UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior
Departamento de Tecnología Electrónica y de las Comunicaciones

CONTRIBUTIONS TO MOBILE ROBOT NAVIGATION BASED ON EMBEDDED SYSTEMS AND GRID MAPPING

PHD DISSERTATION

Nafiseh Osati Eraghi

Madrid, June 2015

Universidad Autónoma de Madrid
Escuela Politécnica Superior

PhD in Computer and Telecommunication Engineering

Contributions to Mobile Robot Navigation based on Embedded Systems and Grid Mapping

PHD DISSERTATION

Nafiseh Osati Eraghi

Co-directors:

Dr. Ángel de Castro Martín

Dr. Fernando Jesús López Colino

PhD Thesis

TITLE:	Contributions to Mobile Robot Navigation Based on Embedded Systems and Grid Mapping
AUTHOR:	Nafiseh Osati Eraghi
	Human Computer Technology Laboratory Escuela Politécnica Superior Universidad Autónoma de Madrid
ADVISORS:	Dr. Ángel de Castro Martín Universidad Autónoma de Madrid
	Dr. Fernando Jesús López Colino Universidad Autónoma de Madrid

PRESIDENT:

SECRETARY

VOCALS:

CALIFICATION:

Madrid, , 2015

Acknowledgments

When working on any project, no matter how individual it can be, it will almost certainly require assistance or encouragement from others. My project is no exception to this rule.

First of all I am extremely thankful to Professor Angel de Castro and Professor Fernando Lopez-Colino, for his guidance, support which opened my eyes for the mobile robot navigation of computer science.

I would also like to thank Professor Javier Garrido boss of HCTLab (Human Computer Technology Laboratory) and personnel in it for their cooperation.

Most importantly, I would like to thank my husband, Ali and my daughter, Shahrzad for making it possible for me to fulfil my dream, relieving me in many of life's problems as well as supporting me throughout my decisions. I have also to thank my mother for her tolerance and patience.

Abstract

Path planning is a problem as old as humankind. The necessity of optimizing the resources to reach a location has been a concern since prehistory. Technology has allowed approaching this problematic using new resources. However, it has also introduced new requirements.

This thesis is focused on path planning from the perspective of an embedded system using grid maps. For battery-dependent robots, path length is very relevant because it is directly related to motor consumption and the autonomy of the system. Nevertheless, a second aspect to be considered when using embedded systems is the HW requirements. These requirements comprise floating point units or storage capacity. When computer-based path planning algorithms are directly ported to these embedded systems, their HW requirements become a limitation. This thesis presents two novel path planning algorithms which take into account both the search of the shortest path and the optimization of HW resources. These algorithms are HCTNav and NafisNav.

The HCTNav algorithm was developed using the intuitive approach as trying to reach the goal in a straight trajectory until an obstacle is found. When an obstacle is found, it must be surrounded until the straight path to the goal can be continued, reaching this goal or another obstacle. Considering HCTNav as a path planning algorithm, both possible surrounding trajectories can be explored and then choose the best solution. Therefore, for each obstacle the algorithm finds, there is a branch in the search of the solution. Finally, the algorithm includes an optimization procedure which reduces the length of the obtained paths if it is possible to go between nonconsecutive waypoints in straight line.

The NafisNav algorithm evolves from a depth-first search. For each iteration of the algorithm, the straight trajectory to the goal position is verified. If this trajectory is not available, the algorithm selects from the unexplored neighbor cells the closest one to the target. If two neighbors were at the same distance, the algorithm would branch evaluating both alternatives. This algorithm includes a backtracking procedure just in case it finds a dead end. Finally, from every possible solution, the algorithm proposes the one that, after optimization, provides the shortest path.

The new algorithms have been evaluated and compared with the most extended algorithms of the state of the art: Dijkstra and A*. The two chosen evaluation metrics have been final path length and required dynamic memory. HCTNav provides an average penalization in the path length of 2.1% and NafisNav has this penalization increased to 4.5%. However, these algorithms present a decrease of the memory requirements of a 19% for HCTNav and of a 49% for the NafisNav algorithm.

Resumen

La planificación de una ruta es un problema casi tan antiguo como la humanidad. La necesidad de optimizar esfuerzos para alcanzar un objetivo ha sido una gran preocupación desde la prehistoria. La tecnología ha permitido abordar la solución de esta problemática con nuevos medios, pero también ha planteado otros requisitos distintos.

Esta tesis aborda el problema de navegación desde la perspectiva de los sistemas empotrados en entornos de mapas de rejilla. En todo robot dependiente de batería, la longitud final es un factor relevante porque se traduce directamente en el consumo de los motores y repercute en la autonomía del sistema. No obstante, un segundo factor que aparece al utilizar sistemas empotrados es el uso de recursos HW, ya sean unidades de coma flotante o capacidad de almacenamiento. Cuando se intenta adaptar los algoritmos diseñados para ser ejecutados en un ordenador nos enfrentamos a una gran demanda de estos recursos. La tesis plantea dos algoritmos novedosos que tienen en cuenta tanto la búsqueda de un camino lo más corto posible como la optimización de recursos HW: HCTNav y NafisNav.

El algoritmo HCTNav se desarrolló siguiendo el movimiento intuitivo de quien trata de ir en línea recta hasta que encuentra un obstáculo y lo rodea hasta que puede continuar en línea recta hasta el destino, o en caso contrario hasta otro obstáculo. Dado que se trata un algoritmo de planificación, se puede plantear rodear el obstáculo por ambos lados y elegir cuál es la mejor opción. Por lo tanto, cada obstáculo genera una bifurcación en la búsqueda de solución. Este algoritmo incluye un proceso de optimización por el que se reduce el recorrido final si se pueden saltar puntos intermedios viajando en línea recta.

El algoritmo NafisNav plantea una búsqueda en profundidad modificada. En cada iteración se intenta alcanzar el destino verificando si se puede alcanzar en línea recta. En caso de no poder alcanzarlo, se avanza al vecino, de entre los contiguos no explorados, aplicando un criterio de mínima distancia al objetivo. Si hubiera dos candidatos posibles, la búsqueda se bifurca, evaluando ambas opciones. Por último, se incluye un proceso de retroceso para el caso en el que se llegara a un punto sin salida. De entre las soluciones posibles se presenta aquella que, tras la optimización, obtiene el mínimo recorrido.

Los nuevos algoritmos han sido evaluados y comparados con los algoritmos más extendidos en el estado del arte: Dijkstra y A*. Los dos criterios utilizados han sido la longitud final del camino y el espacio de memoria que se necesita. HCTNav tiene una penalización promedio del 2,1 % en la longitud de la solución, mientras que NafisNav aplica una penalización promedio del 4,5 %. HCTNav obtiene una reducción del consumo de memoria del 19 % comparado con la mejor solución entre Dijkstra y A*. NafisNav mejora estos resultados con una reducción del 49 %.

General index

Figures index

Tables index

Pseudo code index

Chapter 1. Introduction and motivation

1.1. Problem Statement and Overview

There is no doubt about the importance of the ability to navigate in environments for mobile devices. Navigation prevents collisions and unsafe conditions. Most robots have as a goal finding special places in their own environment. Path planning is one of the most important issues in navigation that it is used in many applications, such as games, robotics or navigators. Path planning in robotics means the ability to determine the route from the robot's current position towards a goal position. The objective of path planning is to create a collision free path from the starting point to the given goal in an environment with obstacles. The most common objective in any navigation algorithm is to find the solution with the minimum path length. But nowadays low cost robots are commonly used. They are characterized by low energy supply and low computational resources. Therefore, there are three objectives in path planning: minimum path length, low energy consumption and low computational resources. Therefore, path planning for navigation in these robots is still an open challenge. Energy consumption is important, because recharging or changing the power supply or batteries can be difficult or even impossible. Computational resources include processing and memory capacity. Due to hardware limitations, memory usage is limited on many low cost robots and processing necessities has a direct impact on execution time. Depending on the application, the trade-off between the shortest path, CPU time and memory usage changes considerably. It is not the same problem to navigate using a computer or console than using an embedded system. Also, execution time is the least important parameter in this case as the execution time of the algorithm in the μprocessor is not significant compared to the robot's movement time. Therefore, the two most important aspects in embedded systems are memory consumption and path length.

This study aims to provide a review of the current state of the art on mobile robot navigation on embedded systems and grid mapping as well as to propose new path planning algorithms intended for embedded systems, comparing them to classic algorithms.

1.2. Thesis Objectives

The main objective of this Ph.D. thesis is the optimization of navigation algorithms for low-cost and low-resourced robots. Navigation is a well-known research field, and there are many works related to path planning and algorithms for navigation optimization. However, when we deal with low-cost robots, the existing solutions are hard to fit and to make them work within the specifications of this kind of robots.

The first objective is to investigate on the path planning algorithms and their classification. This objective will require a deep study on path planning algorithms for navigation.

The second objective is the modelling environment for path planning algorithms that is used in navigation of the mobile robots.

The third objective is the optimization, if possible, of existing path planning and navigation algorithms. The existing approaches have been proven to find the shortest path between two positions. The objective is to optimize the standard implementation of these algorithms to the limitations of the robots.

The forth objective is the design and implementation of new navigation algorithms especially created for mobile robots with the objective of minimizing memory consumption while keeping an acceptable path length.

1.3. Thesis Contributions

The main contributions of this thesis are:

- HCTNav algorithm: This algorithm is a new path planning algorithm aiming to minimize hardware resources. This algorithm finds in most cases the shortest path from the origin node to the target node without collision with obstacles.

- NafisNav algorithm: This algorithm is also a new path planning algorithm. The results of this algorithm show a trade-off between path length and memory usage.
- Comparison of HCTNav and NafisNav algorithms with Dijkstra and A* algorithms: The minimum path length and the maximum dynamic memory of the proposed algorithms are compared with the results of Dijkstra and A* algorithms.

1.4. Thesis organization

This thesis addresses path planning algorithms that are one of the most important issues in navigation. For a better understanding, different types of path planning algorithms and environment modelling are introduced at the beginning. Then two new path planning algorithms are presented and the implemented algorithms are individually tested and experimental results are given in each chapter. In the final chapters of this thesis results are discussed and the conclusions are drawn.

- **Chapter 2: State of the art**
  This chapter provides an introduction to path planning algorithms by presenting two different types of these algorithms. Each type introduces important and common algorithms that belong to that category. Examples and pseudo codes are enumerated for better understanding of each algorithm.
- **Chapter 3: Environment modelling**
  This chapter considers the environment modelling for navigation of the mobile robot. For this purpose a grid map is defined and then there is a discussion on graph generation from a grid map.
- **Chapter 4: HCTNav algorithm**
  This chapter describes a novel algorithm for path planning designed for low cost robots. The different parts of this algorithm are implemented and defined along with pseudo code of each part. Also the experimental results are shown. At the end of this chapter details about anomalies in path length are discussed through examples.
- **Chapter 5: NafisNav algorithm**
  Another new path planning algorithm is presented in this chapter. This algorithm focuses even more on reducing memory consumption, although at the expense of path length. Its procedures are introduced and implemented. The method of this algorithm is shown by some examples. This chapter is finished with experimental results.

- **Chapter 6: Conclusions and future work**

  This chapter resumes the most important aspects of the accomplished work. Additionally suggestions for possible future research are provided.

- **Appendix A: Small maps set**

  This appendix includes figures of 42 maps.

- **Appendix B: Big maps set**

  The Big maps set are presented in this appendix. It includes 12 maps.

Chapter 2. State of the art

The ability to navigate in environments is important for mobile devices. Navigation prevents collisions and unsafe conditions. Most robots have as goal finding special places in their own environment. Path planning is a component of robot’s navigation, so that it is a main issue among robot control research areas. In a path planning algorithm, there is a map of the environment with some obstacles. The mobile robot moves from the start point towards the target point and attempts to find the optimal collision-free path between them. In most path planning algorithms, the most important problem is to generate a collision-free path from the origin point to the goal point.

Some of the path planning algorithms is sensor-based. These algorithms divide into two branches: Global path planning algorithms and Local path planning algorithms. The global path planning algorithms use genetic algorithms and A* algorithms. Global path planning algorithms need complete information of the environment and create a full path from the start point towards the target point before the robot starts to move. In other words, first the mobile robot builds a global model based on sensor data and second uses it for the path planning algorithm [1, 2]. In this case, the robot has to do a heavy computational work. One of the global path planning models is the Piano’s mover problem [3] so that complete information is on the geometry. In the Piano’s mover problem, it is assumed a computer-aided design (CAD) model of a house and a piano as input for the algorithm. The piano should be transferred from one room to another room without hitting anything. The robot path planning usually ignores dynamics and other limitations. It is focused mainly on moving and rotating the piano without any collision for transferring from the start point to the target point. The new approaches used in global path planning are Genetic Algorithm [4, 5], Neural Network [6] and Ant Colony Optimization (ACO) [7, 8].

Local path planning algorithms [9, 10] create paths while the robot is moving. Indeed, the algorithm can create a new path responding to changes in the environment. There is a simple assumption in local path planning algorithms. If there are no obstacles: the shortest path from the start point towards the target point is a straight line. It means, the robot will move along this path until an obstacle is found. In here, the path planning algorithm is used for continuing the path towards the target point. The local path planning algorithms are simpler than the global path planning algorithms regarding to their implementation.

There are many different algorithms focused on path planning and navigation. In the local path planning algorithms, the robot moves aside of the obstacle when an unknown obstacle is found. Obviously, each of these algorithms has their own advantages [11].

This thesis focuses on different algorithms of path planning algorithms and then it will apply two approaches for solving the path planning problem: 1) HCTNav algorithm and 2) NafisNav algorithm. Both algorithms belong to global path planning algorithms for mobile robots, so the map is known a priori.

The remainder of this chapter is organized into two sections. Global path planning algorithms are presented in section 2.1 and the local path planning algorithms are described in section 2.2 respectively.

2.1. Global path planning algorithms

These algorithms use a graph search algorithm for finding a path from the start point to the target point in the visibility graph. A visibility graph [12] is defined in path planning of robots and it is a graph of visible locations that includes set of obstacles and points in a real map. Each node of graph is as a point location and each edge is as a visible connection between them so that any obstacle does not go through between them in corresponding graph. The visibility graph is used for determining the shortest path. A basic algorithm for searching in a graph is the depth/breadth-first search. Dijkstra and A* algorithms are usually used for finding the shortest path. In global path planning algorithms a graph is completely known before navigation starts. Also they belong to global path planning algorithms and Dijkstra and its variations, such as A*, are most well-known algorithms in path planning.

The reminder of this section is organized as follows. The depth first search algorithm is described in subsection 2.1.1 and the breadth first search algorithm is presented in subsection 2.1.2. Subsection 2.1.3 is shown the Dijkstra algorithm and subsection 2.1.4 is described the A* algorithm.

2.1.1. The depth first search algorithm

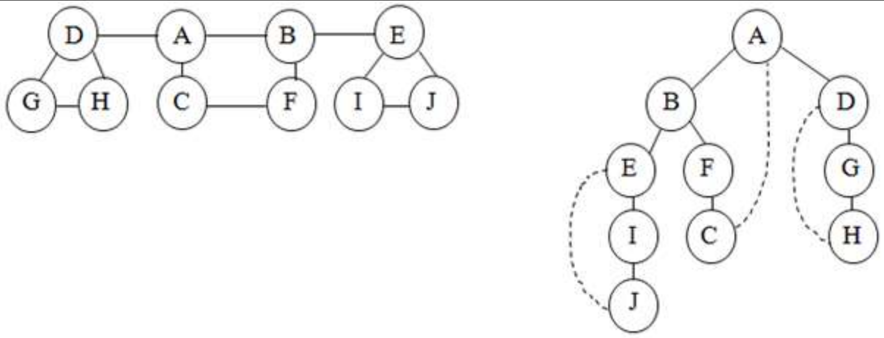
The depth first search is a search algorithm for graphs or trees [13, 14]. This algorithm searches from the start point as far as possible in each branch before backtracking. This algorithm prioritizes the search of a node's descendants before exploring its neighbors. The algorithm uses a heap pushing current node's descendants and continues with the heap's top node. It means that if it is seen an unvisited node, it puts the node in heap and when backtracking is done, the node is ejected from the heap. Therefore during execution of the algorithm, first element from the heap is considered. The pseudo code of algorithm is shown in code 2.1 for two cases, 'recursive' and 'non-recursive':

Code 2.1. The depth first search algorithm pseudo codes

```
Input: a graph G, vertex v
Procedure DFS(G,v): (recursive)
        label v as visited
        For all edges from v to w in G.neighborEdges(v) do   //w is a vertex
             If vertex w is not labeled as visited then
                  call DFS(G,w)
output: all vertices from v as visited
```

```
Input: a graph G, vertex v
Procedure DFS(G,v): (non-recursive)
        Heap S;
        push(v) in S
        While S is not empty
             v ←S.pop()
             If (v is not labeled as visited)
                label v as visited
                For all edges from v to w in G.neighborEdges(v) do  //w is a vertex
                   S.push(w)
output: all vertices from v as visited
```

In a graph as G(V, E), with a set of vertices as V and a set of edges as E, time complexity includes two parts: 1) time complexity for exploring of all vertices is O(|V|), 2) time complexity for exploring both sides of all edges is O(|E|). Thus the time complexity of this algorithm is linear by size of its input and usually takes O(|V|+|E|). The result of exploring vertex A on its corresponding graph is shown in fig 2.1. The dotted edges aren't considered for main tree obtained from graph, because their vertices are previously visited and they are called back edge.



Order of visitation	Queue contents after processing node
	[A]
A	[BCD]
B	[CDEF]
C	[DEF]
D	[EFG]
E	[FGI]
F	[GI]
G	[IH]
I	[HJ]
H	[J]
J	[ ]

Fig 2.1. The result of explore from vertex A in a depth-first search of the graph [13]

2.1.2. The breadth first search algorithm

The breadth first search (BFS) is a search algorithm for trees or graphs [14]. Also it is an extremely useful searching technique and an archetype for many important search algorithms. It uses a queue to perform the search. The queue works according by FIFO (First Input, First Output). This algorithm works on directed graphs and also undirected graphs. It explores from the root of tree or a start node of a graph and then its adjacent nodes. In the next step, it explores the next level of these adjacent nodes. The breadth first search considers all edges that have the same length than the previous level.

The breadth first algorithm acts layer by layer and in this algorithm, two layers are active at any time. One layer is in layer d and another one is in layer d+1.

The pseudo code of the breadth first search algorithm [13] is shown in code 2.2.

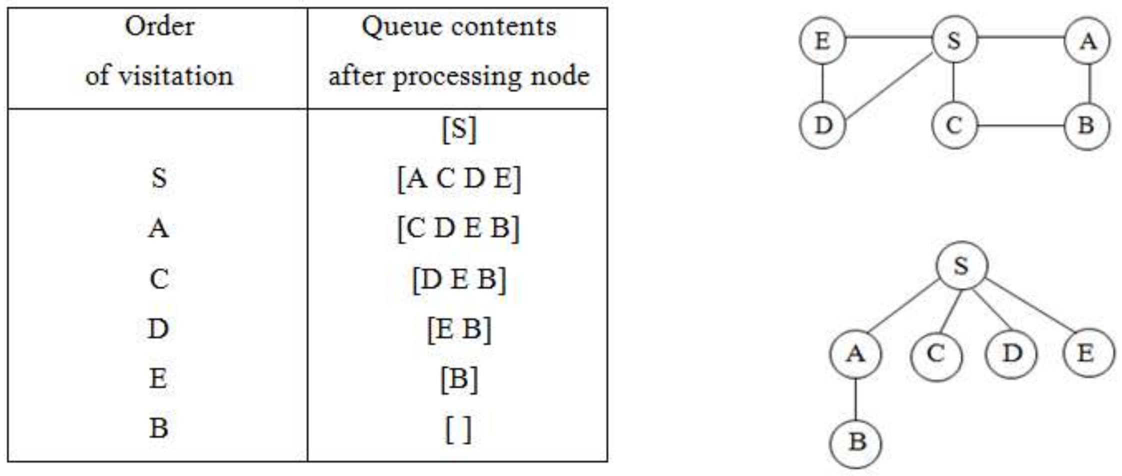
If G as a graph with a set of vertices as V and a set of edges as E, so the time complexity of this algorithm is O(|V|+ |E|), that is linear time. Each vertex is put on the queue only one time, when it is first met, therefore there are 2|V| queue operations. The rest of operations are done in an internal loop. In the execution time, this loop considers each edge only once and therefore the time complexity is O(|E|).

The result of the breadth first search on a graph and its corresponding tree is shown in fig 2.2.

Code 2.2. The breadth first search algorithm pseudo code [13]

```
Input: Graph G = (V,E) directed or undirected and vertex s ϵ V
For all u ϵ V:
    dist(u) = ∞

dist(s) = 0
Q = [s]   (queue containing just s)
While Q is not empty:
    u = eject (Q)
    For all edges (u,v) ϵ E:
        If dist(v) =∞:
            Inject (Q,v)
            dist(v) = dist(u) + 1        // dist(u) is the distance from s to u
Output: All vertices u reachable from s.
```



Order of visitation	Queue contents after processing node
	[S]
S	[A C D E]
A	[C D E B]
C	[D E B]
D	[E B]
E	[B]
B	[ ]

Fig 2.2. The result of breadth first search on the graph [13]

2.1.3. The Dijkstra algorithm

Dijkstra algorithm, named after its inventor, Edsger W. Dijkstra in 1956 and published in 1959 [15]. It is a graph search algorithm for solving the problem of finding the shortest path between the origin and the target nodes in a graph with non-negative edge path costs, by creating a shortest path tree. This algorithm is often used as a base routine in navigation of maps on real world. The Dijkstra algorithm is a graph search algorithm similar to BFS and DFS. The Dijkstra algorithm works on non-negative weighted graphs whereas BFS and DFS work on unweighted graphs. Also, the Dijkstra algorithm uses a priority queue instead of a regular queue used in BFS and it prioritizes nodes while it considers ledge lengths. Note that DFS uses a stack.

Description of the algorithm is as follows:

Suppose there are some nodes in a graph and distance between some nodes is given. It is asked to find the shortest path from the origin node to the target node. In this algorithm, three values for each node are used.

1. Distance value: It is the distance to the origin node.
2. Previous value: The previous visited node.
3. Visited value: The state of the node, if it is visited or it is unvisited.

For initialization in the first stage, distance value is defined zero for the origin node and previous value undefined. Second stage, for rest of nodes, distance value and previous value are determined to infinity and undefined respectively and all nodes are unvisited. In the next stage, for all the nodes is done the following process:

First, a node with unvisited value is selected with the minimum distance value. Next, its visited value changes to visited state. Then for all unvisited neighbors of this node, distance value of neighbor is changed with the distance of node in addition distance from this neighbor to the node, if this distance value was less than previous distance value of neighbor. Also the previous value of neighbor is modified to the selected node. This process is continued until the selected node is the target node. In this stage, the shortest path creates by this iteration: in the first, the current node is the target node, and as long as the previous value is defined, the current node is inserted to begin of path and then the current node is changed to the previous value.   At the end, there is the shortest path from the origin node to the target node.

Example 2.1: Finding the shortest path is shown between from (a) to (h) by Dijkstra algorithm. It is applied Dijkstra algorithm according to the above description. The graph, the origin and the target nodes are indicated as fig. 2.3.

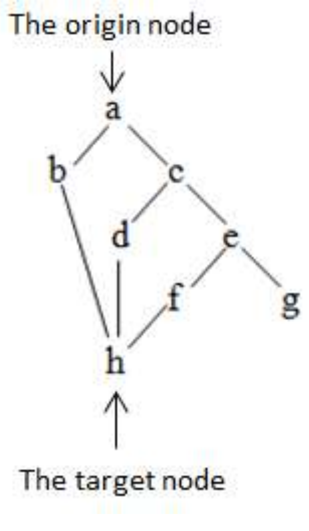


Fig. 2.3: the origin node and the target node in a graph

Suppose the distance between each two neighbor nodes is unit (D=1). The origin node is (a) and the target node is (h). The object is to find the shortest path from (a) to (h) by Dijkstra algorithm. The previous as (P), distance as (D) and visited as (V) values are determined for every node according by bellow table:

	a	b	c	d	e	f	g	h
P	u	u	u	u	u	u	u	u
D	0	∞	∞	∞	∞	∞	∞	∞
V	f	f	f	f	f	f	f	f

Value ‘u’ is used instead of undefined, value ‘f’ as false and value‘t’ as true. According to the algorithm, the unvisited node with the minimum distance is selected. So, in here the origin node (a) is selected and its visited value (V) is changed to visited or‘t’ (true). (a) has only two unvisited neighbors (b) and (c). For the first neighbor (b): D(a)+D(a, b)=0+1=1 , D(b)= ∞ and 1<∞ and so D(b)=1. Also P(b) = a. For the second neighbor (c): D(a)+D(a, c)=0+1=1 , D(c)= ∞ and 1<∞ and so D(c)=1 , also P(c) = a.

	a	b	c	d	e	f	g	h
P	u	a	a	u	u	u	u	u
D	0	1	1	∞	∞	∞	∞	∞
V	t	f	f	f	f	f	f	f

This process is continued according by follows:

From {b, c, d, e, f, g, h} (c) is selected and (c) has two unvisited neighbor (d) and (e). Also V(c) = t.

D(c)+D(c, d)=1+1=2, 2<∞ $\rightarrow$ D(d)=2, P(d)=c.

D(c)+D(c, e)=1+1=2, 2<∞ $\rightarrow$ D(e)=2, P(e)=c.

	a	b	c	d	e	f	g	h
P	u	a	a	c	c	u	u	u
D	0	1	1	2	2	∞	∞	∞
V	t	f	t	f	f	f	f	f

From{b, d, e, f, g, h} (c) is selected V(b)= t, D(b)+D(b, h)=1+1=2, 2<∞ $\rightarrow$ D(h)=2, P(h)=b.

	a	b	c	d	e	f	g	h
P	u	a	a	c	c	u	u	b
D	0	1	1	2	2	∞	∞	2
V	t	t	t	f	f	f	f	f

From {d, e, f, g, h} (d) is selected V(d)= t,

D(d)+D(d, h)=2+1=3, 3>2 $\rightarrow$ previous value(h) and distance value (h) aren't changed.

	a	b	c	d	e	f	g	h
P	u	a	a	c	c	u	u	b
D	0	1	1	2	2	∞	∞	2
V	t	t	t	t	f	f	f	f

From{e, f, g, h} (e) is selected V(e)= t, D(e)+D(e, f)=2+1=3 , 3<∞ $\rightarrow$ D(f)=3 , P(f)=e.

D(e)+D(e, g)=2+1=3, 3<∞ $\rightarrow$ D(g)=3, P(g)=e.

	a	b	c	d	e	f	g	h
P	u	a	a	c	c	e	e	b
D	0	1	1	2	2	3	3	2
V	t	t	t	t	t	f	f	f

From {f, g, h} (h) is selected V(h)= t, but it is the target node. So the shortest path must be created.

The current node=the target node (h),

P(h) = defined, Path= {h}, the current node= (b)

P(b) = defined, Path = {b, h}, the current node= (a)

P(a) = undefined,

Therefore the shortest path is equal to {a, b, and h}.

Dijkstra algorithm obtains the shortest path in a weighted graph with vertices V and edges E [16]. Also this graph can be directed. Denote G= (V, E) as a weighted graph. In this algorithm, all edge weights must be nonnegative that w(u, v)≥0 for each edge (u, v)€ E, so u and v are the two nodes of edge. The run time of Dijkstra algorithm depends on implementation of the min-priority queue. If Dijkstra algorithm doesn't use the min-priority queue, the run time is $O(V^2 + E) = O(V^2)$. But if it uses the min-priority queue with a Fibonacci heap, the run time is O(VlogV + E). Therefore the run time for Dijkstra algorithm is lower than the Bellman-Ford algorithm [17]. This algorithm is very similar to Dijkstra algorithm, being the main difference that, the edge weights may be negative and its run time is O(VE). As a result Dijkstra algorithm is implemented with min-priority queue for having best run time [18, 19]. The Dijkstra algorithm has a high accuracy but it has time space overhead, because Dijkstra algorithm traverses all nodes of a map for obtaining the shortest path from the start node to the target node. Nevertheless it is used widely used in network optimization and transportation [20].

Nomenclature of symbols:

- *Q*: it is a min-priority queue and it is includes nodes of graph.
- *P*: It is the previous node of node i that was visited.
- *D*: It is the distance of node i from the origin node.
- *V*: if node i has been visited, the value of this element is true otherwise it is false.

Its implementation can be described as follows:

Dijkstra algorithm keeps a min-priority queue Q and three operations are done: Add-priority (first for in pseudo instructions), Obtaining-Min (section a of while in pseudo instructions) and Decrease-Key (subsection 2 of command if in section c of while). They (*D* and *V*) are initialized with *Di* = ∞, *Vi* =false and *Pi*= ∞. First the origin node is added to queue and after in each iteration the closest node to node i that it is unvisited, it is added to queue. The queue is prioritized with values of *D* (the add-priority operation). In each iteration, a node is selected from queue that it has minimum distance so that *Di* is the smallest value and *Qm* is a node (the obtaining-min operation), then it is removed from queue and *Vm*=true. Then for every neighbor *ni* of this node *Qm*, if *Vni*=false (if node *ni* has not been visited). It is obtained a value equal with *Dm*+distance-between (*m*,*ni*). If this value is lower than *Dni*: first, the new value replaces with old value; second, *Pni*← *Qm* and; third, it is decreased the priority of *Qni* in queue with reordering *Qni* in queue (the decrease-key operation). These operations repeat while the queue isn’t empty or don’t reach to target node.

The instructions to implement this algorithm are presented in Code 2.3.

Code 2.3. Dijkstra algorithm pseudo code

```
input: the origin-node and the target-node
for all Pi and Di and Vi do
    a. Pi ← ∞, Di← ∞, Vi←false
    b. if Di =origin-node then  Di ←0
Qi←origin-node
 while Qi≠target-node do
    a.Calculate the smallest element in Q then
        mD← Di  ,    Qm← Qi
        Vm←true           //removing Qm from Q
    b.if  Qm= target-node then
        i.path← target-node
        ii. while Pm≠origin-node do
            1.if  Pm=origin-node then
                path ← path+ origin-node  ,  break
            2.path ← path+Pm  ,   Qm←Pm
       iii.Break
    c. for all neighbor ni of Qm do
        i. if   Vni = false then
            1.alt ← Dm +distance-between(m,ni)
            2.if alt < Dni then
                Dni←alt , Pni←Qm , reorder  Qni in Q
output: the shortest path from the origin-node to the target-node
```

Dijkstra algorithm is similar to a breadth-first search and the Prim's algorithm [21] for calculating minimum spanning trees, but in breadth-first search, graph is unweighted. The greedy process is used in Prim's algorithm same to Dijkstra algorithm. A greedy algorithm [13, 14] typically is used for optimization of complex problems. It creates a solution step by step and each step selects next step that it looks the best selection in that moment. The greedy algorithms don't always create the global optimal solution, but they can create a local optimal solution. Also they are the powerful and efficient techniques in the most problems that make simply solution for the complex problems. A minimum spanning tree in Prim's algorithm is connected to all nodes in graph, but in Dijkstra algorithm is concerned with only two nodes [18, 14]. The Dijkstra algorithm creates the shortest path from the origin node to every other node. Therefore it obtains the minimum distance tree between two nodes, but the Prim's algorithm obtains the minimum spanning tree for a given graph that connects all nodes when the sum of all costs is the minimum possible. Chao et al. [22] developed the Dijkstra algorithm for a real urban traffic network abstraction to the network diagram in graph theory. It is proved to reduce the computational time through simulation. It reduces the large number of meaningless operations and improves the efficiency of the search in the networks based on graph. In [23] an Energy Saving-oriented Routing Algorithm based on Dijkstra (ESRAD) in Wireless Sensor Networks (WSN) is presented. It is shown the efficiency of the proposed algorithm (ESRAD) for searching the best navigation scheme with least energy consumption to transmit information between any of two nodes in WSNs. In this paper, first is presented an improved Energy Saving -oriented Least-hop Routing Algorithm (ESLHA) to search the shortest path of energy consumption with least nodes from start node to target node and because the ESLHA does not consider the energy consumption in transmission process, it is used ESRAD. This ESRAD is demonstrated to save great amount of energy. But their scheme was not implemented in a grid-based maps or real mobile robots. In [24] is proposed a hybrid multi-path routing algorithm for industrial wireless mesh networks. The Ant Colony Optimization (ACO) routing protocol improves with enhanced Dijkstra algorithm. Also it increases efficiency in route set up and improves redundant route exploration and failure link maintenance. But their work focuses on the improvements of graph routing to meet demands of industrial applications and not routing in the grid maps. Pushpalatha et al. [25] proposed a Dijkstra algorithm that can estimate distance information among the sensor nodes and found the shortest path position estimation between the source and the destination in WSN. Their method is used to estimate all sensors' relative locations by the use of Dijkstra algorithm to compute the relative positions of sensors with low cost and high error tolerance. It is demonstrated that their algorithm works well for near uniform radio propagation. Also it shows that the Dijkstra algorithm is the best algorithm to find out the shortest path position estimation method in WSN. But they didn't measure the memory size in their experiments and all of experiments are done only in near uniform radio propagation.

2.1.4. The A* algorithm

Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute first described the algorithm in 1968 [26]. It is an extension of Dijkstra Algorithm. The A* algorithm is more efficient and obtains higher speed than Dijkstra algorithm in path finding, because it applies a heuristic function in its algorithm. The A* algorithm will find shortest path between the start node and the target node on a map.

There are many types of A* algorithms. The A* algorithm searches the map by exploring the different nodes for creating the shortest path. Each node has three properties that they are called f, g, and h. They are sometimes referred to as the fitness, goal, and heuristic values, respectively. They are defined as follows:

- **g** is the cost for getting to this node from the start node.
- **h** is the estimated cost to get from this node to the target node.
- **f** is the sum of g and h. f is the best guess for the cost of the best path between the start and the target node.

Value g is zero for the first node, because there isn't any cost for the first node. Value h for each application is calculated differently. For the grid based map problems, it is a combined cost of the horizontal and vertical differences that is called Manhattan Distances. It is a reasonable guess for value h. The Manhattan Distance is calculated as follows. In this formulate, (tx,ty) is the coordinate of the target node and (sx, sy) is the coordinate of the start node [16, 27].

$$h= | tx-sx| + | ty-sy|$$

The time complexity of A* depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path). In the following formula h(x) is the heuristic function. $|h(x)-h^*(x)|=O(\log h^*(x))$

The h* is the optimal heuristic, the exact cost to get from x to the goal point. On the other hand, the error of h won't grow faster than the logarithm of h* (perfect heuristic), so that specifies the real distance from x to the goal point.

The A* algorithm keeps for each node's value f(x) = g(x) + h(x) but in Dijkstra algorithm h(x)=0. Therefore A* algorithm is a generalization of Dijkstra algorithm. This algorithm tries to combine the advantage offered by Dijkstra and breadth-first search algorithms. It also achieves better performance (with respect to time) by using heuristics than Dijkstra algorithm. So it reduces the size of the subgraph that must be searched.

The main advantage of the A* algorithm is higher speed in convergence and it is due to depth-first traversal rules and heuristic [20, 28].

The A* algorithm works as follows:

First, the origin node is selected as current node and f, g and h values are determined for it. A* algorithm needs a way to keep track of the nodes. So the nodes to be considered are kept in a set, called openset. At the beginning, the origin node is put to the openset, and after that all of its neighbor's nodes are considered, it is moved from the openset to another set called the closedset. Closedset keeps the nodes that are visited and there is no need to revisit. In each iteration, a node is selected with the minimum g value. If this node is the target node, a path is found. If openset is empty, there is not any path from the origin node to the target node. At the moment, f, g and h values are determined for every neighbor of the selected node. For each neighbor considered, whether it is in the openset or in the closedset, if it has lower g value, the openset or closedset is updated (the parent of this node changes) and otherwise this neighbor is added to openset. Then the selected node is moved from the openset to the closedset and the process is continued. The end condition is defined when it reaches the target node or the openset is empty. This algorithm is the most efficient algorithm because it uses both the shortest path information from the origin node and the shortest path information towards the target node. The instructions to implement this algorithm are presented in Code 2.4.

Code 2.4. A* algorithm pseudo code

```
input: the origin node and the target node
Current node ← the origin node
Determine f, g and h values for current node.
openset ←current node
B ← select the node from the openset with the minimum g value
        a.  if B=the target node then exit // the path is found
        b.  if openset=∅ then exit // there isn't any path
C ← a neighbor node from B
        a.  Determine f, g and h values for C
        b.  if C ϵ openset or C ϵ closedset then
              i.   if the C has minimum g value then
                       Update openset or closedset with new g value
                       //the parent of C changes
              ii.      else openset ← C
        c.  repeat step 5 for all neighbors of B.
Move B from the openset to the closedset and repeat from step 4.
output: the shortest path from the origin node to the target node
```

Example 2.2: The A* algorithm is applied for finding the shortest path from the origin node (s) to the target node (t) as shown in fig. 2.4.

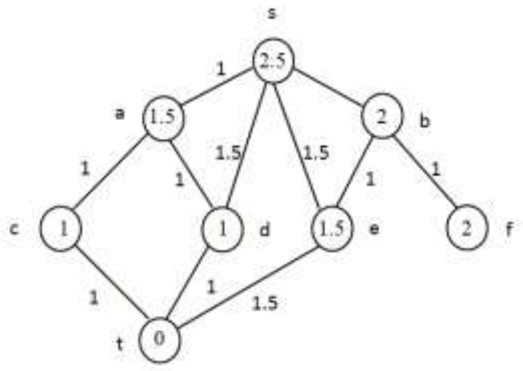


Fig. 2.4: the origin node as s and the target node as t in the graph

The values inter each node are h value and the values on the edges are the distance between two nodes.

Step 1: The current node = s

Step 2: g(s) = 0, because (s) is the origin node. h(s) = 2.5→ f(s)= g+h= 0+2.5= 2.5

Step 3:

openset	s
f	2.5

Step 4: B = s

Step 5: C = a, g(a)=1 , h(a) = 1.5→ f(a) =1 + 1.5 = 2.5 ,

openset	s	a
f	2.5	2.5

C = d, g(d)=1.5 , h(d) = 1→ f(d)=1.5 + 1= 2.5 ,

openset	s	a	d
f	2.5	2.5	2.5

C = e, g(e)=1.5 , h(e)=1.5 →f(e)=1.5+1.5=3,

openset	s	a	d	e
f	2.5	2.5	2.5	3

C = b, g(b)=1 , h(b)=2 → f(b)=1+2=3

openset	s	a	d	e	b
f	2.5	2.5	2.5	3	3

Step 6:

closedset	s
f	2.5

openset	a	d	e	b
f	2.5	2.5	3	3

Step 4: B= a

Step 5: C= c, g(c)= 1+1=2 , h(c)= 1 → f(c)= 2+1=3,

openset	a	d	e	b	c
f	2.5	2.5	3	3	3

C= d, g(d)=1+1=2 , h(d)= 1 → f(d)=2+1=3,

openset	a	d	e	b	c
f	2.5	2.5	3	3	3

g new>g old and d ϵ openset→ the parent of d does not change.

Step 6:

closedset	s	a
f	2.5	2.5

openset	d	e	b	c
f	2.5	3	3	3

Step 4: B= d

Step 5: C= t, g(t)= 1+1.5=2.5 , h(t)= 0 → f(t)= 2.5+0=2.5,

openset	d	e	b	c	t
f	2.5	3	3	3	2.5

Step 6:

closedset	s	a	d
f	2.5	2.5	2.5

openset	e	b	c	t
f	3	3	3	2.5

Step 4: B=t and B= target node→ a path is found.

Final path: t → d → s

Fig 2.5 provides a briefly representation of A* algorithm.

