the collection of values of G and H for every node. G is the cost of movement from the Source node to the current node, H is an estimation of the cost from the labeled grid map to the Target node. The node that is closer to the Target node has lower costs than nodes that are further from the Target, this biases the search in the direction of the Target [8]. As a heuristic function, H has a relationship with whether the method is A* Algorithm or not. The pseudocode for A* Algorithm is given in Figure 2.16.

```
Initialize array open_list[]
Initialize array closed_list[]
Put the Source node on the open_list[]
While the open_list[] is not empty
{
   Find the node with the least value of F on the open_list[], call it 'q'
   pop q off the open_list[]
   For each successor
        If successor is the Target node, stop the search
              successor_G = q_G + distance between successor and q
              successor_H = distance from goal to successor
              successor_F = successor_G + successor_H
        If a node with the same position as successor is in the open_list[]
              which has a lower F than successor, skip this successor
        If a node with the same position as successor is in the closed_list[]
        which has a lower F than successor, skip this successor
        otherwise, add the node to the open_list
   End
   push q on the closed_list[]
}
```

Figure 2.16: General pseudocode for A* Algorithm

3.5.1 Description of A* Algorithm step by step

1. Add the beginning node to the open list.
2. Repeat the following:
    a. Look for the lowest F cost node on the open list. We refer to this as the current node.
    b. Switch it to the closed list.
    c. For each of the four node neighbors to this current node.
        - If it is a blockage, or it is on the closed list, ignore it.

- If the current node is not on the open list, add to the open list. Get the current node the parent of this node. Take the *F*, *G*, and *H* costs of the node.
- If it is on the open list now, compare to see if this path to that node is better, using *G* cost. *A* lower *G* cost suggests that this is a better path. If so, change the parent of the node to the current node and recalculate the *G* and *F* scores of the node.

d. Stop when you: Add the Target node to the closed list. In that case, the path has been found, or Fail to find the Target node *'T'*, and the open list is empty. In this case, there is no path.

3. Save the path. Go backward from the Target node go from each node to its parent square until you reach the starting square. That is your path.

3.5.2 The main idea of A* Algorithm

1. Seek *F*'s values from nodes that have not been searched, and use an ascending stack to save them.
2. Choose the smallest one as current node;
3. Pull out the head element from the queue, figure out the values of all current node neighbors, and then put F in stack;
4. Loop 1 to 3 steps until stack is empty or we had found the Target.

When the shortest path between two nodes on the grid map have been found using Euclid function. As shown in the below Equation (3.2).

$$\text{Euclid distance H (n)} = \sqrt[2]{(X_1 - X_2)^2 + (Y_1 - Y_2)^2} \qquad \text{Eq. (3.2)}$$

H is an estimation of the cost value between two nodes, from the current node to the Target node in the grid map. And (x_1, x_2, y_1, and y_2) are the coordinates for two nodes, where (x_1, y_1) the current node coordinates and (x_2, y_2) the Target node coordinates. By the Euclid distance will find the shortest path quickly.

3.5.3 Representation

A* Algorithm uses a Best-First Search method and finds a minimum cost from Source node *'S'* to Target node *'T'*. As A* Algorithm traverses the grid map, it follows a path of the lowest required total cost or distance, preserving a sorted priority queue of alternate path segments along the way. It uses a heuristic cost function of the current node to determine the order in which the exploration visits nodes in the Maze Router Problem. The cost function is a sum of two values. Firstly, the past path cost function, which is the known distance from the Source node *'S'* to the current node. Second, a future path cost function, which is an admissible "heuristic" distance from the current node to the Target node *'T'*. The $H(n)$ part of the $F(n)$ function must be an admissible heuristic; that is, it must not overestimate the distance to the Target node. In such a case, A* can be, roughly speaking, implemented more efficiently, no node needs to be processed more than once. Thus, fewer nodes are visited and hence less computation time is needed as it is shown in the Figure 2.17.

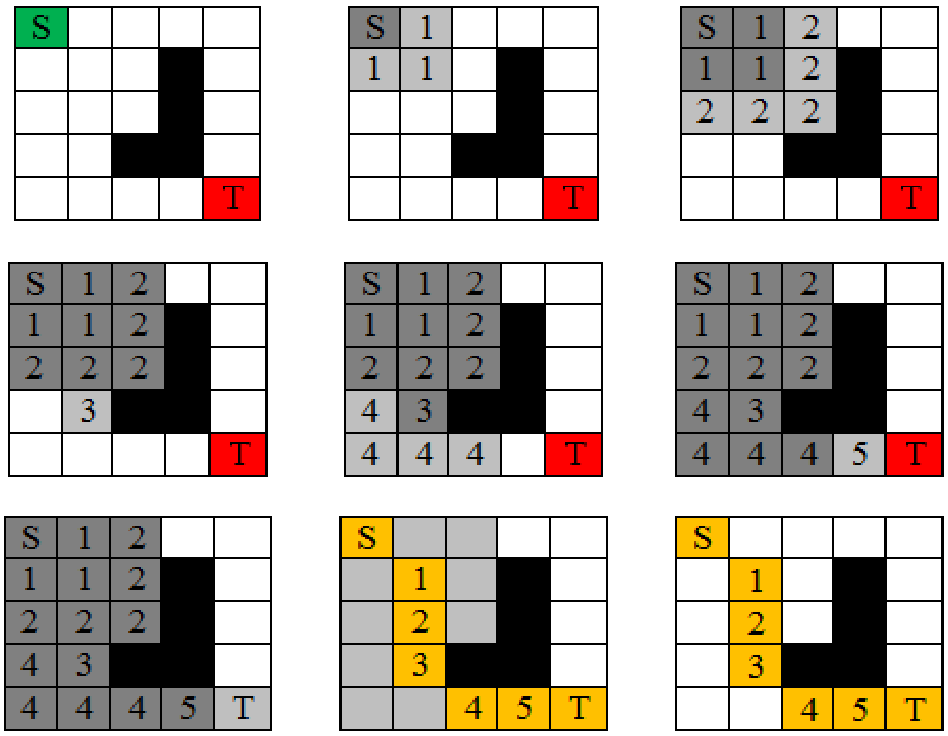


Figure 2.17: Example of the A* Algorithm. The nodes colored in light gray are the Open List, and the dark gray nodes denotes the Close List, and the orange nodes denotes the final path from ‘*S*’ to ‘*T*’

3.5.4 Simple example to A* Algorithm

In this section, a simple Maze Router Problem will be used to find out the shortest path between two terminals with A* Algorithm; the size of the Maze is (11x11). The black color represents a blocked node, and the white color – nonblocked node. The Source node *'S'* in the northwest has (1, 1) as its coordinates, and the Target node *'T'* in the southeast has (9, 9) as its coordinates. Depending on the costs, the moves in the Maze can be not only horizontal or vertical, but also diagonal. This Algorithm defines the two variables (***G*** and ***H***). ***G*** is the cost to move from the Source node to the Current node. While ***H*** is an estimation of the cost to move from the Current node to the Target node using the Euclid distance [26]. Moreover, the two lists have been used: Open List containing the nodes to explore, and Closed List containing the processed nodes as it is shown in Figure (2.18) below.

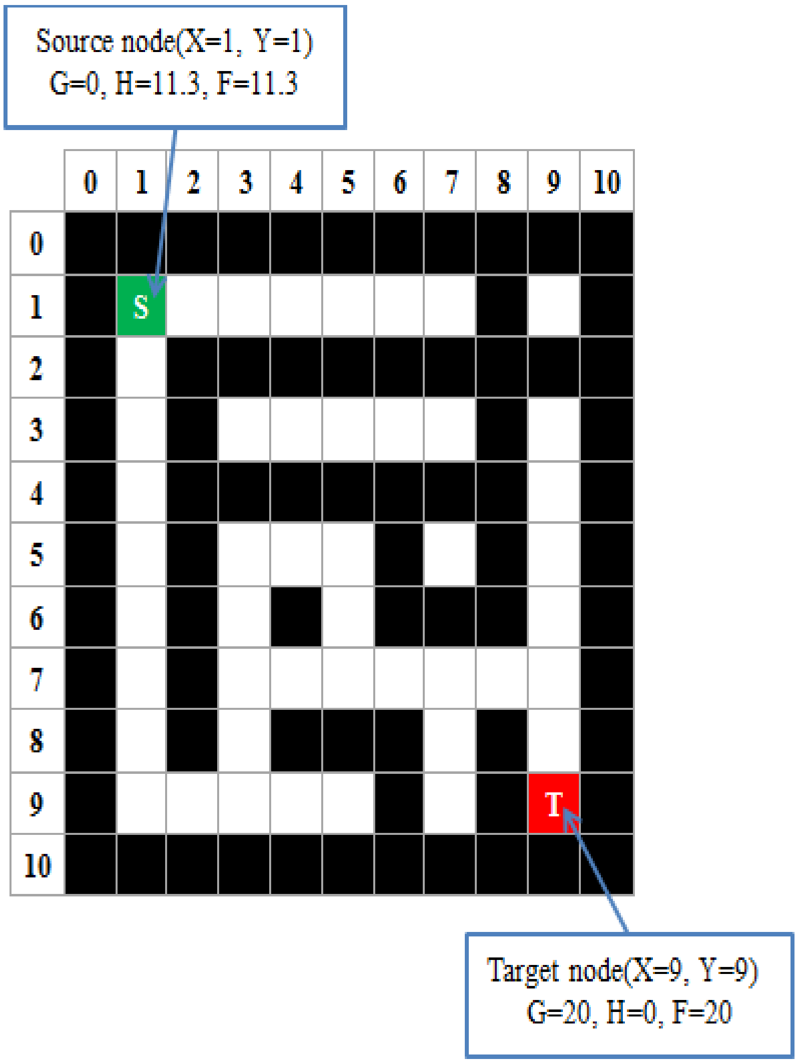
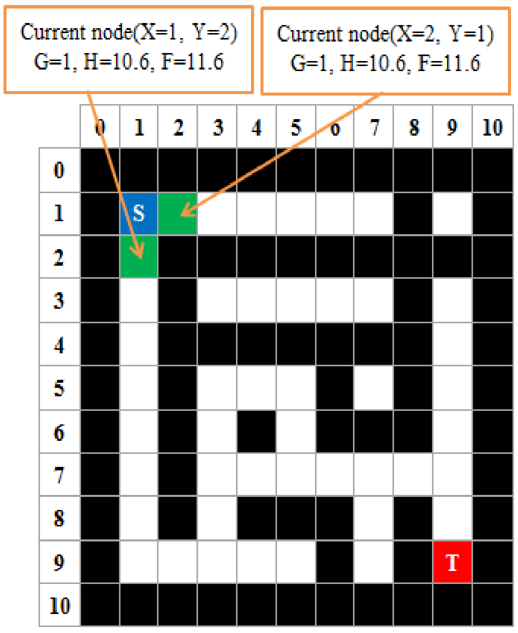
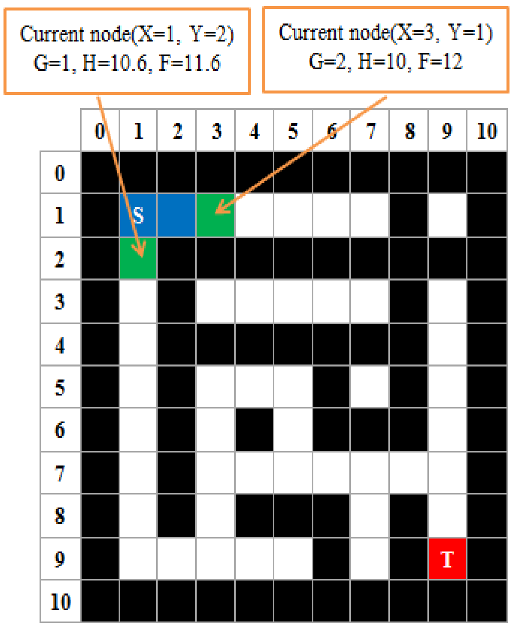


Figure 2.18: A Simple Maze Router Problem

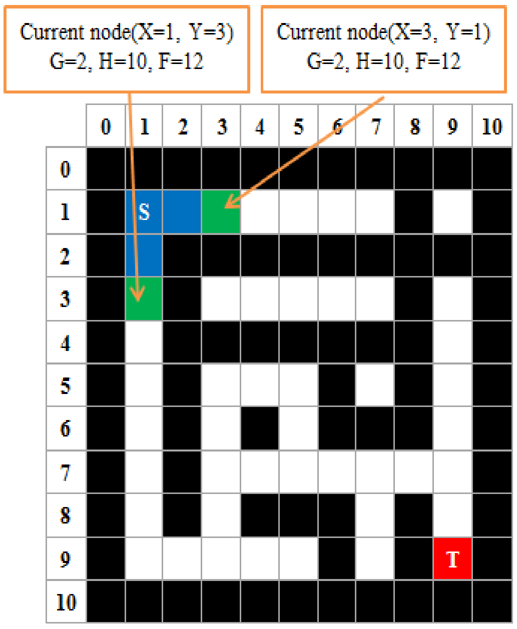
A* Algorithm Start with the Source node in the Open List and nothing in the Closed List. The first round of this Algorithm starts by processing our first node from the open list, which is the Source node and removes it from the Open List and appends it to the Closed List. Retrieve the list of neighbor's node and we start processing them. The Source node has four neighbor nodes whose coordinates are (*1, 0*), (*0, 1*), (*2, 1*) and (*1, 2*). (*1, 0*) and (*0, 1*) are the blocked node so we add to Close List. But (*2, 1*) and (*1, 2*) are reachable or not in the Closed List, so we process them. And calculate values of *G* and *H* for them. *G = 1* as we need to processing two node East and South from the Source node. Then found the value of *H = 10.6* by using Euclid distance, and use the value of *H* and *G* to find the sum value for *F = G + H = 1 + 10.6 = 11.6*. And choose the minimum cost for *F* between two nodes, if the numbers are equal we choose the node that is located on the eastern side. Then removed Source node from the Open List and add to Close List then add neighbor’s node to the open list as shown in Figure (2.19) from (a) to (b). Then continue with the node in the Open List having the lowest cost *F*(*n*)= *G* (*n*) + *H* (*n*). And remove the current node it from the Open List and add to Close List and add neighbor's nodes that have the minimum cost to Open List as shown in (c) of the Figure (2.19). The node with coordinates (*3, 7*) has three neighbors, one node in the Closed List and two nodes in the Open List: (*3, 6*) and (*4, 7*). The first node removed from the Open List is (*4, 7*) because F is equal to 18.3. This proves that this Algorithm is better as shown in the Figure (2.19) from (d) to (e). The next node processing in the Open List is (*9, 9*) coordinates and it is the Target node, so we have found our path. It is easy to display the path. We just have to follow the agent up to the starting node. Our path is highlighted in blue on the Maze as shown in (f) of the Figure 2.19.



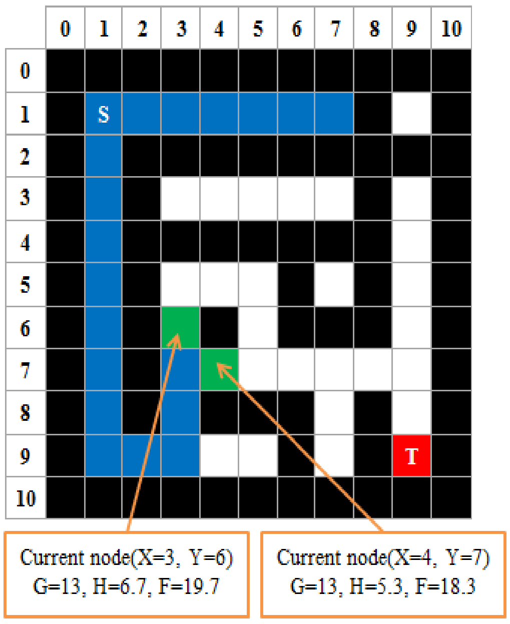
a. Add Source node to Close List and Process two neighbors node



b. Choose the East neighbor node and add current node to Close List

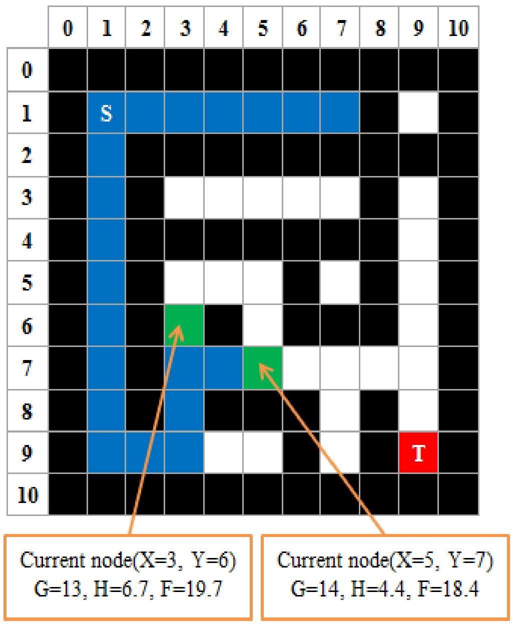


c. Choose minimum cost value

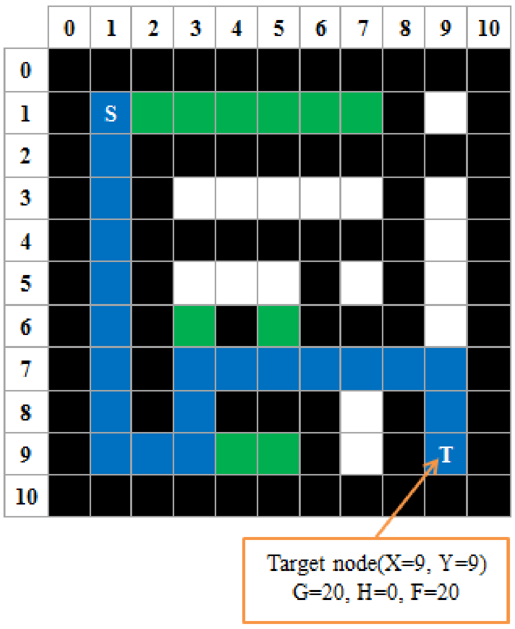


d. The current node has two node in the Open List

Figure 2.19.1: Illustration A* Algorithm step by step for solving Maze Router Problem



e. The East neighbor node has minimum cost value



f. Find Target node and Determine The shortest path

Figure 2.19.2: Illustration A* Algorithm step by step for solving Maze Router Problem