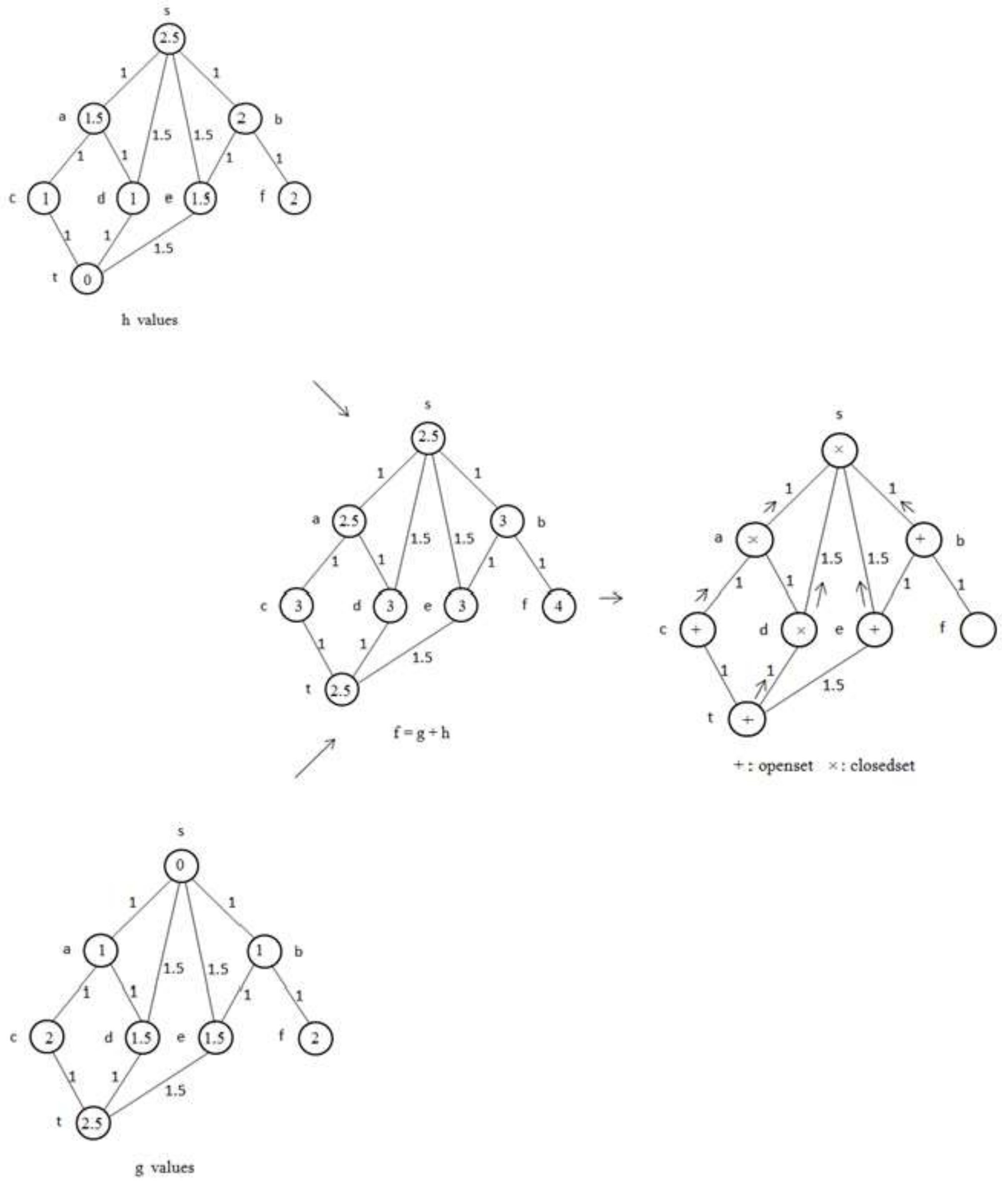


Fig. 2.5: Applying stages of A* algorithm on sample grid map briefly

There is an abundance of algorithmic research related to path planning in mobile robots. In [29] is shown the time complexity depends on quality of the heuristic function and if the heuristic function returns an exact cost to reach a target, in this case the time complexity is linear along the solution depth. Many types of A* algorithms exist for mobile robots on grid based maps. The most of the research on A* algorithm or its derivatives is focused on the speed up of this algorithm for navigation. The time metric is considered in these investigations as the advantage of the algorithm, because A* algorithm consumes a large amount of memory. Ananya et al. [30] improved real time A* algorithm for path planning. Their strategy is designed in a grid map form of an unknown environment with static obstacles. They optimized path solution with three cost functions: the distance, time and energy metrics. Also, Kumar Das et al. [31] implemented the heuristic A* algorithm for a mobile robot in an unknown environment. They considered the distance and time metric as the cost function. In [32] Cazenave presents some optimizations for A* and IDA* using an array of stacks instead of a priority queue in A*. Results show a higher speed in execution. The authors in [33] propose a modified A* algorithm to generate safe paths for robots. It considers the robot size and the direction of arrival (DOA) as parameters for avoiding sharp turns around corners and unsafe diagonal moves. Yuksel et al. [34] presented and implemented three well-known path planning algorithms (Breadth-first algorithm, A* algorithm and Dijkstra algorithm) for a mobile robot on grid based map for one-goal and multi-goal states. They compared these algorithms using CPU time, path length and memory (the sum of the visited cells). It is shown for maps with different cost cells and with one-goal cell, the A* algorithm is best in CPU time and size of memory. Also it is shown for maps with different cost cells and with multi-goal cells A* is best in CPU time, but it doesn’t guarantee for the shortest path. Dijkstra algorithm in this state guarantees the shortest path, but it has worse CPU time than A* algorithm. It is presented for maps with the same cell cost with one-goal cell and multi-goal cells; using Breadth-first algorithm is the best if the CPU time is the important issue. But if the size of memory is an important issue, the A* algorithm can be a better selection. In [35] is presented an effective pathfinding algorithm for mobile robot navigation. It is derived from the A* algorithm that can find the minimum path cost based on heuristic estimations with obstacles avoidance capability. It is utilized in mobile robot navigation in grid based maps. The main advantage of their work is creating low cost paths with fast computational process and easy deployment. In [36] by using A* algorithm is improved the performance to creating optimal alignments. Alignment of multiple DNA or protein sequence is a main problem in computational biology. In [37] a method of global path planning by using GA (Genetic algorithm) [5, 6] and A* algorithm for mobile robots is proposed. This approach is proved which has better performance than Dijkstra algorithm in term of speed or computational time and its solution quality.

2.2. Local path planning algorithms

Dijkstra/A* algorithms are used for global path finding algorithms, when all the information of the map is known a priori. However, when not all the information is available or may change, local path planning algorithms must be used. Among local algorithms, the bug family is very popular. The bug family algorithms are local path planning algorithms, because it isn't any knowledge of environment. They are a simple planning algorithm with an easy implementation, so that can create a collision-free path from the start point to the target point, if it exists. They work in 2-D environment with unknown obstacles. The bug's family show a better performance than other algorithms, but their performance is related to shape of obstacles. According to shape of obstacles and environment a different bug's family algorithms may be better [38].

These algorithms have only local information that is acquire through sensing. It is inspired by insects. They have three assumptions about the robot mobile:

- The robot is a point. In fact, they don't consider any size for robot in the implementation.
- The robot has a perfect localization.
- Their sensors are precise. It means the robot has essentially tactile (contact) sensing.

Clearly, these three assumptions can't be used for real robots. But they could be considered as a main component of an algorithm so that includes the former assumptions.

There are different types of bug family algorithms. It is differed depending on their behavior of obstacle avoidance.

Another popular local path planning algorithm is the potential field algorithm. This approach considers the obstacles as repulsive force sources and the target as attractive force sources. At the end, the path is created by the composition of two forces. This way is simple mathematically and able to create a collision free path in a dynamic environment. The main problem of this algorithm is local minima. It is for locations of trap in map, which prevent the mobile robot reaches to real target location. But this problem is solved by different techniques.

The reminder of this section is organized as follows. The Bug family algorithms are described in subsection 2.2.1 and the potential field algorithm is presented in subsection 2.2.2.

2.2.1. Bug family algorithms

The bug algorithms such as Bug1 and Bug2 [39] work according to two simple behaviors:

- "Move towards the target point", as long as the robot can do it and while an obstacle is not found,
- "Follow the obstacle boundary" till the robot is possible to "move towards the target point" again. Therefore path planning algorithm is a continuous online process that merges locality and globally convergence assumptions, i.e., to head the target point whenever possible.

The Bug1 and Bug2 algorithms are sensor-based path planning algorithms, also their advantage are minimum memory requirements, because they are simple [40, 41].

2.2.1.1. Bug1 algorithm

The Bug1 algorithm was the first bug algorithm created by Lumelsky and Stepanov in 1984 [39]. In this algorithm, first it is considered the existence of a collision-free path from the start point towards the target point [42, 43]. But if this path does not exist and an obstacle is found, in this case, the algorithm enters into a second phase. When the robot encounters an obstacle, it moves clockwise sense (default) in external edge of obstacle and recorded the distance between the current point and the target point in each motion. This process is continued until it is done a complete swing around of obstacle. In this state, it is determined the point that it has the minimum distance towards the target point and it turns again to the first stage. If the path is clear from this point towards the target point, the algorithm terminates. If a second obstacle appears, this process is repeated. This algorithm is inefficient, only ensures the robot reaches to the target point, if the path exists. The path from the start point to the target point is shown in fig. 2.6. Also pseudo code of the bug1 algorithm is indicated in code 2.5. In this pseudo code, H_i is the i^{th} meet point. This is the first point of contact between the mobile robot and the i^{th} obstacle and L_i is the i^{th} leave point. This is the point at which the mobile robot leaves the i^{th} obstacle.

The drawback of the algorithm is not considering the width of the robot. This can be a serious problem if the distance between two obstacles is narrower than the robot's width.

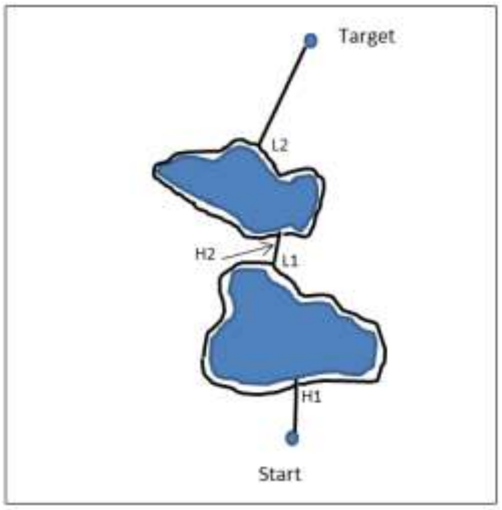


Fig. 2.6: the path from the start point to the target point by Bug1 algorithm

Code 2.5: Bug1 pseudo code

```
input: the start and the target
$L_0$ ←start; i ← 1   // i is obstacle number
Loop
    repeat move on a straight line from $L_{i-1}$ to target
    until target is reached or obstacle is met at $H_i$
     if target is reached then exit with success
    repeat turn boundary saving point $L_i$ with shortest distance to target
    until target is reached or $H_i$ is met again
    if target is reached then exit with success
    turn boundary from $H_i$ to $L_i$ along shortest path
    if move on straight line from Li toward target moves into obstacle then exit with failure
    else i ← i + 1
output: the shortest path from the start to the target
```

In [38] the different types of Bug algorithms are compared and simulated. The results indicate that Bug1 has the worst results when compared to Bug2 and DistBug (see below).

2.2.1.2. Bug2 algorithm

The Bug2 algorithm was also created by Lumelsky and Stepanov in 1984 [39]. The bug2 algorithm is an improved version of the Bug1 algorithm. Also, this algorithm is the greedy algorithm that generates a path from the start point towards the target point with a constant slope (m). The slope ‘m’ and y-intercept ‘c’ are given by (1) and (2) respectively. The slope or gradient of a straight line is a constant number which describes the steepness and the direction of a straight line. It is usually represented by ‘m’. The y-intercept is where a straight line intersects the y-axis.

$$m = \frac{y2-y1}{x2-x1} \qquad (1)$$

$$c = y - mx \qquad (2)$$

In Bug2 algorithm, first it is considered a straight and collision-free path from the start point to the target point. Otherwise if it is found an obstacle, the robot follows external edge of obstacle from this point in clockwise sense (default) and checks continuously its slope from the current point to the target point. When this slope becomes equal to initial slope (between the start point and the target point), in this case, it is considered again, existence straight path towards the target point or finding new obstacle and these operations is repeated until reach to the target point [40, 44]. The path from the start point to the target point is shown in fig. 2.7.

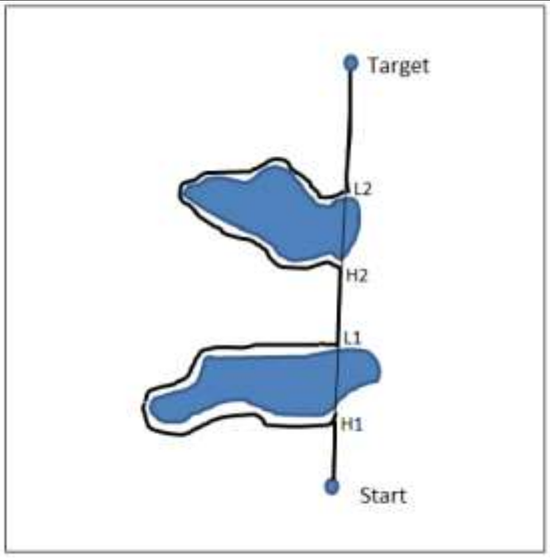


Fig. 2.7: the path from the start point to the target point by the Bug2 algorithm

Also, the pseudo code of the Bug2 algorithm is presented in code 2.6. In this pseudo code d is the distance between two nodes, such as d(Q,target) is the distance between the Q node and the target node. Also Q is an arbitrary point on the obstacle boundary.

Code 2.6: Bug2 pseudo code

```
input: the start and the target
$L_0$ ← start, i ← 1
loop
repeat move on a straight line from $L_{i-1}$ to target
until target is reached or obstacle is met at $H_i$
if target is reached then exit with success
repeat follow boundary
until
  (a)  target is reached or
  (b)  m-line is met again at Q such that $Q \neq H_i$, d(Q , target) < d($H_i$ , target), an
       Line (Q,target) does not cross the current obstacle at Q or
  (c)  $H_i$ is met again
 if target is reached then exit with success
else if $H_i$ is re-encountered then exit with failure
else  $L_i$ ← Q , i← i + 1
output: the shortest path from the start to the target
```

In [2], when the shape and the coordinate of obstacles are not known for the mobile robot and it knows only itself and the target locations. It is shown that information is enough for reaching the target and the path planning algorithm works correctly. Also unlike Bug2 algorithm, Bug1 algorithm does not need to know the current coordinate of the mobile robot. But the Bug2 algorithm is more efficient than the Bug1 algorithm and also has less execution time for reaching to the target point. In [45] the performance of Bug1 and Bug2 algorithms is limited by worst-case lower bound and worst-case upper bound on the path length.

In [38] the different types of Bug algorithms are compared and simulated. The conclusion is that Bug2 has better results than Bug1.

2.2.1.3. DistBug algorithm

The DistBug algorithm was invented by Kamon and Rivlin in [46]. DistBug algorithm is final improved version of Bug algorithm families. DistBug algorithm is a local path planning algorithm together global information that ensures convergence and that is found path from the start point towards the target point. In this algorithm, the robot navigates the path from the start point to the target point in shorter distance than before two algorithms and also has less execution time [47, 48].

In this algorithm, the robot has a maximum detection range is R and it has two modes for motion, boundary-following and motion-to-target. First, the robot navigates towards T (the target point) till it meets an obstacle. Then, its boundary-following mode is done in the clockwise sense (default). During the boundary following mode, the robot saves the minimum distance, dmin(T), to T obtained since the last hit point. The robot senses the distance in free space, F, which is a distance of an obstacle from the robot's location, X, in the direction of T. If no obstacle is sensed, F is set to R. The robot leaves the obstacle boundary mode only while the path to T is clear or the equation, $d(X,T) - F \leq dmin(T) - Step$, is satisfied where d(X,T) is the distance from X to T, and Step is a predefined constant.

The path from the start point to the target point is shown in fig. 2.8.

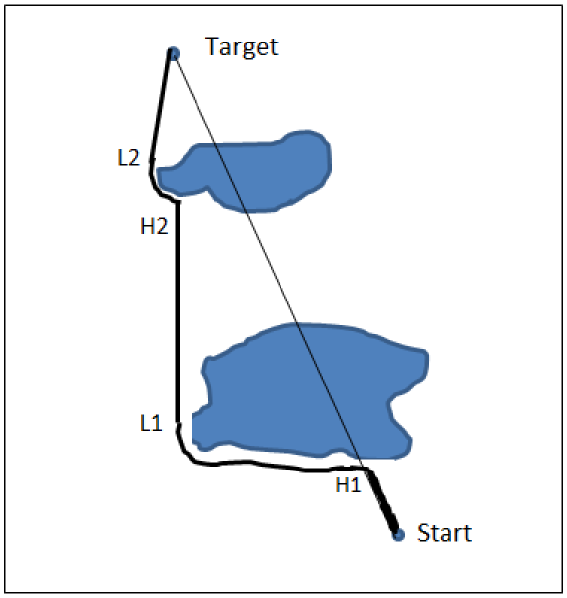


Fig. 2.8: the path from the start point to the target point by DistBug algorithm

The DistBug algorithm is described as followed:

1. Go straight to the target point, until one of the following does:
   a) The target point is met. Stop.
   b) An obstacle is met. Go to step 2.
2. Select the boundary following mode and Follow the obstacle boundary while recording the minimal distance to T dmin(T) and sensing the distance in free space F until one of the following does:
   a) The target point is met (The target is visible) d(X,T) – F ≤0. Stop.
   b) The free range in the mode to the target guarantees that the next hit point will be closer to the target point than the last hit point (The range-based leaving condition holds) d(X,T) – F ≤ dmin(T) – Step. Go to step 1.
   c) The robot completed a loop around the obstacle. The target point is unreachable. Stop.

In [38] the different types of Bug algorithms are compared and simulated. It is used the empirical data for comparing the performance between Bug1, Bug2, and DistBug algorithms. Also they applied it on a mobile robot that provides the simulation environment by MobileSim. It is indicated that the DistBug has the best path length. In [49] is considered the different types of the Bug family. It is shown the performance of them can change depending on the environment. Also the Bug families are ranked from simple to complex according to implementation complexity in order: Bug2, Bug1 and DistBug.

2.2.2. The potential field algorithm

The Artificial Potential Field approach [50] is a local path planner method that was introduced by Khatib. The mobile robot navigates from the start location towards the target location. This approach works in the dynamic environment so that it permits any variation to occur, so it does not need full knowledge of the environment. This characteristic makes it a useful algorithm for real time path planning. Also, this algorithm is flexible and adaptable. This method is based on attractive and repulsive forces. In potential fields, the goal is considered as a global minimum potential value, and all obstacles as high valued potential fields. The movement of the mobile robot is determined by the potential values present in its path. The movement is done from high to low potentials. The mobile robot is represented as a particle in the configuration space q moving under the influence of an artificial potential field U(q). In this formulation, q is state of robot in the map. The superscript T represents “transpose”.

$$q = (x,y)^T$$

The artificial force or F(q) in each iteration is due to the potential field. These formulas are according by [41]. The potential field is created by the sum of an attractive potential Uatt pushing the robot to the target location and a repulsive potential Urep pushing the robot away from obstacles.

$$U(q) = Uatt(q) + Urep(q)$$

The force vector is evaluated by the particle to be the negative gradient of the potential field.

$$F(q) = -\nabla U(q) = -\nabla Uatt - \nabla Urep$$

Where ∇U is the gradient vector of U evaluated at the robot position. Therefor Uatt is the “attractive” potential for moving to the target and Urep is the “repulsive” potential for avoid obstacles. The force is defined as the sum of the two attractive and repulsive force vectors, Fatt and Frep, respectively, i.e.

$$F(q) = Fatt(q) + Frep(q)$$

2.2.2.1 Attractive potential field

The attractive potential field is responsible for the potentials that attract or pull the robot towards the target location. At all locations in the environment the action vector will point to the target location.
The attractive field is defined as a parabolic form:

$$\text{Uatt}(q) = \frac{1}{2}\gamma\rho^2 \text{ goal}(q)$$

With γ a positive scalar and ρ goal(q) is the distance from q to the target location.

The function Uatt is assumed nonnegative and reaches its minimum at qgoal where ∇ Uatt(q_{goal})=0. Since Fatt is different everywhere in the configuration space, the attractive force can be defined as follows:

$$\text{Fatt}(q) = -\gamma\nabla \text{ Uatt}(q) = -\gamma\rho \text{ goal}(q)\ \nabla\text{goal}(q) = -\gamma\rho\ (q-q_{goal})$$

2.2.2.2 Repulsive potential field

The repulsive potential is used to make a potential boundary surrounding obstacles that can't be crossed by the mobile robot. Also, this potential doesn't affect the movement of the robot when it is adequately far from the obstacles. We denote d = $|q-q_0|$ to represent the distance between the robot and the obstacle. The formula proposed by [50] is given:

$$\text{Urep}(q) = \begin{cases} \frac{1}{2}\mu\left(\frac{1}{d} - \frac{1}{d_0}\right)^2 & \text{if } d < d_0 \\ 0 & \text{if } d > d_0 \end{cases}$$

In above formula q is the robot position and q_0 is the obstacle position. Note that d_0 is the influence distance of the force, and μ is an adjustable constant. Hence, there is the corresponding repulsive force function:

$$\text{Frep}(q) = \begin{cases} \frac{1}{2}\mu\left(\frac{1}{d} - \frac{1}{d_0}\right)\frac{q-q_0}{d^3} & \text{if } d < d_0 \\ 0 & \text{if } d > d_0 \end{cases}$$

The attractive force and repulsive force is shown in fig. 2.9. In this figure, Fr is the sum of all attractive and repulsive forces.

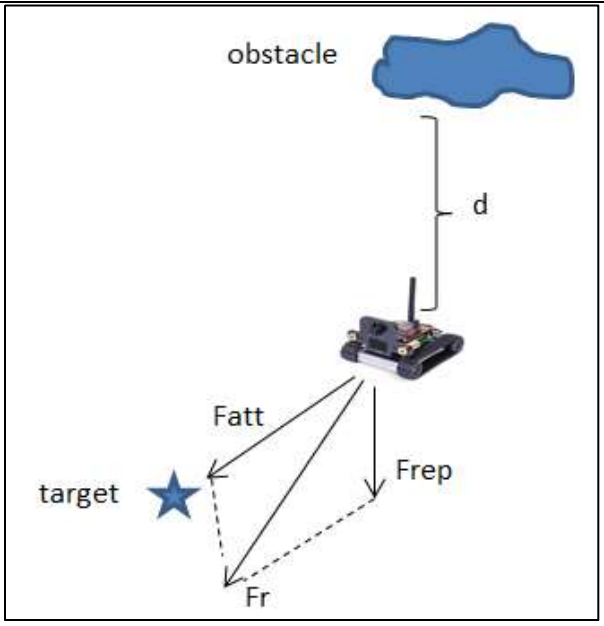


Fig. 2.9: the attractive and repulsive forces in the potential field algorithm

Potential Field algorithm has many advantages as a general technique for the mobile robot path planning. This approach is implemented simply, therefore behavior of the mobile robot easy to predict. Each field is independent and calculating under the field is not relevant. So the potential field algorithm can work in parallel systems. Another advantage is real time environment or dynamic environment. Therefore the potential field algorithm is a good approach for local path planning especially in a dynamic environment.

2.2.2.3. Local minima problem

The main problem of the potential field algorithm is local minima. This problem is created when the mobile robot cannot reach to target location and it falls in a “trap”. There are different examples of this case. A common example is for U shaped obstacles, when it locates between the mobile robot and the target. In this state F_r equals zero in a point from straight trajectory between the mobile robot and the target, because the repulsive force from the obstacle neutralizes the attractive force from the target and therefore the mobile robot will stop. It is shown in fig. 2.10.

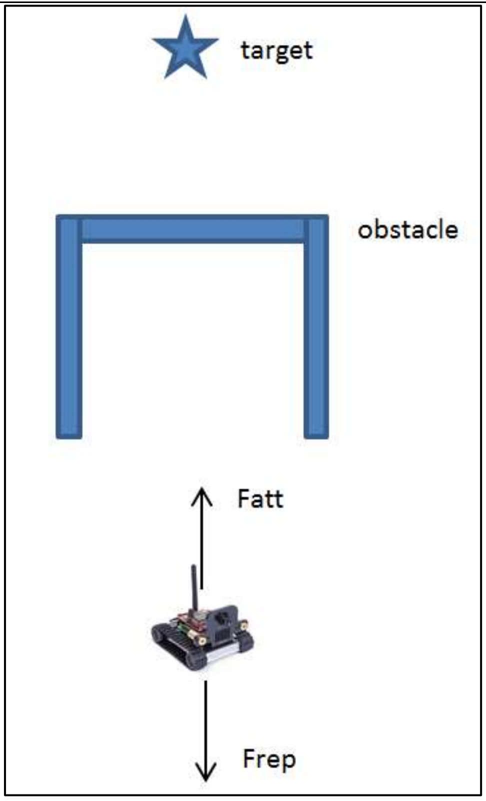


Fig. 2.10: An example with local minima problem in U shape obstacle.

Some solutions for dealing with local minima are:

- Wave-front planner: this technique acquires simplest solution for local minima problem. But it is only for environments which are represented with grids [41].
- Waiting method: this technique is suitable for dynamic environments, where the target and the obstacles are moving. Thus the mobile robot waits until the target location or the obstacle location change [51].
- Injection of Random Noise: this solution uses a random potential field, so that the random values are given to all points. The effect is that the robot "bumps" out of the local minimum.
  This field acts as “reactive grease” [52].

- Avoid-Past Behavior: this solution introduces a cyclic behavior where the mobile robot wants to return towards the local minima. In this approach, the local minima problem is solved and the mobile robot finds a way towards the target position and it is attracted to other places [53].
- Navigation Templates (NaT): in this technique, the direction of tangential fields set the direction of vector sum all potential fields in real time. In this approach, the obstacles are avoided by a force vector from the points in the same direction [54].

Chapter 3. Environment modelling

3.1. Introduction

Map building is another important element for mobile robot navigation, because selecting an appropriate model of the environment can have a great impact in energy saving and performance of path planning. In global path planning, unlike local path planning, the input size and its complexity are important. As a result for fewer memory consumption and higher speed of path planning algorithm, a good and precise representation of the environment is necessary. The common representation approaches collect precise geometric data from the environment. This information is usually obtained from several sensors [55]. Some metric approaches are very different such as configuration space [56], Voronoi diagrams [57, 58], segment models [59], generalized cones [60], vertex models [61], grid models [62, 63] and convex polygon models [64]. These models can include information of the relation between the different places, even if they are still unexplored. The reason is their geometry correspondence to the environment's geometry, which is an important advantage of them. But some of them have low precision in large scale environments. In recent years, most researchers have been interested on the use of grid models for mobile robot navigation or path planning in games. The utilization of the grid maps is very simple for calculating and updating. But they have a main drawback which is the large amount of data require for modelling large scale spaces. As a result, the path planning algorithm consumes a large amount of memory. One of the usual ways to reduce the amount of data from grid maps is processing them for getting a variable resolution partitioning; in this case the environment is divided into subareas of cells for displaying homogeneous regions. Each region is called a node. In [65], the partitioning is implemented using quadtrees. Also in [66], the distribution of the obstacles has effect on optimizing of the partitioning. Arleo et al. [67] presented low complexity of the learned variable resolution partitioning, so that it conducts into saving memory and time resources. Their approach has only one drawback, that it cannot display irregularly shaped regions or the regions which are circular with high accuracy. Therefore there is a trade-off between accuracy and efficiency. Chˆaari et al. [68] designed a tabu search planner algorithm for solving the path planning problem in a grid environment. The path length is evaluated in different map sizes (small, medium and large scales) and compared to A* and GA. The advantage of their algorithm is that it reduces the execution time. The main reason for selecting grid maps for their works is integrating their path planner into the Robot Operating System (ROS) framework for robot application development. In ROS convention, the grid map is displayed with two dimensional matrix called occupancy grid map. A. Elfes [69] presented an overview of the grid maps and its usage in some problems such as path planning, multiple sensor. Nowadays [70, 71] the grid based maps are utilized as fundamental paradigm for mobile robot navigation and path planning algorithms, because they create a geometrical structure of the real world and their usage is easy. Advantages [72] of using grid based maps are listed as follows:

- Easy building, representation and maintenance.
- Recognition of places based on geometric is non-ambiguous and view point independent.
- Facilitate computation of the shortest paths.

Disadvantages [72] of them are:

- Planning inefficient, memory consumption.
- Require precise determination of the robot’s position.
- Poor interface for most symbolic problem solvers.

The remainder of this chapter is organized into two sections. Grid map definition is presented in section 3.2 and graph generation is displayed in subsection 3.3 respectively.

3.2. Grid map definition

A grid map is a map that is defined on a network or a framework of regularly spaced horizontal and vertical lines superimposed for locating points. They include the squares on a map.

In this thesis, some considerations are taken into account regarding the cell size and obstacles representation.

In our case, the grid size is chosen so that a robot can rotate inside the cell without going out of it. Indeed the mobile robot can rotate in every direction without colliding with the walls of cell. Therefore, grid size and robot size must be consistent as depicted in fig. 3.1.

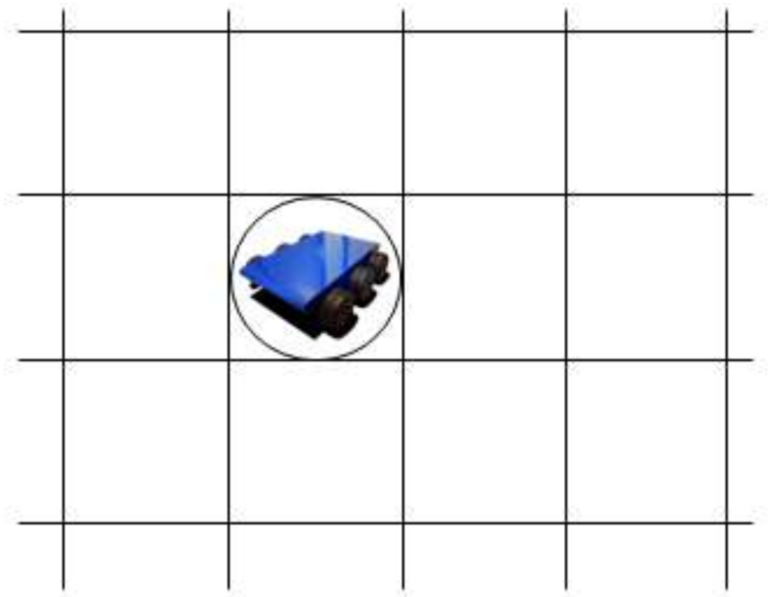


Fig. 3.1. Grid size and robot size must be consistent

Regarding obstacles representation, in this thesis a cell is considered occupied if all of the cell or a part of the cell is occupied by an obstacle. Therefore, each cell is considered fully occupied or fully free. Designing of path planning algorithm in this case is easier when compared to other representation such as probability of being occupied. An example is shown in fig 3.2.

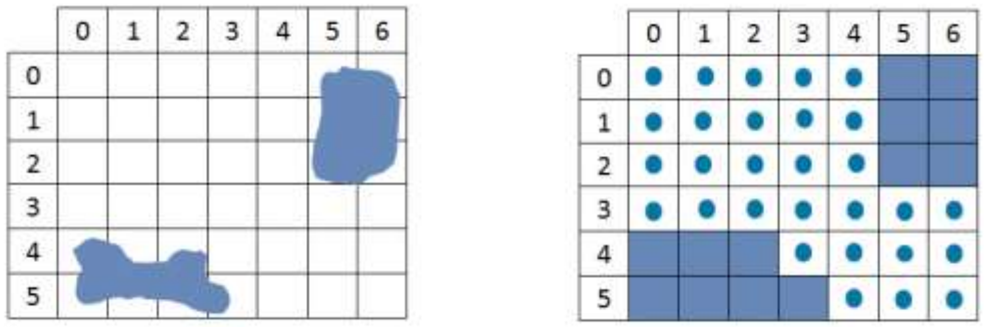


Fig. 3.2. A real map and its equivalent grid map.

As seen above, the workspace is divided in two kinds of grid cells via their values: free space or fill space according to the actual position of the obstacles. In this presentation of the environment, the trajectory can be demarcated as a sequence of cells which starts at the origin node and ends at the target node. In total, grid based map is used to gain simplicity of indoor environment's representation and also among different approaches, this method is popular and simple for implementation.

In this work, a binary matrix is used for modeling the environment as a 2D grid based map. This model allows important savings in processing and memory usage. Every cell of this matrix can be empty or occupied. The zeroes represent empty cells in the grid map whereas the ones mean obstacle cells from the grid map, whether the cell is partially or completely occupied. The circles on the grid map in fig. 3.2 show all possible places that the robot can occupy in the grid map. The grid based map is divided into equal size cells. Every cell in the map has a coordinate that is shown with C(x,y). This cell C(x,y) can be filled by one or zero depending on if it is occupied or free. In other words, C(x,y) is displayed as C(x,y) ϵ [0,1]. The coordinate of each row from the map is in the domain x$\in [0, r-1]$ and the coordinate of each column from map is in the domain $y \in [0, c-1]$, such that 'r' represents the number of rows and 'c' the number of columns. So the mobile robot in motion can meet with occupied cell or empty cell.

$$C(x, y) = \begin{cases} 1 & \text{if } C(x,y) \text{ is occupied} \\ 0 & \text{if } C(x,y) \text{ is empty} \end{cases}$$

The length of each cell is considered one unit (L=1). Converting the binary matrix to a 2D grid based map is shown in fig. 3.3.

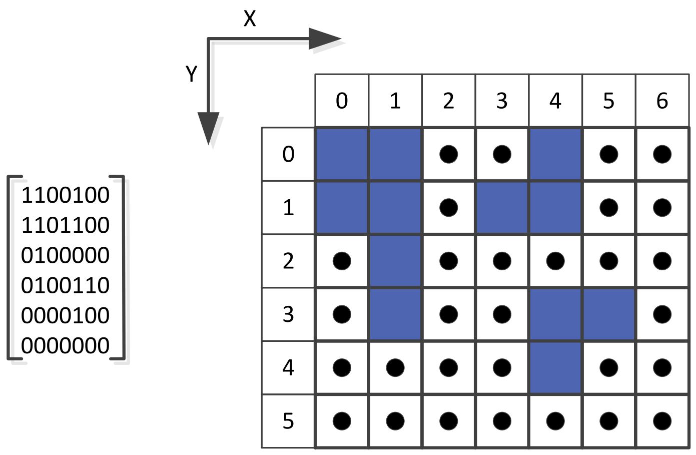


Fig. 3.3. Binary representation of a 2D grid based map. The dots show the coordinates where the robot may be positioned.

3.3. Graph generation

After the grid map is obtained from the binary matrix, a graph is created from the grid map only for Dijkstra and A* algorithms, because they are graph-based. Other algorithms (HCTNav and NafisNav) aren't graph-based and are implemented on grid maps directly. For the graph generation, every empty cell is considered as a node of the graph. To define the edges of each node, only the adjacent free cells are considered. An edge is created between two adjacent cells if the robot can navigate between them. Therefore, a node can have up to eight edges (see fig. 3.4). This decision was made to minimize memory consumption reducing graph spread. The number of edges corresponding to a one-hop is calculated as follows:

$$\#\text{edges (n-hop)}_{\text{each node}} = (2N+1)^2-1$$

$$\#\text{edges (1)} = 8$$

$$\#\text{edges (2)} = 24$$

In this formula, N is the length of hop. The number of edges is O(n^2), so to save memory resources (as it will be seen in chapters 4 and 5, the main drawback of these algorithms) the value of 'n' was kept to the minimum.

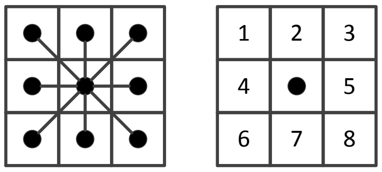


Fig. 3.4. The 8 neighbors for center cell and numbers of them

In fig. 3.5 is represented a graph with 8 edges and 6 vertexes that it is created from a grid map. Every single-vertex is a free cell from grid map that it is identified with its coordinates.

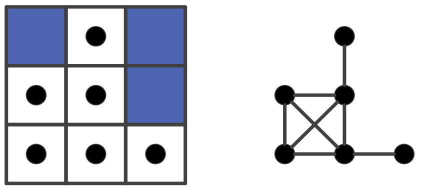


Fig. 3.5. A section of a grid map and its equivalent navigation graph

All path planning algorithms are considered in next chapters implemented on a 2-D grid structure where a grid may contain an obstacle or be free. Some cells in the grid map are empty, so that the mobile robot has to make its path through these empty cells of the grid map in each iteration from the path planning algorithm as a trajectory of movement towards the specified target. Furthermore, this proposed presentation of the environment can help path planning algorithms to use less memory and to be easily implemented.

Chapter 4. HCTNav algorithm

4.1. Algorithm description

HCTNav is a path planning algorithm which arises from a bio-inspired navigation approach. The objective of HCTNav algorithm is to reduce memory consumption. Considering a flat surface, the robot will try to reach its goal position in a straight line. If this line collides with an obstacle, there are two possibilities to surround it. As it is a path planning approach, previous to any real movement, both options can be explored. The obstacle will be surrounded along its perimeter until a position that allows continuing the straight movement towards the goal position is found. The new straight line may reach the goal position, finishing the path, or a new obstacle, which would lead to a new surrounding process. As it can be noted, for each obstacle that is found in the path, it splits into two branches, creating a tree of possible paths. Every path is explored, some of them lead to a possible solution and others fall into a bottleneck. Finally, all valid paths are optimized and the shortest one is returned as the algorithm's solution.

HCTNav algorithm was first presented in [73]. HCTNav presented lower memory requirements than Dijkstra and A* in the same map conditions, which is its main objective. A deeper study was presented in

[74], not only comparing HCTNav to Dijkstra and A* in terms of memory, but also path length and explaining why are they produced. HCTNav shows a good trade-off between path length and memory consumption, being a suitable path planning algorithm for embedded systems. All the details of the HCTNav algorithm and its comparison to Dijkstra and A* are presented in the rest of the chapter.

The algorithm implementation is presented in section 4.2. The examples are shown in section 4.3 and the experimental results are presented in section 4.4.

4.2. Algorithm implementation

The first task of the algorithm is to pre-process the map, defining all cells occupied by obstacles and the perimeter of these obstacles. Next, the algorithm starts checking if the straight path between the origin and the target cells is free of obstacles. As it was presented before, the size of a cell is the minimum area that fits the robot and allows it to turn any angle, considering its holonomic design. When the movement is horizontal or vertical, only a row or a column of the map needs to be verified. However, when the movement requires any kind of diagonal, the algorithm requires checking a corridor, width enough for allowing the robot to travel without colliding with an obstacle (see subsection 4.2.4). As the algorithm was created to be used in low-resource robots, which may lack of a floating point unit, the implementation of the procedure “navbresenham” that establishes the cells that are in the corridor is based on the Bresenham algorithm [75], and explored in section 4.2.3 This algorithm is an integer-based algorithm to draw lines in pixel monitors. More details about how the corridor is defined can be found in subsection 4.2.4.

If an obstacle is found in this corridor, the nearest perimeter cell is set as a path split and the points_surrounding procedure starts. This procedure uses the resources obtained in the preprocessing method, sections 4.2.1 and 4.2.2. Two paths are created from the split point, one in each direction of the obstacle. For both paths, the algorithm iterates along the perimeter cells verifying that the split point can be reached and the same obstacle is in the linear trajectory towards the goal point, see section 4.2.5. When either of these two conditions is broken, the algorithm defines an intermediate node and continues either towards the goal or surrounding the obstacle.

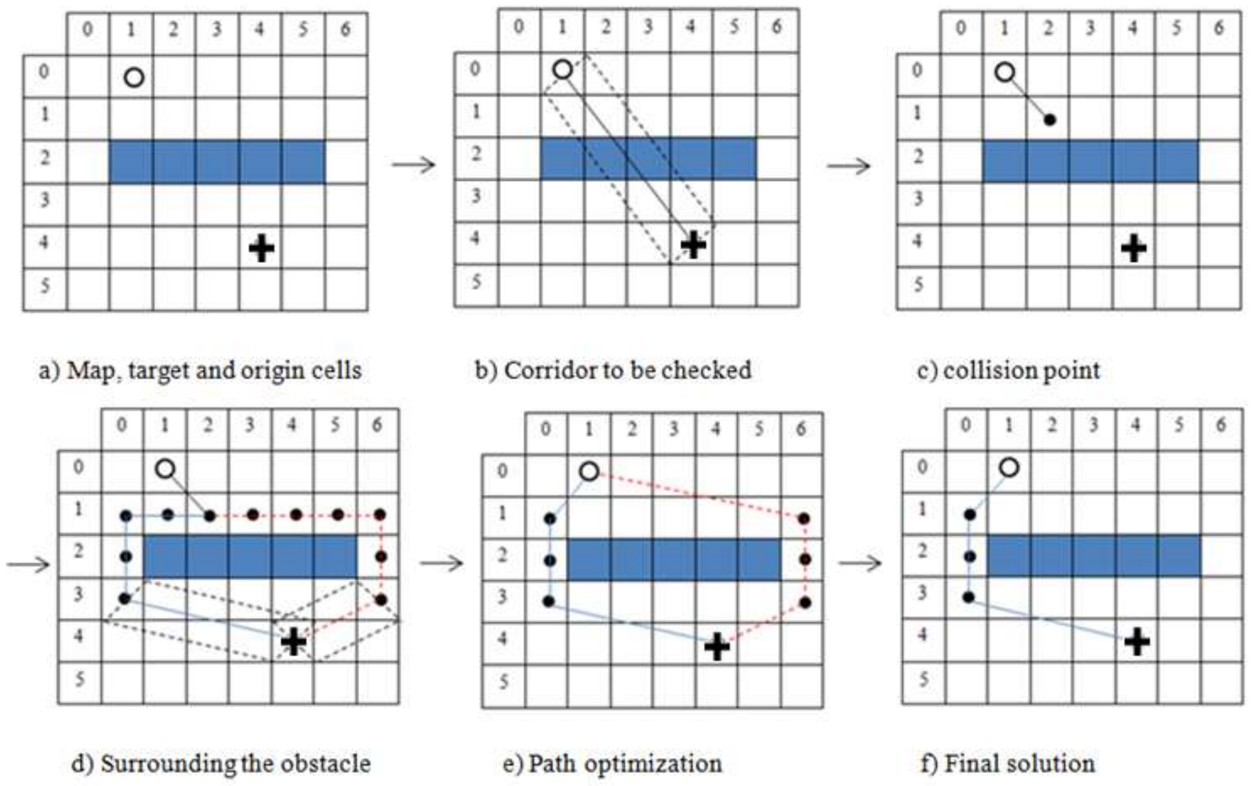

a) Map, target and origin cells
b) Corridor to be checked
c) collision point
d) Surrounding the obstacle
e) Path optimization
f) Final solution

Fig. 4.1. The sequence steps of HCTNav algorithm. The origin point is marked with a circle and the target node with a plus.

Each cell in a grid map is equivalent to a node in a graph tree. So instead of using the concept of cells, the concept node may be used.

When every branch has been explored, the algorithm obtains a tree connecting the origin node, the split nodes, the intermediate nodes and the destination node. The valid paths are optimized and the shortest is proposed as the solution. It must be noted that for path lengths, the square distance between nodes is used for comparison instead of the real distance. This is required to avoid the floating point operations required for the square root. The sequence steps of HCTNav algorithm is indicated on an example in fig. 4.1. Code 4.1 includes the pseudo code of the HCTNav algorithm.

Code 4.1. HCTNav algorithm pseudo code

```
input: the origin-node (ox,oy), the target node (tx,ty) and the map
call Procedure  obstacle_detection(map)
n←call Procedure  points_surrounding(map)        // n is the obstacle number aside the origin node
if navbresenham(ox,oy,tx,ty) = clear then
      i.  $d^2$ ← $(tx-ox)^2+(ty-oy)^2$
     ii.  path ← (ox,oy), (tx,ty)
    iii.  length ← $d^2$
else
      i.  (nx,ny) ← (ox,oy)
     ii.  push (nx,ny)
    iii.  while navbresenham(nx,ny,tx,ty) ≠ clear or stack isn't empty  do
              pop(nx,ny)
              if (n=0) then
                (nx,ny) ← call Procedure  collision_point(nx,ny,tx,ty)
              else  (nx,ny) ←(ox,oy)
              (px,py)(qx,qy) ← call Procedure  create_paths(nx,ny,tx,ty)
               push (px,py)
               push (qx,qy)
              connect (nx,ny) to (px,py)
              connect (nx,ny) to (qx,qy)
              add to tree (nx,ny)
optimized_tree← call Procedure optimization(tree)
find shortest path from optimized_tree
output: shortest path from the origin node to the target node
```

The organization of the remaining part of this section is as follows. The obstacle_detection procedure is presented in subsection 4.2.1 and the points_surrounding procedure is described in subsection 4.2.2. The bresenham procedure is shown in subsection 4.2.3. The navbresenham procedure is shown in subsection 4.2.4. The collision_point procedure is presented in subsection 4.2.5. The create_paths procedure is displayed in subsection 4.2.6 and an optimization procedure used in all the previous paths is presented in section 4.2.7.

4.2.1. The obstacle_detection procedure

This procedure detects all obstacles in a given grid map. The input of procedure is grid map and the output of the procedure is the count and the number of obstacles, a relation between each occupied cell and the related obstacle is established.

The result of this procedure is used in the next procedure.

This procedure is inspired by two-pass technique of connected-component labeling problem in graph theory and image processing [76]. But in this procedure, this technique is modified according to limitations of grid map and for detection all obstacles on grid map and the new parts is added to it.

This procedure includes four steps:

Step 1 is first pass, of the two-pass technique, and for each occupied cell is determined a number. If the values of four neighbors (west neighbor, west north neighbor, north neighbor and east north neighbor) are zero, a number is given to the current cell value (this number is a counter and it is initialized with zero and in each iteration is increased one unit). If only one of the neighbors is occupied with an obstacle, its value is replaced in the current cell value and if the more neighbors are occupied with obstacle, the minimum value of them is found and replaced in the current cell value. An occupied cell and its four neighbors are shown in fig. 4.2.

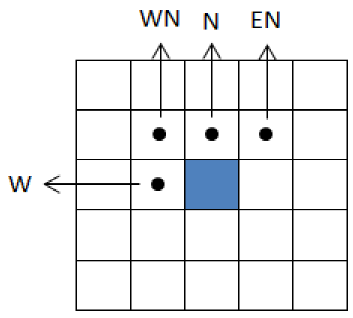
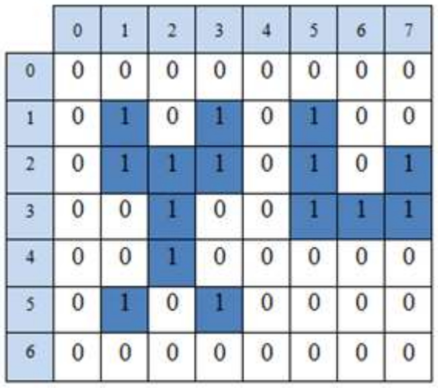


Fig. 4.2. An occupied cell and its four neighbors(west neighbor, west north neighbor, north neighbor and east north neighbor as shown respectively 'W', 'WN', 'N' and 'EN')

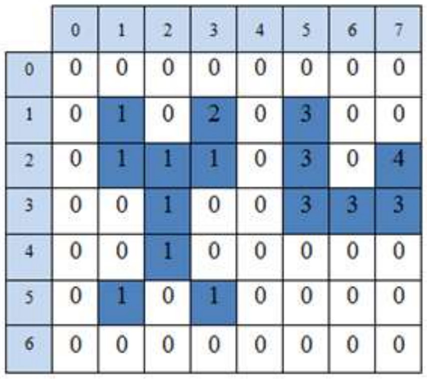
A grid map with two obstacles is shown in fig. 4.3. First the value of free cells is zero and the value of occupied cells is one.



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	0
2	0	1	1	1	0	1	0	1
3	0	0	1	0	0	1	1	1
4	0	0	1	0	0	0	0	0
5	0	1	0	1	0	0	0	0
6	0	0	0	0	0	0	0	0

Fig. 4.3. A sample grid map

Note the grid map is traversed row to row and if in this traverse, the occupied cell was met, the algorithm is applied on it (see fig. 4.4).

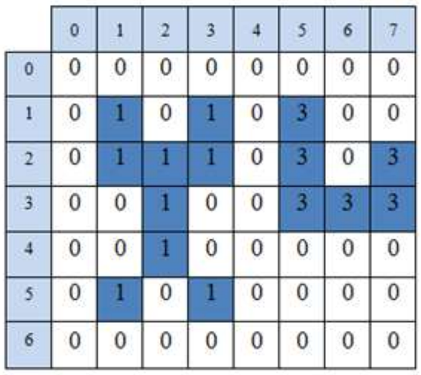


	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	0	2	0	3	0	0
2	0	1	1	1	0	3	0	4
3	0	0	1	0	0	3	3	3
4	0	0	1	0	0	0	0	0
5	0	1	0	1	0	0	0	0
6	0	0	0	0	0	0	0	0

Fig. 4.4. A representation of step 1 of obstacle_detection procedure

Step 2 is the second pass of the two-pass technique. If each of four neighbors had a value opposed with the current cell value, for all previous cells from first row until current row (in row traversing) the value of the traversed cell equals the neighbor value. In this case the current cell value is replaced in that cell. It is occurred in cell(3,2) and cell(7,3) of fig. 4.4 and the result of step 2 is shown in fig. 4.5.

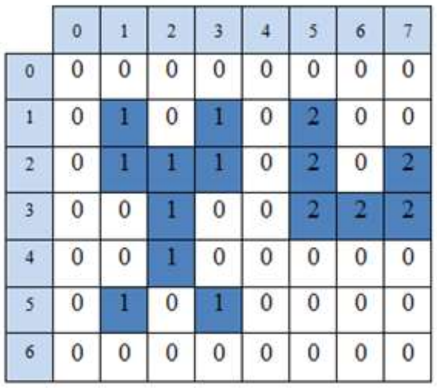
Step 1 and step 2 are repeated for all occupied cell on grid map.



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	3	0	0
2	0	1	1	1	0	3	0	3
3	0	0	1	0	0	3	3	3
4	0	0	1	0	0	0	0	0
5	0	1	0	1	0	0	0	0
6	0	0	0	0	0	0	0	0

Fig. 4.5. Applying step 2 of obstacle_detection procedure

Step 3 is applied on the obstacles number and they will be sequential in end of part. For the example, if the obstacles number are {1, 3, 8, 4, 14, 11, 12}, after applying this step on them, the obstacles number will be {1, 2, 3, 4, 5, 6, 7}. It is indicated in fig. 4.6.



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	2	0	0
2	0	1	1	1	0	2	0	2
3	0	0	1	0	0	2	2	2
4	0	0	1	0	0	0	0	0
5	0	1	0	1	0	0	0	0
6	0	0	0	0	0	0	0	0

Fig. 4.6. Step 3 of obstacle_detection procedure

Step 4 is the recognition of the cells that belong to each obstacle and count them. This step is applied on fig. 4.6 and the result is:

Obstacle 1={cell(1,1), cell(3,1), cell(1,2), cell(2,2), cell(3,2), cell(2,3), cell(2,4), cell(1,5), cell(3,5) }

Obstacle 2={cell(5,1), cell(5,2), cell(7,2), cell(5,3), cell(6,3), cell(7,3)}

The pseudo code of this algorithm is brought in code 4.2.

Nomenclature of symbols:

- *k*: It is count of obstacles.

- *h*: It is count of rows on grid map.
- *w*: It is count of columns on grid map.
- *cell(j,i)*: It is value of cell in column j and row i on grid map.
- *f1*: If west neighbor of current cell occupied with obstacle, so f1 equals one.
- *f2*: If west north neighbor of current cell occupied with obstacle, so f2 equals one.
- *f3*: If north neighbor of current cell occupied with obstacle, so f3 equals one.
- *f4*: If east north neighbor of current cell occupied with obstacle, so f4 equals one.
- *count*: It is count of obstacles.
- *temp*: It is a temporary variable that stores value of a cell.
- *n*: It is count of cells belong to considered obstacle.

Its implementation can be described as follows:

For each of occupied cell is done step 1 and step 2. The cells are traversed row to row. In the first step (lines a-g.i of pseudo code), each of four neighbors is considered and if each of them occupied with obstacle, then its f value equals one and if values f of four neighbors were zero, in this case, counter k increases one unit and number of the current cell equals k. But if only one value f of four neighbors was one, then the current cell equals neighbor cell value and if the value f of two or three or four neighbors were one, the minimum value of neighbors is found and replaced in the current cell. In the second step (line g.ii of pseudo code), if value of each four neighbors opposed with the current cell value, for all previous cells from first row until current row if the traversed cell value equals the neighbor cell value, in this case the current cell value is replaced in that cell. After the second pass, all f values equal zero and the algorithm is repeated.

But in this procedure, step 3 and step 4 are added to the two-pass technique depending on our need.

Third step (lines 3 and 4 of pseudo code) guarantees the obstacles number be sequential (1, 2, 3 …). If this step is not applied on code, it is possible the obstacles number do not be sequential (1, 3, 4, 6 …). In this part, the count of obstacles is initialized by one, then for each cell of map first its value is stored in temp variable and then if its value opposed zero and count variable and also it is greater than count variable, the count variable is increased one unit and replaces in that cell, then if temp value is greater than count variable, for each cell on map if its value equals temp variable, in this case the count variable is replaced in it.

Fourth step (line 5 of pseudo code) is recognized the cells that belong to each obstacle and count them. In this reason, each cell of map equals to one obstacle number, the row number and the column number of that cell is stored and at the end, also count of cells belong that obstacle (n variable) is stored.

Code 4.2. The obstacle_detection algorithm pseudo code

```
 input: grid map
1)  k←0
2)  for i←0 to h do                              // the count of rows is h
      for j←0 to w do                            // the count of columns is w
        if (cell(j,i)is occupied) then
          a)   if (j-1≥0) then if (cell(j-1,i)is occupied) then f1←1
          b)   if (j-1≥0 and i-1≥0) then if  (cell(j-1,i-1)is occupied) then f2←1
          c)   if (i-1≥0) then if (cell(j,i-1)is occupied) then f3←1
          d)   if (j+1<w and i-1≥0) then if (cell(j+1,i-1)is occupied) then f4←1
          e)   if ((f1=0 and f2=0) and (f3=0 and f4=0)) then k←k+1, cell(j,i)←k
          f)    if (only value f from one of four neighbors equals one) then
                   cell(j,i)← neighbor cell value
          g)   if (value f from two or three or four neighbors equal one) then
                      i.   cell(j,i)← find the minimum neighbors value
                     ii.   for m←1 to 4 do                // m is neighbor number
                             if (m neighbor value ≠ cell(j,i) )then
                               for row←0 to i do  // the current row number is i
                                  for column←0 to w do
                                     if (cell(column,row)=neighbor value) then
                                       cell(column,row)← cell(j,i)
          h)    f1←0, f2←0, f3←0, f4←0
 3)  count←1,                                    // count of obstacles store in count variable
 4)  for row←0 to h do
        for column←0 to w do
          temp←cell(column,row)
          if (temp≠0) then
            if (temp≠count and temp>count) then count←count+1, cell(column,row)←count
               if (temp>count) then
                 for a←0 to h do
                    for b←0 to w do
                      if (cell(b,a)temp) then cell[b][a] ←count
 5)  for a←0 to count do
          n←0
          for i←0 to h do
            for j←0 to w do
              if (cell(j,i)=a) then n←n+1, store j and i from each cell belong to obstacle 'a'
           store n                               // count cells belong to obstacle 'a'
 output: each cell of obstacle with its number and coordinate, the entire cells that belong to each obstacle and count them.
```

4.2.2. The points_surrounding procedure

In this procedure, points surrounding of each obstacle are recognized. Input of this procedure is the given grid map and a cell and the output of this procedure is the points surrounding of each obstacle and the obstacle number aside the input cell.

This procedure in HCTNav algorithm is applied after the previous procedure. For all occupied cells on grid map and for each obstacle number, the eight neighbors surrounding are checked, if everyone is free obstacle, then it is stored as point surrounding of that obstacle.

This procedure ensures that when the robot advance to the target cell, it move aside an obstacle unless a new obstacle is detected.

When this procedure is performed on fig.4.10, the following results are obtained:

Points surrounding of obstacle 1={cell(0,0), cell(1,0), cell(2,0), cell(3,0), cell(4,0), cell(0,1), cell(2,1), cell(4,1), cell(0,2), cell(4,2), cell(0,3), cell(1,3), cell(3,3), cell(4,3), cell(0,4), cell(1,4), cell(3,4), cell(4,4), cell(0,5), cell(2,5), cell(4,5), cell(0,6), cell(1,6), cell(2,6), cell(3,6), cell(4,6)}

Points surrounding of obstacle 2={cell(4,0), cell(5,0), cell(6,0), cell(4,1), cell(6,1), cell(7,1), cell(4,2), cell(6,2), cell(4,3), cell(4,4), cell(5,4), cell(6,4), cell(7,4)}

Also the pseudo code of this procedure is shown as code 4.3.

Code 4.3. The points_surrounding algorithm pseudo code

```
input: the grid map and cell (xi,yi)
for a←0 to count do                    //count of obstacles store in count variable
   for i←0 to h do                     //the count of rows is h
      for j←0 to w do                  //the count of columns is h
         if (cell(j,i) occupied with obstacle) then
            if (cell(j,i)=a) then      // 'a' is an obstacle number
               for each of eight neighbors do
                  if (neighbor value is free) then
                     store neighbor coordinate as point surrounding of obstacle 'a'
                     if ((xi,yi) is equal to neighbor coordinate) then
                        n←a            // the obstacle number is stored in n
output: the points surrounding of each obstacle and n variable
```

4.2.3. The bresenham procedure

The algorithm was developed by Jack E. Bresenham in 1962 at IBM [75]. This algorithm is intended for rapid and proper line drawing. It only uses integer operations. The bresenham algorithm is used in computer graphics widely. Also it is applied in the plotters and the graphics hardware of modern graphics cards [77].

The bresenham algorithm is described here briefly.

The bresenham algorithm gets help from straight line equation (1) for line drawing.

$$y=mx+b \qquad (1)$$

The bresenham algorithm is described on an example as shown in fig 4.7. Suppose, we want to draw a straight line between pixel (0,0) and pixel (3,2). After drawing point (0,0), it is decided which points (1,0) and or (1,1) must be painted in the next step. This question is answered by the Bresenham algorithm and a decision parameter (p_k) that shows the distance between each pixel and the real line. The only problem of this algorithm is its dependency to the line’s slope.

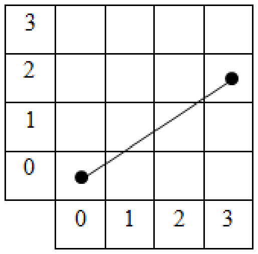


Fig. 4.7. A straight line between pixel (0, 0) and pixel (3, 2)

In a pixel display, to draw a segment between two pixels, the algorithm decides by a decision parameter (p_k) in every column if it has to advance to the next row or to stay in the same one. The initial value of p_k is $p_0=2\Delta y-\Delta x$. The decision parameter (p_k) is determined in step k which pixel is painted ((x_k,y_k) or (x_k,y_k+1)).

As a result:

- If p_k is negative, then the pixel in y_k is closer to line and the pixel (x_k,y_k) is painted. In this case only x increases and $p_{k+1}= p_k +2\Delta y$
- If p_k is positive, then the pixel in y_k+1 is closer to line and the pixel (x_k,y_k+1) is painted. In this case both x and y increase and $p_{k+1}= p_k +2(\Delta y-\Delta x)$

The pseudo code of this algorithm is shown in code 4.4.

Code 4.4. The bresenham algorithm pseudo-code [78]

```
input: (x1, y1) and (x2,y2)
xi←x1, yi←y1
xj←x2, yj←y2
if (x1>x2) then
   a. xi←x2, yi←y2
   b. xj←x1, yj←y1
Δx←|xj-xi|
Δy←|yj-yi|
k←2Δy
m←2(Δy-Δx)
p←2Δy-Δx
draw pixel (xi,yi)
while (xi<xj) do
     i.  xi←xi+1
     ii. if (p<0) then
              p←p+k
         else
              a) yi←yi+1
              b) p←p+m
     iii. draw pixel (xi,yi)
output: the drawn pixels
```

The pseudo code of the bresenham algorithm is adapted to the concept of grid map and the movement of real robot. This pseudo code is shown in code 4.5.

In this thesis, grid maps are used and, for this reason, the concept of cells on grid maps is used instead of pixel. Moreover if specified cells in drawing a Bresenham line are free or in other words they are not occupied with obstacle therefore can draw a cell in the grid map. The input of the pseudo code is two cells (origin and target) and the output is variable "find" which indicating whether the path is free or occupied and cell (xt,yt). If all specified cells in bresenham algorithm are free, output is true and otherwise output is false. In this pseudo code, inside the loop, (xi,yi) stores in (xt,yt) and it is used in the collision_point procedure of subsection 4.2.5. This cell (xt,yt) is the last cell before to collide the obstacle.

Note, this procedure cannot be used alone for diagnosing a straight path between two cells in a grid map and for this reason is introduced next procedure. The reason is that a real robot is not a point, but it has a determined size. The entire robot must cross through empty space, not only its middle point.

Code 4.5. The adapted bresenham algorithm pseudo-code

```
input: (x1,y1) and (x2,y2)
xi←x1, yi←y1
xj←x2, yj←y2
find←true
if (x1>x2) then
        a. xi←x2, yi←y2
        b. xj←x1, yj←y1
Δx←|xj-xi|
Δy←|yj-yi|
k←2Δy
m←2(Δy-Δx)
p←2Δy-Δx
if (xi, yi) does not occupied then store cell (xi,yi)
else find←false
while (xi<xj) do
 i.   xi←xi+1
 ii.  if (p<0) then
          p←p+k
      else
          a) yi←yi+1
          b) p←p+m
iii.  if (xi,yi) does not occupied then
          a) store cell (xi,yi)
          b) (xt,yt) ←(xi,yi)
      else find←false , break
 return find
 output: find and (xt,yt)
```

Now, regarding to before formula and pseudo code is performed bresenham algorithm on example in fig 4.7.

(x1,y1)=(0,0)   (x2,y2)=(3,2)

(xi,yi)=(0,0)   (xj,yj)=(3,2)

Δx=|xj-xi|=3 , Δy=|yj-yi|=2 , k=2(Δy)=4 , m=2(Δy-Δx)= -2 , p=2Δy-Δx=1

Store cell (0,0) , (xt,yt) ← (0, 0)

0<3 → xi=xi+1=0+1=1

```
            p≥0 → yi=yi+1=0+1=1 , p=p+m=1-2=-1
            Store cell (1,1) , (xt,yt) ← (1,1)
1<3 → xi=xi+1=1+1=2
            p<0 → p=p+k=-1+4=3
            Store cell (2,1) , (xt,yt) ← (2,1)
2<3 → xi=xi+1=2+1=3
            p≥0 → yi=yi+1=1+1=2 , p=p+m=3-2=1
            Store cell (3,2) , (xt,yt) ← (3,2)
3≥3 → quit while
```

The stored cells in adapted bresenham algorithm are shown in fig. 4.8.

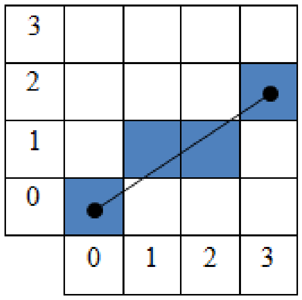


Fig. 4.8. The stored cells in adapted bresenham algorithm for drawing a straight line between (0,0) and (3,2)

4.2.4. The navbresenham procedure

This procedure diagnoses the existence of a collision-free straight trajectory between two cells for the movement of mobile robot in a grid map. Because the robot is not a point and it is a solid object, we have to check all cells allocated in a corridor between two cells for determining a safe or collision-free path between them. For this reason, existence of a collision-free straight trajectory is considered in a corridor between two cells in this procedure as shown in fig. 4.9.

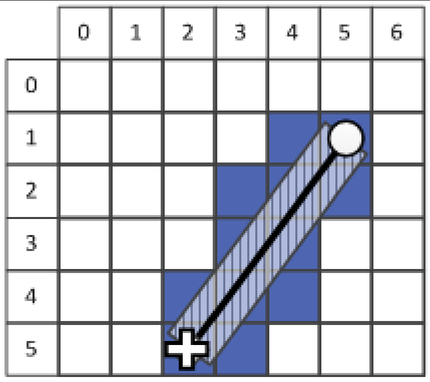


Fig. 4.9. Example of the corridor to be checked for robot navigation. The scratched area is the real corridor and the dark cells are the ones that must be checked to avoid collisions.

In order to obtain all the visited cells (totally or partially) by a robot moving in the proposed corridor, multiple individual bresenham lines have to be included. The exact number of lines depends on the relative position of the origin and target nodes as follows:

- If the path between two cells is horizontal or vertical, it is enough to only check the main bresenham line between them.
- If the path between two cells has a slope of 45°, Δx equals Δy, in this state three bresenham lines must be checked (the main bresenham line and two internal bresenham lines).
- Otherwise, it must check five bresenham lines (the main bresenham line, two internal bresenham lines and two external bresenham lines).

According to the x and y values of input cells, the internal and external bresenham lines are defined in four cases. So, case 1 is defined when (xi≤xj and yi≤yj), the corridor is defined with the five bresenham lines (a main bresenham line, two internal bresenham lines and two external bresenham lines). It is shown in fig. 4.10. Case 2 (xi>xj and yi≤yj), case 3 (xi≤xj and yi>yj) and case 4 (xi>xj and yi>yj) are obtained from case 1 mirrors. Note, the main bresenham line from (xi,yi) to (xj yj) is the same one in the four cases.

Case 1) if xi≤xj and yi≤yj → First internal bresenham line from (xi+1,yi) to (xj,yj-1)
Second internal bresenham line from (xi,yi+1) to (xj-1,yj)
First external bresenham line from (xi+2,yi) to (xj,yj-1)
Second external bresenham line from (xi,yi+1) to (xj-2,yj)

Case 2) if xi>xj and yi≤yj → First internal bresenham line from (xi-1,yi) to (xj,yj-1)

Second internal bresenham line from (xi,yi+1) to (xj+1,yj)

First external bresenham line from (xi-1,yi) to (xj,yj-2)

Second external bresenham line from (xi,yi+2) to (xj+1,yj)

Case 3) if xi≤xj and yi>yj → First internal bresenham line from (xi,yi-1) to (xj-1,yj)

Second internal bresenham line from (xi+1,yi) to (xj,yj+1)

First external bresenham line from (xi,yi-2) to (xj-1,yj)

Second external bresenham line from (xi+1,yi) to (xj,yj+2)

Case 4) if xi>xj and yi>yj → First internal bresenham line from (xi,yi-1) to (xj+1,yj)

Second internal bresenham line from (xi-1,yi) to (xj,yj+1)

First external bresenham line from (xi,yi-1) to (xj+2,yj)

Second external bresenham line from (xi-2,yi) to (xj,yj+1)

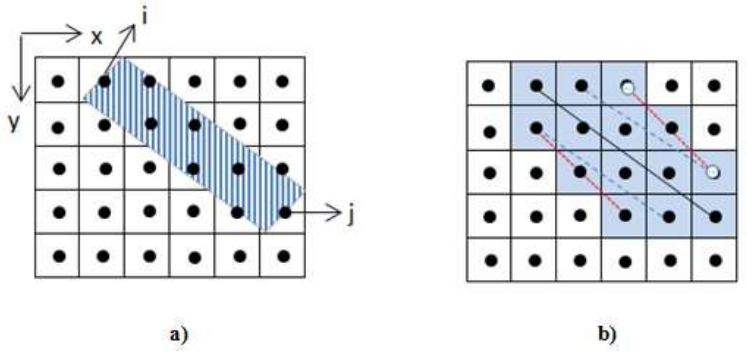


Fig.4.10.a)The corridor between cell (xi,yi) and cell (xj,yj). b) The main bresenham line is the solid line (black line), the internal bresenham lines are the dash-lines (blue lines) and the external lines are dot-lines (red lines).

The different corridors are illustrated in fig.4.11.

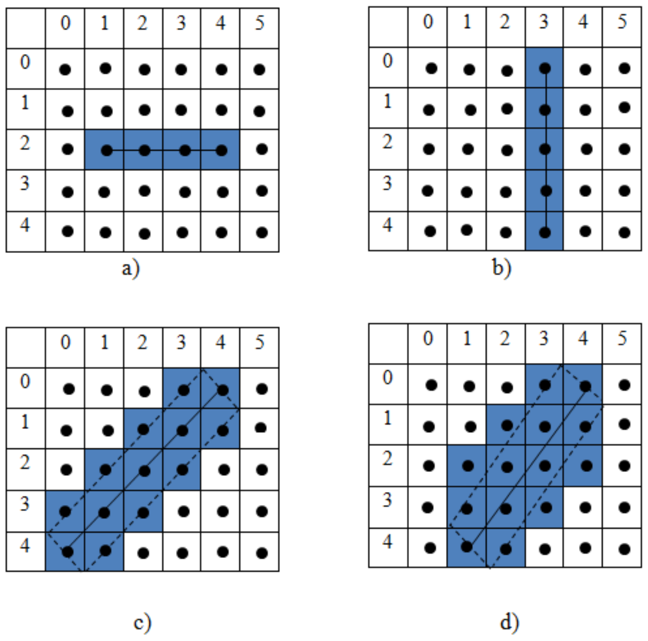


Fig. 4.11. A presentation of navbresenham algorithm for the different cases: a) an horizontal corridor b) a vertical corridor c) a corridor with slope=45° d) a corridor with slope≠45°.

Input of this procedure is cell (xi,yi) and cell (xj,yj). The objective of this procedure is to diagnose a collision-free straight path between two cells. If this path is found, output of the procedure is true and otherwise output of the procedure is false.

First, this algorithm considers if the values x or the values y be equal together or in other word two cells are in horizontal or vertical trajectory, in this case only the main bresenham line is checked. Otherwise two internal bresenham lines are checked in addition to the main bresenham line, and if the absolute difference of the x values is different from the absolute difference of the y values (the slope of corridor is not 45°) then the two external bresenham lines are checked in addition to the previous bresenham lines.

Note that the output of the bresenham algorithm is true, if this line can be drawn using only empty cells. In the navbresenham algorithm, if all those bresenham lines (one, three or five) can be drawn according to the condition of the algorithm, then it is concluded that there is not any obstacle in the corridor between the two cells and there is a collision-free path between them. The pseudo code of this algorithm is presented in code 4.6.

Code 4.6. The navbresenham algorithm pseudo-code

```
input: cell (xi,yi) and cell (xj,yj)
if (xi=xj) then found← call Bresenham(xi,yi,xj,yj)
else if (yi=yj) then found← call Bresenham(xi,yi,xj,yj)
    else
        Determine cell (xi,yi) and cell (xj,yj) are in which case
        b1← call Bresenham(xi,yi,xj,yj)
        b2← call first internal Bresenham line
        b3← call second internal Bresenham line
        if (|xi-xj|≠|yi-yj|) then
            b4 ← call first external Bresenham line
            b5 ← call second external Bresenham line
        else
            b4 ←true, b5←true
        if (b1= true and b2= true and b3= true and b4= true and b5= true) then
            find← true
        else
            find ← false
 return find
 output: find
```

4.2.5. The collision_point procedure

The objective of this procedure is to determine the collision cell of a corridor with an obstacle. The corridor is considered between the current cell and the target cell. Input of this procedure is the current cell and the target cell. Output of this procedure is the collision point.

The bresenham and navbresenham procedures are used in this procedure. In fact, the information is given by the bresenham procedure, which gives as one of its outputs the last empty cell of the bresenham line when an obstacle is found (see subsection 4.2.3). The pseudo code of this algorithm is presented in code 4.7.

Code 4.7. The collision_point algorithm pseudo code

```
input: the current cell (xi,yi) and the target cell (xj,yj)
(xm,ym) ← (xi,yi) , c←0
repeat
  1) if (c >0) then
       (xj,yj) ← (xm,ym)
  2) if (xi=xj) then
       find← call Bresenham(xi, yi, xj, yj)
       if (find=false  and cell(xt,yt)=0) then (xm,ym) ← (xt,yt)
       else (xm,ym) ← (xi,yi)
  3) else if (yi=yj) then
            find← call Bresenham(xi, yi, xj, yj)
            if (find=false  and cell(xt,yt)=0) then (xm,ym) ← (xt,yt)
            else (xm,ym) ← (xi,yi)
          else if cell i and cell j are in case k then
              a)  b1← call Bresenham(xi, yi, xj, yj)
              b)   if (b1=false) then (xm,ym) ← (xt,yt)
              c)  b2← call first internal Bresenham line
              d)   if (b2=false and b1=true) then (xm,ym) ← (xt,yt)
              e)  b3← call second internal Bresenham line
              f)   if (b3=false and b1= true) then (xm,ym) ← (xt,yt)
              g)   if (b1=true and b2= true and b3= true) then
                       if (|xi-xj|≠|yi-yj|) then
                            i.  b4 ← call first external Bresenham line
                            ii.  if (b4=false) then (xm,ym) ← (xt,yt)
                            iii. b5 ← call second external Bresenham line
                            iv.  if (b5=false) then (xm,ym) ← (xt,yt)
  4) c ← c+1
until navbresenham(xi, yi, xm, ym)=false
output: the collision cell (xm,ym)
```

The bresenham procedure stores in (xt,yt) the last empty cell of the line when an obstacle is found (see code 4.2). The collision_point procedure calls the necessary bresenham lines depending on whether the corridor is horizontal or vertical, 45°, or other slope as explained in subsection 4.2.4. It uses the last empty cell of the bresenham line from the current cell to the target if this line finds an obstacle. However, sometimes a line finds no obstacle, even if there is an obstacle in any of the other lines considered for the full corridor. In those cases, the called “internal” lines are considered first, and if they also don’t find an obstacle, then the called "external" lines are considered. The line from current cell to the target cell has precedence for determining the collision cell because this is the most centered point of collision. The collision cells obtained with the auxiliary lines are only considered when the central line is empty.

As an example, the collision point specified in fig. 4.12. In this fig. 4.12., the corridor is displayed by dash line. In this case, the central line finds the collision cell.

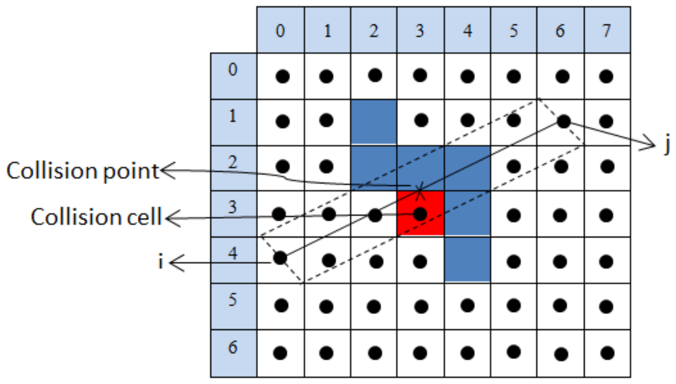


Fig. 4.12. A representation of collision_point algorithm

4.2.6. The create_paths procedure

This procedure creates different paths from the collision cell. In essence, this procedure ensures the mobile robot moves around the obstacle. Input of this procedure is the collision cell and the target cell. The outputs are the different possibilities for movement. After detecting the collision cell, the mobile robot must move around the obstacle until it has a straight line to the target cell or it meets a new collision cell in a new obstacle. A representation of this algorithm is illustrated in fig. 4.13.

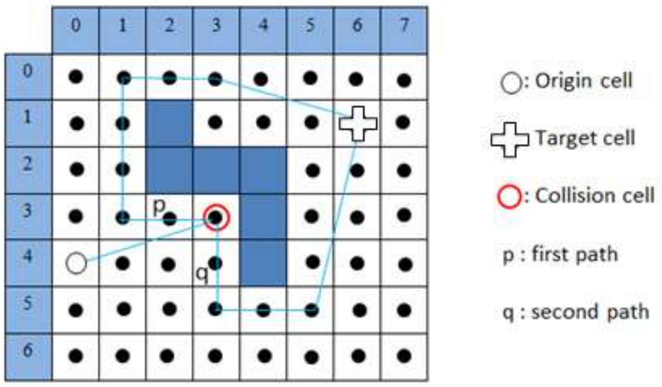


Fig. 4.13. A representation of create_paths algorithm

The pseudo code of this algorithm is presented in code 4.8. In this pseudo code, first, the obstacle number aside the collision cell is determined by calling the points_surrounding procedure. In this pseudo code, c is the number of paths and each path sets by a flag and bottom path, top path, left path and right path set by bflag, tflag, lflag and rflag respectively. Note first path is specified by cell p and also second path is determined by cell q. The movement in four directions is considered. In the movement in each direction, if the next cell is aside the same obstacle of the collision cell, the movement is possible.

Code 4.8. The create_paths algorithm pseudo code

```
Input: the collision cell (nx,ny) and the target cell (tx,ty)
c←0
n←points_surrounding(map, cell (nx,ny)) //n is the obstacle number aside the input cell
bflag←0                                //bottom movement
If (cell[nx][ny+1]=0) then
    If (cell[nx+1][ny]≠0 or cell[nx-1][ny]≠0) then bflag=1
    If (cell[nx+1][ny+1]≠0 or cell[nx-1][ny+1]≠0) then bflag=1
    m←points_surrounding(map, cell (nx,ny+1))
If (bflag=1  and n=m) then (px, py) ← (nx,ny+1), c←c+1
tflag←0                                //top movement
If (cell[nx][ny-1]=0) then
    If (cell[nx+1][ny]≠0 or cell[nx-1][ny]≠0) then tflag=1
    If (cell[nx+1][ny-1]≠0 or cell[nx-1][ny-1]≠0) then tflag=1
    m←points_surrounding(map, cell (nx,ny-1))
If (tflag=1 and n=m) then
   If (c=0) then (px, py) ← (nx,ny-1), c←c+1
   else  (qx, qy) ← (nx,ny-1), c←c+1
lflag←0                                //left movement
If (cell[nx-1][ny]=0) then
    If (cell[nx][ny-1]≠0 or cell[nx][ny+1]≠0) then lflag=1
    If (cell[nx-1][ny-1]≠0 or cell[nx-1][ny+1]≠0) then lflag=1
    m←points_surrounding(map, cell (nx-1,ny))
If (lflag=1 and n=m) then
   If (c=0) then (px, py) ← (nx-1, ny), c←c+1
  else (qx, qy) ← (nx-1,ny), c←c+1
rflag←0                                //right movement
If (cell[nx+1][ny]=0) then
    If (cell[nx][ny-1]≠0 or cell[nx][ny+1]≠0) then rflag=1
    If (cell[nx+1][ny-1]≠0 or cell[nx+1][ny+1]≠0) then rflag=1
    m←points_surrounding(map, cell (nx+1,ny))
If (rflag=1 and n=m) then
   If (c=0) then (px, py) ← (nx+1,ny), c←c+1
  else  (qx,qy) ← (nx+1,ny), c←c+1
output: (px,py) and (qx,qy)
```

4.2.7. The optimization procedure

The objective of this procedure is to process both the solution provided by the Dijkstra and A* algorithms and the different paths of the HCTNav tree. The input of this procedure is the path tree and its output is the optimized tree. The basis of the procedure is that it may be possible for the robot to travel from a node to another node of the path skipping the intermediate nodes of the path. This procedure leads to a reduction of the path lengths.

The procedure iterates along the nodes of the path checking if a node i can reach, without any collision, to the node i+2, i+3, etc., until the target node or a collision is found. Then every intermediate node is removed from the path. The pseudo code of this algorithm is found in Code 4.9.

To make a fair comparison between HCTNav, Dijkstra and A* every algorithm's solutions are processed by this procedure. Considering the graph used by Dijkstra and A*, their solutions can be only improved when a diagonal (45^o movement) is preceded or succeeded by a vertical or horizontal path and no obstacle is found between these two lines. An example of the application of this procedure is found in fig. 4.14.

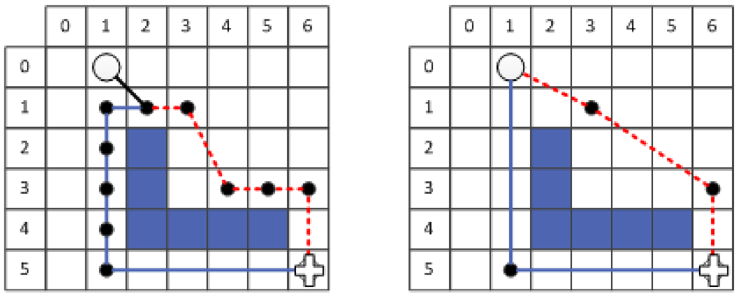


Fig. 4.14. The left figure shows the tree created by HCTNav and the right figure shows the final paths after optimization. The origin point is marked with a circle and the target node with a plus.

Code 4.9. The optimization procedure pseudo code

```
Input: the path tree
(nx,ny) ← (ox,oy) //origin
while (nx,ny) ≠ (tx,ty)  do        //target not reached
    i ← 2
    while navbresenham(nx,ny,nx+i,ny+i) = clear do
        i ← i+1
    i ← i -1
    create edge (nx,ny,nx+i,ny+i)
    (nx,ny) ← (nx+i,ny+i)
Output: the optimized tree
```

4.3. Experimental results

In this section a comparison is presented between HCTNav algorithm and the A* and Dijkstra algorithms, which were presented in previous sections. This section will focus on comparing the proposed algorithm with the A* and the Dijkstra algorithms. Three aspects can be compared: 1) final path length, 2) memory requirements and 3) execution time. The last aspect, execution time, is the least relevant in our case as the execution time of the algorithm in the μprocessor is not significant compared to the robot's movement time. Therefore, first a test suit is presented in subsection 4.3.1 which is used in both of experiments (path length comparison and dynamic memory comparison). Then subsection 4.3.2 and subsection 4.3.3 will show the results regarding the final path length and the memory requirements of each algorithm.

4.3.1. Test suite

The three algorithms have been executed for the following test set:

- A set of 42 small maps. The size of the maps is 15 columns and 10 rows.
- A set of 12 large maps. These large maps have been created as the scaled and repeated versions of three of the former small maps. The size of these maps is 120 columns and 80 rows, 64 times larger than the equivalent small maps.

The small maps set and the big maps set are included in Appendix A and Appendix B respectively. Each algorithm has been implemented in a full program that receives the map as a plain text file, the origin and target positions. The three algorithms have been implemented using ANSI-C language, aiming for its execution in an embedded processor. The result is the sequence of cells for the final path or a message showing that there is no possible solution.

4.3.2. Comparison and discussion on the path length results of HCTNav versus A* and Dijkstra algorithms

The first comparison focuses on comparing the path length provided by the three algorithms. The three algorithms have been executed for every possible path in each map, registering the final path length. To ensure an equitable comparison, the optimization procedure has been also applied to the solution provided by the Dijkstra and A* implementations. It must be noted that although the initial solutions provided by Dijkstra and A* have the same length, the paths can be different. Hence, the final path lengths of these two algorithms may differ in some cases after optimization.

For each path, the resulting path lengths have been compared leading to 13 different possible combinations of results. An explanation is necessary about these cases. The optimization procedure is applied to the three algorithms and the results are compared after this optimization. Before the optimization, path length obtained from the Dijkstra algorithm and the A* algorithm equal, although the paths can be different. But after applying optimization procedure on both algorithms sometimes the values are different.

In case 1 ($PL_{HCTNav}=PL_{A*}=PL_{Dijkstra}$), the path length of the three algorithms is equal. This case is significant for analyzing, because it shows the three algorithms obtain the shortest path length. Case 2 ($PL_{HCTNav}=PL_{A*}<PL_{Dijkstra}$) shows the states that the path length of the HCTNav algorithm is equal to the path length of the A* algorithm, but these paths are shorter than the obtained path of the Dijkstra algorithm. The obtained path length of the Dijkstra and A* algorithms is equal, but after applying the optimization procedure, the path length is different. In case 3 ($PL_{HCTNav}=PL_{Dijkstra}<PL_{A*}$), the path length of the HCTNav algorithm is equal to the path length of the Dijkstra algorithm and the obtained paths are shorter than the obtained path of the A* algorithm. This case is reverse of case 2. Case 4 ($PL_{HCTNav}<PL_{A*}=PL_{Dijkstra}$) is important and shows the superiority of the HCTNav algorithm than the Dijkstra and A* algorithms, because the obtained path of the HCTNav algorithm is shortest. The worst

path length of the HCTNav algorithm is shown in these cases 11 ($PL_{HCTNav} > PL_{A*} = PL_{Dijkstra}$), case 12 ($PL_{HCTNav} > PL_{A*} > PL_{Dijkstra}$) and case 13 ($PL_{HCTNav} > PL_{Dijkstra} > PL_{A*}$).

For the cases in which there are differences between the algorithms, the penalization in the path length has been measured. The penalization, measured as percentage of the increase in path length against the best solution is calculated. We need to measure the penalization introduced by each algorithm. For this reason, the following values are obtained:

Penalization of HCTNav: $$((\sum_{i=0}^{n} \frac{(PL_{HCTNav})_i - MIN(PL_{HCTNav}, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{HCTNav}, PL_{A*}, PL_{Dijkstra})_i})/n) \cdot 100$$

Penalization of A* : $$((\sum_{i=0}^{n} \frac{(PL_{A*})_i - MIN(PL_{HCTNav}, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{HCTNav}, PL_{A*}, PL_{Dijkstra})_i})/n) \cdot 100$$

Penalization of Dijkstra: $$((\sum_{i=0}^{n} \frac{(PL_{Dijkstra})_i - MIN(PL_{HCTNav}, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{HCTNav}, PL_{A*}, PL_{Dijkstra})_i})/n) \cdot 100$$

4.3.2.1. Comparison and discussion in the small maps set

The average path lengths for each of the small maps are presented in table 4.1. Table 4.2 includes for each of these 13 combinations, the percentage of paths and the definition of the case considering the results of the algorithms. Also, the average penalization for each case is presented in this table.