Chapter 4

EXPERIMENTAL RESULTS

4.1 Introduction

In this chapter, the experimental results show how to use the four Algorithms described in the previous chapter to solve the Maze Router Problem, and find shortest path *'S'* node to *'T'* node. Ten problems of different sizes were used. Only four of them will be given in this chapter.

4.2 Solutions to the given problems

In this part, four problems with different sizes are used to determine the Source node and Target node in solving Maze Router Problem. The results in the Table (1.5) shows the shortest distance and execution time of the Algorithms used for the solution of the given problems.

4.2.1 Problem1

The Problem 1 is a 5x5 sized of Maze Router Problem as shown in the Figure (3.1), with Source node located in (*x=1*, *y=1*) and Target node located in (*x=5*, *y=5*). In the process of finding the shortest path between *'S'* node and *'T'* node on a Maze, the Connected-Component Labeling, Depth-First Search, Lee and A* Algorithms have been used.

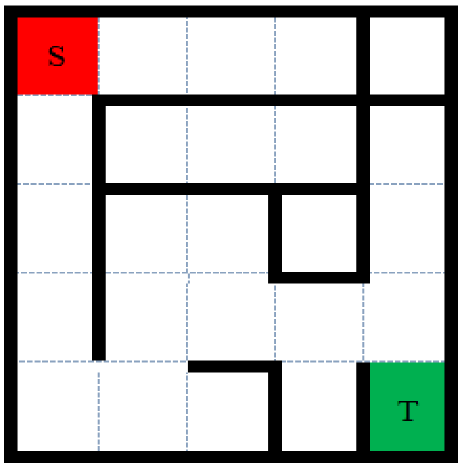


Figure 3.1: Maze Router Problem *1* with (*5x5*)

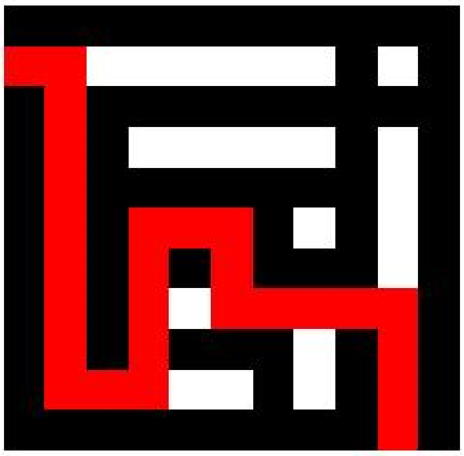


Figure 3.2: Maze Router Problem solved by CCL Algorithm

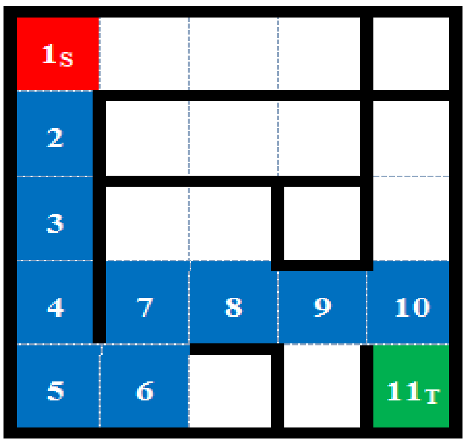


Figure 3.3: Maze Router Problem solved by DFS Algorithm

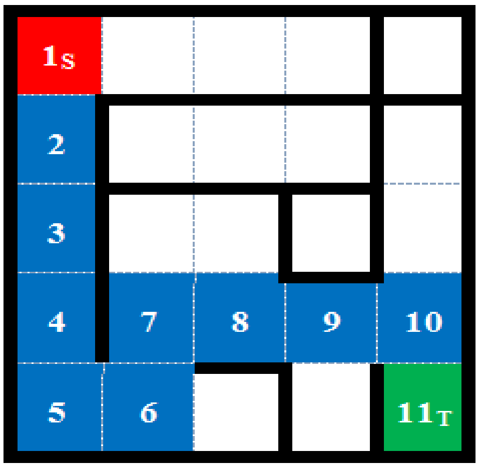


Figure 3.4: Maze Router Problem solved by Lee Algorithm

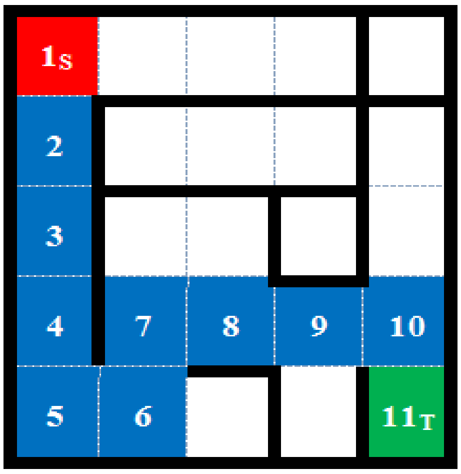


Figure 3.5: Maze Router Problem solved by A* Algorithm

Table 1.1: Comparison between Algorithms for Problem *1*

Problem *1*	CCL	DFS	Lee	A*
Distance	*13*	*11*	*11*	*11*
Elapsed time per second	*2.2744*	*0.0013*	*0.0886*	*0.0500*

From the Table (1.1) and Figures (3.2, 3.3, 3.4, 3.5) for problem *1*, we can observe that DFS, Lee and A* Algorithms can solve the Maze Router Problem with the shorts distance. As a result we can conclude from Table (1.1) that DFS solves the given problem in a fast way with shortest distance.

4.2.2 Problem 2

The Problem 2 is a *10x10* sized of Maze Router Problem as shown in the Figure (3.6), with Source node located in (*x*=*1*, *y*=*1*) and Target node located in (*x*=*10*, *y*=*10*). In the process of finding the shortest path between *'S'* node and *'T'* node on a Maze, the Connected-Component Labeling, Depth-First Search, Lee and A* Algorithms have been used.

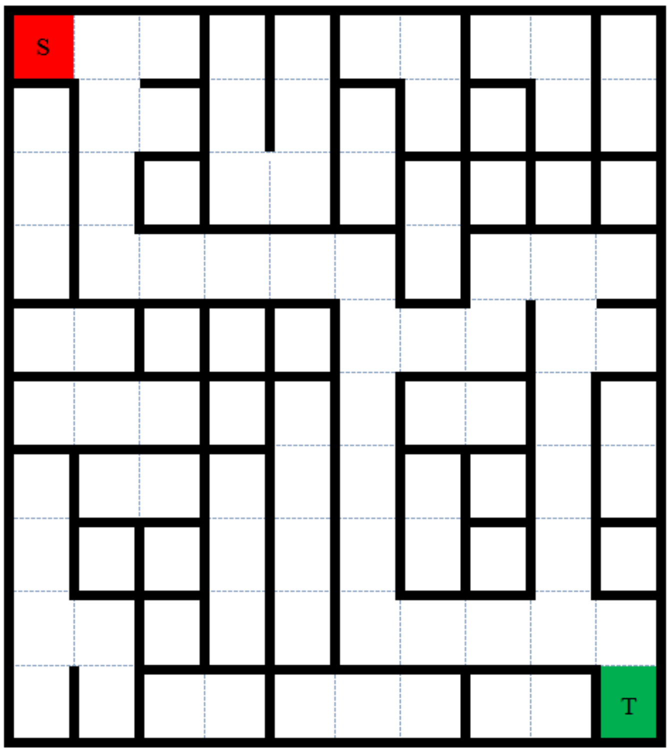


Figure 3.6: Maze Router Problem *2* with (*10x10*)

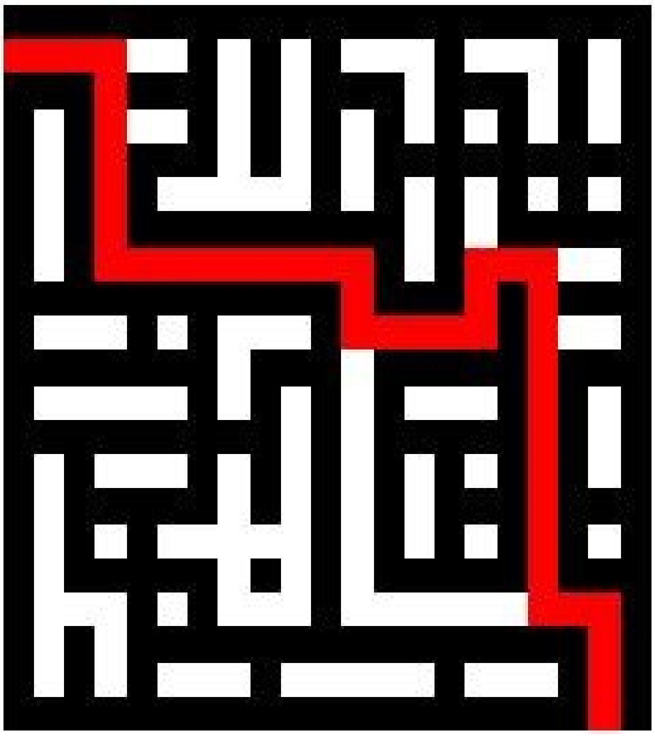


Figure 3.7: Maze Router Problem solved by CCL Algorithm

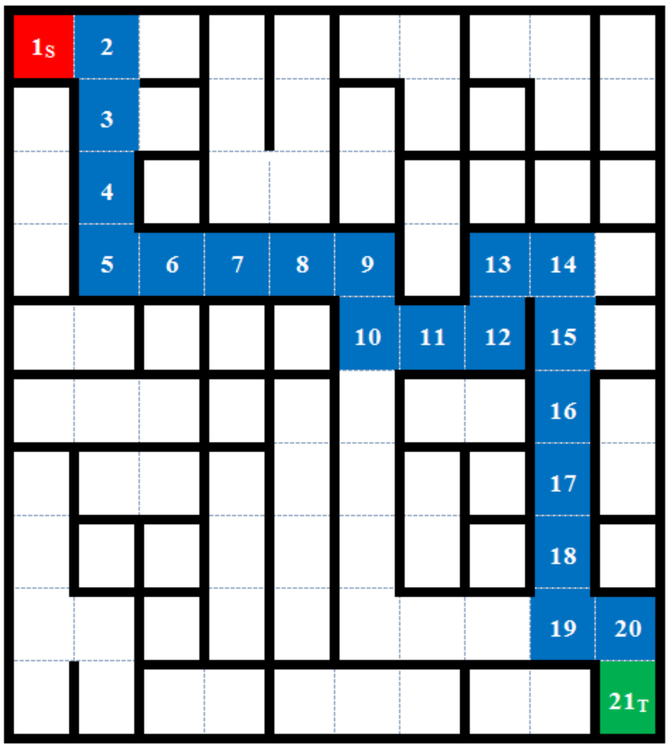


Figure 3.8: Maze Router Problem solved by DFS Algorithm

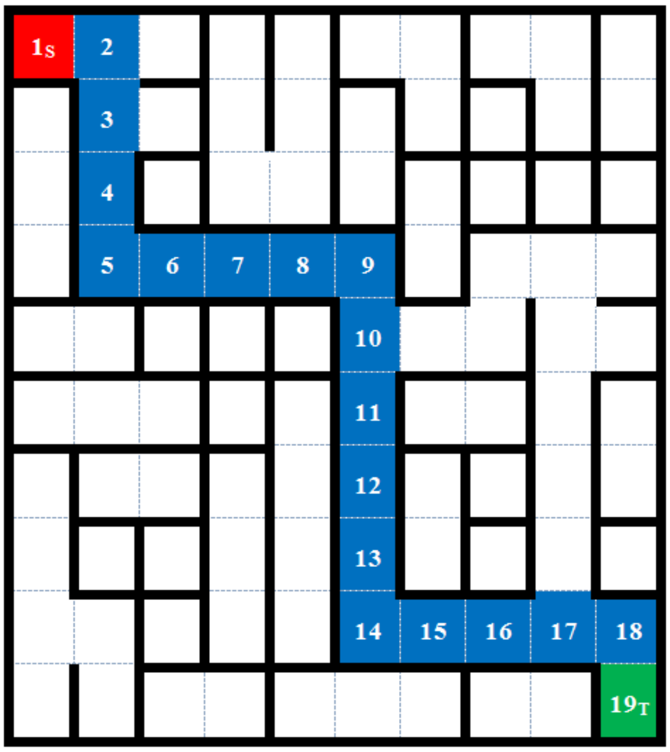


Figure 3.9: Maze Router Problem solved by Lee Algorithm

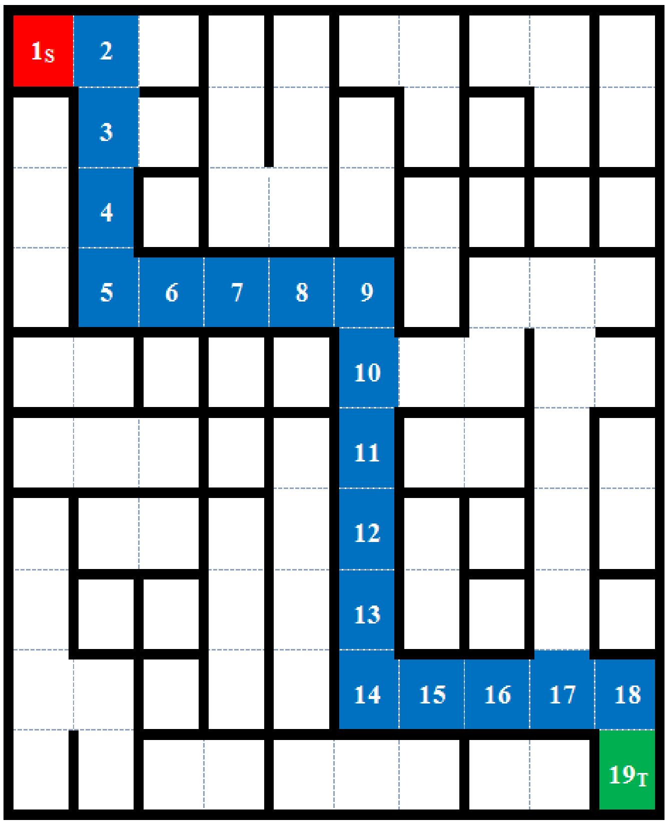


Figure 3.10: Maze Router Problem solved by A* Algorithm

Table 1.2: Comparison between Algorithms for Problem *2*

Problem *2*	CCL	DFS	Lee	A*
Distance	*21*	*21*	*19*	*19*
Elapsed time per second	*2.2044*	*0.0050*	*0.1138*	*0.0913*

From the Table (1.2) and figures (3.7, 3.8, 3.9, 3.10) above, we can observe that Lee and A* Algorithms solve the Maze Router Problem with the shorts distance. As a result, we can conclude from Table (1.2) that A* Algorithm solves the given problem in the fastest way with the shortest distance.

4.2.3 Problem 3

The Problem 3 is a 20x20 sized of Maze Router Problem as shown in the Figure (3.11), with Source node located in (*x=1*, *y=1*) and Target node located in (*x=20*, *y=20*). In the process of finding the shortest path between *'S'* node and *'T'* node on a Maze, the Connected-Component Labeling, Depth-First Search, Lee and A* Algorithms have been used.