Algorithm	Average path lengths
A*	7.89
Dijkstra	7.90
HCTNav	7.89

Table 4.1. Average path lengths for the small map set

The first conclusion of the presented results is that the average path length of HCTNav is identical to the average path length of A* (7.89) and it is even less than the average path length of Dijkstra (7.90) (see table 4.1). This is a good reason for selecting the HCTnav algorithm.

Case	State	Average of percentages	Penalizations		
			HCTNav	A*	Dijkstra
1	$PL_{HCTNav}=PL_{A*}=PL_{Dijkstra}$	87.93	0.00%	0.00%	0.00%
2	$PL_{HCTNav}=PL_{A*}<PL_{Dijkstra}$	6.21	0.00%	0.00%	1.88%
3	$PL_{HCTNav}=PL_{Dijkstra}<PL_{A*}$	2.16	0.00%	2.33%	0.00%
4	$PL_{HCTNav}<PL_{A*}=PL_{Dijkstra}$	0.58	0.00%	1.46%	1.46%
5	$PL_{Dijkstra}<PL_{HCTNav}<PL_{A*}$	0.10	0.25%	1.03%	0.00%
6	$PL_{HCTNav}<PL_{A*}<PL_{Dijkstra}$	0.48	0.00%	0.68%	1.61%
7	$PL_{HCTNav}<PL_{Dijkstra}<PL_{A*}$	0.25	0.00%	3.21%	1.02%
8	$PL_{HCTNav}=PL_{Dijkstra}>PL_{A*}$	0.03	0.44%	0.00%	0.44%
9	$PL_{Dijkstra}<PL_{HCTNav}=PL_{A*}$	0.07	0.75%	0.75%	0.00%
10	$PL_{A*}<PL_{HCTNav}<PL_{Dijkstra}$	0.12	0.24%	0.00%	0.54%
11	$PL_{HCTNav}>PL_{A*}=PL_{Dijkstra}$	1.49	2.13%	0.00%	0.00%
12	$PL_{HCTNav}>PL_{A*} > PL_{Dijkstra}$	0.04	0.79%	0.20%	0.00%
13	$PL_{HCTNav}>PL_{Dijkstra} > PL_{A*}$	0.54	2.40%	0.00%	0.60%

Table 4.2. Mean percentage for each case obtained and path penalization for each algorithm in the small maps set

The second conclusion is that the HCTNav, A* and Dijkstra algorithms obtain the same path length in most cases (87.93%, case 1 in table 4.2). Even in the cases in which there are differences, those differences are small, being its mean difference under 2.5% in path length for the worst case (case 13 in table 4.2). In fact, A* has slightly better results, because the most common case in which HCTNav is worse that A* (case 13, see table 4.2) has a penalization of 2.40%, while the most common opposite case (case 7) has a penalization of 3.21%. Also, the normalized penalization in cases 11, 12 and 13 is 2.17 % for small maps set according to formula (1).

$$\text{Normalized penalization in case i to j}=\frac{\sum_{n=i}^{j}(\text{P\_HCTNav in case n})(\text{Average of percentage in case n})}{\sum_{n=i}^{j}\text{Average of percentage in case n}} \qquad (1)$$

Table 4.3 merges the results comparing Dijkstra versus HCTNav and Table 4.4 makes the same comparison between A* and HCTNav.

State	cases	Sum of average percentages
$PL_{HCTNav}=PL_{Dijkstra}$	1, 3, 8	90.12
$PL_{HCTNav}<PL_{Dijkstra}$	2, 4, 6, 7, 10	7.64
$PL_{Dijkstra}<PL_{HCTNav}$	5, 9, 11, 13, 12	2.24

Table 4.3. Comparison path length of HCTNav and Dijkstra algorithms

State	cases	Sum of average percentages
$PL_{HCTNav}=PL_{A*}$	1, 2, 9	94.21
$PL_{HCTNav}<PL_{A*}$	3, 4, 5, 6, 7	3.57
$PL_{A*}<PL_{HCTNav}$	8, 10, 11, 12, 13	2.22

Table 4.4. Comparison path length of HCTNav and A* algorithms

Table 4.3 shows that the path length of the HCTNav and Dijkstra algorithms is identical in 90.12% and table 4.4 shows that the path length of the HCTNav and A* algorithms is equal in 94.21%.

The main conclusion is that all of these algorithms are feasible solutions according to path length and the differences are very small. Anyway, the decision of which algorithm should be used in a low cost robot can be taken considering other factors, like memory usage.

4.3.2.2. Comparison and discussion in the big maps set

The average path lengths for each of the big maps are presented in table 4.5. Table 4.6 includes for each of these 13 combinations, the percentage of paths and the definition of the case considering the results of the algorithms. Also, the average penalization for each case is presented in this table.

Algorithm	Average path lengths
A*	55.78
Dijkstra	55.58
HCTNav	56.74

Table 4.5. Average path lengths for the big maps set

As seen in table 4.5, the average path length of the HCTNav (56.74) is very close to the A* (55.78) and Dijkstra algorithm (55.58).

Case	State	Average of percentages	Penalizations		
			HCTNav	A*	Dijkstra
1	$PL_{HCTNav}=PL_{A*}=PL_{Dijkstra}$	54.34	0.00%	0.00%	0.00%
2	$PL_{HCTNav}=PL_{A*}<PL_{Dijkstra}$	7.16	0.00%	0.00%	0.94%
3	$PL_{HCTNav}=PL_{Dijkstra}<PL_{A*}$	5.61	0.00%	1.64%	0.00%
4	$PL_{HCTNav}<PL_{A*}=PL_{Dijkstra}$	0.97	0.00%	0.78%	0.78%
5	$PL_{Dijkstra}<PL_{HCTNav}<PL_{A*}$	1.93	0.54%	1.72%	0.00%
6	$PL_{HCTNav}<PL_{A*}<PL_{Dijkstra}$	1.86	0.00%	0.63%	1.21%
7	$PL_{HCTNav}<PL_{Dijkstra}<PL_{A*}$	7.86	0.00%	2.21%	0.73%
8	$PL_{HCTNav}=PL_{Dijkstra}>PL_{A*}$	0.61	0.17%	0.00%	0.17%
9	$PL_{Dijkstra}<PL_{HCTNav}=PL_{A*}$	0.74	0.57%	0.57%	0.00%
10	$PL_{A*}<PL_{HCTNav}<PL_{Dijkstra}$	0.78	0.23%	0.00%	0.75%
11	$PL_{HCTNav}>PL_{A*}=PL_{Dijkstra}$	4.91	2.70%	0.00%	0.00%
12	$PL_{HCTNav}>PL_{A*} > PL_{Dijkstra}$	6.45	3.53%	0.50%	0.00%
13	$PL_{HCTNav}>PL_{Dijkstra} > PL_{A*}$	6.78	3.74%	0.00%	0.39%

Table 4.6. Mean percentage for each case obtained for each algorithm in the big maps set

Table 4.6 shows that more than 80% paths are identical or shorter than the Dijkstra or A* algorithms. A* has slightly better results, because the most common case in which HCTNav is worse than A* (case 13) has a penalization of 3.74%, while the most common opposite case (case 7) has a penalization of 2.21%. Also, the normalized penalization in cases 11, 12 and 13 is 3.38 % for big maps according to formula (1) in previous subsection.

Table 4.7 merges the results comparing Dijkstra versus HCTNav and Table 4.8 makes the same comparison between A* and HCTNav.

State	cases	Sum of average percentages
$PL_{HCTNav}=PL_{Dijkstra}$	1, 3, 8	60.56
$PL_{HCTNav}<PL_{Dijkstra}$	2, 4, 6, 7, 10	18.63
$PL_{Dijkstra}<PL_{HCTNav}$	5, 9, 11, 13, 12	20.81

Table 4.7. Comparison path length of HCTNav and Dijkstra algorithms

State	cases	Sum of average percentages
$PL_{HCTNav}=PL_{A*}$	1, 2, 9	62.24
$PL_{HCTNav}<PL_{A*}$	3, 4, 5, 6, 7	18.23
$PL_{A*}<PL_{HCTNav}$	8, 10, 11, 12, 13	19.53

Table 4.8. Comparison path length of HCTNav and A* algorithms

4.3.3. Comparison and discussion on the dynamic memory results of HCTNav versus A* and Dijkstra algorithms

The second comparison focuses on the memory that each program requires for solving the problem. All three algorithms have been implemented using dynamic memory for storing the map, generating the graph (when required) and generating the solution. The measurement of the allocated memory is done using the Valgrind application. The results are processed with Massif-msprint tool.

4.3.3.1. Comparison and discussion in the small maps set

Similar to the previous experiment, we have measured the memory for every possible path in each map of the set. Fig. 4.15 shows the maximum memory in each map for the small-map set.

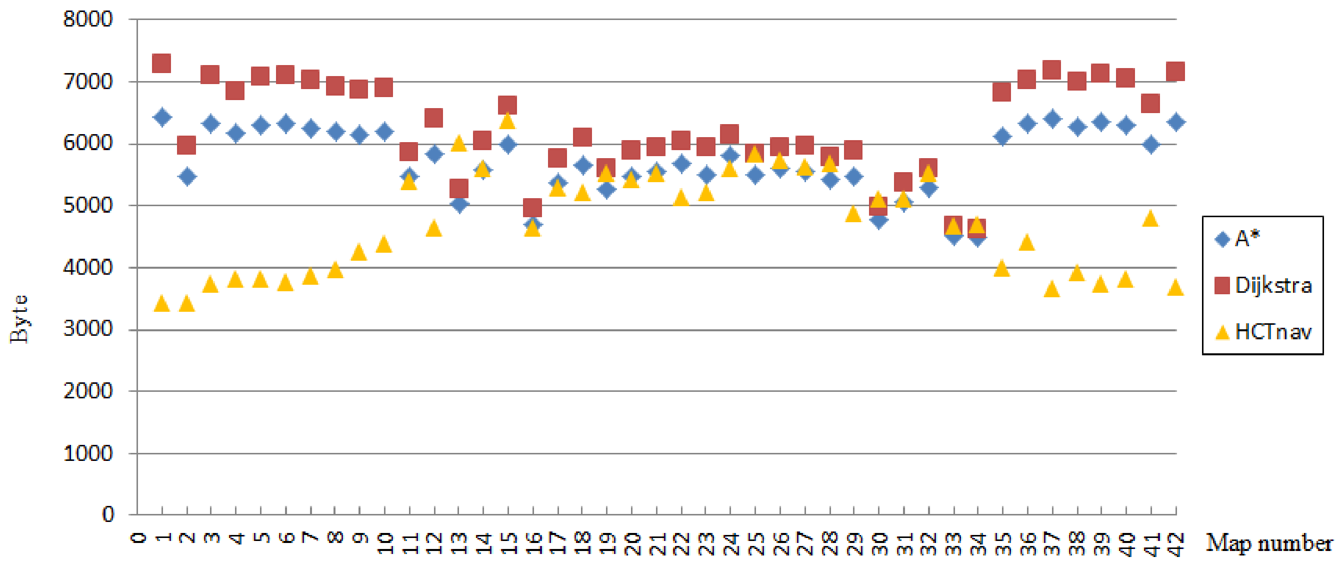


Fig. 4.15. Comparison of the maximum memory requirement for each small map

Fig. 4.15 shows that dynamic memory of the Dijkstra and A* algorithms is the maximum in maps which have few obstacles and many free cells and it is the minimum in maps which have least free cell (see all small map in Appendix B). For example, the dynamic memory of the Dijkstra algorithm is the minimum in map-34 (4620 Byte) and the dynamic memory of the A* algorithm is the minimum in map-34 (4484 Byte). Also, the dynamic memory of the Dijkstra algorithm is the maximum in map-01 (7272 Byte) and the dynamic memory of the A* algorithm is the maximum in map-01 (6424 Byte). Map-34 has least free cells whereas map-01 has the maximum free cell. But about HCTNav algorithm, its dynamic memory is the minimum in maps which have the least obstacles and it is the maximum in maps which have the most obstacles. For this reason, HCTNav has a minimum dynamic memory in map-01 (3432 Byte) and map-02 (3432 Byte), because they have no obstacles and it has a maximum dynamic memory in map-15 (6372 Byte), because it has the maximum number of obstacle (28) in this map set.

Note the total number of considered paths from the origin cell to the target cell in the HCTNav is exponential in terms of the number of obstacles according to this formula:

$$\text{Number of all paths} = 2^{\text{number of obstacles}}$$

E.g. if the number of obstacles is the three, there are the eight paths from the origin cell to the target cell.

The critical value is the maximum, because is the value it has to be considered for robot design. The memory peaks of each algorithm is shown in table 4.9 for all small maps set.

Type algorithm	Memory peaks (Byte)
A*	6424
Dijkstra	7272
HCTNav	6372

Table 4.9. Comparison memory peaks of three algorithms for all small maps set

Table 4.9 shows that memory peak of the HCTNav (6372 Byte) is lower than the memory peaks of the Dijkstra (7272 Byte) and A* (6424 Byte) algorithms.

4.3.3.2. Comparison and discussion in the big maps set

Fig. 4.16 shows the maximum memory in each map for the big-map set.

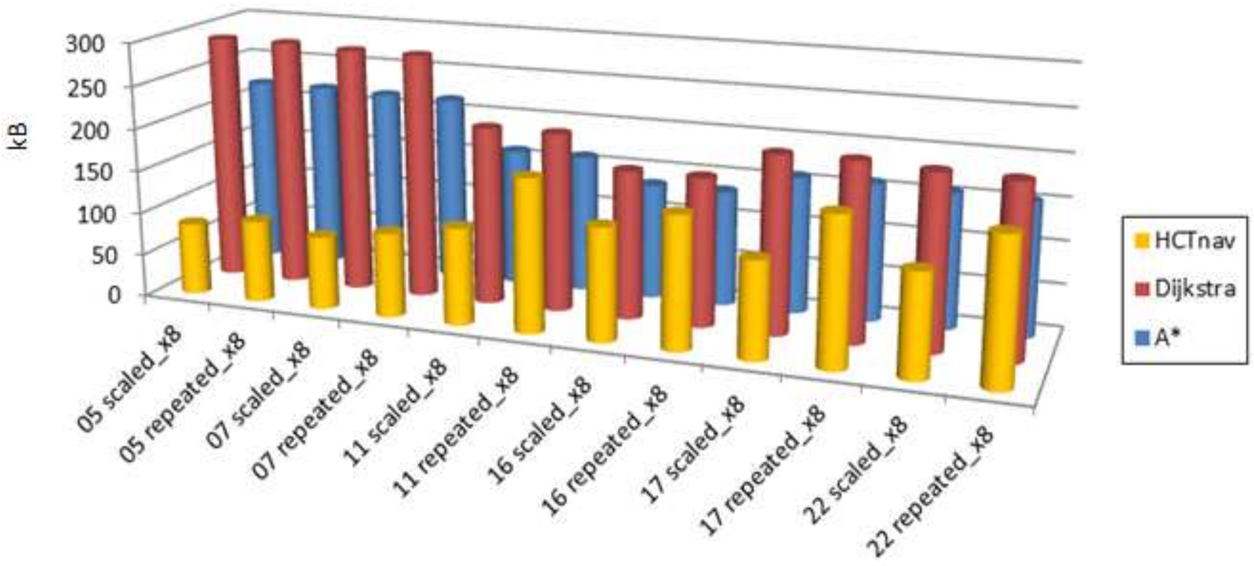


Fig. 4.16. Comparison of the maximum memory requirement for each big map

This map set includes 12 maps. They are defined in terms of two types of scaling. The first type maps are created by scaling 8 times each map, they are called scaled_x8. The second type maps are created by replicating the map by the 8 times, these are called repeated_x8.

As seen in fig. 4.16, the Dijkstra and A* algorithms have a minimum dynamic memory in map-16 scaled_x8 and map-16 repeated_x8, because these maps have the least free cells and these algorithms have their maximum dynamic memory in map-05 scaled_x8, because this map has the most free cells. The HCTNav algorithm has a minimum dynamic memory usage in map-05 scaled_x8 which has the minimum number of obstacles. It has its maximum dynamic memory usage in map-11 repeated_x8, as it has the maximum number of obstacles in the map set.

The memory peaks of each algorithm is shown in table 4.10 for all big maps set.

Type algorithm	Memory peaks (kB)
A*	219.25
Dijkstra	289.28
HCTNav	177.86

Table 4.10. Comparison memory peaks of three algorithms for all big maps set

Table 4.10 shows that memory peak of the HCTNav (177.86 kB) is lower than the memory peaks of the Dijkstra (289.28 kB) and A* (219.25 kB) algorithms.

It can be surprising that Dijkstra or A* can have worse results than HCTNav in some cases, because Dijkstra and A* should always give the shortest path. That is only true when the graph is a perfect one, i.e. all nodes are connected to all other nodes. However, such a graph would use an enormous memory for grid maps, especially for big maps. That is why the simplification of only using one-hop connections in the graph is used, so Dijkstra and A* can be implemented in low cost systems. If not, the differences in memory consumption would have been much bigger, because even a minimum one-hop-step graph requires a considerable amount of memory, while the differences in path length would be very similar. For a fair comparison, we have chosen to use one-hop connections, because it is the optimal implementation of Dijkstra and A* for embedded systems.

4.4. Path length discussion through examples

In order to understand how different path lengths are obtained for the three algorithms, cases 2, 4, 11, 12, and 13 are being presented in this section. For this purpose, two small maps are used. Map-05 is simple with one obstacle and map-17 is a complex one with nine obstacles. Small maps are used for the sake of clarity.

4.4.1. Example of case 2

In case 2 ($PL_{HCTNav}=PL_{A*}<PL_{Dijkstra}$), the path length of A* algorithm is equal to the path length of HCTNav algorithm and lower than the path length of Dijkstra algorithm. This case is shown in two examples. First, this case is considered in map-05 with the origin cell (0,0) that is shown with circle and the target cell (5,9) that is shown with plus in fig. 4.17.

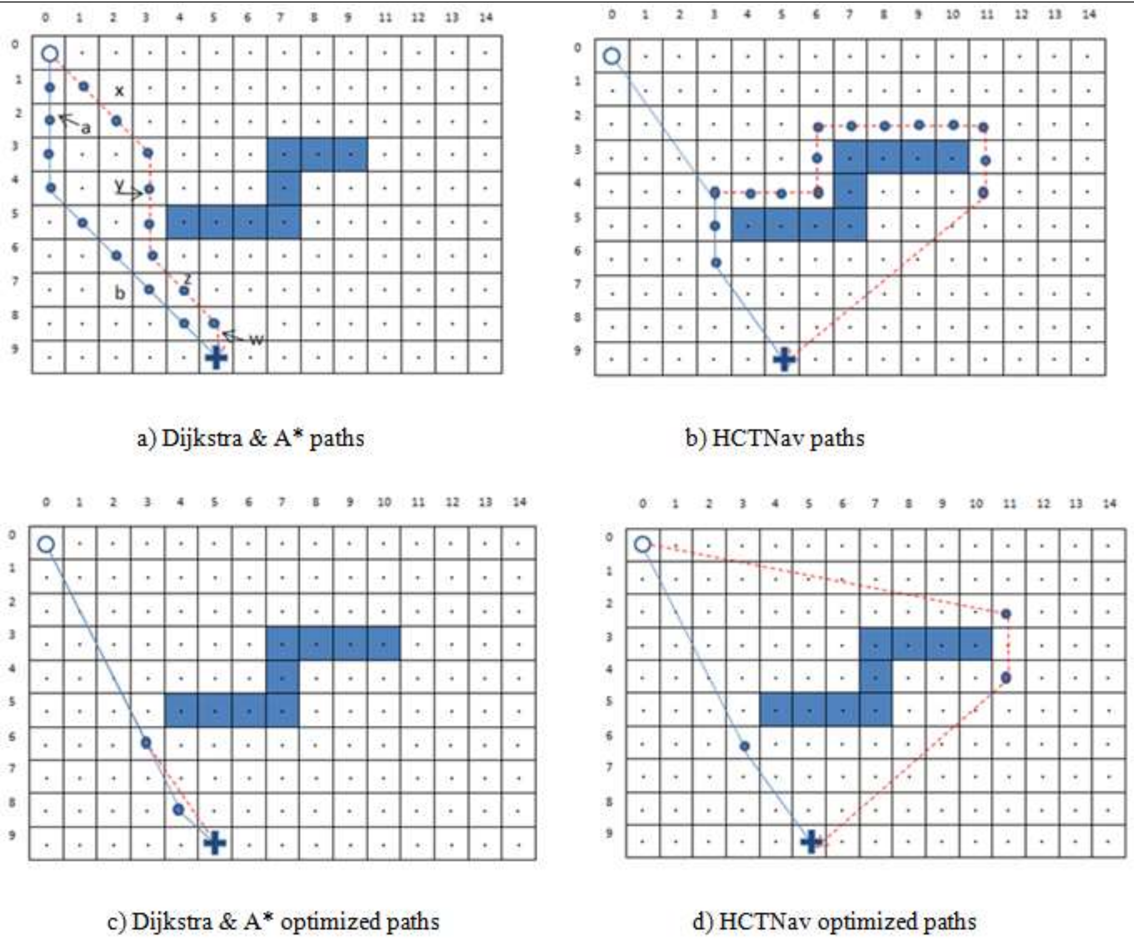


Fig. 4.17. Example of case 2 in map-05. a) The Dijkstra path is the solid line and the A* path is the dash line. b) Path 1 and path 2 in HCTNav algorithm. c) The optimized paths in Dijkstra and A* algorithms. d) The optimized paths in HCTNav algorithm.

The results of Dijkstra and A* algorithms prior to optimization are shown in fig. 4.17.a). The black line is for the Dijkstra algorithm and the dash line is for the A* algorithm. In the HCTNav algorithm, the collision point is found and obstacle is surrounded loading to two paths. The obtained paths are shown in fig. 4.17.b). As it can be seen in fig 4.17.a) a=y+w and b=x+z, therefore a+b=x+y+z+w=11.071. But after applying the optimization procedure on both results fig. 4.17.c) is obtained. In the fig. 4.17.c) the optimization path in the Dijkstra algorithm is {(0,0), (3,6), (4,8), (5,9)} and the optimization path in the A* algorithm is {(0,0), (3,6), (5,9)}. Sometimes this procedure has good impact on Dijkstra or A* and sometimes is ineffective, it depends on which path is obtained from the algorithm. After optimization on the paths in HCTNav algorithm, fig. 4.17.d) is obtained. The path length is represented for every algorithm before and after optimization from (0,0) to (5,9) in table 4.11.

Type of algorithm	Before optimization	After optimization
A*	11.071	10.314
Dijkstra	11.071	10.358
HCTNav path1	10.606	10.314
HCTNav path2	24.810	20.991

Table 4.11: The path length comparison in the three algorithms from (0,0) to (5,9)

Another example of case 2 is shown on map-17 in fig. 4.18. The path from the origin point= (3,4) to the target point= (10,8) in the three algorithm. Fig. 4.18.a) shows the results of Dijkstra and A* algorithms prior to optimization. Black (solid) line is for Dijkstra algorithm and black/white (dash) line is for the A* algorithm. In HCTNav algorithm, the obtained paths are shown in fig. 4.17.b).

After applying the optimization procedure on results of the Dijkstra and A* algorithms, fig. 4.18.c) is obtained. After optimization on results of the HCTNav algorithm, fig. 4.18.d) is obtained. The path length is represented for every algorithm before and after optimization from (3,4) to (10,8) in table 4.12.

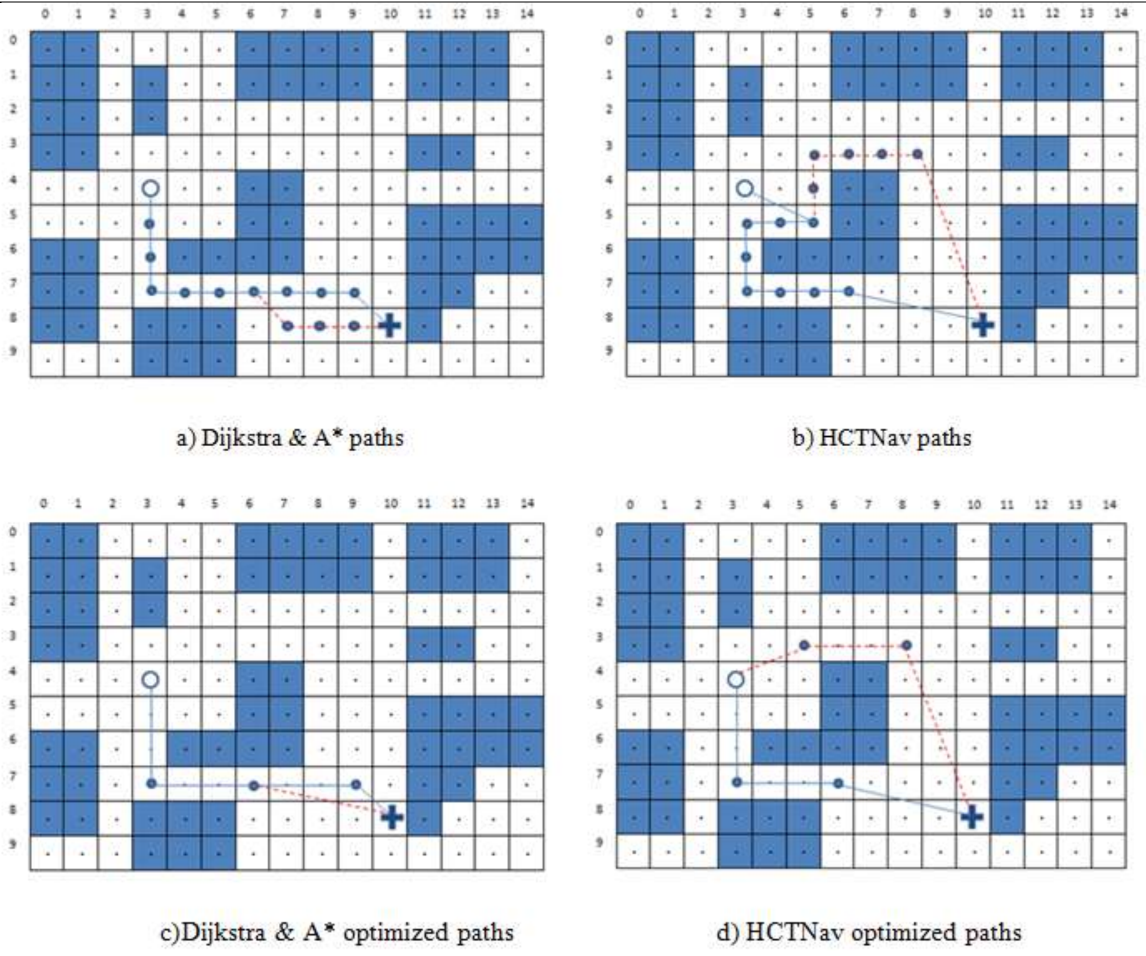


Fig. 4.18. Example of case 2 in map-17. a) The Dijkstra path is the solid line and the A* path is dash line. b) Path 1 and path 2 in HCTNav algorithm. c) The optimized paths in Dijkstra and A* algorithms. d) The optimized paths in HCTNav algorithm.

Type of algorithm	Before optimization	After optimization
A*	10.414	10.123
Dijkstra	10.414	10.414
HCTNav path1	13.359	10.123
HCTNav path2	12.621	10.621

Table 4.12: The path length comparison in the three algorithms from (3,4) to (10,8)

4.4.2. Example of case 4

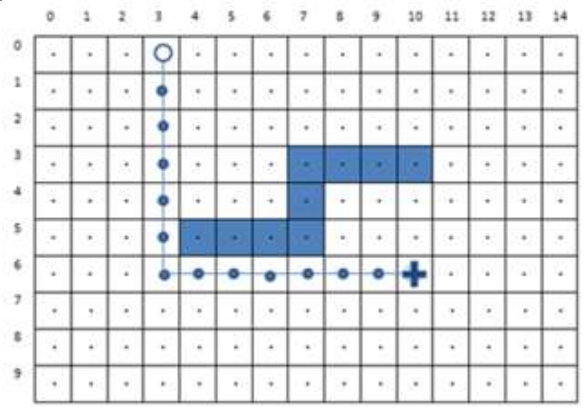
In case 4 ($PL_{HCTNav}<PL_{A*}=PL_{Dijkstra}$), the path length of HCTNav algorithm is lower than the both A* and Dijkstra algorithms. First example of case 4, the path from (3,0) to (10,6) is shown according to fig. 4.19 for every three algorithms. Initial path of the Dijkstra and A* algorithm is equal and it is shown in fig. 4.19.a). After applying the HCTNav algorithm obtains two paths, fig. 4.19.b). It can be seen in fig. 4.19.c) that A* and Dijkstra obtain the same solution before and after optimization. The initially worse case of the path1, leads to a better solution after the optimization.

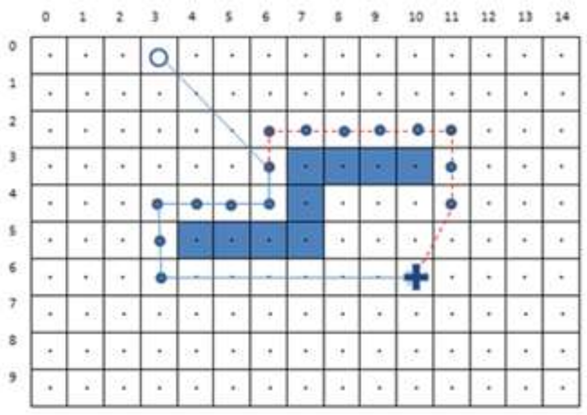
In here the optimization procedure only has effect on the solution obtained by the HCTNav algorithm. After optimization, the resulting paths in the HCTNav algorithm are shown in fig. 4.19.d).

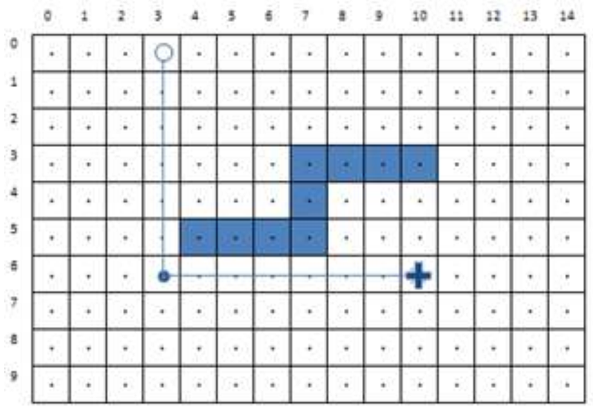
The path length is represented before and after the optimization in every three algorithms is considered in table 4.13.

Type of algorithm	Before optimization	After optimization
A*	13	13
Dijkstra	13	13
HCTNav path1	14.479	12.482
HCTNav path2	17.243	13

Table 4.13: The path length comparison in the three algorithms for path from (3,0) and (10,6)


a)Dijkstra & A* paths


b) HCTNav paths


c) Dijkstra & A* optimized paths

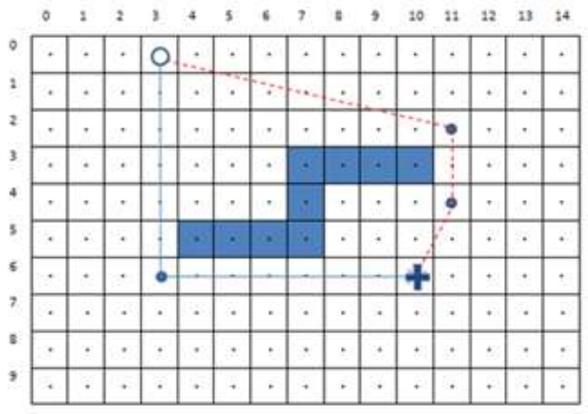
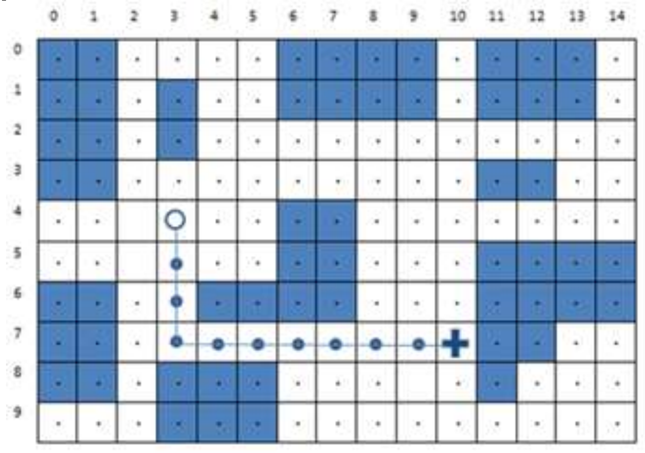
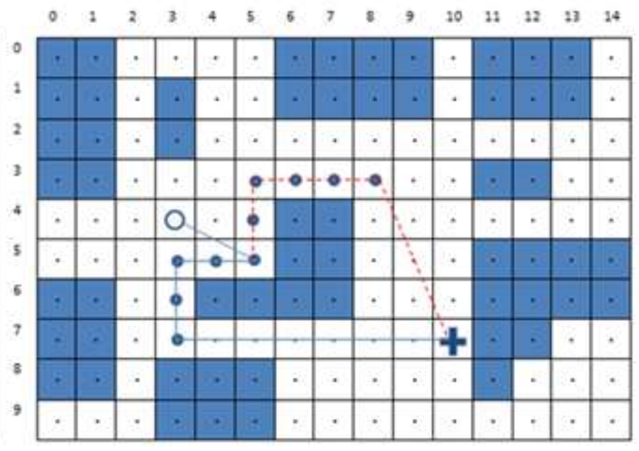
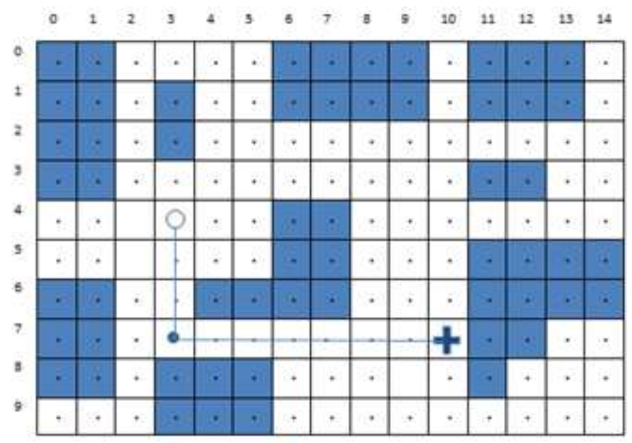

d) HCTNav optimized paths

Fig. 4.19. Example of case 4 in map-05. a) The initial path in the Dijkstra and A* algorithm. b) Path 1 and path 2 in HCTNav algorithm. c) The optimized path in Dijkstra and A* algorithms. d) The optimized paths in HCTNav algorithm.

Another example of case 4 is shown on map-17 in fig. 4.20. The path from the origin point= (3,4) to the target point= (10,7) in the three algorithm. The initial path of the Dijkstra and A* algorithms is shown in fig. 4.20.a). In the HCTNav algorithm, the initial paths are shown in fig. 4.20.b). As can be seen that A* and Dijkstra obtain the same path after optimizing in fig. 4.20.c), and it does not change. But, the new paths are created after optimization by the HCTNav algorithm for creating a better solution. The paths of HCTNav algorithms are shown in fig. 4.20.d) after optimization. The path length is represented before and after the optimization in every three algorithms is considered in table 4.14.


a) Dijkstra & A* paths


b) HCTNav paths


c) Dijkstra & A* optimized paths

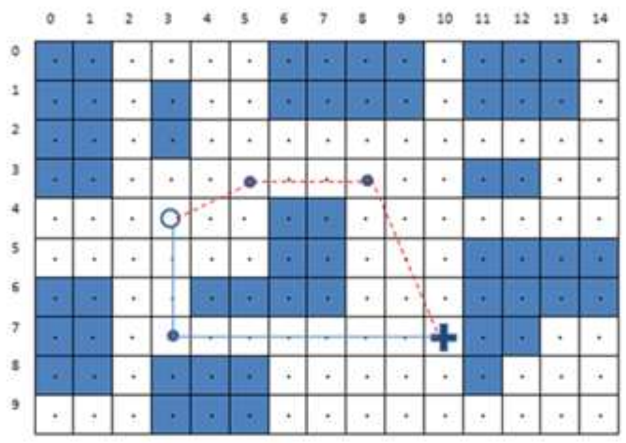

d) HCTNav optimized paths

Fig. 4.20. Example of case 4 in map-17. a) The initial path in the Dijkstra and A* algorithm.b) Path 1 and path 2 in HCTNav algorithm. c) The optimized path in Dijkstra and A* algorithms. d) The optimized paths in HCTNav algorithm.

Type of algorithm	Before optimization	After optimization
A*	10	10
Dijkstra	10	10
HCTNav path1	11.708	9.708
HCTNav path2	13.236	10

Table 4.14: The path length comparison in the three algorithms for path from (3,4) and (10,7)

4.4.3. Example of case 11

In cases 11 (PL_{HCTNav}>PL_{A*}=$PL_{Dijkstra}$), the path length of HCTNav algorithm is longer than the path length of A* algorithm and Dijkstra algorithms. An example of case 11 is shown on map-05 in fig. 4.21. The origin cell is (0,2) and the target cell is (5,8). The initial paths of the Dijkstra and A* algorithms are shown in fig 4.21.a) and the initial path of the HCTNav algorithm are presented in fig. 4.21.b). When the optimization procedure is applied to the obtained solutions of the Dijkstra and A* algorithms, the paths are the same as shown in fig. 4.21.c). The optimized paths in the HCTNav algorithm are presented in fig. 4.21.d). The path length is displayed before and after the optimization in every three algorithms in table 4.15.

Type of algorithm	Before optimization	After optimization
A*	8.071	7.817
Dijkstra	8.071	7.817
HCTNav path1	8.434	7.828
HCTNav path2	24.211	20.211

Table 4.15: The path length comparison in the three algorithms from (0,2) to (5,8)

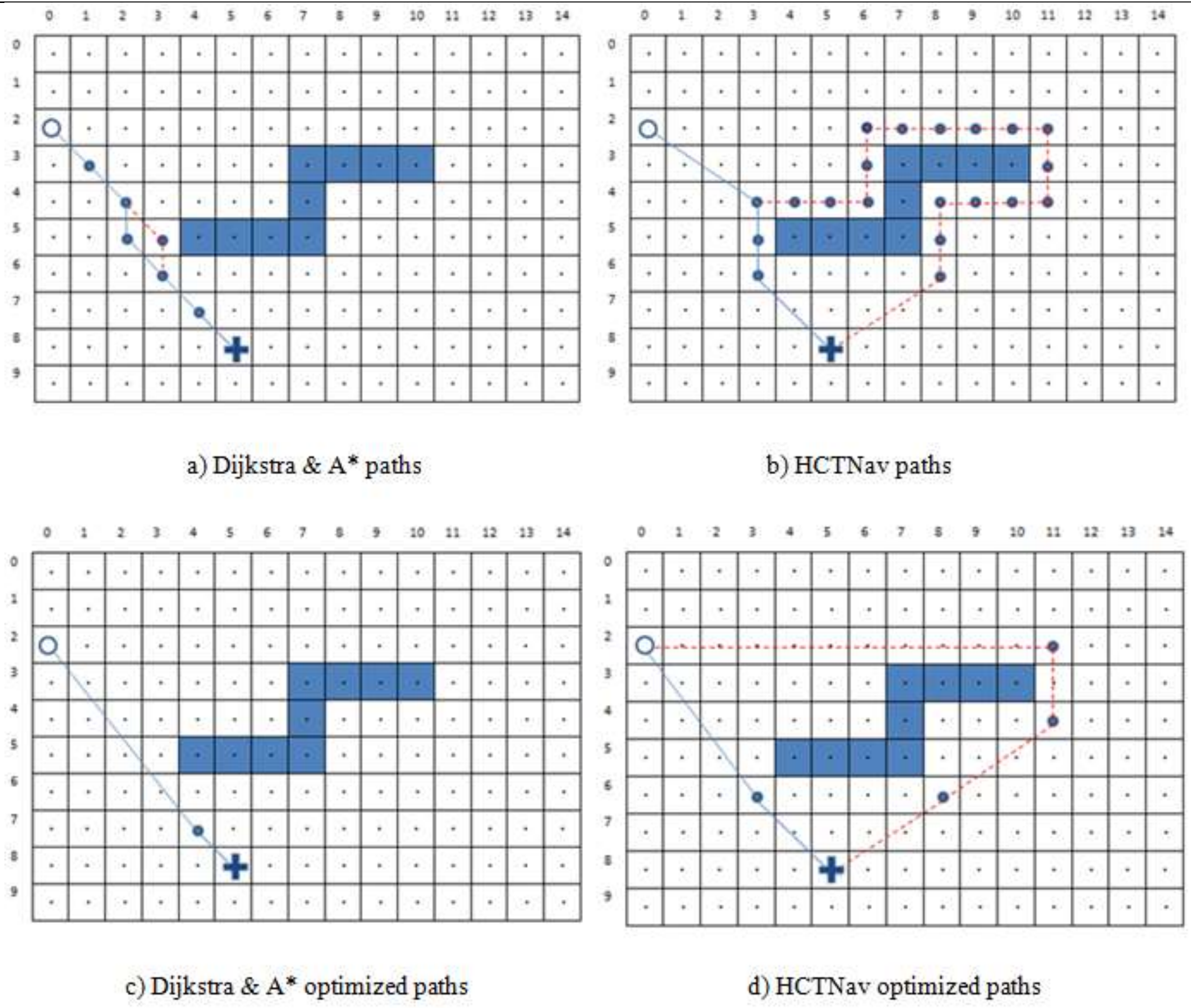


Fig. 4.21. Example of case 11 in map-05. a) The Dijkstra path is the solid line and the A* path is dash line. b) Path 1 and path 2 in HCTNav algorithm. c) The optimized paths in Dijkstra and A* algorithms. d) The optimized paths in HCTNav algorithm.

4.4.4. Example of case 12

In case 12 ($PL_{HCTNav} > PL_{A*} > PL_{Dijkstra}$), the path length of HCTNav algorithm is longer than the path length of A* algorithm and also the path length of A* algorithm is bigger than Dijkstra algorithm. It is presented an example of case 12 for the origin node (5,7) and the target node (5,0) in fig. 4.22. The initial paths in the Dijkstra and the A* algorithms is represented respectively with black and dash lines in fig 4.22.a).

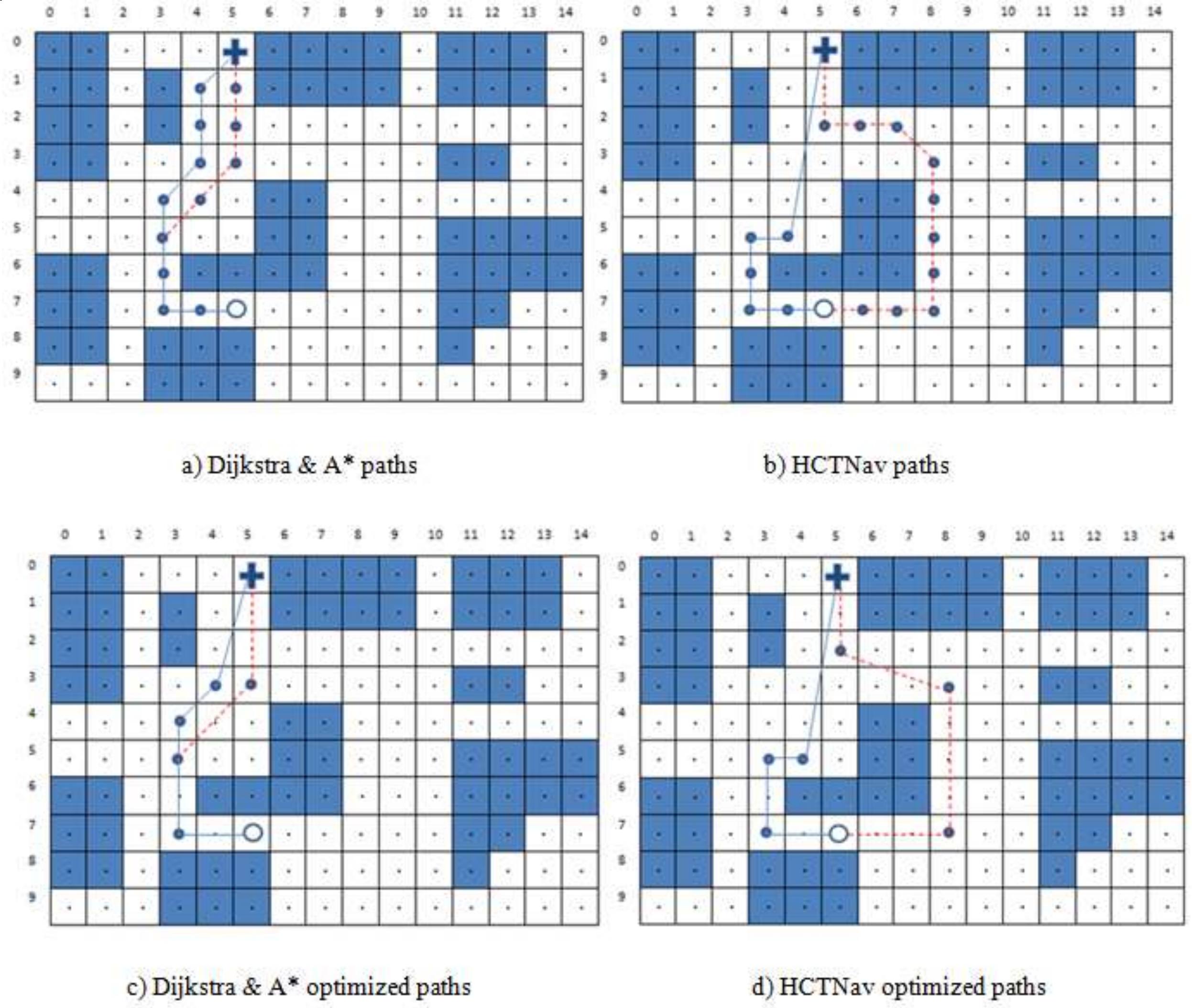


Fig. 4.22. Example of case 12 in map-17. a) The Dijkstra path is the solid line and the A* path is dash line. b) Path 1 and path 2 in HCTNav algorithm. c) The optimized paths in Dijkstra and A* algorithms. d) The optimized paths in HCTNav algorithm.

It is created two paths with the application of the HCTNav algorithm on the origin and the target nodes. Two paths respectively with the black and the dash lines are denoted in fig. 4.22.b). After applying the optimization procedure on results of the Dijkstra and A* algorithms is obtained fig. 4.22.c). The optimized paths in the HCTNav algorithm are shown in fig. 4.22.d). The path length is displayed before and after the optimization in every three algorithms in table 4.16.

Type of algorithm	Before optimization	After optimization
A*	9.828	9.828
Dijkstra	9.828	9.576
HCTNav path1	10.099	10.099
HCTNav path2	12.414	12.162

Table 4.16: The path length comparison in the three algorithms from (5,7) to (5,0)

4.4.5. Example of case 13

In case 13 ($\mathrm{PL}_{\mathrm{HCTNav}}$>$\mathrm{PL}_{\mathrm{Dijkstra}} > \mathrm{PL}_{\mathrm{A*}}$), the path length of HCTNav algorithm is longer than the path length of Dijkstra algorithm and also the path length of Dijkstra algorithm is longer than A* algorithm. An example of case 13 is presented for the origin node (5,5) and the target node (11,4) in fig. 4.23. The paths in the Dijkstra and the A* algorithms is represented respectively with black and dash lines in fig. 4.23.a). Three paths are created with applying the HCTNav algorithm on the origin and the target nodes. The paths are shown with solid line, dashed line and dot line respectively. It is represented in fig. 4.23.b). Fig. 4.23.c) is obtained after applying the optimization procedure on the results of the Dijkstra and A* algorithms. The optimized paths in the HCTNav algorithm are shown in fig. 4.23.d). The path length is displayed before and after the optimization for every algorithm in table 4.17.

Type of algorithm	Before optimization	After optimization
A*	8.414	8.236
Dijkstra	8.414	8.414
HCTNav path 1	9	9
HCTNav path 2	15	14.099
HCTNav path 3	14.236	13.606

Table 4.17: The path length comparison in the three algorithms from (5,5) to (11,4)

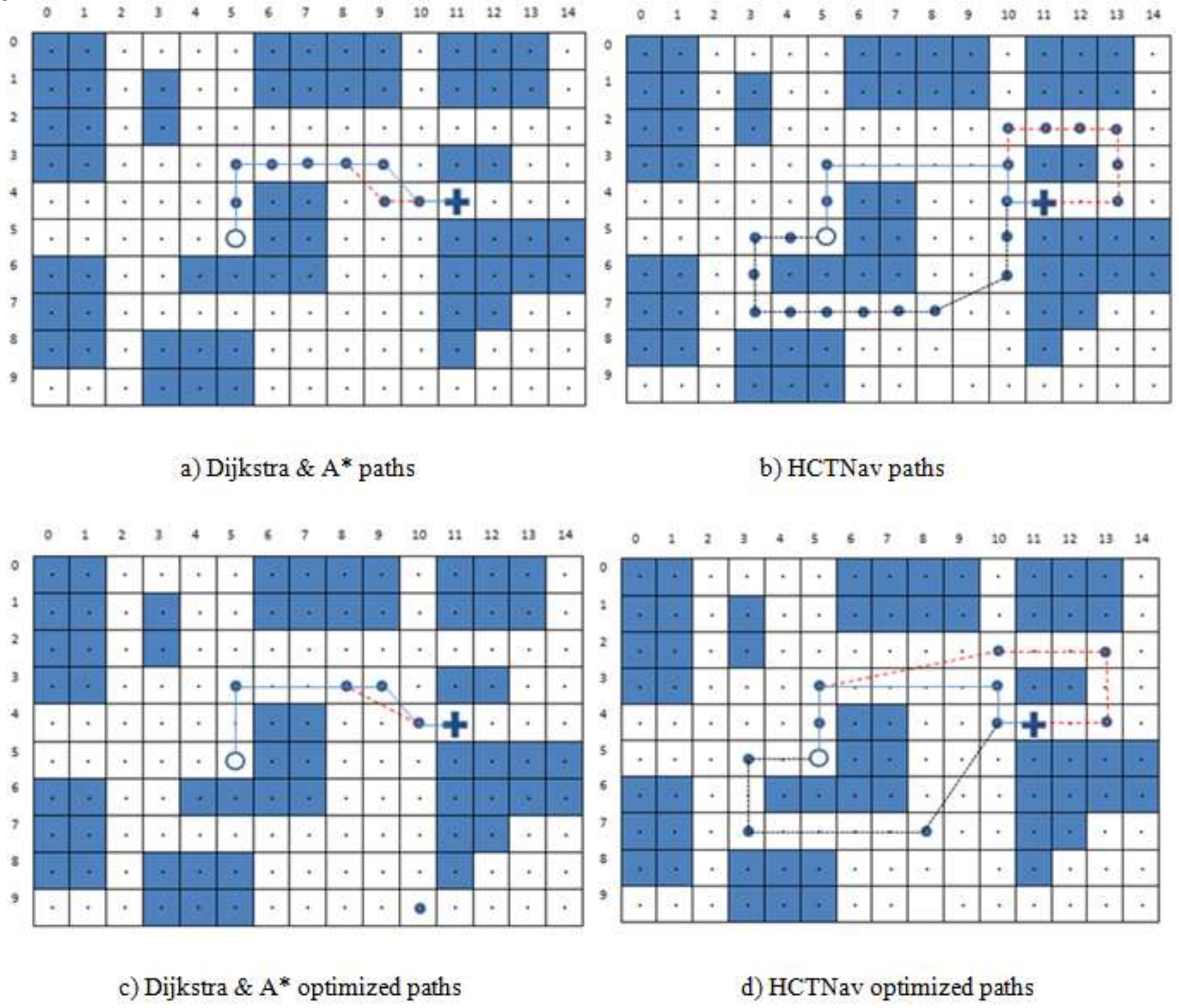


Fig. 4.23. Example of case 13 in map-17. a) The Dijkstra path is the solid line and the A* path is dash line. b) Path 1 and path 2 in HCTNav algorithm. c) The optimized paths in Dijkstra and A* algorithms. d) The optimized paths in HCTNav algorithm.

Chapter 5. NafisNav algorithm

5.1. Algorithm description

In nature, all creatures trying to reach a specified point move towards it, selecting points closer to the target each step. NafisNav algorithm is inspired in this simple rule. On the other hand, most path planning algorithms consider parameters for measuring quality: minimum path length and minimum execution time. As memory consumption is very important in low-cost robots and embedded systems, reducing the needed memory should be a primary goal for these systems. The performance of the proposed algorithm will be evaluated through experiments and comparison with well know algorithms (Dijkstra and A*).

This algorithm, NafisNav, was designed to be run in low-cost robots for indoor navigation on grid based maps. In contrast to traditional path planning algorithms, this algorithm is rather simple and has the following characteristics.

The main idea is similar to the algorithm that a blind person would use. Although the complete grid map is available, the algorithm does not make a priori analysis of the complete map (does not “see” the complete map, only its immediate surroundings), which is the part that consumes more memory in this kind of maps for other algorithms. Dijkstra algorithm always makes the whole graph of the map, and that is its main memory usage, analyzing parts of the maps that are not needed. According to this idea, the algorithm simply moves towards the target from the origin node without considering obstacles until they are found, as a blind person would do. The neighbor cell with the shortest distance to the target is always chosen for obtaining the trajectory, but only among those not already used for the path until now. In this way, loops are avoided. This is true also when obstacles are found. In those cases, some of the neighbor cells are not available (obstacles), but the available neighbor cell with the minimum distance to the target is chosen, even if that distance is greater from the distance of a previous cell in the path (possible when surrounding obstacles). By default only one path from the origin node to the target node is calculated, with an exception: when two neighbors have the same distance to the target node. In this case, one is selected and the other cell is put in a temporary memory, which is called “stack” for future execution. This stack works as a FILO (First Input Last Output), basically as a common stack.

The advantage of the NafisNav algorithm is that it requires less memory, especially in simpler maps where a path can be found without exploring the complete map. The main drawback is that it can lead to non-optimal paths. When reaching an obstacle there is no analysis of other possible solutions, only choosing the available cell with the minimum distance to the target, even if the total path length is finally higher. Therefore, it is expected that the proposed algorithm obtains good memory results at the expense of path length minimization, especially in more difficult maps.

Other important circumstance that must be taken into account is that the idea presented can lead to deadlocks, points in which there is no unused neighbor cell for continuing the search. Therefore, in order to assure that at least one solution is found (even if non-optimal) a backtracking technique is included for exiting from deadlocks. In this case, the algorithm “moves” back to an earlier cell with unvisited neighbors. It is important to take into account that a robot would run the complete algorithm before moving (it would need some milliseconds), so there would be no apparent backtracking, as deadlocks are eliminated during the algorithm, not during the movement.

The NafisNav algorithm was presented in [78]. Also its results in terms of path length and memory consumption were compared with Dijkstra and A* algorithms. It was shown that it proposes a tradeoff between path length and memory consumption with very good results in terms of memory consumption.

This chapter is organized as follows. Section 5.2 presents the implementation of the algorithm. After this section, graphical examples of the complete algorithm are included for a better understanding of the algorithm in section 5.3 where several examples are presented. Finally, section 5.4 presents the results of the experimentation.

5.2. Algorithm implementation

Once the general idea of the algorithm is presented, the details of the algorithm are presented through a pseudo code (Code 5.1) and some additional explanations.

Before each movement, including the first one, if there is a straight trajectory from the current node to the target node (see navbresenham procedure in subsection 4.2.2), the algorithm is finished. Regarding distance calculation, the Euclidean distance is not directly used, but its square. The Euclidean distance from (xi, yi) to (xj, yj) is:

$$D(i,j) = \sqrt{(xi - xj)^2 + (yi - yj)^2}$$

In this work, the square distance, denoted as D^2, is used. In this way, no square root is needed, decreasing computational needs and execution time without affecting the obtained path.

When no straight path is available, the origin is included in the path by calling the insert_node procedure. Then the origin node is considered the current node. After that, a loop is executed while there isn't a straight trajectory from the current node to the target node or the stack isn't empty, which means the algorithm must go on. The following step is, usually, looking for more nodes for the path. However, if there are not any available neighbors for going on it may be because of two reasons. A deadlock has been reached, so the deadlock_exit procedure is called, or a second path is being looked for but cannot be obtained because nodes cannot be repeated in different paths (see later footprint). We know we are in that case when there is one path already calculated and the stack is empty. That is a break condition, finishing the algorithm with the available path or paths.

The main part of the algorithm is choosing the next current point among unvisited neighbors (those not already included in the path). Only the eight immediate neighbors are considered. The unvisited neighbor with the shortest distance to the target is always chosen. If one node has two neighbors with the same minimum distance to the target node, then the path is divided into two branches from that node. One path is continued and the other will be calculated later, adding the other node to the stack (push).

After each new node is added to a path the algorithm checks if there is a straight line to the target from the new node. In that case, the algorithm would be finished for the current path, and would go on with the stack if not empty. After calculating each path, the footprint from every node will be included. It means that in the next path, previously selected nodes (footprint) won't be selected again. There is an exception for the footprint: the surrounding nodes of the target node clear their footprints. Therefore, it is possible to reach the target node even if a previous path had used that final step, but the rest of multiple paths will not share any other nodes except the ones adjacent to the target.

Once the path (or paths) has been obtained, it is optimized. Most of the nodes of each path are just one step from each other, as neighbor cells are used during the algorithm. But possibly some not neighbor nodes can be connected directly. This is done in the optimizating_paths procedure which is identical to the optimization procedure in subsection 4.2.7. The sequence steps of NafisNav algorithm are shown in fig 5.1.

As seen, four procedures are used during the main algorithm: deadlock_exit, navbresenham, optimizing_path and insert_node. The navbresenham and optimizing_path procedures are identical to the navbresenham and optimization procedures in the previous chapter. So, the rest of this section is outlined as follows. Subsection 5.2.1 gives a description of deadlock_exit components and its operation in further detail. The insert_node procedure is described in subsection 5.2.2.

Code 5.1. NafisNav algorithm pseudo code

```
Input: the origin node (ox,oy), the target node (tx,ty) and the map
read the origin node (ox,oy), the target node (tx,ty) and the map
if navbresenham(ox, oy, tx, ty) = clear then
    $d^2 \leftarrow (tx-ox)^2+(ty-oy)^2$ , path ← (ox,oy), (tx,ty) , length ← $d^2$
else
   1) call procedure insert_node(ox,oy)
   2) (cx,cy) ← (ox,oy)
   3) while navbresenham(cx, cy, tx, ty) ≠ clear or stack isn't empty do
      a) if there aren't any neighbors for (cx,cy) then
               if there is at least one least one path and stack isn't empty then
                    break
               call procedure deadlock_exit(map, cx, cy)
      b) min ← infinity
      c) for all neighbors (nx,ny) ϵ (cx,cy) do
           i.   $d^2 \leftarrow (tx-nx)^2+(ty-ny)^2$ , length ← $d^2$
           ii.  if $d^2$ < min then
                  min ← $d^2$ , (mx,my) ← (nx,ny)
           iii. if $d^2$ = min then
                  push (nx,ny)
           iv.  call procedure insert_node(mx,my)
      d) (cx,cy) ← (mx,my)
      e) if navbresenham(cx, cy, tx, ty) = clear then
           i.   if stack is empty then
                    call procedure insert_node(tx,ty)
           ii.  else
                    call procedure insert_node(tx,ty)
                    pop(nx,ny) , (cx,cy) ← (nx,ny)
                    repeat the before path until before of this node(nx,ny)
           iii. clean footprint the target node and its neighbors
   4) optimized_paths ← call procedure optimizing_path(all of paths)
   5) find shortest path from optimized_paths
output: shortest path from the origin node to the target node
```

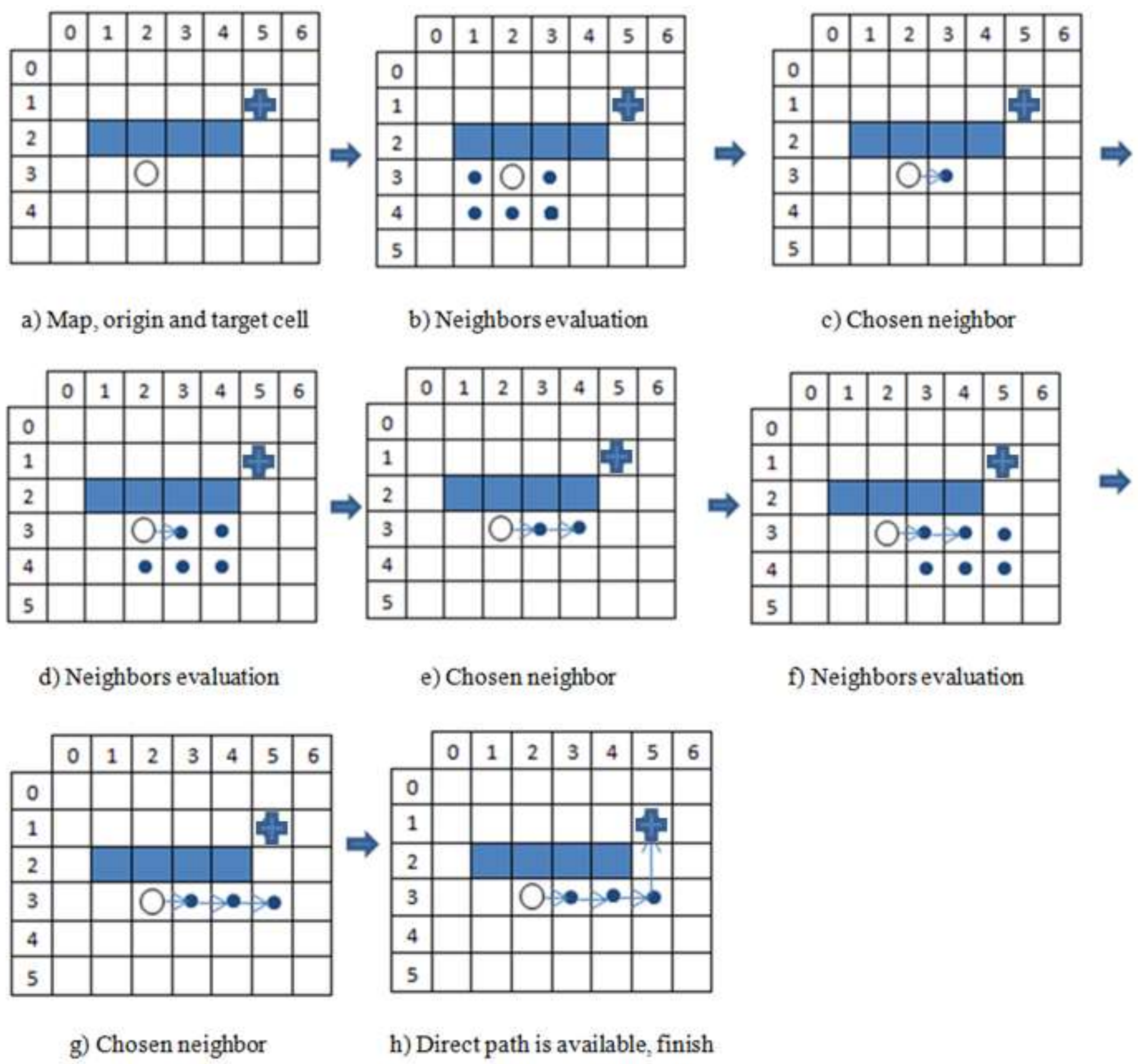


Fig. 5.1. The sequence steps of NafisNav algorithm. The origin point is marked with a circle and the target node with a plus. The neighbors are marked with point.

5.2.1. The deadlock_exit procedure

When there is no available neighbor cell that has not already been used in the path, it means that a deadlock has been reached. At that time, this procedure starts. The objective of this procedure is to exit from the deadlock. The input of this procedure is the given map and the cell that deadlock occurs in and the output of this procedure is the exiting cell of deadlock.

It is a backtracking technique used for exiting from deadlocks. This procedure goes back in the sequence of nodes until it finds one that has unvisited neighbors. That node is called "the promising node". This procedure does two operations: exiting from the deadlock by recognizing the promising node and inserting the new path from the start node to this node. This is illustrated in fig. 5.2, where an example of a deadlock and a promising node are shown. The backtracking is also shown in fig. 5.3. The pseudo code of this procedure is shown in Code 5.2.

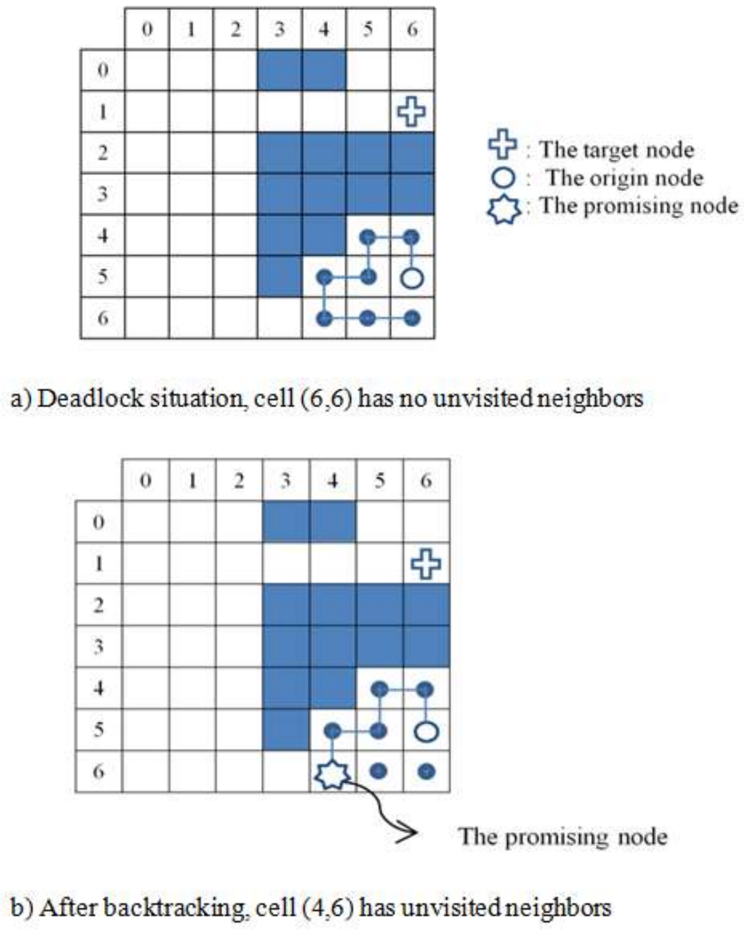

0 1 2 3 4 5 6
0 1 2 3 4 5 6
: The target node
: The origin node
: The promising node
a) Deadlock situation, cell (6,6) has no unvisited neighbors
The promising node
b) After backtracking, cell (4,6) has unvisited neighbors

Fig. 5.2. Example of the deadlock_exit procedure

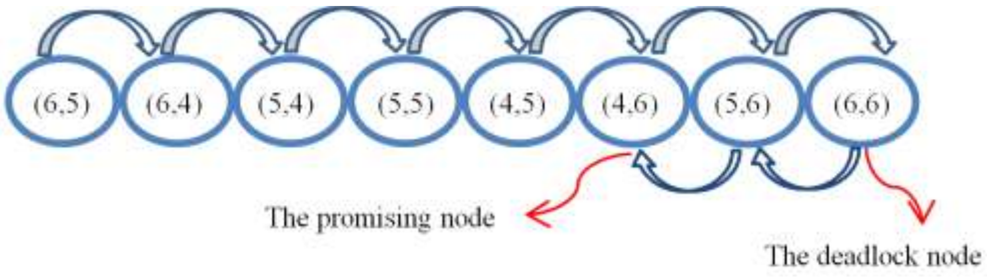


Fig. 5.3. The path diagram from the origin node (6,5) to the deadlock node (6,6) and backtracking to the promising node (4,6)

Code 5.2. The deadlock_exit procedure pseudo code

```
Input: map and the cell (cx,cy) that deadlock occurs in it
for i=node _number-1 to 0 step -1 do
    if  i isn't occupied with obstacle and  isn't filled with footprint  then
        if  there is possibility movement to one side then
            for j=0 to i do
                call procedure insert_node (xj,yj)  // the nodes of previous path
            for all neighbors (nx,ny) ϵ i do
                $d^2 \leftarrow (tx-nx)^2+(ty-ny)^2$ ,  length $\leftarrow d^2$
                if $d^2 <$ min then
                    min$\leftarrow d^2$ , (mx,my)$\leftarrow$(nx,ny) , (cx,cy) $\leftarrow$ (mx,my)
                if $d^2$ =min  then
                        push(nx,ny)
                call procedure insert_node(mx,my)
        else
            continue
        break
Output: the cell (cx, cy) for exiting of deadlock
```

5.2.2. The insert_node procedure

In this procedure, a node (xi, yi) is added to the path. If the node to be included is the target point, then it means a new path has been finished, as shown in code 5.3.

Code 5.3. The insert_node pseudo code

```
input: (xi,yi)
add (xi,yi) to the path
if (xi=xt and yi=yt) then        // (xt,yt) is the target cell
        create a new path
```

5.3. Graphical examples

This section presents some graphical examples of the NafisNav algorithm working. The objective is to help understanding the details of this algorithm.

5.3.1. E.1: Differences in path length depending on relative position

This is a double example with the same obstacle, showing that sometimes longer paths than optimal ones are proposed by the algorithm. The initial path according by this algorithm is shown in fig. 5.4.a). After each movement, it will remain footprint from every node. It means that the relevant cell is not selected in the next path. Then, the optimization procedure is applied on path and the shortest path is determined. It is shown in fig. 5.4.b).

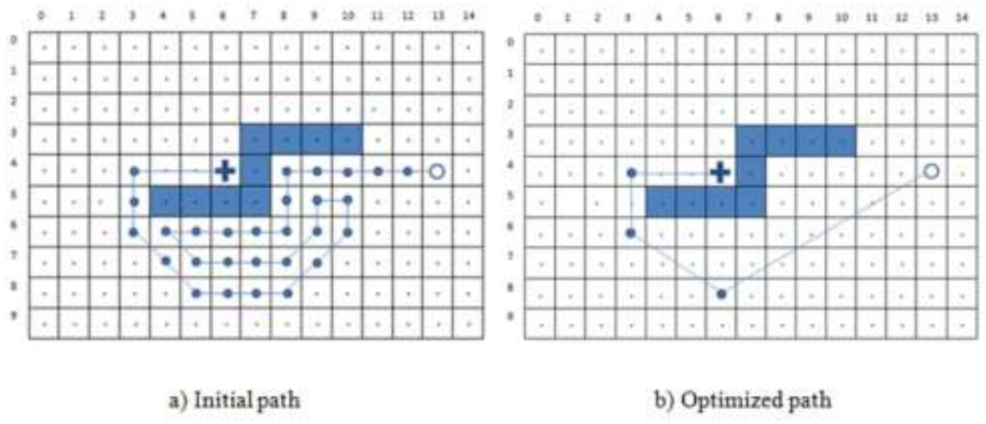


Fig. 5.4. The initial and optimized paths from (13,4) to (6,4) in map-05

There is other example swapping the origin cell and the target cell as shown in fig. 5.5. The initial path is displayed in fig. 5.5.a) and the optimized path is distinguished in fig. 5.5.b).

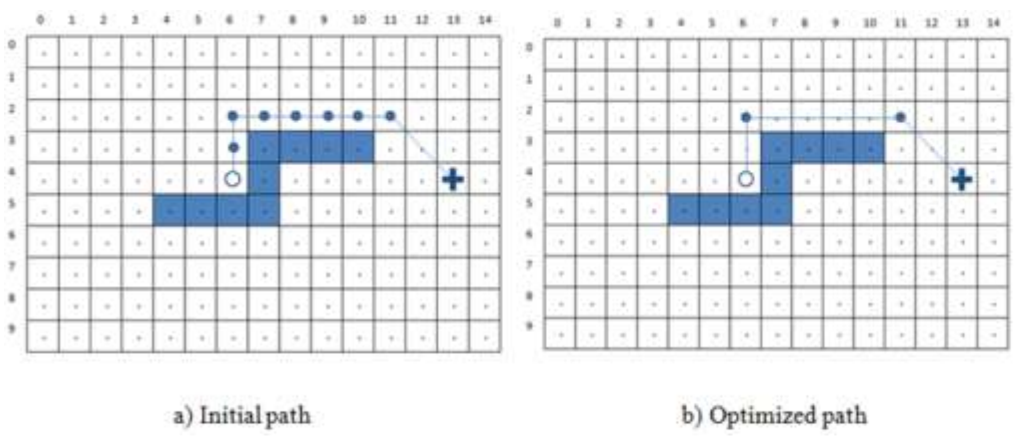


Fig. 5.5. The initial and optimized path from (6,4) to (13,4) in map-05

As it can be seen, NafisNav algorithm does not always get the shortest path from target to origin. The reason is that it only explores one possible path in most cases, therefore not detecting other possible shorter paths. However, exploring less parts of the map has an obvious advantage in terms of memory consumption.

5.3.2. E.2: Two neighbors with the same potential

In this example, the origin cell has two neighbors with the same distance towards the target cell, therefore one neighbor cell is pushed in the stack and the path is continued from other side as shown in fig. 5.6.a). When the first path reaches to the target cell, another neighbor is poped from the stack and the second path is started as shown in fig. 5.6.b). At the end, the optimized paths are shown in fig. 5.6.c).

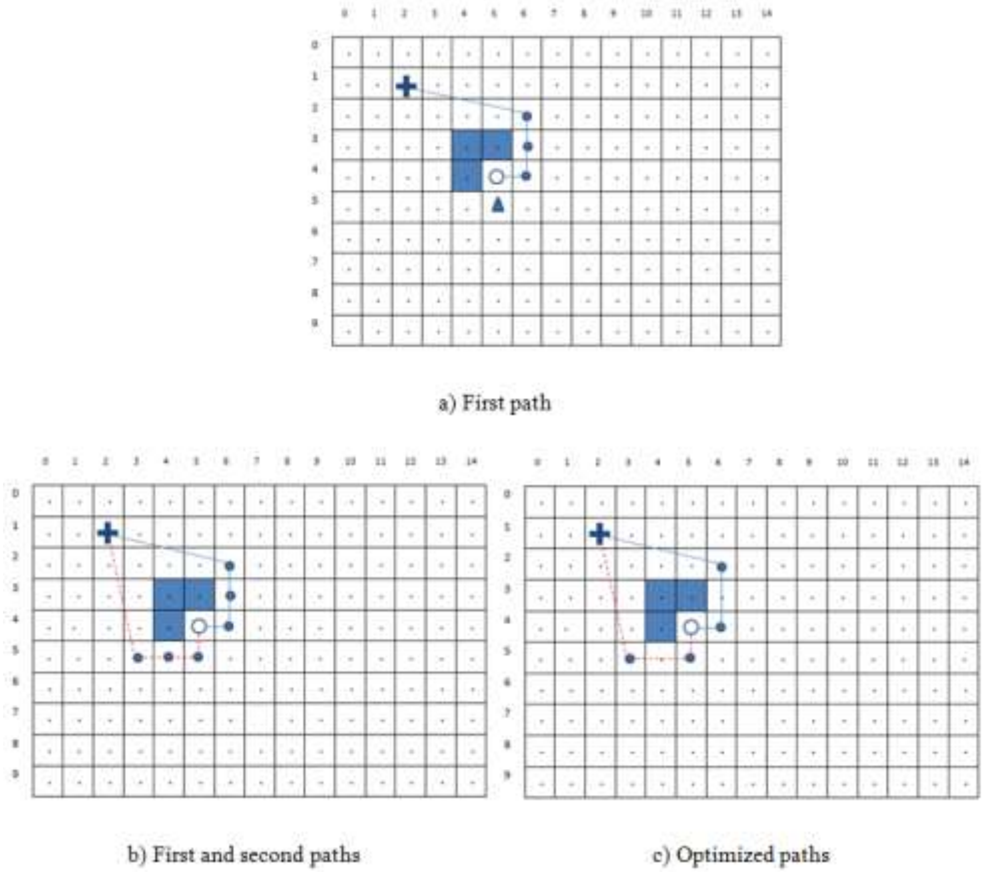


Fig. 5.6. The initial and optimized paths from (5,4) to (2,1). The pushed cell is marked with a triangle.

5.3.3. E.3: Example of deadlock solution

Let’s consider the map in fig. 5.7. It has one obstacle with a potential deadlock. This example shows the basic working of the deadlock_exit procedure, but also how deadlocks are avoided and when additional paths are calculated.

The origin node is (7,9) and the target node is (7,0) as shown in fig. 5.7. First the straight path between the origin node and the target node is checked. As observed in fig. 5.7.a) there is no straight path from the origin to the target node. So the origin node is added to the path. After that the origin node is replaced with the current node. In here, there are five neighbors for the origin node as shown fig. 5.7.a). The node among neighbors with the minimum distance to the target node is selected. In this example, node (7,8) is selected and it is added to the path. Then it is replaced to the current node as shown in fig. 5.7.b). The path is continued, but the deadlock is reached as shown in fig. 5.7.c).

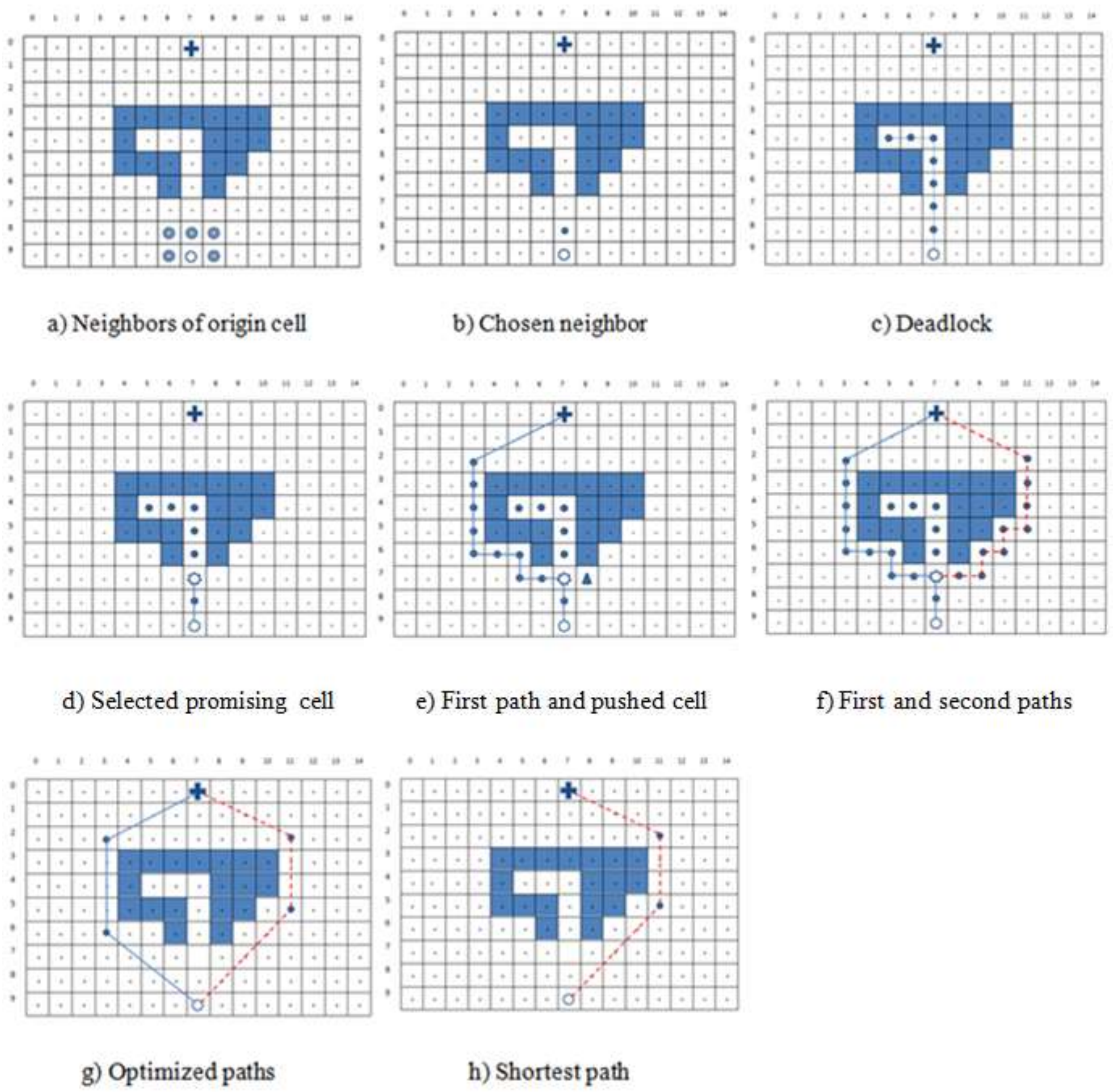

a) Neighbors of origin cell
b) Chosen neighbor
c) Deadlock
d) Selected promising cell
e) First path and pushed cell
f) First and second paths
g) Optimized paths
h) Shortest path

Fig. 5.7. The initial and optimized paths from (7,9) to (7,0). The pushed cell and the promising cell are marked with a triangle and a star respectively.

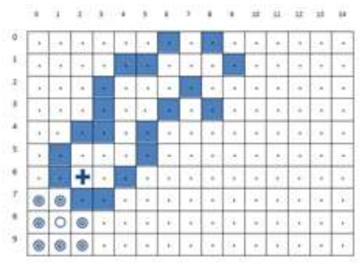
In this case, the current node is moved back until a node that has at least one possibility for motion is reached. In this example, it is node (7,7). This movement towards back is shown in fig. 5.7.d). This approach is called backtracking technique and (7,7) is called the promising node as mentioned earlier. This promising node is marked with a star in fig. 5.7.d). In this node, two nodes ((8,7) and (6,7)) with the equal minimum distances to the target node are found. One of them is added to the path and the path is continued. The other node is pushed to the stack and is marked with a triangle as shown in fig. 5.7.e). After the first path is finished, the stack is checked and, as it is not empty, therefore the upper node of the stack (8,7) is pop. Next the previous path is repeated to before this node and the path is continued as shown in fig. 5.7.f). At the end, both paths are optimized as shown in fig. 5.7.g) and the shortest path is selected as shown in fig. 5.7.h)

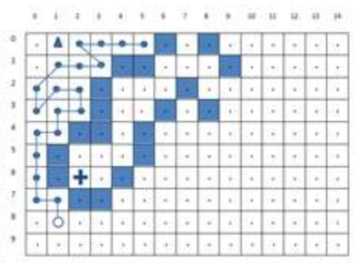
5.3.4. E.4: Multiple deadlocks

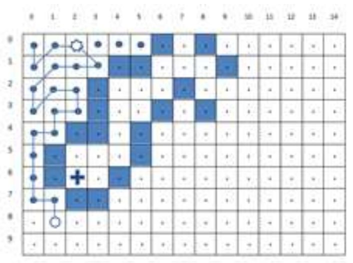
This is an example of when multiple deadlocks are reached during the algorithm execution. The original output of the algorithm is a really long path, but it is a good example of how the optimization leads to a very acceptable final solution.

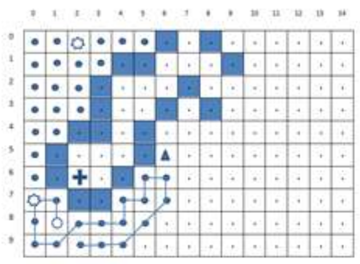
In this example, the origin node is (1,8) and the target node is (2,6) as shown in fig. 5.8. Regarding to the main presented algorithm in code 5.1, first the straight path from the origin node to the target node is checked. In here, there isn't a straight trajectory from the origin node to the target node. Therefore, the origin node is added to the path and then the distance for each neighbor to the target node is determined and chosen the neighbor with the minimum distance. In this example, there are seven neighbors as shown in fig. 5.8.a) and (1,7) is chosen from them and it is added to the path. These operations are continued until it reaches to a deadlock as shown in fig. 5.8.b). Furthermore there are two neighbors with minimum distance to the target node in (2,0), they are (1,0) and (3,0). So the path is continued from one of them and other node (1,0) pushes in the stack. When the deadlock occurs, in this case, the path is come back to rear that it is reached to the promising node as shown in fig. 5.8.c). In here, that is (2,0) and the path is continued from this node. But the new path leads to the deadlock in (0, 0), too. So it is used again the backtracking technique and it is distinguished the second promising node (0,7) as shown in fig. 5.8.d). This new path is continued and again a deadlock is reached in (2,9). Note for (6,6) there are two nodes with same minimum distance to the target. So one of them (6,5) pushes in the stack. The next promising node is (4,9) and the new path is shown in fig. 5.8.e).