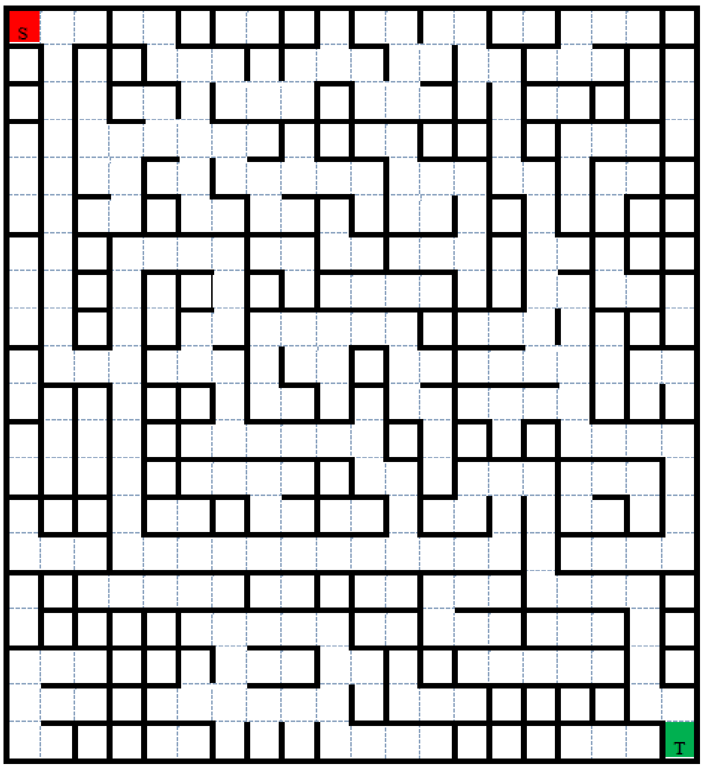


Figure 3.11: Maze Router Problem *3* with (*20x20*)

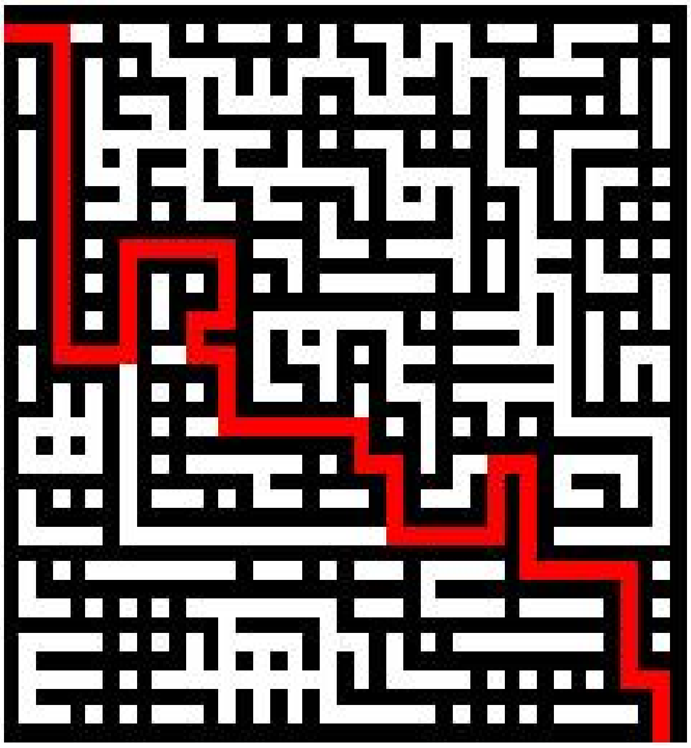


Figure 3.12: Maze Router Problem solved by CCL Algorithm

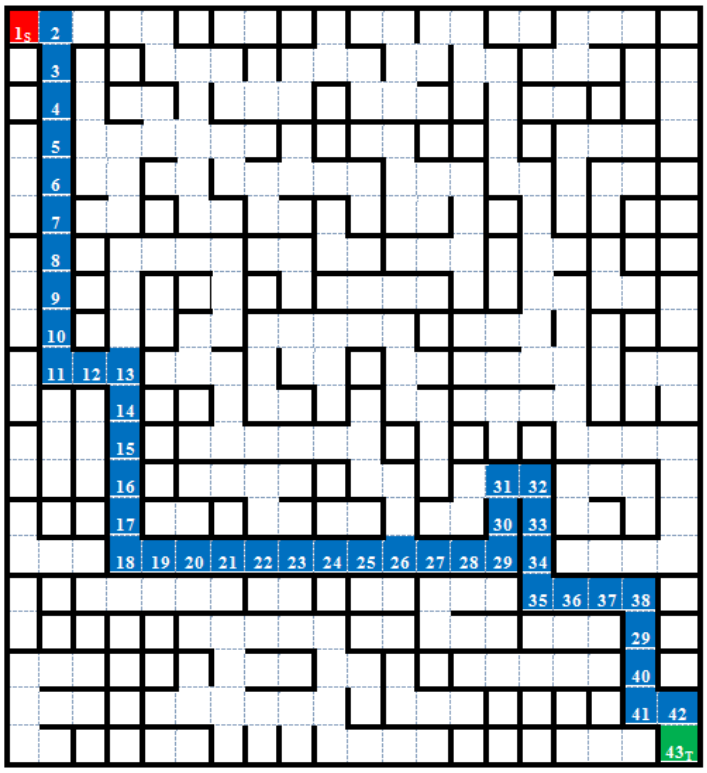


Figure 3.13: Maze Router Problem solved by DFS Algorithm

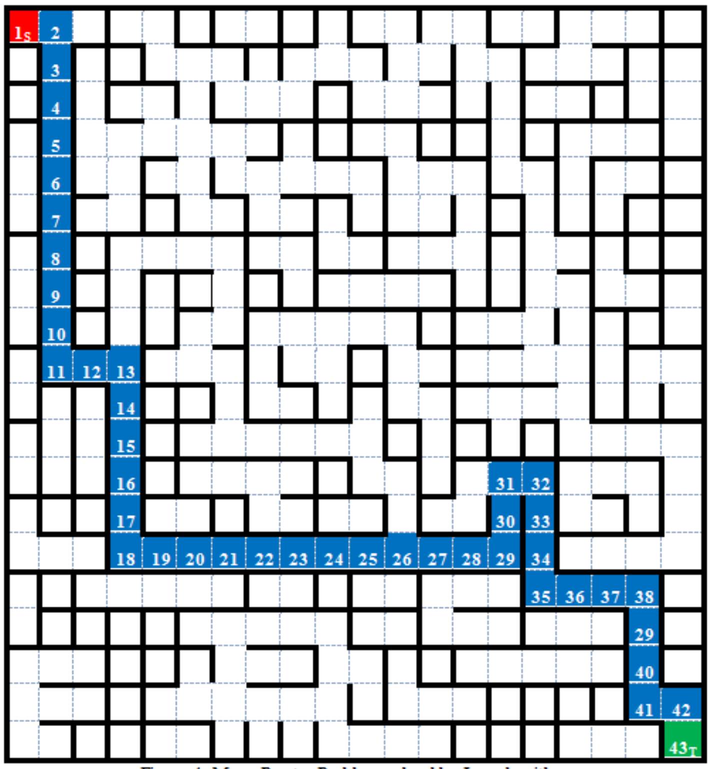


Figure 3.14: Maze Router Problem solved by Lee Algorithm

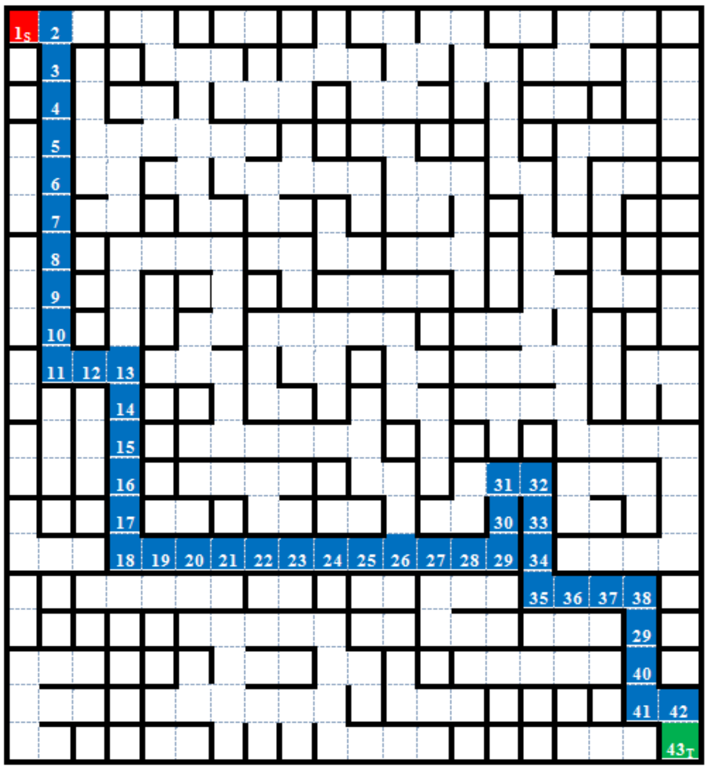


Figure 3.15: Maze Router Problem solved by A* Algorithm

Table 1.3: Comparison between Algorithms for Problem *3*

Problem *3*	CCL	DFS	Lee	A*
Distance	*51*	*43*	*43*	*43*
Elapsed time per second	*2.2852*	*0.0391*	*0.1730*	*0.1474*

From the Table (1.3) and Figures (3.12, 3.13, 3.14, 3.15) we can observe that DFS, Lee and A* Algorithms solve the Maze Router Problem with the shorts distance. As a result, we can conclude from Table (1.3) that DFS Algorithm solves the given problem in the fastest way with the shortest distance.

4.2.4 Problem 4

The Problem 4 is a *40x40* sized of Maze Router Problem as shown in the Figure (3.16), with Source node located in (*x=1*, *y=1*) and Target node located in (*x=40,* *y=40*). In the process of finding the shortest path between *'S'* node and *'T'* node on a Maze, the Connected-Component Labeling, Depth-First Search, Lee and A* Algorithms have been used.

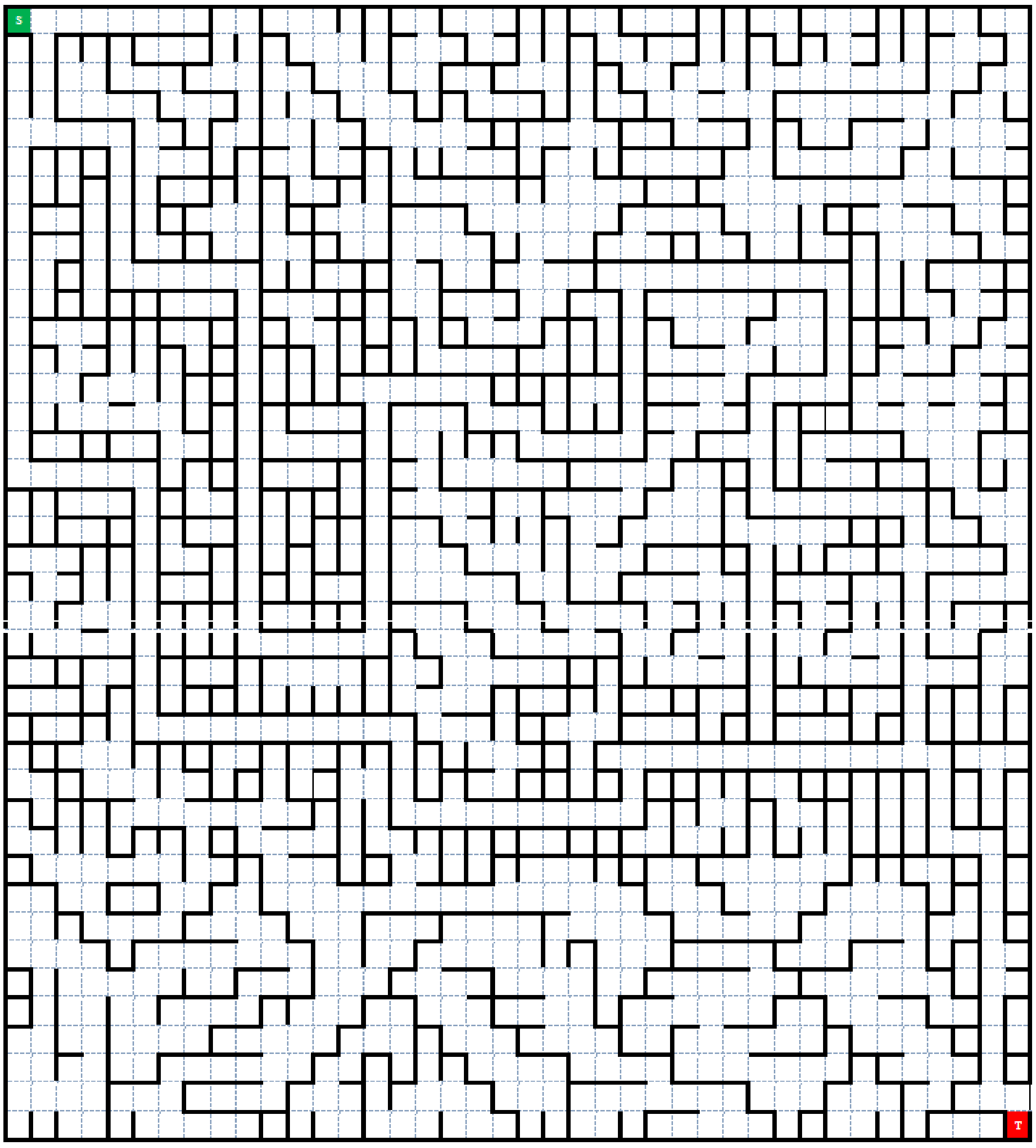


Figure 3.16: Maze Router Problem 4 with (40x40).