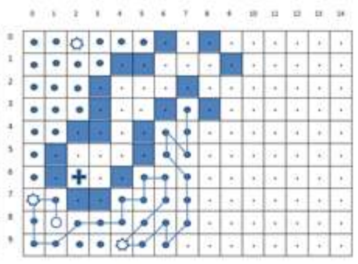
This path reaches to other deadlock and a fourth promising node is (7,4) as shown in fig. 5.8.f) and the path is continued from this node. While the path is finished, as the stack isn't empty, the upper node (6,5) of the stack is pop, but it was previously visited. Therefore the next node (1,0) is pop, but also it was previously visited. Now, the stack is empty and there is a path, therefore the path is optimized as shown in fig. 5.8.g) and the algorithm is finished.

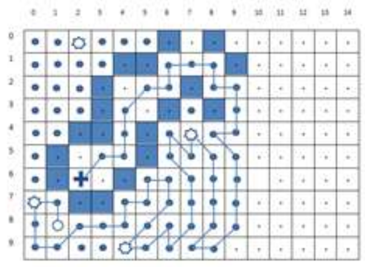

a) Neighbors of origin node


b) Deadlock


c) Promising node and new deadlock


d) Promising node and new deadlock


e) Promising node and new deadlock


f) Promising node and new path

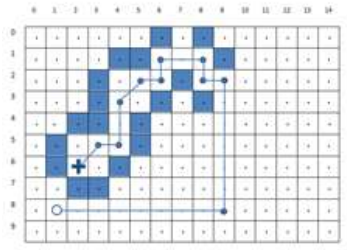

g) Optimized path

Fig. 5.8. The sequence steps of NafisNav algorithm from the origin node (1,8) to the target node (2,6).

5.3.5. E.5: Multiple visited pushed nodes

This example shows that any new path is not created if the pushed nodes have been visited previously.

The origin node is (6,4) and the target node is (5,2) as shown in fig. 5.9.a). In the first path, two nodes are pushed in the stack, they are (4,5) and (7,5) respectively. First, when the path gets to (5,5), there are two options with the same minimum distance to the target node. The path is continued from (6,5), and (4,5) is pushed in the stack. Second when the path reaches the node (7,4), there are two nodes with the minimum distance to the target node. In this moment, also the path is continued from (8,4) and (7,5) is pushed in the stack as shown in fig. 5.21.

In this way, while the path gets to the target node, then the node (7,5) from the top of the stack is pop, but this node (7,5) was previously visited. Therefore the next node of stack is pop, but also this node (4,5) was previously visited. Because there is at least a path and the stack is empty, therefore the path is optimized as shown in fig. 5.9.b) and the algorithm is finished with a single path.

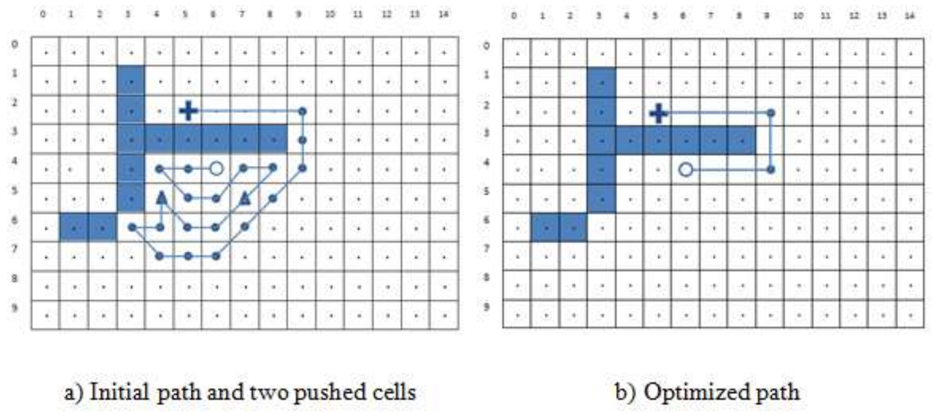


Fig. 5.9. The initial and optimized paths from (6,4) to (5,2)

5.3.6. E.6: Partial map exploration

This is an example of the algorithm working in a map with a high number of obstacles, and how only a small portion of the map is explored, contrary to what other algorithms would do, therefore decreasing memory consumption.

In this example, the first path reaches a deadlock as shown fig. 5.10.a). Therefore deadlock_exit is called. This procedure uses the backtracking technique for exiting the deadlock. After calling deadlock_exit procedure, the current node is (12,9) and the path is continued as shown in fig. 5.10.b) and finally the path is optimized as shown in fig. 5.10.c).

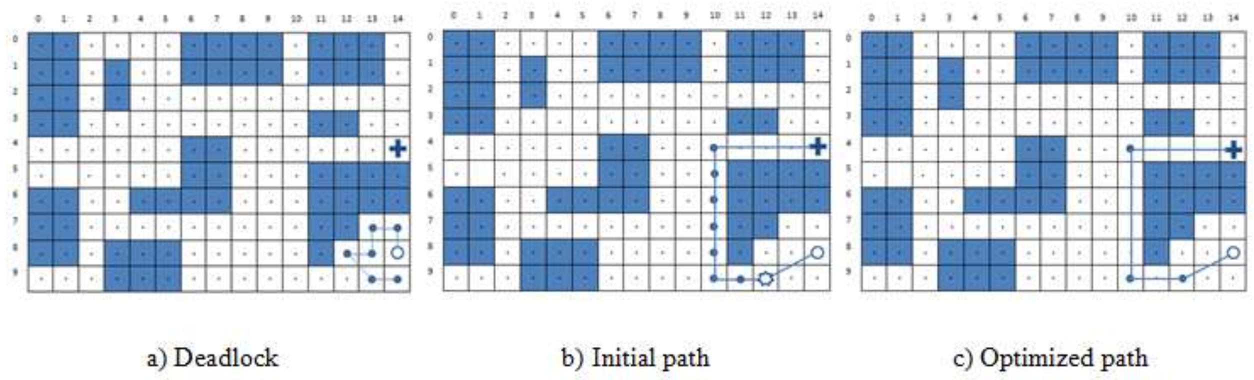


Fig. 5.10. The initial and optimized paths from (6, 4) to (5, 2)

5.4. Experimental results

The NafisNav algorithm is based on a depth-first algorithm which avoids visited nodes and includes a backtrack procedure. This section will focus on comparing the proposed algorithm with the A* and Dijkstra algorithms. Two aspects are compared: 1) final path length, 2) memory requirements.

The three algorithms have been implemented using ANSI-C language. The final code has been compiled using GCC without any optimizing flag.

The test suite is the same one used in the previous chapter (see subsection 4.3.1) which includes two test sets: 1) the small maps set and 2) the big maps set.

Each algorithm has been implemented in a full program that receives the map as a plain text file, the origin and target positions. The result is the sequence of cells for the final path or a message showing that there is no possible solution.

5.4.1. Comparison and discussion on the path length of NafisNav versus A* and Dijkstra algorithms

The first comparison focuses on comparing the path length provided by the three algorithms. The three algorithms have been executed for every possible path in each map, registering the final path length. The optimization procedure has been applied to the solutions provided by the Dijkstra and A* implementations, and also NafisNav. It must be noted that although the initial solutions provided by Dijkstra and A* have the same length before optimization, the paths can be different. Hence, the final path lengths of these two algorithms may differ in some cases after the optimization (see subsection 4.4.1). To ensure an equitable comparison, the sequence of nodes provided by Dijkstra and A* is optimized using the “optimizing_path procedure” presented before. The optimizing_path procedure makes some of the results obtained by Dijkstra and A* different.

5.4.1.1. Comparison and discussion in the small maps set

The average path lengths for the small maps set is presented in table 5.1.

A deep study of the results of the three algorithms leads to 13 different occurrences, depending on the comparison of the provided path length of each algorithm. These have been used for presenting the results. For every case the mean percentage of the number of paths is presented for the small maps set as shown in table 5.2.

Algorithm	Average path lengths
A*	7.89
Dijkstra	7.90
NafisNav	8.35

Table 5.1. Average path lengths for the small maps set

The first conclusion of the presented results shows that the average path length of NafisNav algorithm is (8.35) which is a small difference compared to the average path length of A* (7.89) and Dijkstra (7.90) (see table 5.1).

Also, table 5.2 shows the penalization for each algorithm in the small-map. For each of the 13 different possible cases, the path increase penalization is also presented. This penalization is obtained for each case using the following formulas:

$$\text{Penalization of NafisNav: } \left(\left(\sum_{i=0}^{n} \frac{(PL_{NafisNav})_i - MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i}\right)/n\right) \cdot 100$$

$$\text{Penalization of A* : } \left(\left(\sum_{i=0}^{n} \frac{(PL_{A*})_i - MIN(PL_{NafisNav},, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i}\right)/n\right) \cdot 100$$

$$\text{Penalization of Dijkstra:} \left(\left(\sum_{i=0}^{n} \frac{(PL_{Dijkstra})_i - MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i}\right)/n\right) \cdot 100$$

Case	Result	Average of percentages	Penalizations		
			NafisNav	A*	Dijkstra
1	$PL_{NafisNav}=PL_{A*}=PL_{Dijkstra}$	80.591	0.00%	0.00%	0.00%
2	$PL_{NafisNav}=PL_{A*}<PL_{Dijkstra}$	5.329	0.00%	0.00%	7.67%
3	$PL_{NafisNav}=PL_{Dijkstra}<PL_{A*}$	0.771	0.00%	2.13%	0.00%
4	$PL_{NafisNav}<PL_{A*}=PL_{Dijkstra}$	0.295	0.00%	1.31%	1.26%
5	$PL_{Dijkstra}<PL_{NafisNav}<PL_{A*}$	0.605	1.09%	2.39%	0.00%
6	$PL_{NafisNav}<PL_{A*}<PL_{Dijkstra}$	0.371	0.00%	0.40%	1.17%
7	$PL_{NafisNav}<PL_{Dijkstra}<PL_{A*}$	0.166	0.00%	1.99%	0.62%
8	$PL_{NafisNav}=PL_{Dijkstra}>PL_{A*}$	0.515	1.51%	0.00%	1.51%
9	$PL_{Dijkstra}<PL_{NafisNav}=PL_{A*}$	0.533	2.86%	2.86%	0.00%
10	$PL_{A*}<PL_{NafisNav}<PL_{Dijkstra}$	0.194	0.56%	0.00%	1.23%
11	$PL_{NafisNav}>PL_{A*}=PL_{Dijkstra}$	9.117	20.65%	0.00%	0.00%
12	$PL_{NafisNav}>PL_{A*} > PL_{Dijkstra}$	0.540	7.51%	1.28%	0.00%
13	$PL_{NafisNav}>PL_{Dijkstra} > PL_{A*}$	0.974	18.84%	0.00%	1.33%

Table 5.2. Mean percentage for each case obtained and path penalization for each algorithm in the small maps set

The second conclusion is that the NafisNav, A* and Dijkstra algorithms obtain the same path length in most cases (80.591%, case 1 in table 5.2). A* has the better results, because the most common cases in which NafisNav is worse than A* (case 11, see table 5.2) has a penalization of 20.65%, while the most common opposite case (case 4) has a penalization of 1.31%.

As expected, the depth-first approach of the NafisNav algorithm provides, for some cases, longer paths than Dijkstra and A*. This is meaningful observing cases 11, 12 and 13. The normalized penalization in these three cases is 19.82 % for small maps.

Table 5.3 merges the results comparing Dijkstra versus NafisNav and Table 5.4 makes the same comparison between A* and NafisNav in the small maps set.

State	cases	Sum of average percentages
$PL_{NafisNav}=PL_{Dijkstra}$	1, 3, 8	81.877
$PL_{NafisNav}<PL_{Dijkstra}$	2, 4, 6, 7, 10	6.355
$PL_{Dijkstra}<PL_{NafisNav}$	5, 9, 11, 13, 12	11.768

Table 5.3. Comparison path length of NafisNav and Dijkstra algorithms in the small maps set

State	cases	Sum of average percentages
$PL_{NafisNav}=PL_{A*}$	1, 2, 9	86.453
$PL_{NafisNav}<PL_{A*}$	3, 4, 5, 6, 7	2.207
$PL_{A*}<PL_{NafisNav}$	8, 10, 11, 12, 13	11.34

Table 5.4. Comparison path length of NafisNav and A* algorithms in the small maps set

Table 5.3 shows that the path length of the NafisNav and Dijkstra algorithms is identical in 81.877% and table 5.4 shows that the path length of the NafisNav and A* algorithms is equal in 86.453%.

The main conclusion is that all algorithms are feasible solutions according to path length and the differences are relatively small, although NafisNav can present longer paths in some cases. However, more than 80 % of the paths in the three algorithms are identical (case 1 in table 5.2). Regarding only path length, NafisNav should not be chosen. Anyway, the decision of which algorithm should be used in a low cost robot should be taken considering also other factors, like memory usage.

5.4.1.2. Comparison and discussion in the big maps set

The average path lengths for the big map set is presented in table 5.5. Table 5.6 includes for each of these 13 combinations, the percentage of paths and the definition of the case considering the results of the algorithms and also the average penalization for each case.

Algorithm	Average path lengths
A*	55.78
Dijkstra	55.58
NafisNav	58.10

Table 5.5. Average path lengths for the big maps set

As seen in table 5.5, the average path length of NafisNav (58.10) is close to the A* (55.78) and Dijkstra algorithm (55.58) and it has a low difference compared to Dijkstra and A* algorithms.

Case	Result	Average of percentages	Penalizations		
			NafisNav	A*	Dijkstra
1	$PL_{NafisNav}=PL_{A*}=PL_{Dijkstra}$	51.37	0.00%	0.00%	0.00%
2	$PL_{NafisNav}=PL_{A*}<PL_{Dijkstra}$	6.19	0.00%	0.00%	0.95%
3	$PL_{NafisNav}=PL_{Dijkstra}<PL_{A*}$	2.56	0.00%	1.14%	0.00%
4	$PL_{NafisNav}<PL_{A*}=PL_{Dijkstra}$	0.17	0.00%	0.59%	0.59%
5	$PL_{Dijkstra}<PL_{NafisNav}<PL_{A*}$	5.16	0.84%	1.71%	0.00%
6	$PL_{NafisNav}<PL_{A*}<PL_{Dijkstra}$	0.85	0.00%	0.37%	0.94%
7	$PL_{NafisNav}<PL_{Dijkstra}<PL_{A*}$	2.26	0.00%	1.76%	0.48%
8	$PL_{NafisNav}=PL_{Dijkstra}>PL_{A*}$	0.63	0.47%	0.00%	0.47%
9	$PL_{Dijkstra}<PL_{NafisNav}=PL_{A*}$	2.81	2.81%	2.81%	0.00%
10	$PL_{A*}<PL_{NafisNav}<PL_{Dijkstra}$	1.16	0.30%	0.00%	0.78%
11	$PL_{NafisNav}>PL_{A*}=PL_{Dijkstra}$	8.67	11.29%	0.00%	0.00%
12	$PL_{NafisNav}>PL_{A*} > PL_{Dijkstra}$	9.81	6.03%	1.05%	0.00%
13	$PL_{NafisNav}>PL_{Dijkstra} > PL_{A*}$	8.36	12.65%	0.00%	0.72%

Table 5.6. Mean percentage for each case obtained and path penalization for each algorithm in the big maps set

Table 5.6 shows that more than 70% paths are identical or shorter (cases 1 to 10) than Dijkstra or A* algorithms. A* has the best results, because the most common case in which NafisNav is worse than A* (case 13) has a penalization of 12.65%, while the most common opposite case (case 7) has a penalization of 1.76%. Also, the normalized penalization in case 11, 12 and 13 is 9.79 % for the big maps set.

Table 5.7 merges the results comparing Dijkstra versus NafisNav and Table 5.8 makes the same comparison between A* and NafisNav in the big maps set.

State	cases	Sum of average percentages
$PL_{NafisNav}=PL_{Dijkstra}$	1, 3, 8	54.56
$PL_{NafisNav}<PL_{Dijkstra}$	2, 4, 6, 7, 10	10.63
$PL_{Dijkstra}<PL_{NafisNav}$	5, 9, 11, 13, 12	34.81

Table 5.7. Comparison path length of NafisNav and Dijkstra algorithms in the big maps set

State	cases	Sum of average percentages
$PL_{NafisNav}=PL_{A*}$	1, 2, 9	60.37
$PL_{NafisNav}<PL_{A*}$	3, 4, 5, 6, 7	11
$PL_{A*}<PL_{NafisNav}$	8, 10, 11, 12, 13	28.63

Table 5.8. Comparison path length of NafisNav and A* algorithms in the big maps set

5.4.2. Comparison and discussion on the dynamic memory of NafisNav versus A* and Dijkstra algorithms

The second comparison focuses on comparing the memory that each program requires for solving the problem. All three algorithms have been implemented using dynamic memory for storing the map, generating the graph (when required) and generating the solution. The measurement of the allocated memory is done using the Valgrind application. The results are processed with the Massif-msprint tool.

5.4.2.1. Comparison and discussion in the small maps set

Similar to the previous experiment, we have measured the memory for every possible path in each map of the set. Fig. 5.11 shows the maximum memory in each map for the small-map set.

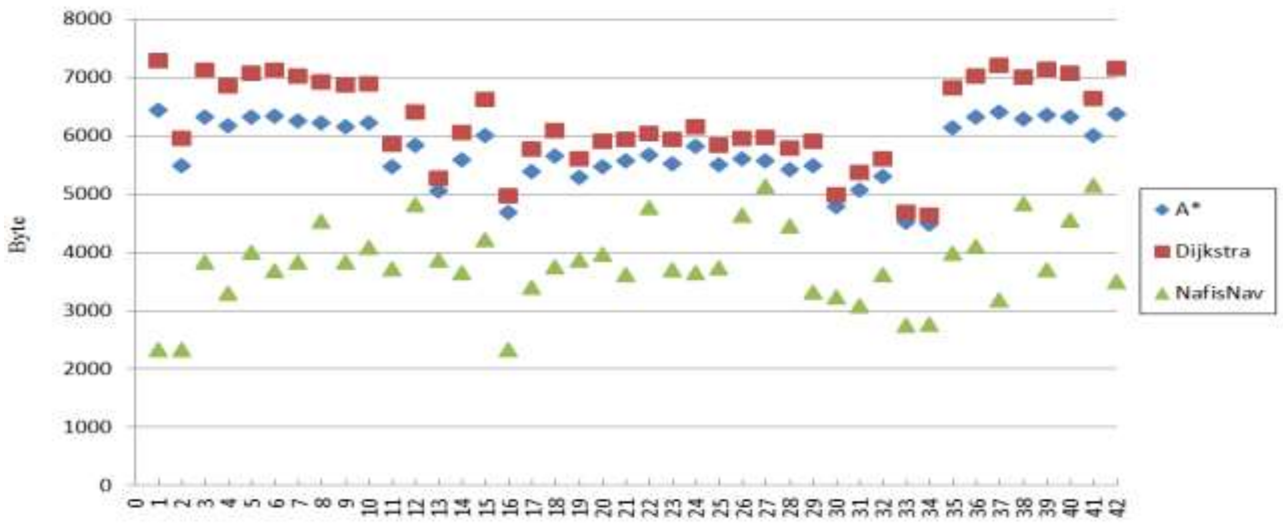


Fig. 5.11. Comparison of the maximum memory requirement for each small map

Fig. 5.11 shows that dynamic memory of the Dijkstra and A* algorithms is maximum in maps which have few obstacles and many free cells and it is the minimum in maps which have fewer free cells (see all small map in Appendix B). For example, the dynamic memory of the Dijkstra algorithm (4620 Byte) and the A* algorithm (4484 Byte) are the minimum in map-34 and the dynamic memory of the Dijkstra algorithm (7272 Byte) and the A* algorithm (6424 Byte) are the maximum in map-01, map-34 has the least free cells whereas map-01 has the maximum free cells. But about NafisNav algorithm, its dynamic memory is the minimum when the unvisited pushed nodes are minimum along the path and it is the maximum when the unvisited pushed nodes in along the path are maximum. For this reason, the NafisNav has a minimum dynamic memory in map-01 (2322 Byte) and map-02 (2322 Byte), because they are not any obstacle and all paths are straight. The NafisNav algorithm has a maximum dynamic memory in map-41 (5150 Byte). The NafisNav reduces the memory usage significantly compared to Dijkstra and A* algorithms, as shown in fig. 5.11.

The memory peaks of each algorithm are shown in table 5.9 taking into account the maximum in any of the maps of the small maps set.

Type algorithm	Memory peak (Bytes)
A*	6424
Dijkstra	7272
NafisNav	5150

Table 5.9. Comparison memory peaks of three algorithms for all small maps set

Table 5.9 shows that memory peak of NafisNav (5150 Byte) is lower than the memory peaks of Dijkstra (7272 Byte) and A* (6424 Byte) algorithms.

5.4.2.2. Comparison and discussion in the big maps set

Fig. 5.12 shows the maximum memory in each map for the big maps set.

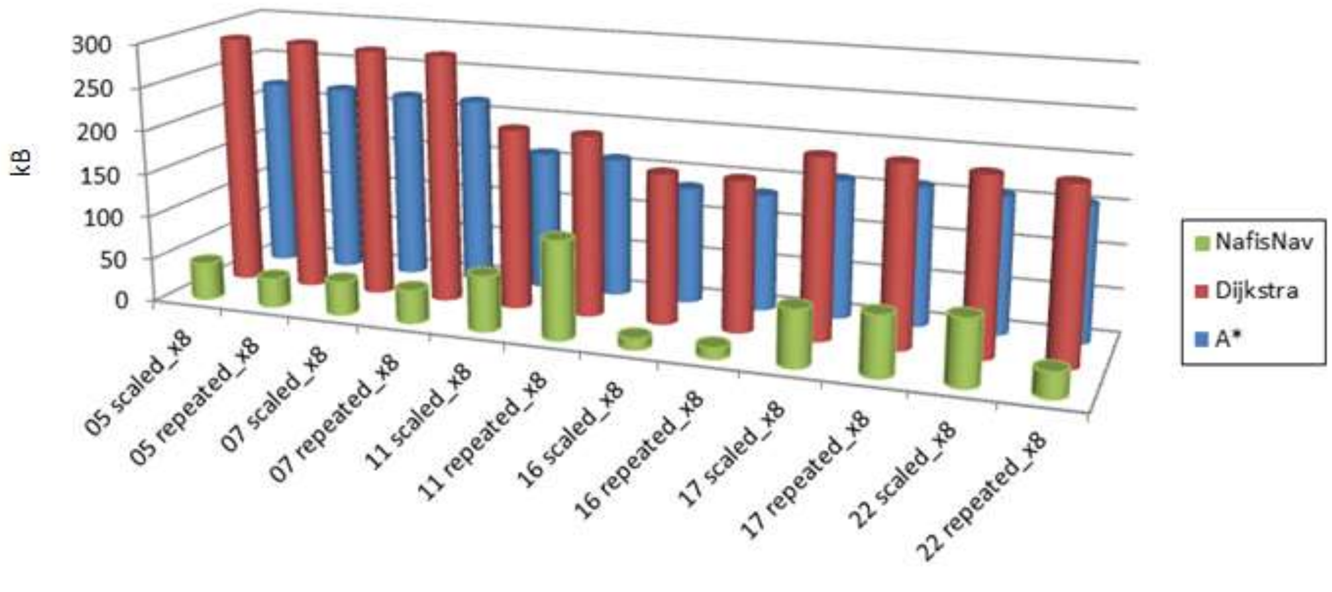


Fig. 5.12. Comparison of the maximum memory requirement for each big map

As seen in fig. 5.12, the Dijkstra has a minimum dynamic memory in map-16 scaled_x8 (171.78 kB) and map-16 repeated_x8 (171.78 kB). Also, the A* has a minimum dynamic memory in map-16 scaled_x8

(134.59 kB) and map-16 repeated_x8 (134.59 kB), because these maps have the least free cells. Dijkstra algorithm has a maximum dynamic memory in map-05 scaled_x8 (289.28 kB). Also, A* algorithm has a maximum dynamic memory in map-05 scaled_x8 (219.25 kB), because it has the minimum number of obstacles. The NafisNav algorithm has a minimum dynamic memory in map-16 repeated_x8 (11.91 kB) and it has a maximum dynamic memory in map-11 repeated_x8 (111.87 kB).

The memory peaks of each algorithm are shown in table 5.10 taking into account the maximum in any of the maps of the big maps set.

Type algorithm	Memory peak(kB)
A*	219.25
Dijkstra	289.28
NafisNav	111.87

Table 5.10. Comparison memory peaks of three algorithms for all big maps set

Table 5.10 shows that memory peak of the NafisNav (111.87 kB) is much lower than the memory peaks of the Dijkstra (289.28 kB) and A* (219.25 kB) algorithms.

The NafisNav algorithm consumes less memory than the Dijkstra and A* algorithms. This is clearer in the big maps set than the small maps set. Furthermore the memory peak of the NafisNav algorithm is much lower than the Dijkstra and A* algorithms. If the algorithms were used in even bigger maps, the difference in memory usage is expected to keep on growing, while the path length penalization would be similar or even smaller. Therefore, the NafisNav algorithm is especially usefull for decreasing memory consumption in bigger maps.

5.4.3. Comparison and discussion of NafisNav and HCTNav algorithms

First comparison focuses on comparing the average path length provided by the four algorithms (HCTNav, NafisNav, Dijkstra and A*). The four algorithms have been executed for every possible path in each map, registering the final path length. The optimization procedure has been applied to the solutions provided by of the four algorithms. The second comparison focuses on comparing the memory that each program requires for solving the problem. All four algorithms have been implemented using dynamic memory for storing the map, generating the graph (when required) and generating the solution. The measurement of the allocated memory is done using the Valgrind application. The results are processed with Massif-msprint tool.

5.4.3.1. Comparison and discussion in the small maps set

The average path lengths for the small maps set are presented in table 5.11.

Algorithm	Average path lengths	Penalization of average path length
HCTNav	7.89	0.00%
NafisNav	8.35	5.83%
Dijkstra	7.90	0.13%
A*	7.89	0.00%

Table 5.11. Average path lengths and penalization of four algorithms for each small map

The main conclusion of the presented results in table 5.11 is that the average path length of NafisNav algorithm is (8.35) is higher than the average path length of HCTNav (7.89) and also Dijkstra (7.90) and A* (7.89). The penalization of average path length for NafisNav is 5.83%.

Also the HCTNav and NafisNav algorithms were compared together in the small maps set as shown in table 5.12. Table 5.12 shows that more than 87% paths are identical (cases 1) in the HCTNav and NafisNav algorithms. NafisNav (case 3) has an average penalization of 18.88% when a worse result is given, while HCTNav (case 2) has an average penalization of 2.18%.

Case	Result	Average of percentages	Penalization	
			NafisNav	HCTNav
1	$PL_{HCTNav}=PL_{NafisNav}$	87.46	0.00%	0.00%
2	$PL_{HCTNav}>PL_{NafisNav}$	0.99	0.00%	2.18%
3	$PL_{HCTNav}<PL_{NafisNav}$	11.55	18.88%	0.00%

Table 5.12. Mean percentage for each case obtained and path penalization for HCTNav and NafisNav algorithms in each small map

Similar to the previous experiment, we have measured the memory for every possible path in each map of the set. Fig. 5.13 shows the maximum memory in each map for the small maps set.

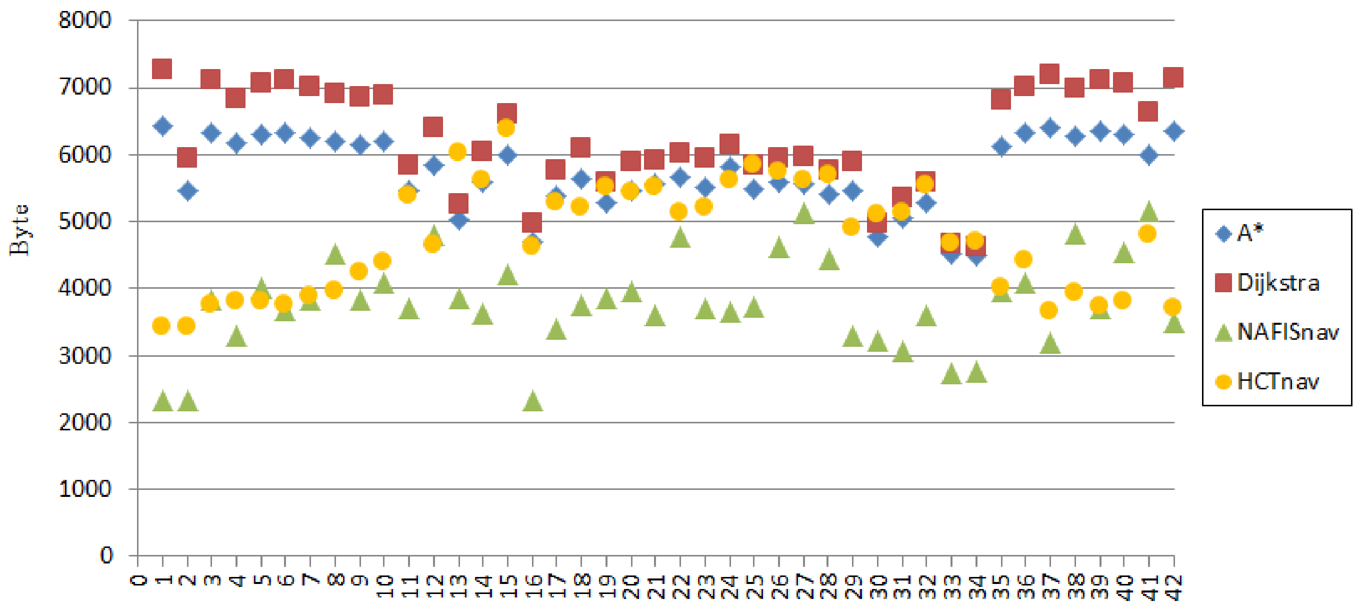


Fig. 5.13. Comparison of the maximum memory requirement for each small map

The dynamic memory of NafisNav is the minimum when the unvisited pushed nodes are minimum along the path and it is the maximum when the unvisited pushed nodes in along the path are maximum. For this reason, the NafisNav has a minimum dynamic memory in map-01 (2322 Byte) and map-02 (2322 Byte), because there are no obstacles and all paths are straight. The NafisNav algorithm has a maximum dynamic memory in map-41 (5150 Byte). But about HCTNav algorithm, its dynamic memory is the minimum in maps which have the least obstacles and it is the maximum in maps which have the most obstacles. For this reason, HCTNav has a minimum dynamic memory in map-01 (3432 Byte) and map-02 (3432 Byte), because they have no obstacles and it has a maximum dynamic memory in map-15 (6372 Byte), because it has the maximum number of obstacles (28) in this map set. The NafisNav reduces the memory usage significantly compared to the HCTNav algorithm as shown in fig. 5.13. Also, fig. 5.13 shows that dynamic memory of the Dijkstra and A* algorithms is maximum in maps which have few obstacles and many free cells and it is the minimum in maps which have fewer free cells (see all small maps in Appendix A). For example, the dynamic memory of the Dijkstra algorithm (4620 Byte) and the A* algorithm (4484 Byte) are the minimum in map-34 and the dynamic memory of the Dijkstra algorithm (7272 Byte) and the A* algorithm (6424 Byte) are the maximum in map-01. Map-34 has the least free cells whereas map-01 has the maximum free cells.

The memory peaks of each algorithm are shown in table 5.13 for all small maps set.

Type algorithm	Memory peak (Bytes)	Penalization of memory peak
HCTNav	6372	-0.81%
NafisNav	5150	-19.83%
Dijkstra	7272	+13.20%
A*	6424	0%

Table 5.13. Comparison memory peaks and penalization of four algorithms for all small maps set

Table 5.13 shows that memory peak of the NafisNav (5150 Byte) is lower than the memory peaks of the HCTNav (6372 Byte), Dijkstra (7272 Byte) and A* (6424 Byte) algorithms. The penalizations are calculated with the respect to the best of Dijkstra and A* (A* in this case). HCTNav obtains very similar results, while NafisNav saves 19.83% of the necessary memory in A*.

5.4.3.2. Comparison and discussion in the big maps set

The average path lengths for the big maps set are presented in table 5.14.

Algorithm	Average path lengths	Penalization of average path length
HCTNav	56.74	2.09%
NafisNav	58.10	4.53%
Dijkstra	55.58	0.00%
A*	55.78	0.36%

Table 5.14. Average path lengths and penalization of four algorithms for the big maps set

As seen in table 5.14, the average path length of NafisNav (58.10) is again higher than HCTNav (56.74) and also Dijkstra (55.58) and A* (55.78) algorithms. Regarding path length, HCTNav algorithm presents somewhat better results, and more importantly, there are no big differencse in some cases. As it was explained before, NafisNav can sometimes present quite longer paths (up to 20% longer in small maps and 10 % in big ones), although most of the time its results in path length are very similar to HCTNav (or Dijkstra and A*). That is why the average path length is not so different. Table 5.14 shows that penalization of average path length in NafisNav is 4.53%, while it is 2.09% in HCTNav.

Also the HCTNav and NafisNav algorithms were compared together in the big maps set as shown in table 5.15. Table 5.15 shows that more than 58% paths are identical (cases 1) in the HCTNav and NafisNav algorithms. NafisNav (case 3) has an average penalization of 8.70% when presenting worse results, while HCTNav (case 2) has an average penalization of 2.84%.

Case	Result	Average of percentages	Penalization	
			NafisNav	HCTNav
1	$PL_{HCTNav}=PL_{NafisNav}$	58.78	0.00%	0.00%
2	$PL_{HCTNav}>PL_{NafisNav}$	11.97	0.00%	2.84%
3	$PL_{HCTNav}<PL_{NafisNav}$	29.25	8.70%	0.00%

Table 5.15. Mean percentage for each case obtained and path penalization for HCTNav and NafisNav algorithms in each big map

Fig. 5.14 shows the maximum memory in each map for the big maps set.

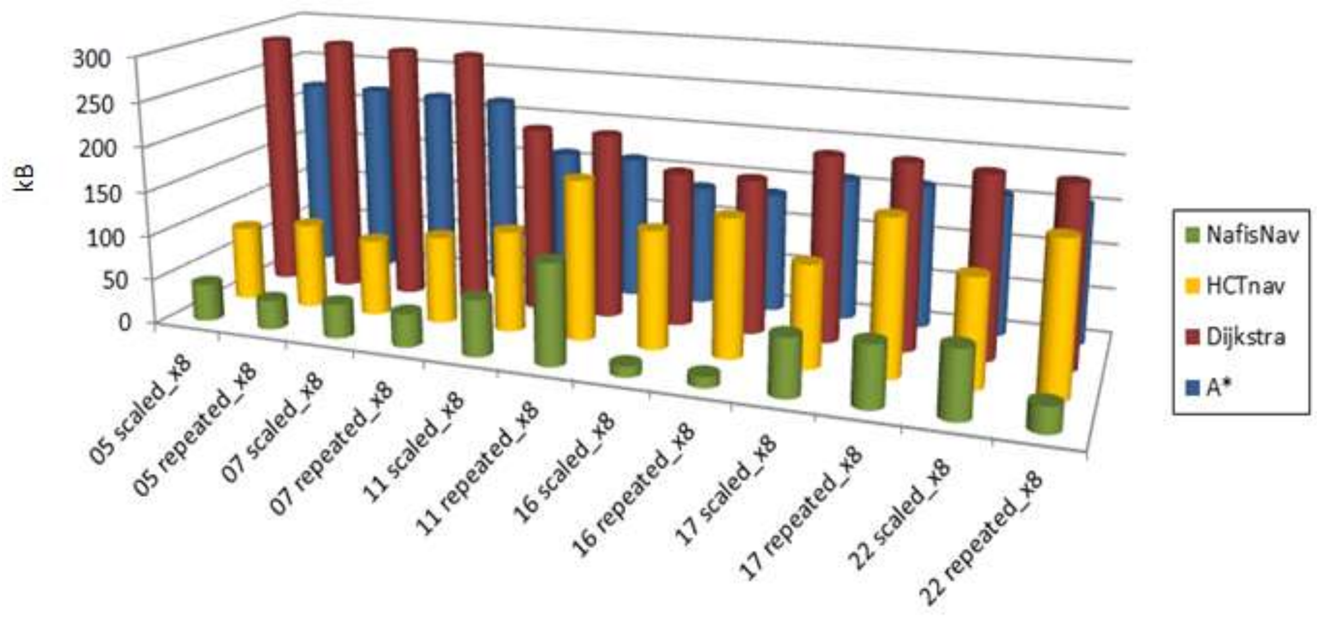


Fig. 5.14. Comparison of the maximum memory requirement for each big map

The NafisNav algorithm has a minimum dynamic memory in map-16 repeated_x8 (11.91 kB) and it has a maximum dynamic memory in map-11 repeated_x8 (111.87 kB). However, the HCTNav algorithm has a minimum dynamic memory in map-05 scaled_x8 (83.9 kB), that has the minimum number of obstacles in this map set and it has a maximum dynamic memory in map-11 repeated_x8 (177.66 kB) that has the maximum number of obstacles in this map set. As seen in fig. 5.14, Dijkstra has a minimum dynamic memory in map-16 scaled_x8 (171.78 kB) and map-16 repeated_x8 (171.78 kB). Also, A* has a minimum dynamic memory in map-16 scaled_x8 (134.59 kB) and map-16 repeated_x8 (134.59 kB), because these maps have the least free cells. Dijkstra algorithm has a maximum dynamic memory in map-05 scaled_x8 (289.28 kB). Also, A* algorithm has a maximum dynamic memory in map-05 scaled_x8 (219.25 kB), because it has the minimum number of obstacles.

The memory peaks of each algorithm are shown in table 5.16 for all big maps set.

Type algorithm	Memory peak (kB)	Penalization of memory peak
HCTNav	177.66	-18.97%
NafisNav	111.87	-48.98%
Dijkstra	289.28	+31.94%
A*	219.25	0%

Table 5.16. Comparison memory peaks and penalization of four algorithms for all big maps set

Table 5.16 shows that memory peak of the NafisNav (111.87 kB) is much lower than the memory peak of the HCTNav (177.66 kB) algorithms and also Dijkstra (289.28 kB) and A* (219.25 kB) algorithms. The penalization is calculated to respect the best of Dijkstra and A*, which is A* in this case. HCTNav saves almost 19% of the needed memory, while NafisNav improves this result up to almost 49%

The NafisNav algorithm consumes the lower memory compared to HCTNav, Dijkstra and A* algorithms. This is clearer in the big maps set than the small maps set. In fact, it is expected that this difference would grow with the size of the maps.

Chapter 6. Conclusions and future work

In this thesis, two new path planning algorithms (HCTNav and NafisNav) are proposed and compared to the state of the art. Both are designed for navigation in grid maps, which are usually indoor environments, running in embedded systems such as low cost mobile robots. Special emphasis has been made in reducing the required hardware resources, which are a bottleneck in embedded systems, especially memory. Other resources, such as CPU time, are not so relevant in mobile robots as the time needed for running the algorithm will be negligible compared to the time needed for the robot to move. However, memory can usually be the main limitation for the maximum size of navigable maps in embedded systems.

In the study of the state of the art, different possibilities of path planning algorithms are presented, being Dijkstra the most known and extended algorithm, and their variants, A* algorithms. They are graph navigation/search algorithms, but can be adapted to grid maps making a conversion of the map to a graph. In this way, both algorithms have been implemented and are used as the main reference for comparison. They have been implemented trying to minimize memory consumption to make a fair comparison with the proposed algorithms, which have been also implemented using the same language, ANSI-C, in order to be run in almost any embedded system.

HCTNav algorithm is based on trying to reach the target from the origin going in straight line. When obstacles are found, two alternative paths are considered surrounding the obstacle in both directions. The obstacle is surrounded until a straight line to the target is found or, if not possible, to a second obstacle. Therefore, up to 2N paths can be calculated, being N the number of obstacles between the origin and the target. Path length results are very similar to the results obtained by Dijkstra and A*. In fact, there are cases in which HCTNav obtains a shorter path. This can be surprising as both algorithms should always find the shortest path. However, the graphs generated from the grid maps only consider connections between neighbor cells in order to minimize memory consumption. A later optimization can directly connect cells that are not neighbors, but not always the shortest possible path is found because of the partial graph initially generated. HCTNav obtains very similar results of path length compared to Dijkstra and A* (2.1% worse), but requires less memory (19%), showing a good compromise between path length and memory consumption.

The other proposed algorithm, NafisNav, is similar to what a blind person would do, using only the information of the immediately surrounding obstacles. The algorithm tries to go in straight line, but if obstacles are found they are surrounded. However, in principle only one path is considered, always choosing the unvisited empty cell neighbor nearest to the target. Only if two neighbors have the same distance to the target a second path is considered. Using only the information of neighbor cells greatly minimizes memory consumption. However, this algorithm can lead to longer paths as it does not explore all possibilities. In most cases, the proposed path has almost the same length than those proposed by the other algorithms, but there are exceptions. Its main advantage is memory consumption, which is 49% lower compared to Dijkstra and A*. The disadvantage is path length, which is 4.5% longer in average, with some cases of paths up to 20% longer.

6.1. Publications derived from this thesis

The results of this thesis have been published in the following papers.