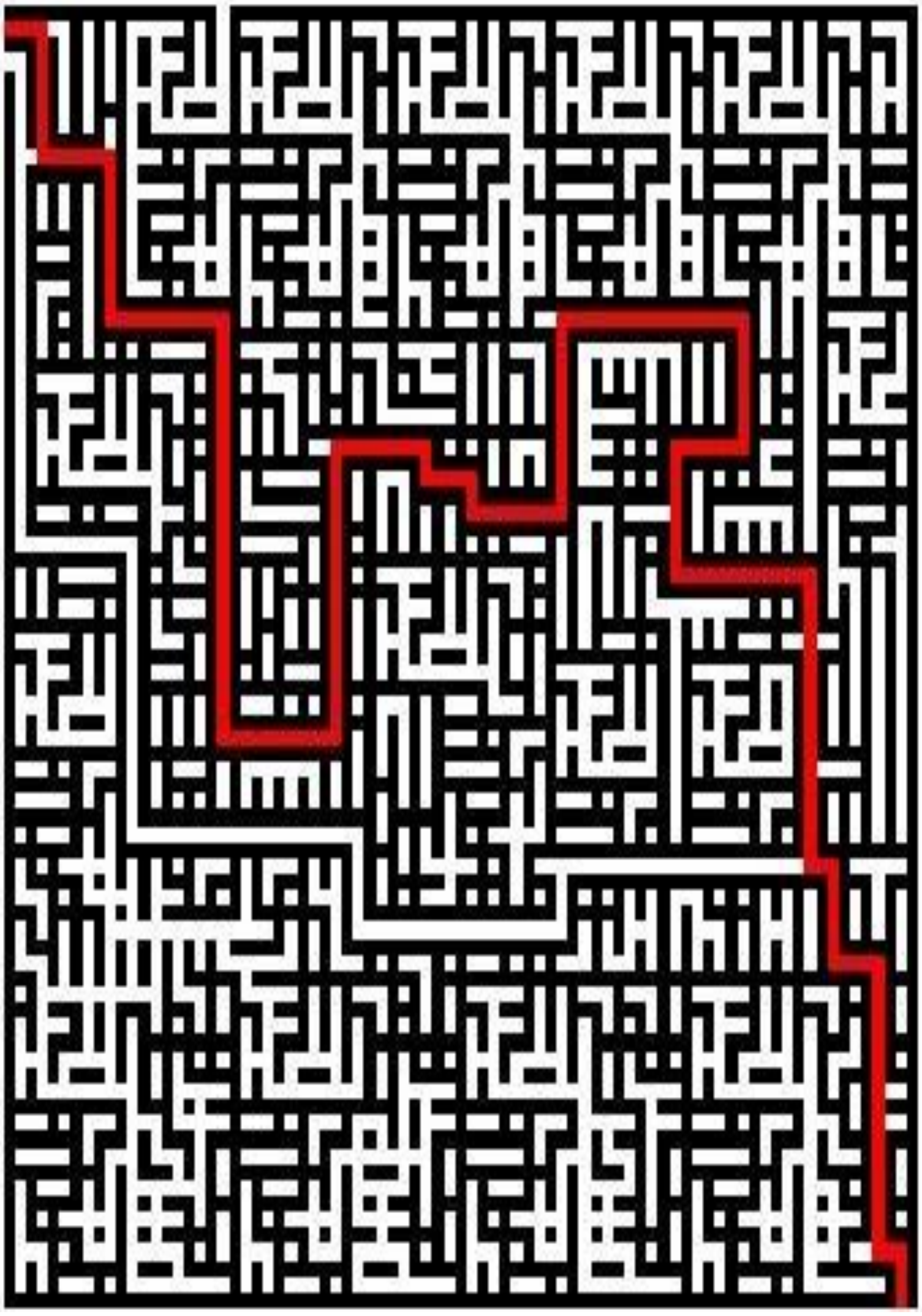


Figure 3.17: Maze Router Problem solved by CCL Algorithm

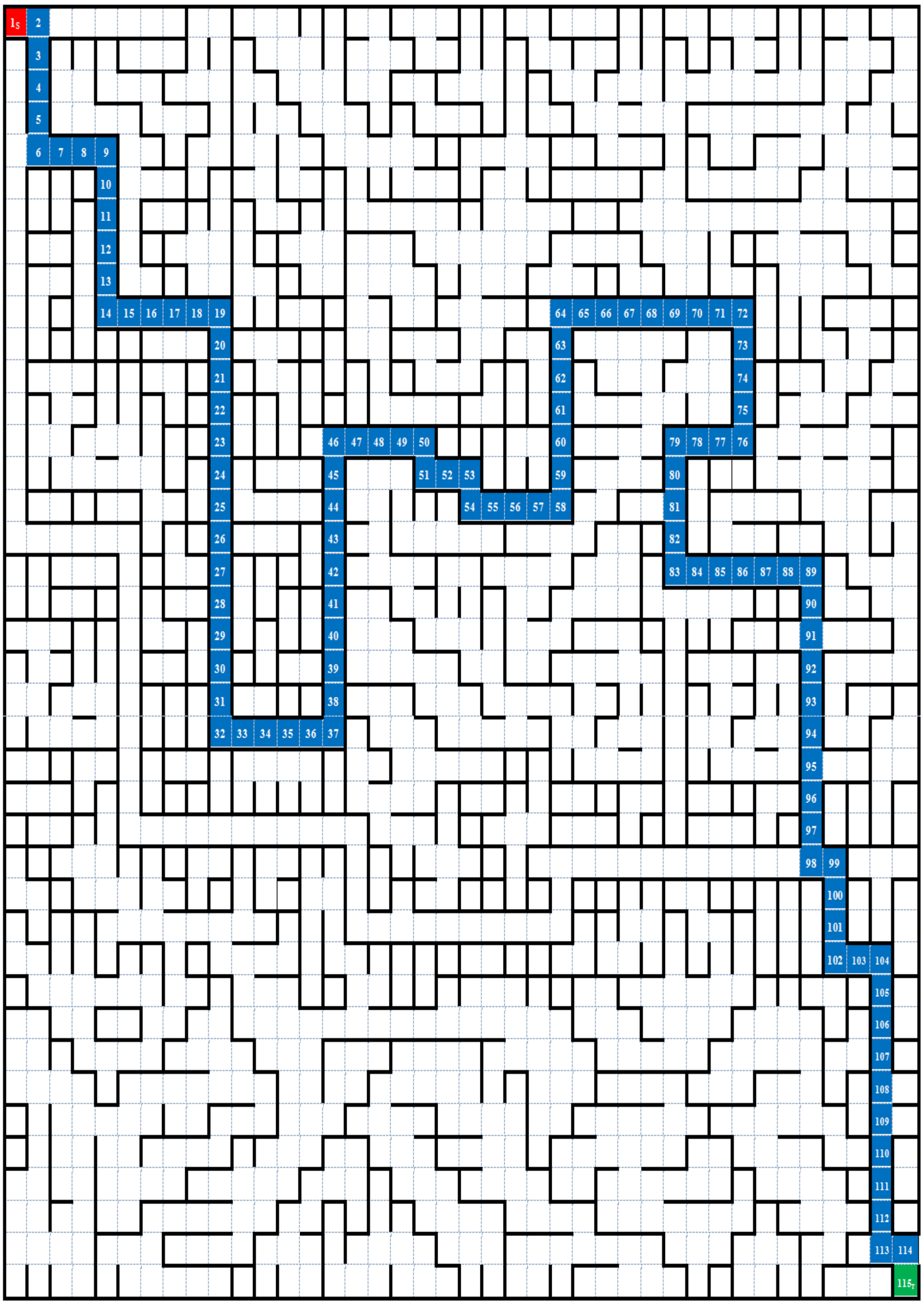


Figure 3.18: Maze Router Problem solved by DFS Algorithm

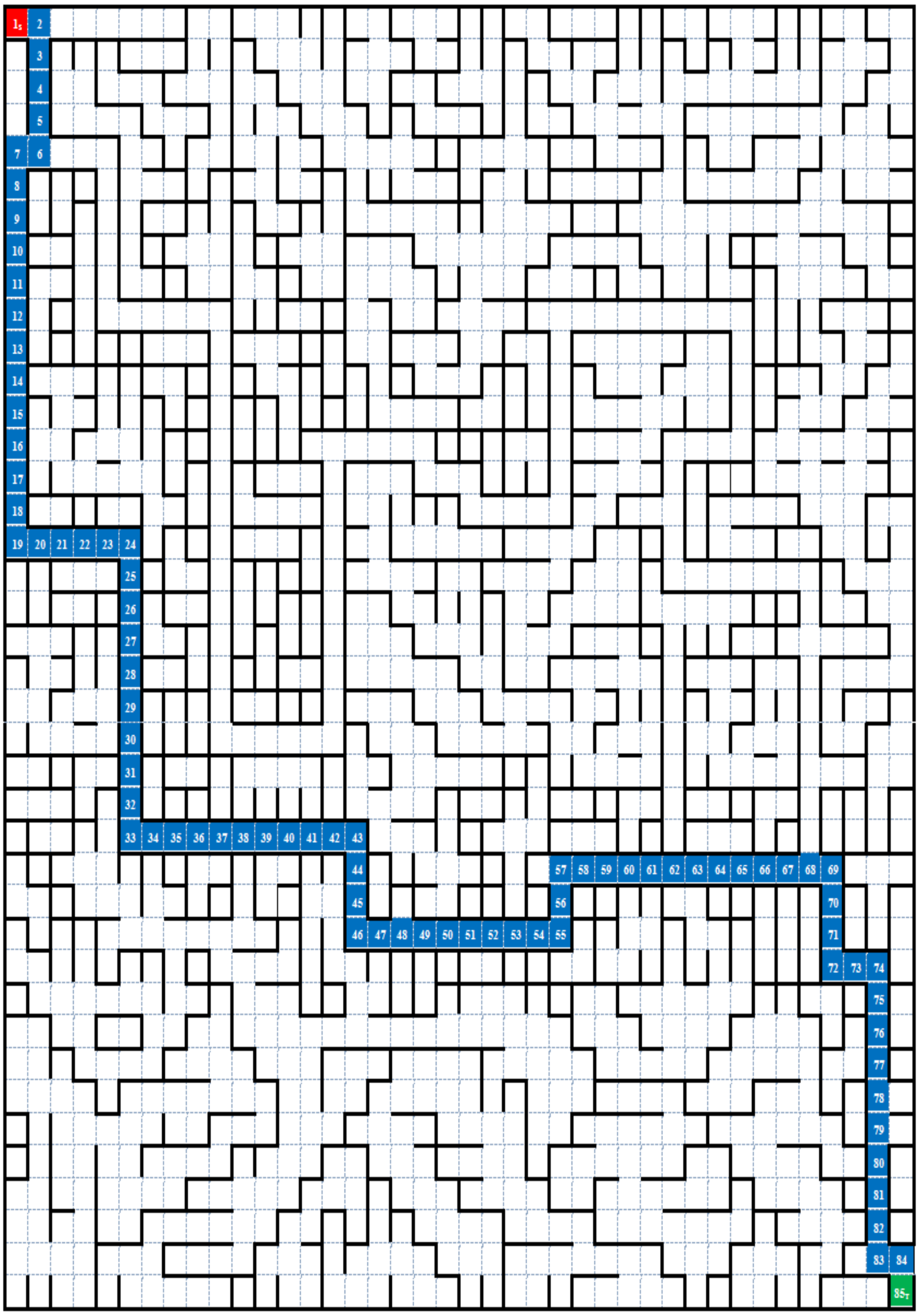


Figure 3.19: Maze Router Problem solved by Lee Algorithm

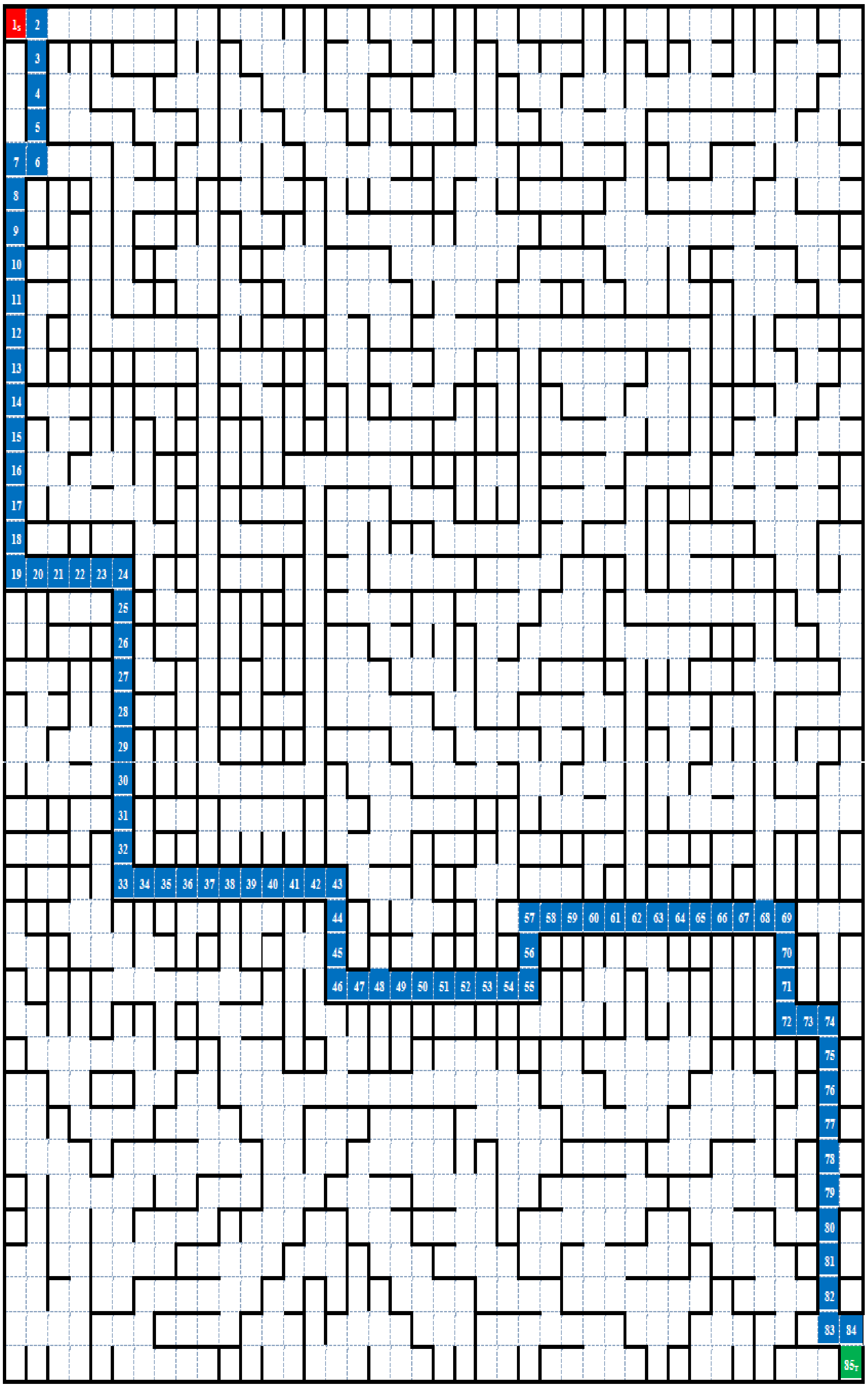


Figure 3.20: Maze Router Problem solved by A* Algorithm

Table 1.4: Comparison between Algorithms for Problem *4*

Problem *4*	CCL	DFS	Lee	A*
Distance	*115*	*115*	*85*	*85*
Elapsed time per second	*2.7019*	*0.5485*	*1.1926*	*0.7084*

From the results obtained, we notice that both A* and Lee Algorithms solves this problem with minimum distance. As a result, we can conclude from Table (1.4) that A* Algorithm solves the given problem in the fastest way with the shortest distance.

Table 1.5: Results for all problems

Problem		CCL	DFS	Lee	A*
Problem *1*	Distance	*13*	*11*	*11*	*11*
	Time	*2.2744*	*0.0013*	*0.0886*	*0.0500*
Problem *2*	Distance	*21*	*21*	*19*	*19*
	Time	*2.2044*	*0.0050*	*0.1138*	*0.0913*
Problem *3*	Distance	*51*	*43*	*43*	*43*
	Time	*2.2852*	*0.0391*	*0.1730*	*0.1474*
Problem *4*	Distance	*115*	*115*	*85*	*85*
	Time	*2.7019*	*0.5485*	*1.1926*	*0.7084*

The Table (1.5) shows the results of A* and Lee Algorithms applied for all the four problems. In each row of the table name of the problem is stated, and it shows that these Algorithms are finds the shortest path between two terminal nodes in all problems.

Chapter 5

CONCLUSION

In this thesis, we described the several Algorithms to solve the Maze Router Problem, and found the shortest path between the two terminal nodes, Source and Target. The Maze Router Problem was represented as a single-layer with the given size and terminal nodes. Furthermore, from the experimental results we can conclude that the Depth First Search, Lee and A* Algorithms are the most successful Algorithms in solving Maze Router Problem (MRP). However, Lee and A* Algorithms found the shortest path for all the problems in our experiments, moreover, A* Algorithm found the shortest path in the fastest possible ways in all the problems, while neither the Depth First search Algorithm nor the Connected Component Labeling Algorithm could not find the shortest path in all the problems. Moreover, the Connected Component Labeling Algorithm is very slow. Thus, the methods that we have used to solve the Maze Router Problem can only be implemented in the single-layer routing model.

REFERENCES

[1] Lee Krystek, “Amazing Mazes”, Last modified, in 2001. "http://www.unmuseum.org/maze.htm".

[2] Luigi Di S. and Andrea B., “A Simple and Efficient Connected Components Labeling Algorithm”, *IEEE Trans*, pp. 322–327, in Sep 29, 1999.

[3] Michael Dillencourt, “A General Approach to Connected-Component Labeling for Arbitrary Image Representations”, *Journal of the ACM (JACM)*, pp. 253-280, Vol. 39 Issue 2, in April 1992.

[4] Vytautas Čyras, “Artificial Intelligence: The Labyrinth Problem Depth First Search”, *Vilnius University: Faculty of Mathematics and Informatics*, pp. 28-30, in May 13, 2014.