Publications related to HCTNav:

- M. Pala, N. Osati Eraghi, F. López-Colino, A. Sanchez, A. de Castro, J. Garrido, "HCTNav: A Path Planning Algorithm for Low-Cost Autonomous Robot Navigation in Indoor Environments", International Journal of Geo-Information, vol. 2, 729-748, Aug. 2013.
- N. Osati Eraghi, F. López-Colino, A. de Castro, J. Garrido, "Path Length Comparison in Grid Maps of Planning Algorithms: HCTNav, A* and Dijkstra", Proceedings of the IEEE Conference on Design of Circuits and Integrated Circuits (DCIS), Nov. 2014.

Publications related to NafisNav:

- N. Osati Eraghi, F. Lopez-Colino, A. de Castro, J. Garrido, "NafisNav: an Indoor Navigation Algorithm for Embedded Systems and based on Grid Maps", Proceedings of the IEEE International Conference on Industrial Technology (ICIT), Mar. 2015.

6.2. Future work

It would be desirable that the work started in this thesis goes on, inducing improvements in the proposed algorithms or different ones, always with the objective of obtaining less expensive robots or mobile nodes for autonomous navigation. The final objective is to introduce small robots in more applications in order to improve life quality.

Some research lines that are open after this thesis are:

- Results of HCTNav and NafisNav in terms of execution time, apart from path length and memory consumption.
- Applying HCTNav and NafisNav to dynamic environments. Both algorithms have been implemented for static grid maps completely known from the beginning. However, in many applications adapting to changing environments is necessary, such as those in which there are people or other moving beings. The objective would be to adapt these algorithms to environments with moving obstacles.
- Creating other new algorithms which may improve the ones proposed here, HCTNav and NafisNav, in terms of memory consumption and/or path length.
- Implementing the proposed algorithms in real mobile robots prototypes, as new problems would probably arise in real environments.

- Applying these algorithms in mobile nodes of wireless sensor networks, considering new aspects such as coverage and optimization.

Capítulo 6. Conclusiones y trabajo futuro

En esta tesis se proponen y comparan con el estado del arte dos nuevos algoritmos de planificación de trayectorias: HCTNav y NafisNav. Ambos están diseñados para su utilización en navegación de mapas de rejilla, normalmente de espacios interiores, y en sistemas embebidos, tales como robots móviles de bajo coste. Se ha hecho especial énfasis en los recursos necesarios para ejecutar dichos algoritmos, ya que son limitados en sistemas embebidos, especialmente la memoria. Hay que tener en cuenta que otros recursos, como el tiempo de ejecución, no son tan relevantes en robots móviles, ya que el tiempo en ejecutar el algoritmo es en cualquier caso muy pequeño en comparación con el tiempo que necesitará el robot para moverse. Sin embargo, la memoria puede limitar el tamaño máximo de los mapas navegables por el robot, y se convierte en el recurso crítico en sistemas embebidos.

En el estudio del estado del arte se revisan las distintas opciones existentes para planificación de trayectorias, haciendo especial hincapié en el algoritmo más conocido y extendido, Dijkstra, así como sus variantes, tales como A*. Aunque se trata de algoritmos pensados para navegación en grafos, se pueden adaptar a mapas de rejilla convirtiendo éstos en grafos. De esta forma, ambos algoritmos se han implementado para servir como puntos de referencia en esta tesis doctoral. En dichas implementaciones se ha buscado minimizar la memoria necesaria, que será un criterio fundamental de comparación con los algoritmos propuestos, de tal forma que las comparaciones sean lo más justas posibles. Los dos algoritmos propuestos, HCTNav y NafisNav, también se han implementado utilizando siempre el mismo lenguaje, ANSI-C, para poder ejecutarse en cualquier sistema embebido.

El algoritmo HCTNav está basado en intentar ir en línea recta desde el origen al destino y, en caso de encontrarse obstáculos, calcular dos rutas alternativas rodeando el obstáculo encontrado por cada lado. El rodeo continúa hasta que se encuentra una línea recta libre hasta el destino o, en caso contrario, a un segundo obstáculo. Por tanto, se crean hasta 2N rutas distintas, siendo N el número de obstáculos entre origen y destino. Los resultados obtenidos en términos de longitud de ruta son muy similares a los obtenidos por los algoritmos de referencia, Dijkstra y A*. De hecho, en algunas ocasiones pueden ser incluso mejores. Es cierto que dichos algoritmos obtienen siempre la ruta más corta posible, pero en grafos completos. Hay que tener en cuenta que los grafos generados a partir de los mapas de rejilla se han hecho considerando sólo conexiones entre celdas vecinas para minimizar la memoria necesaria. Aunque la optimización posterior aplicada a todos los algoritmos puede conectar celdas a más distancia, no siempre se encuentra el camino más corto con las implementaciones propuestas para Dijkstra y A* por la configuración inicial del grafo. Esto se ha hecho así para minimizar la memoria necesaria por dichos algoritmos, que es el segundo criterio de comparación. Aparte de conseguir unos resultados de longitud de ruta algo peores aunque muy similares a Dijkstra y A* (2,1 % de diferencia), HCTNav mejora los resultados de memoria obtenidos por los algoritmos de referencia (19 % menos memoria), mostrando un buen compromiso entre longitud de la ruta y memoria.

El otro algoritmo propuesto, NafisNav, es similar al algoritmo que utilizaría una persona ciega, utilizando sólo la información de los obstáculos inmediatamente adyacentes a la posición actual. Por tanto, se intenta ir en línea recta entre origen y destino, rodeando los obstáculos encontrados. Sin embargo, no se consideran normalmente dos rutas para rodear un obstáculo, sino que se elige la celda libre más cercana al destino en cada caso. Sólo en caso de igualdad en la distancia de varias celdas vecinas hasta el destino se considera más de una ruta. Utilizar sólo la información de las celdas vecinas minimiza en gran medida la memoria necesaria. Sin embargo, este algoritmo puede conducir en algunos casos a resultados claramente peores en términos de longitud de la ruta propuesta al no explorar todas las posibilidades. Aun así, en la mayoría de ocasiones la ruta propuesta tiene la misma longitud que los demás algoritmos usados en la comparación. La ventaja radica en la memoria necesaria para NafisNav, que es notablemente inferior a la necesaria para Dijkstra y A* (49 % menos), en especial en mapas más grandes, y también inferior a la necesaria por HCTNav. Por tanto, este algoritmo es el mejor en términos de memoria, aunque a costa de una penalización en la longitud de la ruta propuesta que puede llegar al 20 % en algunos casos, y de media 4,5 % más larga.

6.1 Publicaciones derivadas de la tesis

Esta tesis ha dado lugar a las siguientes publicaciones.

Publicaciones sobre el algoritmo HCTNav:

- M. Pala, N. Osati Eraghi, F. López-Colino, A. Sanchez, A. de Castro, J. Garrido, "HCTNav: A Path Planning Algorithm for Low-Cost Autonomous Robot Navigation in Indoor Environments", International Journal of Geo-Information, vol. 2, 729-748, Aug. 2013.
- N. Osati Eraghi, F. López-Colino, A. de Castro, J. Garrido, "Path Length Comparison in Grid Maps of Planning Algorithms: HCTNav, A* and Dijkstra", Proceedings of the IEEE Conference on Design of Circuits and Integrated Circuits (DCIS), Nov. 2014.

Publicaciones sobre el algoritmo NafisNav:

- N. Osati Eraghi, F. Lopez-Colino, A. de Castro, J. Garrido, "NafisNav: an Indoor Navigation Algorithm for Embedded Systems and based on Grid Maps", Proceedings of the IEEE International Conference on Industrial Technology (ICIT), Mar. 2015.

6.2 Trabajo futuro

Sería deseable que el trabajo empezado no concluya aquí, sino que induzca a posibles mejoras en los algoritmos propuestos u otros, siempre con el objetivo de que robots o nodos móviles más baratos se puedan utilizar en navegación independiente, sin necesidad de ordenadores centrales. El objetivo final es extender el uso de pequeños robots en más aplicaciones que puedan mejorar la calidad de vida.

Algunas líneas de investigación que quedan abiertas al concluir esta tesis son:

- Resultados de HCTNav y NafisNav en términos de tiempo de ejecución, además de longitud de ruta y memoria necesaria.

- Aplicación de los algoritmos HCTNav y NafisNav a ambientes dinámicos. Ambos algoritmos se han implementado a partir de mapas de rejilla conocidos y fijos desde el comienzo. Sin embargo, en muchas aplicaciones es necesario adaptarse a entornos que pueden cambiar, como aquellos en los que hay personas u otros entes móviles. Por tanto el objetivo sería adaptarlos a entornos con obstáculos móviles.
- Creación de otros algoritmos que mejoren a los ya propuestos, HCTNav y NafisNav, en términos de memoria y/o longitud de ruta.
- Implementación de los algoritmos existentes en prototipos de robots móviles reales, en los que nuevos problemas pueden aparecer en su aplicación a entornos físicos.
- Aplicación a nodos móviles en redes de sensores cubriendo aspectos alternativos como cobertura u optimización.

References

[1] K. Sedighi, H. Ashenay, K. Manikas, T. W. Wainwright, R. L. H. M. Tai, Autonomous local path planning for a mobile robot using a genetic algorithm, In Congress on Evolutionary Computation (CEC2004), IEEEXplore, 2004.

[2] V.J. Lumelski, A. A. Stepanov, Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment, IEEE Transactions on Automatic Control, Vol. AC.31, No. 11, Nov. 1986.

[3] J. T. Schwartz, M. Sharir, on the piano mover's problem II: General techniques for computing topological properties of real algebraic manifolds. Adv. Appl. Math., pp. 298-351, 1983.

[4] N. Sariff, N. Buniyamin, Genetic Algorithm versus Ant Colony Optimization Algorithm: Comparison of Performances in Robot Path Planning Application. In 7th International Conference on informatics in control, automation and robotics (ICICINCO 2010), 15th- 20th June. Madeira, Portugal, 2010.

[5] G. Xiao-Guang, Xiao-Wei Fu, Da-Qing Chen, Genetic-Algorithm-Based Approach to UAV Path Planning Problem, In 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, 2005.

[6] N. Bin, C.X., Z. Liming, X. Wendong, Recurrent Neural Network for Robot Path Planning. Parallel and Distributed Computing: Applications and Technologies, (Springer Berlin / Heidelberg).2004. 3320/2005.

[7] J.E. Bella, P. R. McMullenb, Ant colony optimization techniques for the vehicle routing problem. Advanced Engineering Informatics, pp. 41-48, 2004.

[8] N. Sariff, N. Buniyamin, Comparative Study of Genetic Algorithm and Ant Colony Optimization Algorithm Performances for Robot Path Planning in Global Static Environments of Different Complexities. In 8th IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA2009), December 15-18, Daejon,Korea, 2009.

[9] N. Buniyamin, W.A.J.Wan Ngah , N. Sariff , Z. Mohamad , A Simple Local Path Planning Algorithm for Autonomous Mobile Robots, International Journal of Systems Applications, Engineering & Development, Issue 2, Volume 5, 2011.

[10] E.V. Kumar, M. Aneja, D. Deodhare, Solving a Path Planning Problem in a Partially Known Environment using a Swarm Algorithm, in IEEE International Symposium on Measurements and Control in Robotics, Bangalore, India, 2008.

[11] M. I. Ribeiro, Obstacle Avoidance, in Navigation/Collision Avoidance, Instituto de Sistemas e Robótica, Instituto Superio Técnico, 3rd November 2005.

[12] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, "Chapter 15: Visibility Graphs", Computational Geometry, Algorithms and Applications , Springer-Verlag, pp. 323–333, ISBN 978-3-540-77973-5, 2008.

[13] S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani, Algorithms, McGraw-Hill, pp. 89-112, 2006.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Third Edition, The MIT Press, Cambridge, Massachusetts, London, England, 2009.

[15] EW. Dijkstra, A note on two problems in connexion with graphs. Numer. Math. 1(1), 269–271, 1959.

[16] S. Rabin, AI Game Programming Wisdom, Charles River Media, 2002.

[17] R. Bellman, On a routing problem, Quarterly of Applied Mathematics 16, pp. 87-90, 1958.

[18] R. Diestel, Graph Theory, Graduate Texts in Mathematics. Springer-Verlag, vol. 173, pp. 6–9, 2005.

[19] G. Foux, M. Heymann, A. Bruckstein, Two-dimensional robot navigation among unknown stationary polygonal obstacles, IEEE Transactions on Robotics and Automation 9, 96–102, 1993.

[20] Z. Zhang, Z. Zhao, A Multiple Mobile Robots Path planning Algorithm Based on A-star and Dijkstra Algorithm, International Journal of Smart Home Vol.8, No.3 , pp.75-86 , 2014.

[21] R. C. Prim, Shortest connection networks and some generalizations. In: Bell System Technical Journal, pp. 1389–1401, 36, 1957.

[22] Y. Chao, W. Hongxia, Developed Dijkstra Shortest Path Search Algorithm and Simulation, International Conference on Computer Design And Applications (ICCDA 2010), IEEE 2010, V. 1, pp. 116-119, 2010.

[23] L. Zhang, W. Yang, Q. Rao, W. Nai, D. Dong, An Energy Saving Routing Algorithm Based on Dijkstra in Wireless Sensor Networks, Journal of Information & Computational Science 10:7, pp. 2087–2096, 2013.

[24] Y. Zuo, Z. Ling,Y. Yuan, A hybrid multi-path routing algorithm for industrial wireless mesh networks, EURASIP Journal on Wireless Communications and Networking, 2013.

[25] N. Pushpalatha, B.Anuradha, Shortest Path Position Estimation between Source and Destination nodes in Wireless Sensor Networks with Low Cost, International Journal of Emerging Technology and Advanced Engineering, 2012.

[26] P. E. Hart, N. J. Nilsson, and B. Raphael, A formal basic for the heuristic determination of minimum cost paths, IEEE Trans, Systems Science and Cybernetics, vol. 4, no. 100-107, 1968.

[27] M. Buckland, Programming Game AI by Example, Jones & Bartlett Learning, 2005.

[28] N. J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. International Joint Conference on Artificial Intelligence, Washington, DC. , pp. 509-520, May 1969.

[29] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley Publishing Company, 1984.

[30] D. Ananya, M. priyadarsini, M. Prakruti, K.D. Pradipta, C.M. Subash, Improved real time A* algorithm for path planning of mobile robot in quadrant based environment, International Journal on Advanced Computer Theory and Engineering (IJACTE), 1, 2319-2526, 2012.

[31] P. Kumar Das, A. Konar, R. Laishram, Path planning of mobile robot in unknown environment, Special Issue of IJCCT Vol.1 Issue 2, 3, 4, for International Conference [ACCTA-2010], 2010.

[32] T. Cazenave, Optimizations of data structures, heuristics and algorithms for path-finding on maps, CIG'06 (May 22-24 2006), 2006 IEEE, 2006.

[33] B. M. ElHalawany, H. M.Abdel-Kader, A. TagEldeen, A.E.Elsayed, Z. B.Nossair, Modified A* Algorithm for Safer Mobile Robot Navigation, International Conference on Modelling, Identification & Control (ICMIC), 2013.

[34] T. Yuksel, A. Sezgin, An implementation of path planning algorithms for mobile robots on a grid based map, Electrical & Electronics Engineering Department, Ondokuz Mayls University, Tech. Rep., 2005.

[35] Y. F. Hsu, C. H. Huang, W. R. Huang, W. C. Chu, An Improved Minimum-Cost Pathfinding Algorithm for Mobile Robot Navigation, Journal of Advances in Computer Network, Vol. 1, No. 3, pp. 189-193, September 2013.

[36] R. Zhou, E. Hansen, Multiple sequence alignment using anytime A*, Proc. of the National Conference on Artificial Intelligence, 2002.

[37] CenZeng, Q. Zhang, X. Wei, GA-based Global Path Planning for Mobile Robot Employing A* Algorithm, Journal of Computers, vol. 7, no. 2, February 2012.

[38] A. Yufka, O. Parlaktuna, Performance Comparison of Bug Algorithms for Mobile Robots, 5th International Advanced Technologies Symposium (IATS'09), Karabuk, Turkey , May 13-15, 2009.

[39] V. Lumelsky, P. Stepanov, Effect of Uncertainty on Continuous Path Planning for an Autonomous Vehicle. Proceedings of the 23rd Conference on Decision and Control, pp. 1616-1621, December 1984.

[40] M. Evgeni, R. Ehud, CAUTIOUSBUG: A Competitive Algorithm for Sensory-Based Robot Navigation, Proceedings of 2004 IEEE/RSJ international Conference on Intelligent Robots and Systems (IROS), Japan, vol. 3, pp. 2757-2762, September 28 – October 2, 2004.

[41] H. Choset , K. M. Lynch, S. Hutchinson , G. A. Kantor, W. Burgard , L. E. Kavraki, S.Thrun , Principles of Robot Motion : Theory, Algorithms, and Implementations, MIT Press, vol. 12, no. 3, 2005.

[42] V. Sezer, M. Gokasan, A Novel Obstacle Avoidance Algorithm: Follow the Gap Method, Robotics and Autonomous Systems vol. 60 (9), pp. 1123-1134, 2012.

[43] A. Sankaranarayanan, M. Vidyasagar, A new path planning algorithm for moving a point object amidst unknown obstacles in a plane, Proc. of the IEEE Int. Conf. Robot. Autom. 3, pp. 1930–1936, 1990.

[44] V. Lumelsky, S.Tiwari, An Algorithm for Maze Searching with Azimuth Input, IEEE, 1050- 4729/94$ 03.000, 1994.

[45] V. Lumelsky and A. Stepanov, Path planning strategies for point mobile automation moving amidst unknown obstacles of arbitrary shape, Algorithmica, vol. 2, pp. 403–430, 1987.

[46] I. Kamon, E. Rivlin, Sensory-Based Motion Planning with Global Proofs. IEEE Transaction on Robotics and Automation, Vol. 13, Pages 814-822, December 1997.

[47] M. Zohaib, M. Pasha, R.A. Raiz, N. Javaid, M. Ilahi, R.D. Khan, Control Strategies for Mobile Robot With Obstacle Avoidance, Journal of Basic and Applied Scientific Research (JBASR), vol. 3(4), pp. 1027-1036, 2013.

[48] Y. Horiuchi and H. Nohorio, Evaluation of Path Length Made in Sensor-had Path-Planning with the Alternative Following, In Proc. IEEE ICRA'O1, pp. 909.916, 2001.

[49] J. Ng, T. Bräunl, Performance Comparison of Bug Navigation Algorithms, Journal Intell Robot Syst, pp. 73–84, 50, 2007.

[50] O. Khatib, Real-time obstacle avoidance for manipulators and mobile robots. The International Journal of Robotics Research, 5(1):90-98, 1986.

[51] S.S. GE, Y.J. CUI, Dynamic Motion Planning for Mobile Robots Using Potential Field Method, Kluwer Academic Publishers, Manufactured in the Netherlands, Autonomous Robots 13, pp. 207–222, 2002.

[52] R.C. Arkin, Motor Schema-Based Mobile Robot Navigation, International Journal of Robotics Research, Vol. 8, No. 4, pp. 92-112, August 1989.

[53] T. Balch, R.C. Arkin, Avoiding the Past: A Simple but Effective Strategy for Reactive Navigation, Proceedings from IEEE International Conference on robotics and automation, 1993.

[54] E. Gat, Navigation templates: enhancements, extensions, and experiments, Proceedings of the IEEE International Conference on Robotics and Automation, pp. 541–547, 1993.

[55] H.F. Durrant-Whyte, Integration, Coordination and Control of Multi-sensor Robot Systems, Kluwer, Boston, MA, 1988.

[56] T. Lozano-Perez, Automatic planning of manipulator transfer movements, IEEE Transactions on Systems, Man and Cybernetics, pp. 781–798. 1981.

[57] D. Miller, A spatial representation system for mobile robots, in: Proceedings of the IEEE International Conference on Robotics and Automation, St. Louis, MO, pp. 122–128, 1985.

[58] O. Setter, M. Sharir, D. Halperin, Constructing Two-Dimensional Voronoi Diagrams via Divide-and-Conquer of Envelopes in Space, Proceedings of the IEEE Sixth International Symposium on Voronoi Diagrams, pp. 43-52, 2009.

[59] J. Crowley, Navigation for an intelligent mobile robot, IEEE Journal on Robotics and Automation RA-1, pp. 31–41, (1) 1985.

[60] R.A. Brooks, Solving the find-path problem by good representation of free space, IEEE Transactions on systems, man, and Cybernetics, VOL. SMC-13, NO. 3, pp. 190-197, 1983.

[61] E. Koch, Simulation of path planning for a system with vision and map updating, in: Proceedings of the IEEE International Conference on Robotics and Automation, St. Louis, MO, pp. 146–160, 1985.

[62] H.P. Moravec, Sensor fusion in certainty grids for mobile robots, AI Magazine, pp. 61–74, 1988.

[63] W. Na, C. Ping, Path Planning Algorithm of Level Set based on Grid Modeling, Proceedings of the IEEE International Conference On Computer Design And Applications (ICCDA 2010), pp. 508-510, 2010.

[64] R. Chatila, J. Laumond, Position referencing and consistent world modelling for mobile robots, in: Proceedings of the IEEE International Conference on Robotics and Automation, St. Louis, MO, pp. 138–170, 1985.

[65] A. Zelinsky, A mobile robot navigation exploration algorithm, IEEE Transactions on Robotics and Automation, pp. 707–717, Vol. 8, No. 6, 1992.

[66] D. Chen, R. Szczerba, J. Uhran, A framed-quadtree approach for determining Euclidean shortest paths in a 2D environment, IEEE Transactions on Robotics and Automation, pp. 668–680, Vol. 13, No. 5, 1997.

[67] A. Arleo, J.R. Millán, D. Floreano, Efficient learning of variable-resolution cognitive maps for autonomous indoor navigation, IEEE Transactions on Robotics and Automation, Vol. 15, No. 6, pp. 990–1000, 1999.

[68] I. Chˆaari, A. Koubˆaa, H.Bennaceur, A. Ammar, S. Trigui, M. Tounsi, E. Shakshuki, H. Youssef, On the Adequacy of Tabu Search for Globa Robot Path Planning Problem in Grid Environments, 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014) , 2014.

[69] A. Elfes, Using Occupancy Grids for Mobile Robot Perception and Navigation, IEEE Computer Society, pp. 46-57, 1989.

[70] J.O. Wallgrün, Hierarchical Voronoi Graphs: Spatial Representation and Reasoning for Mobile Robots, chapter 2, Robot Mapping, pp. 11-43, Springer-Verlag Berlin Heidelberg , 2010.

[71] O. Hachour, Path planning of Autonomous Mobile robot, International Journal of systems Applications, Engineering & Development, Issue 4, Volume 2, 2008.

[72] S. Thrun, A. Bucken, Learning Maps for Indoor Mobile Robot Navigation, School of Computer Science, Carnegie Mellon University, Pittsburgh , 1996.

[73] M. Pala, N. Osati Eraghi, F. Lopez-Colino, A. Sanchez, A. de Castro, J. Garrido, HCTNav: A Path Planning Algorithm for Low-Cost Autonomous Robot Navigation in Indoor Environments, International Journal of Geo-Information, vol.2, 729-748, Aug. 2013.

[74] N. Osati Eraghi, F. Lopez-Colino, A. de Castro, J. Garrido, Path Length Comparison in Grid Maps of Planning Algorithms: HCTNav, A* and Dijkstra, Proceedings of the IEEE Conference on Design of Circuits and Integrated Circuits (DCIS), Nov. 2014.

[75] J.E. Bresenham, Algorithm for computer control of a digital plotter. IBM Syst. J., 4, 25–30, 1965.

[76] L. Shapiro, G. Stockman, Computer Vision, Prentice Hall, pp. 69–73, 2002.

[77] D. Hearn, M. P. Baker, Computer Graphics C version, Prentice Hall, 2nd edition, 1997.

[78] N. Osati Eraghi, F. Lopez-Colino, A. de Castro, J. Garrido, NafisNav: an Indoor Navigation Algorithm for Embedded Systems and based on Grid Maps, Proceedings of the IEEE International Conference on Industrial Technology (ICIT), Mar. 2015.

Glossary

2-D:	Two-Dimensional space
ACO:	Ant Colony Optimization
ANSI-C:	American National Standard Institute (ANSI) for the C programming language
BFS:	Breadth First Search
CAD:	Computer-Aided Design
CPU:	Central Processing Unit
DFS:	Depth First Search
DOA:	Direction Of Arrival
ESLHA:	Energy Saving-oriented Least-Hop routing Algorithm
ESRAD:	Energy Saving-oriented Routing Algorithm based on Dijkstra
FIFO:	First Input First Output
FILO:	First Input Last Output
GA:	Genetic Algorithm
GCC:	GNU C Compiler
HCTNav:	HCTLab Navigation
HW:	Hardware
IBM:	International Business Machines corporation
kB:	kilo Byte