[5] Thomas H., Charles E., Ronald L., and Clifford S., “Introduction to Algorithms, 2nd”, “MIT Press and McGraw-Hill”, “Depth First Search”, pp. 540–549, in 2001.

[6] Lee, Y., "An Algorithm for Path Connections and Its Applications", *IRE Transactions on Electronic Computers*, EC-10 (2): pp. 346–365, in 1961.

[7] Huang-Yu Chen and Yao-Wen Chang, “Global and Maze Routing”, *Verification and Test. America*, pp. 687-700, in 2009.

[8] P. H., N. N., and B. R., “A formal basis for the heuristic Determination of minimum cost paths”, *IEEE Trans*, Vol. SCC-4, pp. 100–107, in July 1968.

[9] R. F., ” The Lee path connection Algorithm”, *IEEE Trans. On computers*, VOL. c-23, no. 9, pp. 907-914, in Sept. 1974.

[10] H. S. and M. T., "Efficient Component Labeling of Images of Arbitrary Dimension Represented by Linear Bintrees", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, pp. 10-579, in 1988.

[11] H. C. Lee and Fikret Ercal, “Fast Algorithms for Maze Routing on an RMESH”, *Department of Computer Science University of Missouri*, in 1996.

[12] Yun-Ru Wu, Ming-Chao Tsai and Ting-Chi Wang, ”Maze Routing with OPC consideration”, *IEEE Trans*, Vol. 1, pp. 198 – 203, in Jan 18, 2005.

[13] R. Fisher, S. Perkins, A. Walker and E. Wolfart, "Connected Component Labeling", in 2003. *"http://homepages.inf.ed.ac.uk/rbf/HIPR2/label.htm"*.

[14] Charles P. Trémaux, “Trémaux tree and Planarity", *Electronic Notes in Discrete Mathematics*, pp. 169–180, in March 2008.

[15] Even, Shimon, “Graph Algorithms”, *Cambridge University Press*, pp. 46–48, in 2011.

[16] R. S., “Algorithms in C++: Graph Algorithms”, *Pearson Education*, in Jan 6, 2002.

[17] Donald K., “The Art of Computer Programming”, *Addison-Wesley*, in 1968.

[18] Kinniment D.J., "Performance comparison of conventional and backtracking Algorithms in circuit routing", *Electronic Circuits and Systems, IEE Proceedings G*, Vol. 127, Issue: 6, pp. 309 - 312, in Nov 11, 2008.

[19] Rob A. Rutenbar, ”ASIC Layout: Routing by Maze Search “, *CMU*, pp. 18-760, in Spring 1999.

[20] S. Hur, A. Jagannathan, and J. Lillis, “Timing-Driven Maze Routing”, *IEEE TRANS*, Vol. 19, pp. 234–241, NO. 2, in Feb 2000.

[21] Hightower and D., “The Lee router revisited”, *ICCAD*, pp.136-139, in 1993.

[22] Ian Watson, Chris Kirkham and Mikel Luján, “A Study of a Transactional Parallel Routing Algorithm”, *IEEE Trans*, pp. 388-398, in Sept. 2007.

[23] Daniel D., Peter S., D. Schultes and D. Wagner, "Engineering route planning Algorithms", *Springer Berlin Heidelberg*, Vol. 5515, pp. 117–139, in 2009.

[24] Z., W., Ch., R. L., "Finding shortest paths on real road networks: the case for A*", *International Journal of Geographical Information Science*, pp. 531–543 in 2009.

[25] H. E., Nilsson J. and Raphael B. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Trans on Systems Science and Cybernetics*, pp. 100-107, in Feb 12, 2007.

[26] R. Venkateswaran and P. Mazumder, “Routing Algorithms for VLSI Design”, University of Michigan, Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, in 1991.