NafisNav: Nafiseh Navigation

NaT: Navigation Templates

ROS: Robot Operating System

SW: Software

WSN: Wireless Sensor Networks

Appendix A

Small maps set

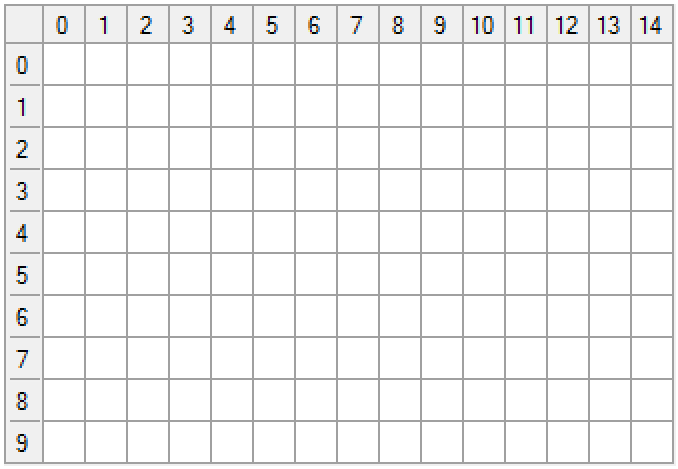


Fig. A.1. Map-1

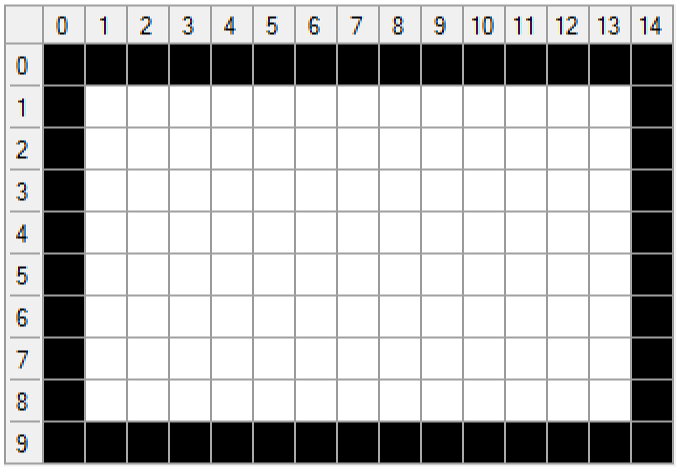


Fig. A.2. Map-2

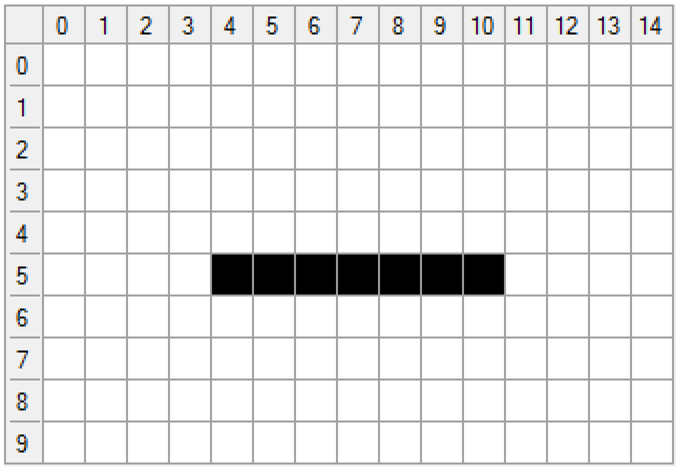


Fig. A.3. Map-3

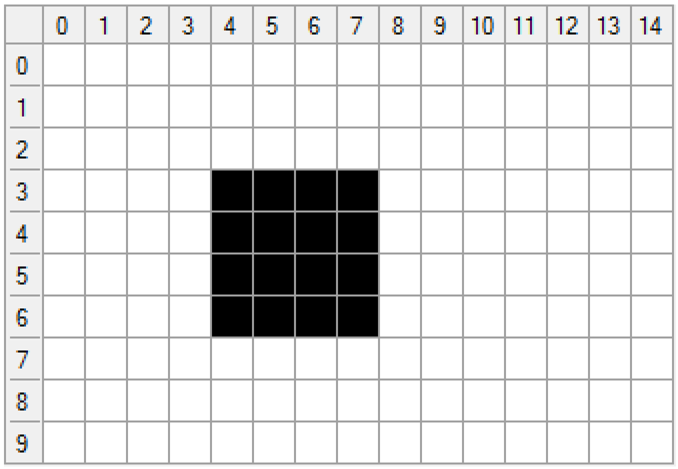


Fig. A.4. Map-4

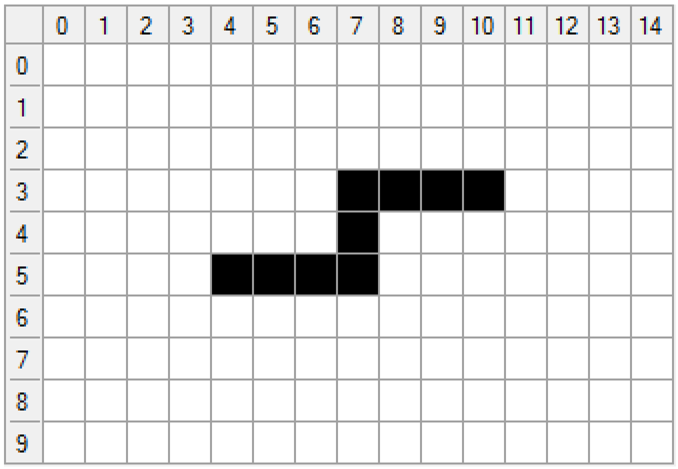


Fig. A.5. Map-5

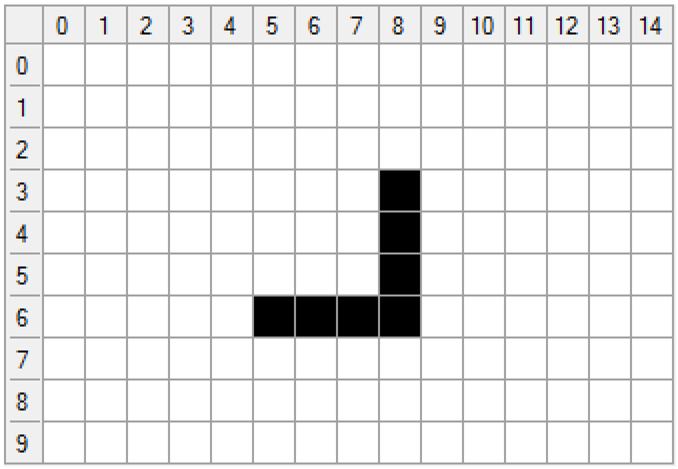


Fig. A.6. Map-6

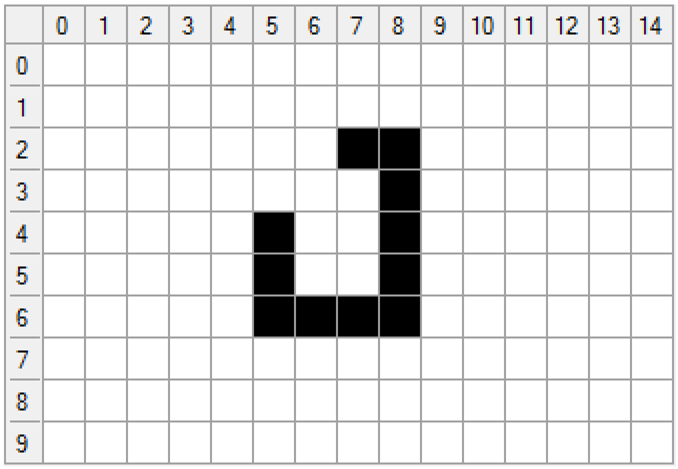


Fig. A.7. Map-7

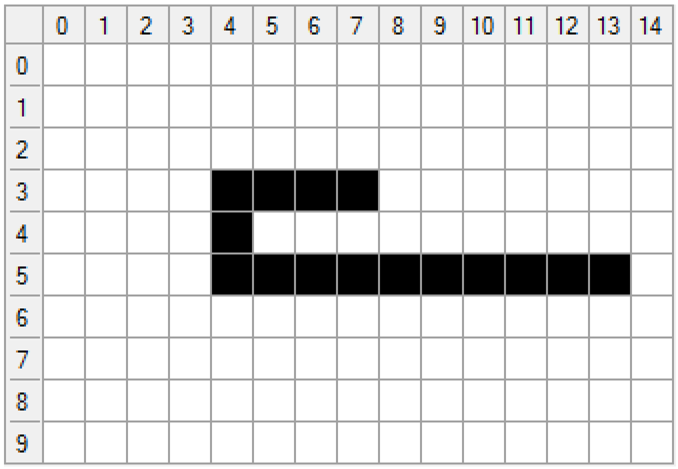


Fig. A.8. Map-8

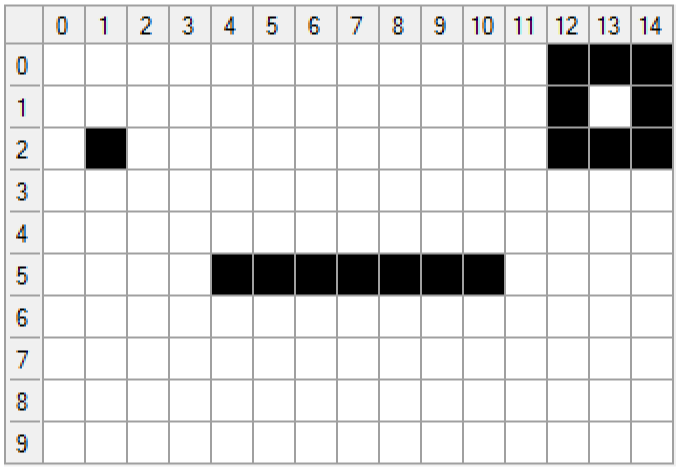


Fig. A.9. Map-9

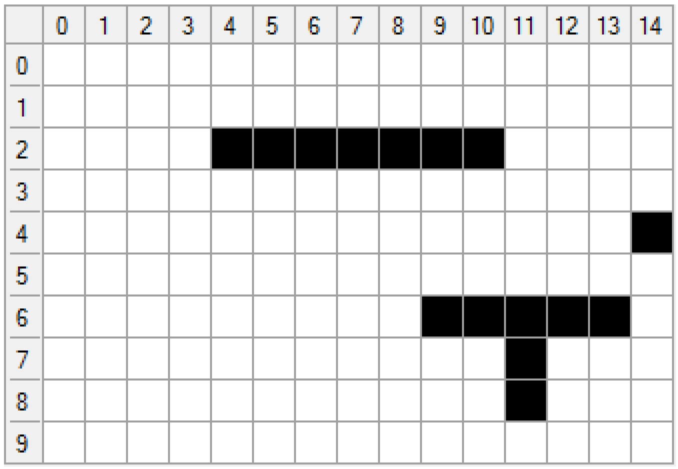


Fig. A.10. Map-10

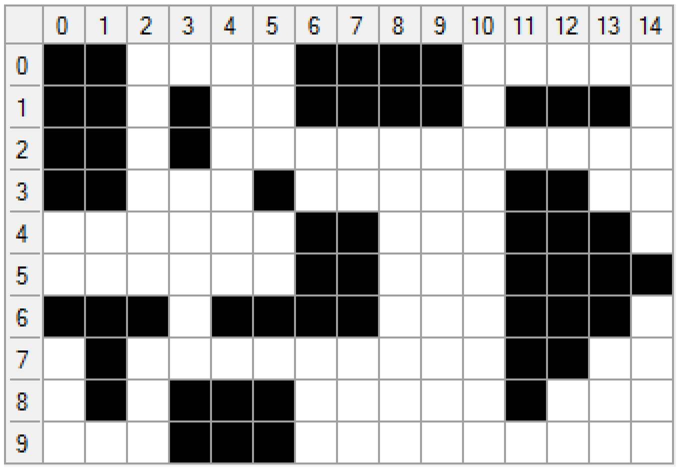


Fig. A.11. Map-11

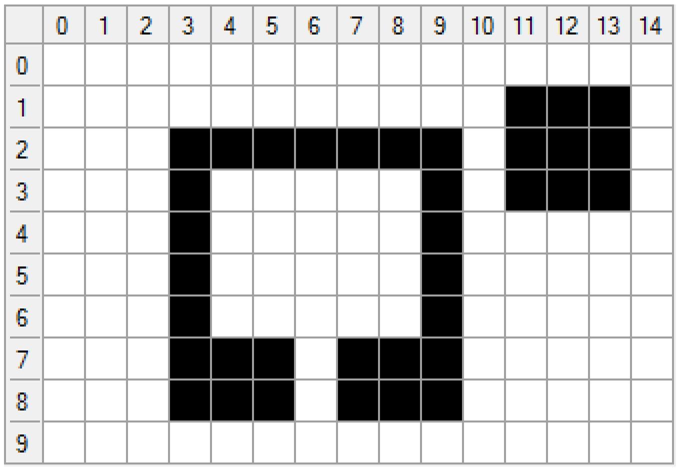


Fig. A.12. Map-12

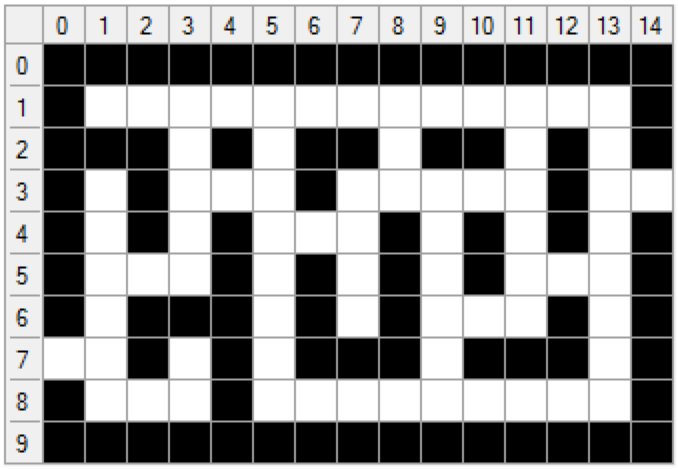


Fig. A.13. Map-13

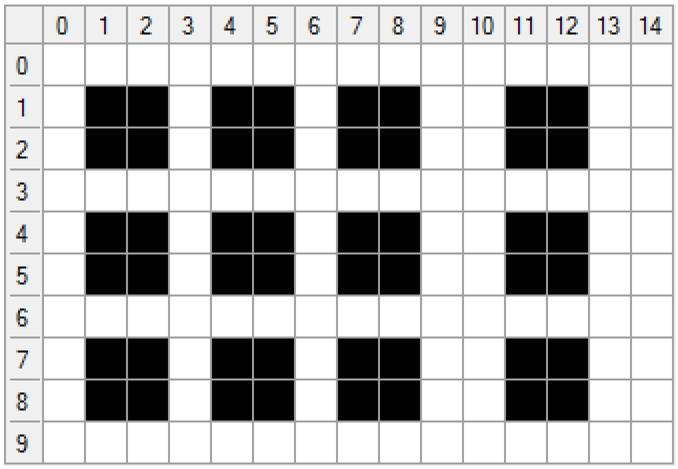


Fig. A.14. Map-14

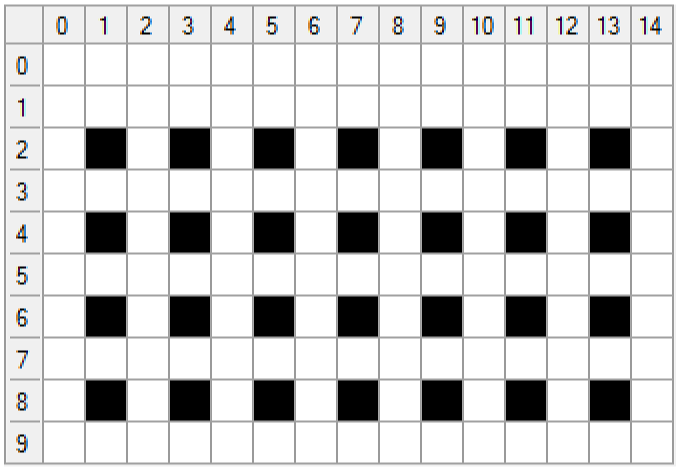


Fig. A.15. Map-15

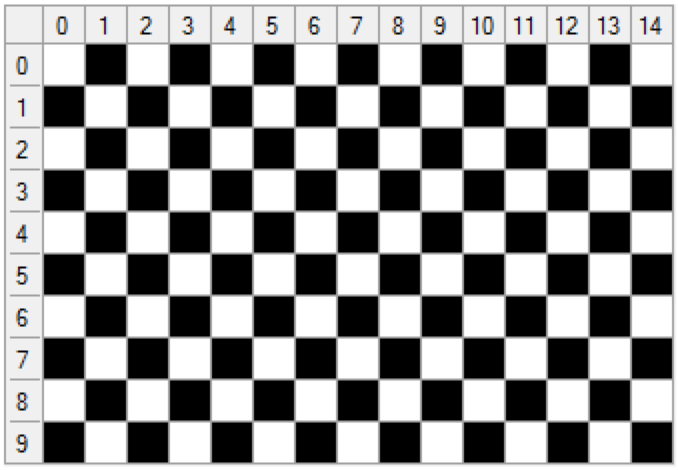


Fig. A.16. Map-16

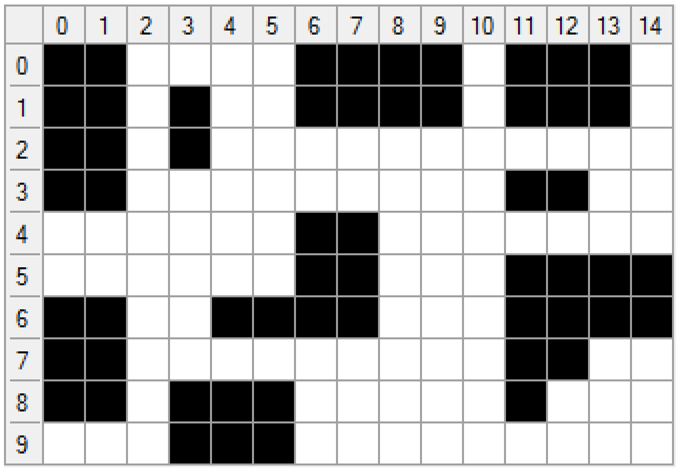


Fig. A.17. Map-17

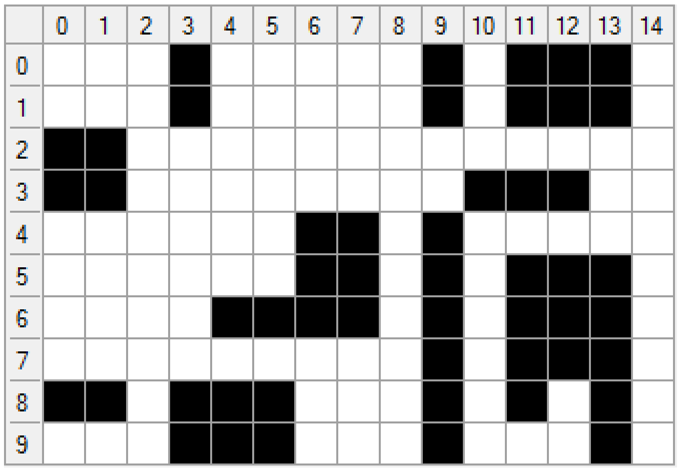


Fig. A.18. Map-18

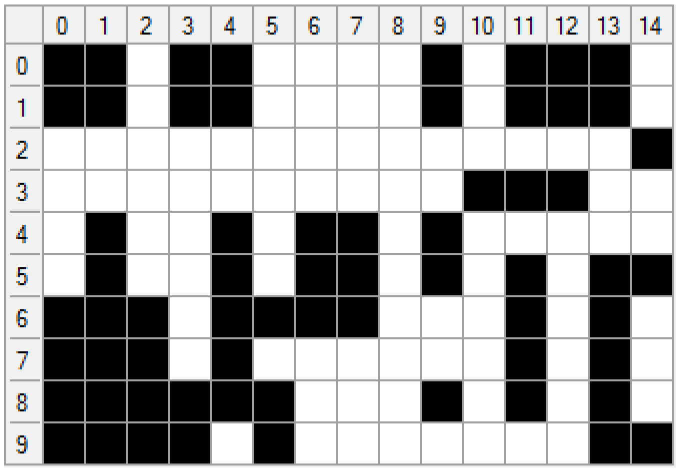


Fig. A.19. Map-19

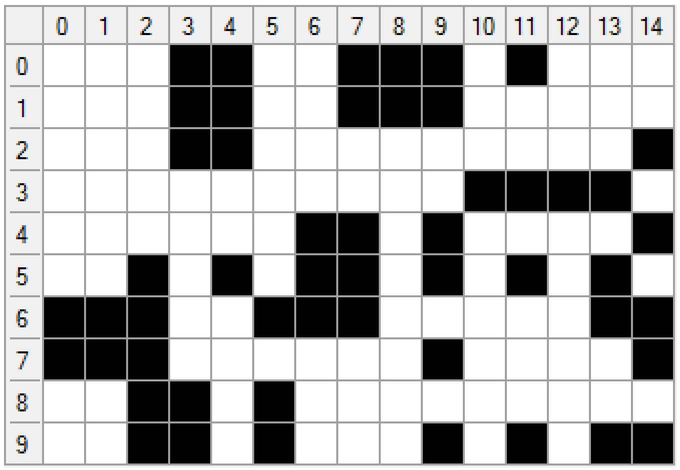


Fig. A.20. Map-20

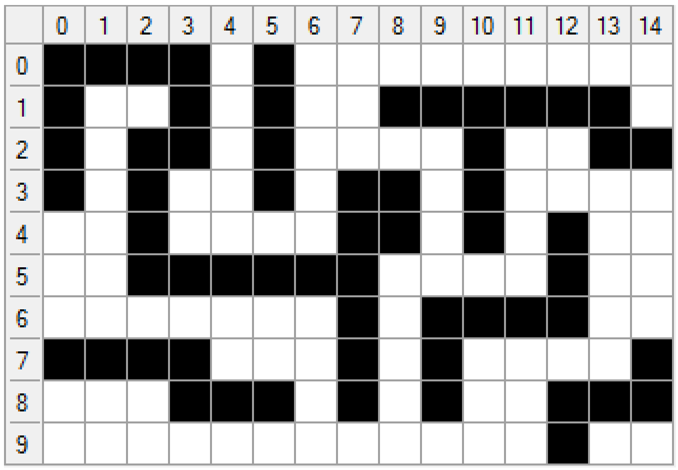


Fig. A.21. Map-21

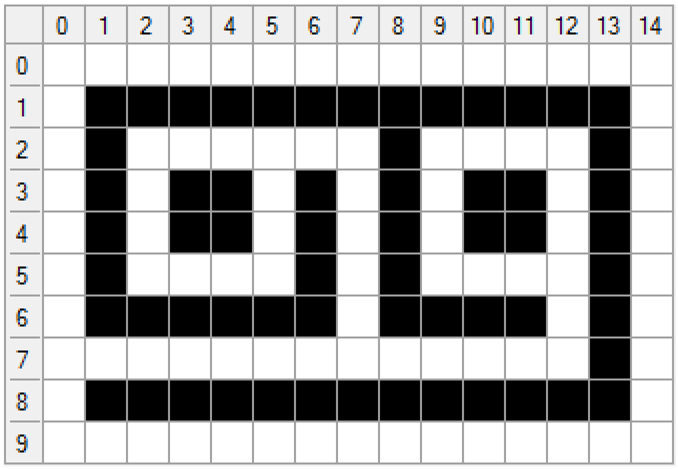


Fig. A.22. Map-22

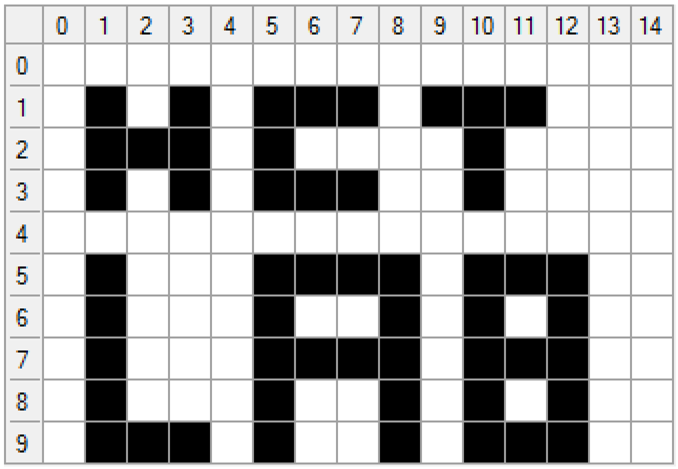


Fig. A.23. Map-23

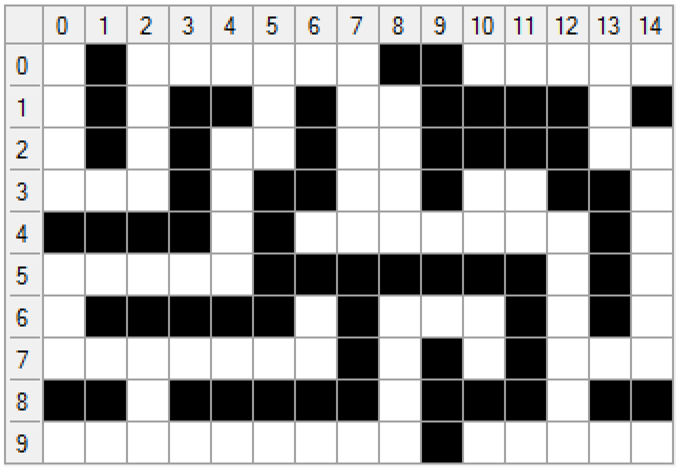


Fig. A.24. Map-24

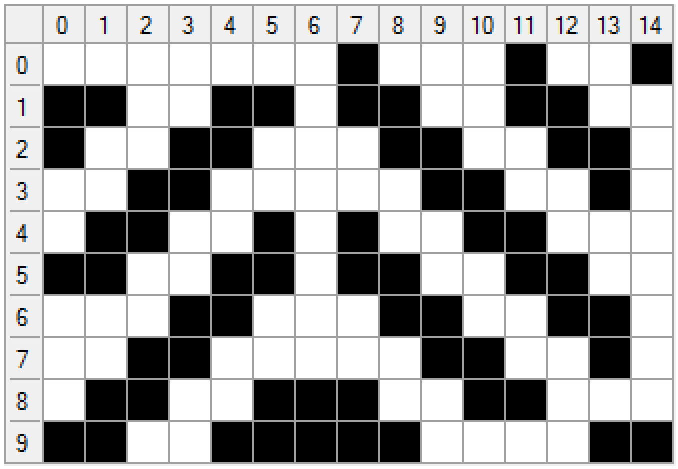


Fig. A.25. Map-25

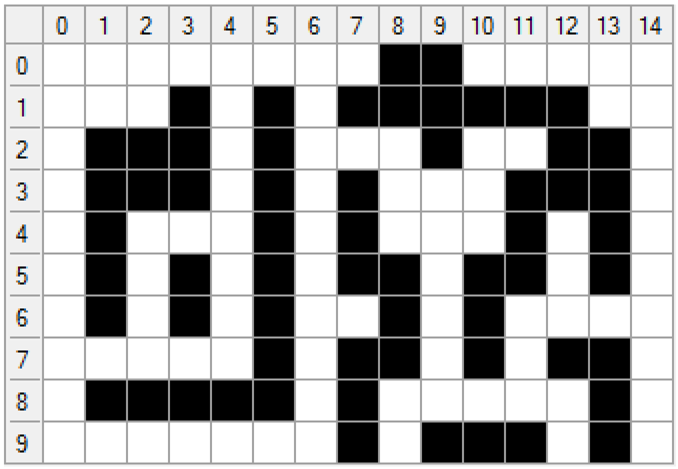


Fig. A.26. Map-26

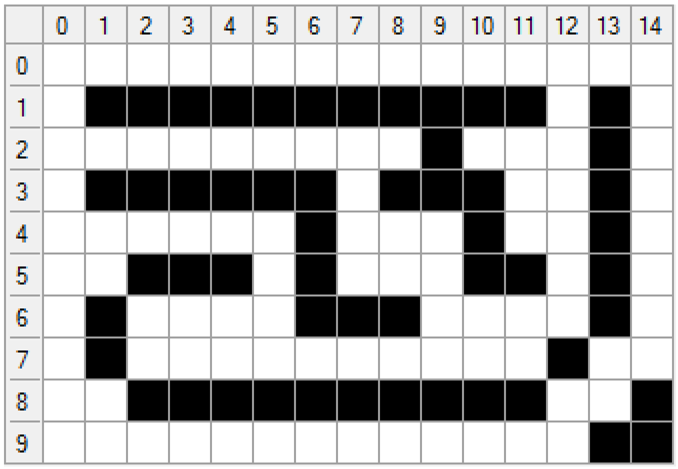


Fig. A.27. Map-27

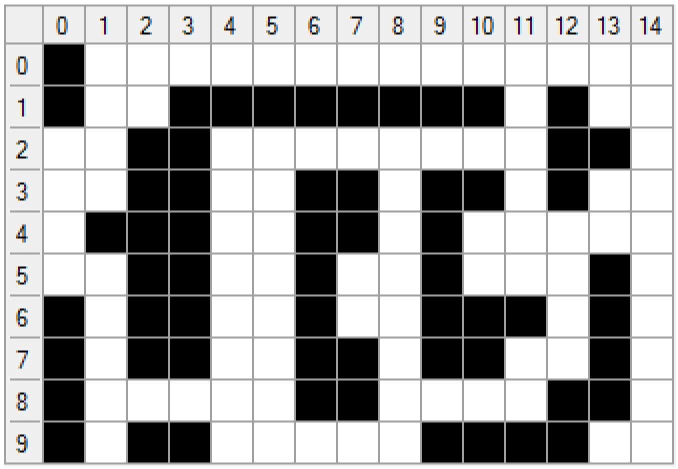


Fig. A.28. Map-28

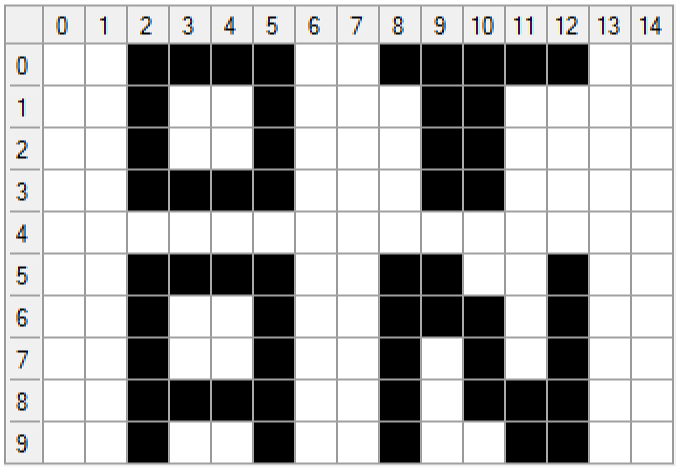


Fig. A.29. Map-29

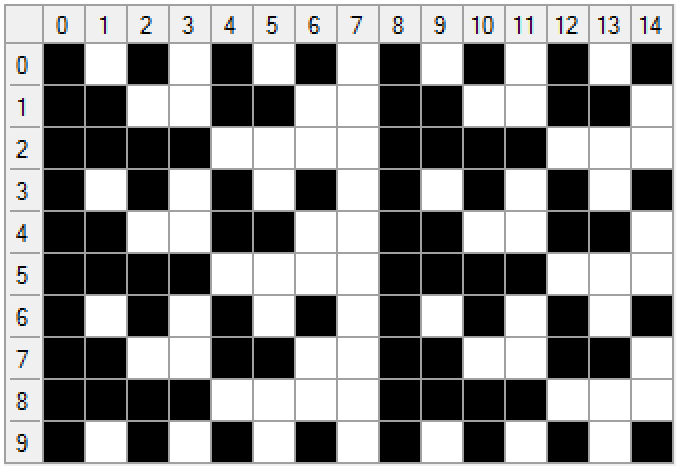


Fig. A.30. Map-30

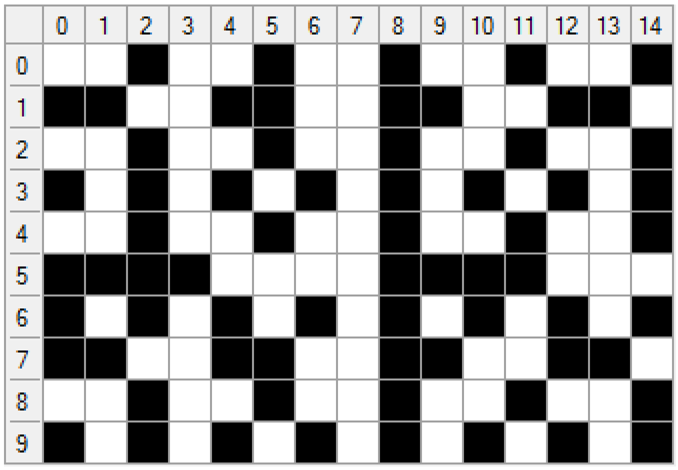


Fig. A.31. Map-31

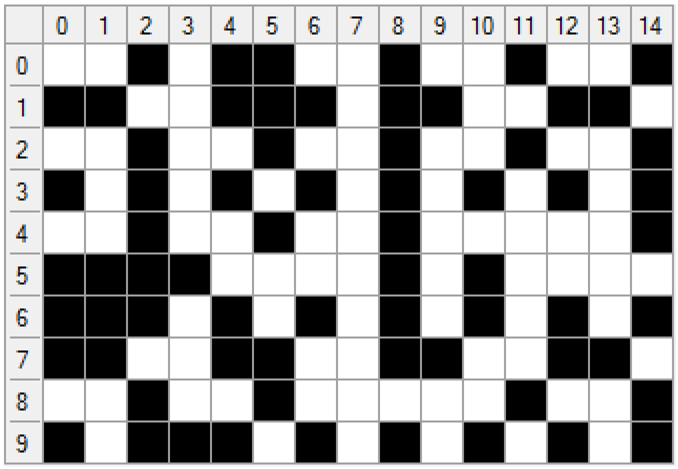


Fig. A.32. Map-32

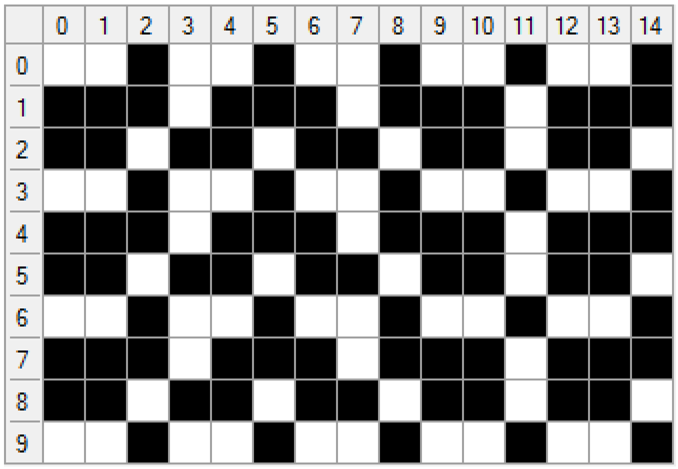
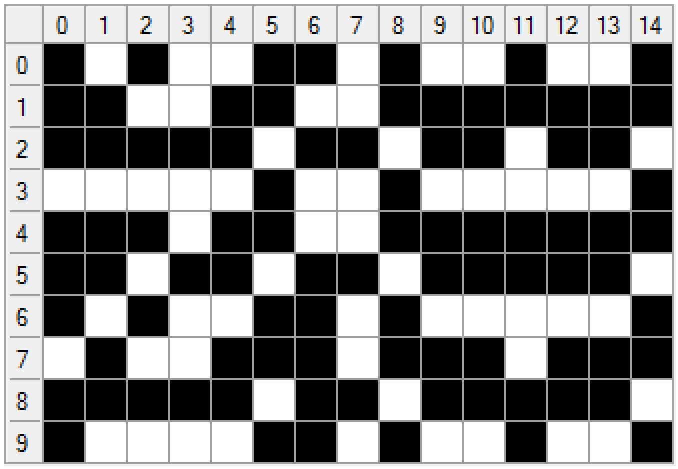
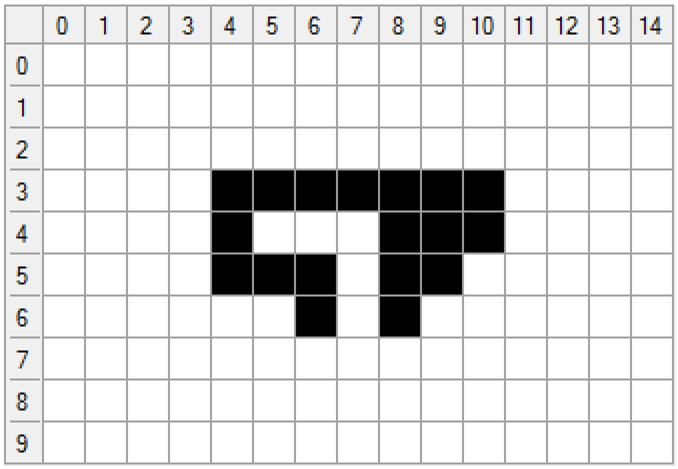


Fig. A.33. Map-33



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	■		■			■	■		■			■			■
1	■	■			■	■			■	■	■	■	■	■	■
2	■	■	■	■	■		■	■		■	■		■	■	
3						■			■						■
4	■	■	■		■	■			■	■	■	■	■	■	■
5	■	■		■	■		■	■		■	■	■	■	■	
6	■		■			■	■		■						■
7		■			■	■	■		■	■	■		■	■	■
8	■	■	■	■	■		■	■		■	■	■	■	■	
9	■					■	■		■			■			■

Fig. A.34. Map-34



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0															
1															
2															
3					■	■	■	■	■	■	■				
4					■				■	■	■				
5					■	■	■		■	■					
6							■		■						
7															
8															
9															

Fig. A.35. Map-35

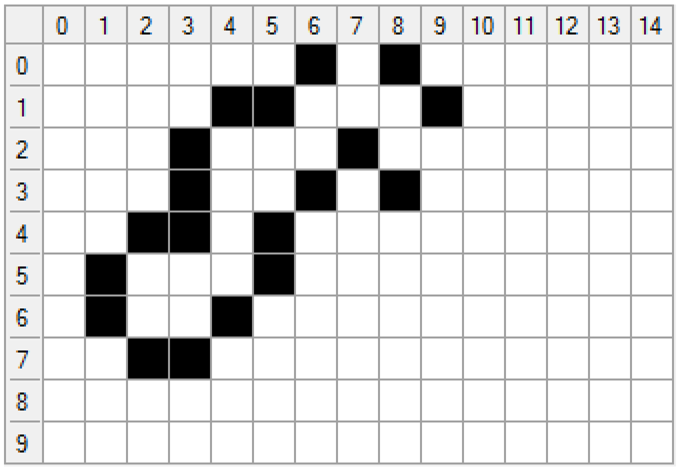


Fig. A.36. Map-36

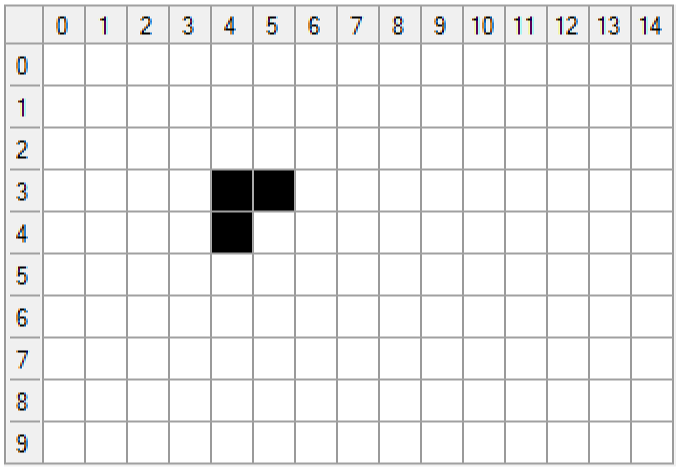


Fig. A.37. Map-37

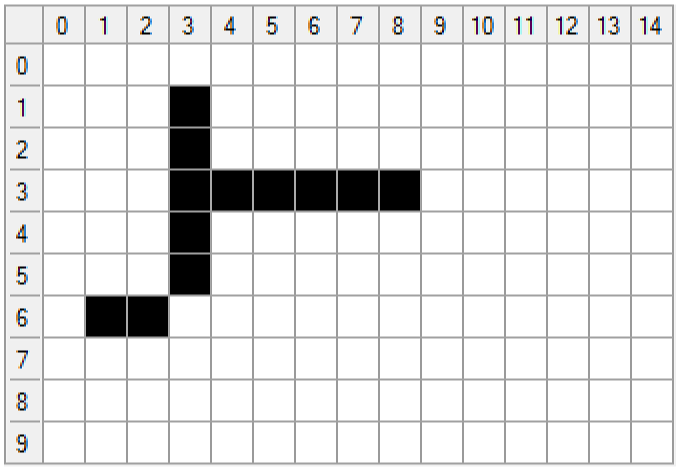


Fig. A.38. Map-38

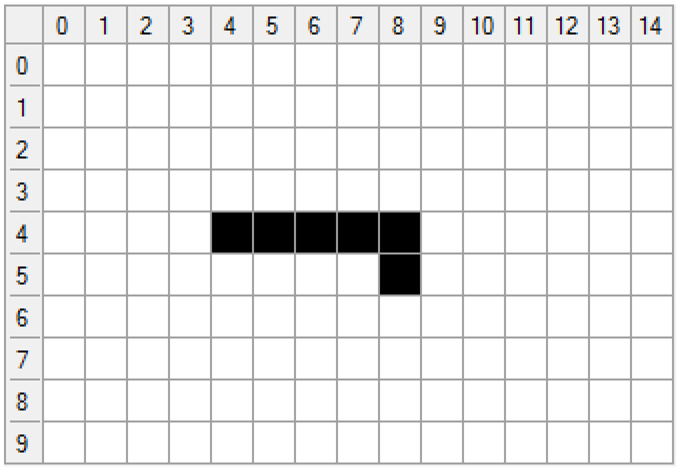


Fig. A.39. Map-39

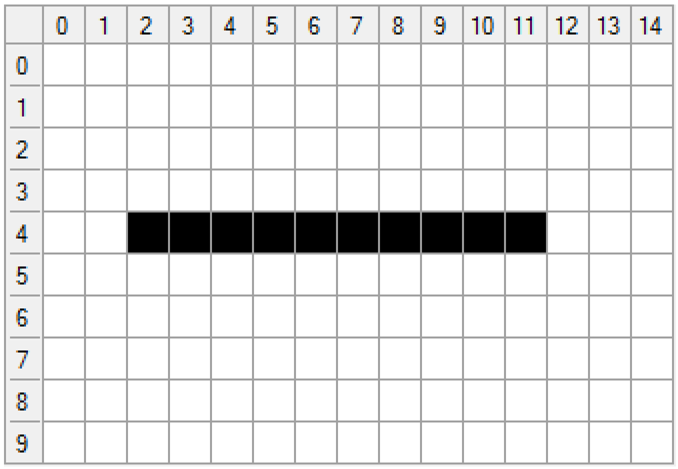


Fig. A.40. Map-40

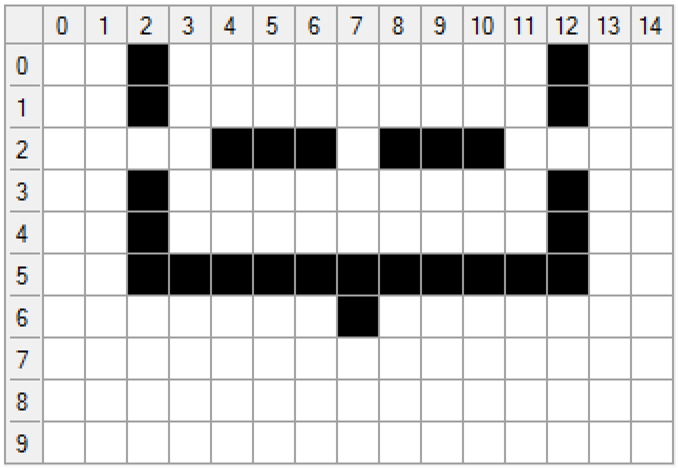


Fig. A.41. Map-41

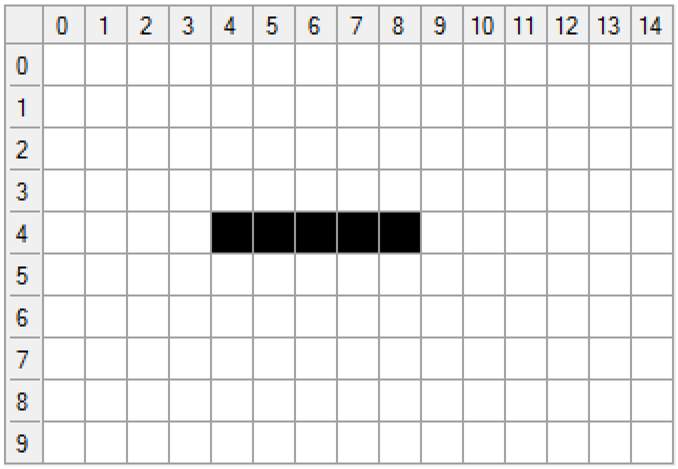


Fig. A.42. Map-42

Appendix B

Big maps set

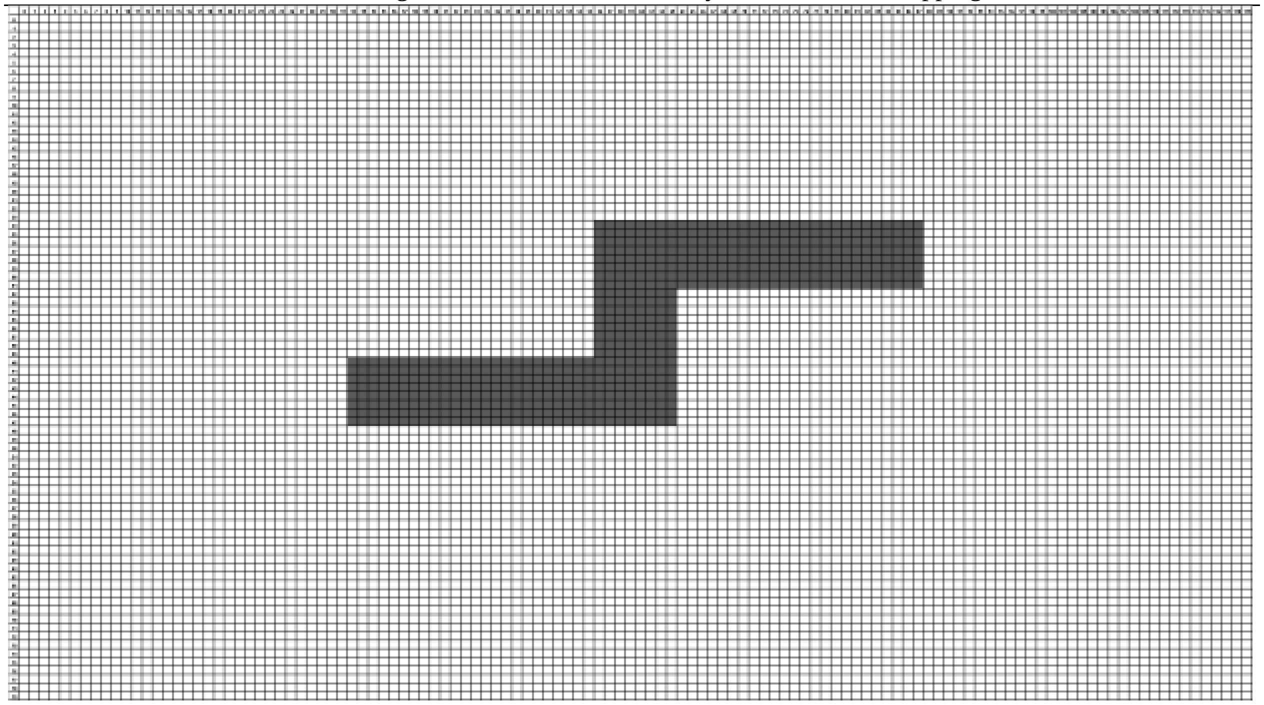


Fig. B.1. Map-05 scaled_x8

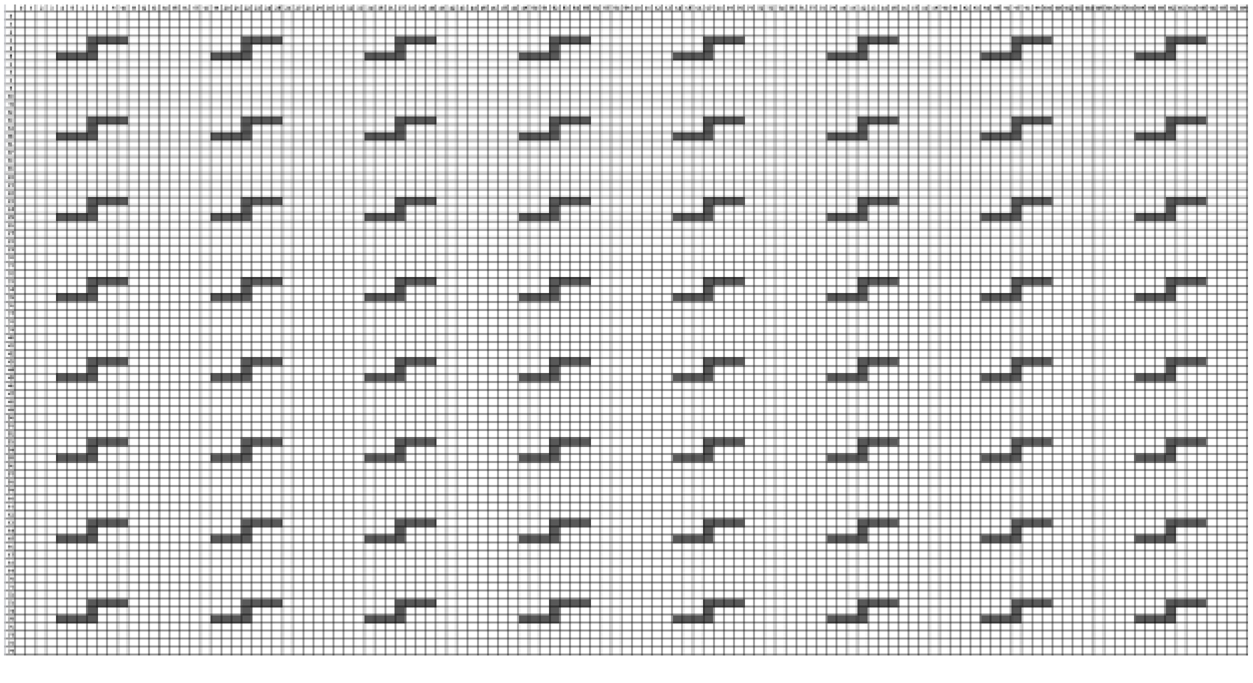


Fig. B.2. Map-05 repeated_x8

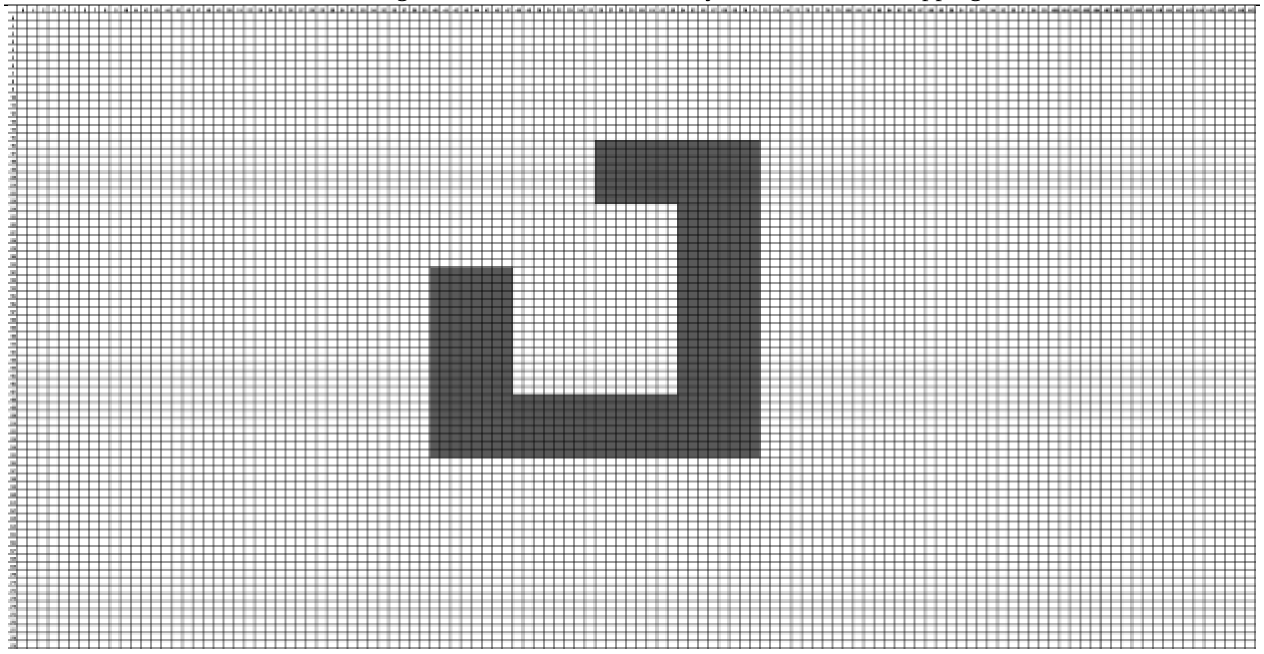


Fig. B.3. Map-07 scaled_x8

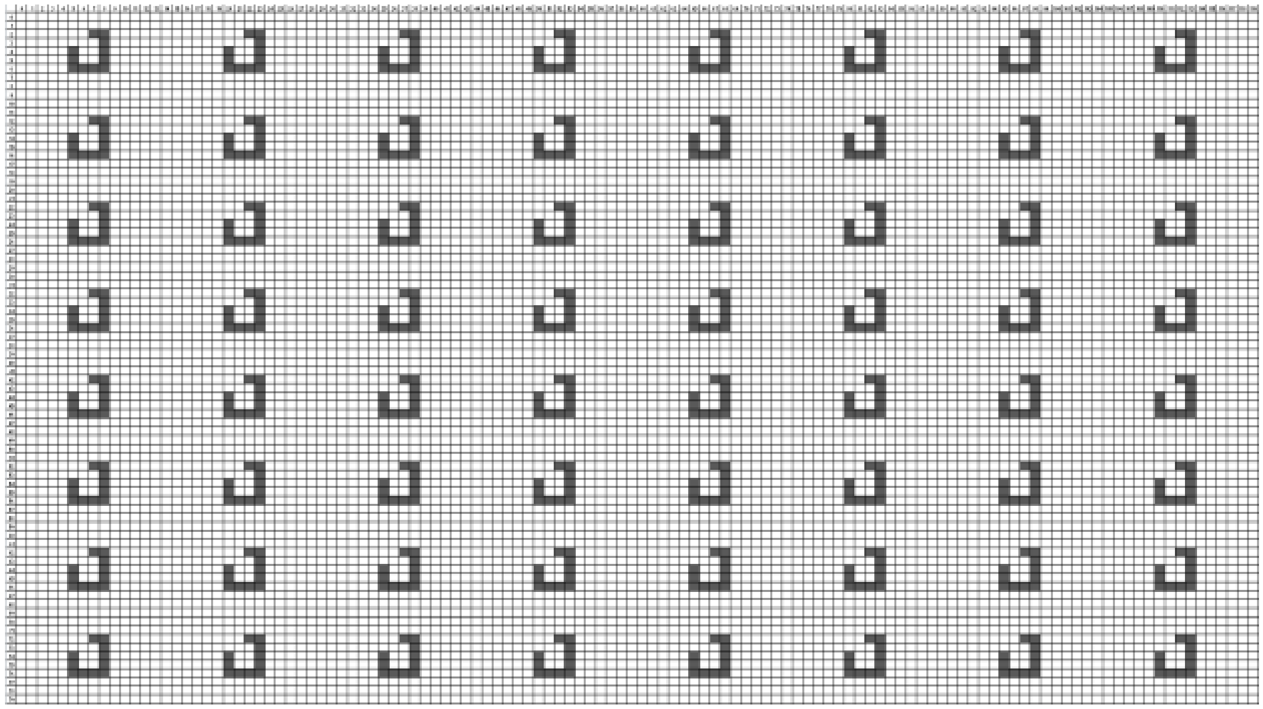


Fig. B.4. Map-07 repeated_x8

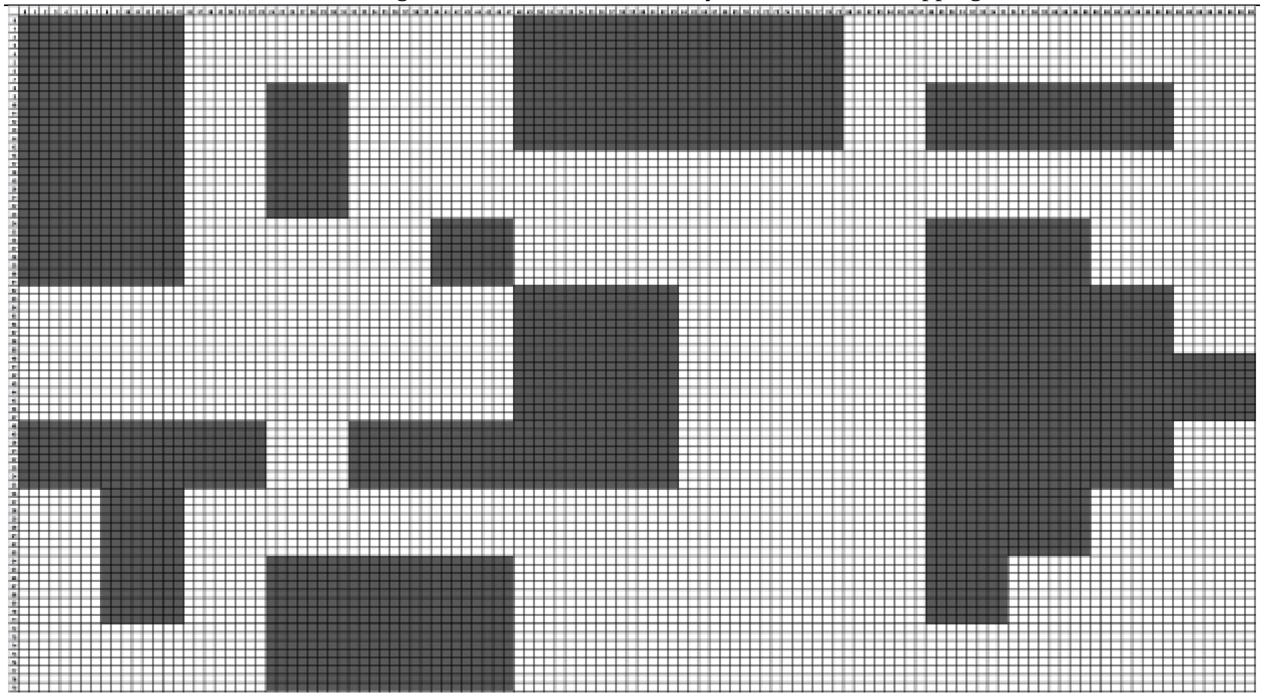


Fig. B.5. Map-11 scaled_x8

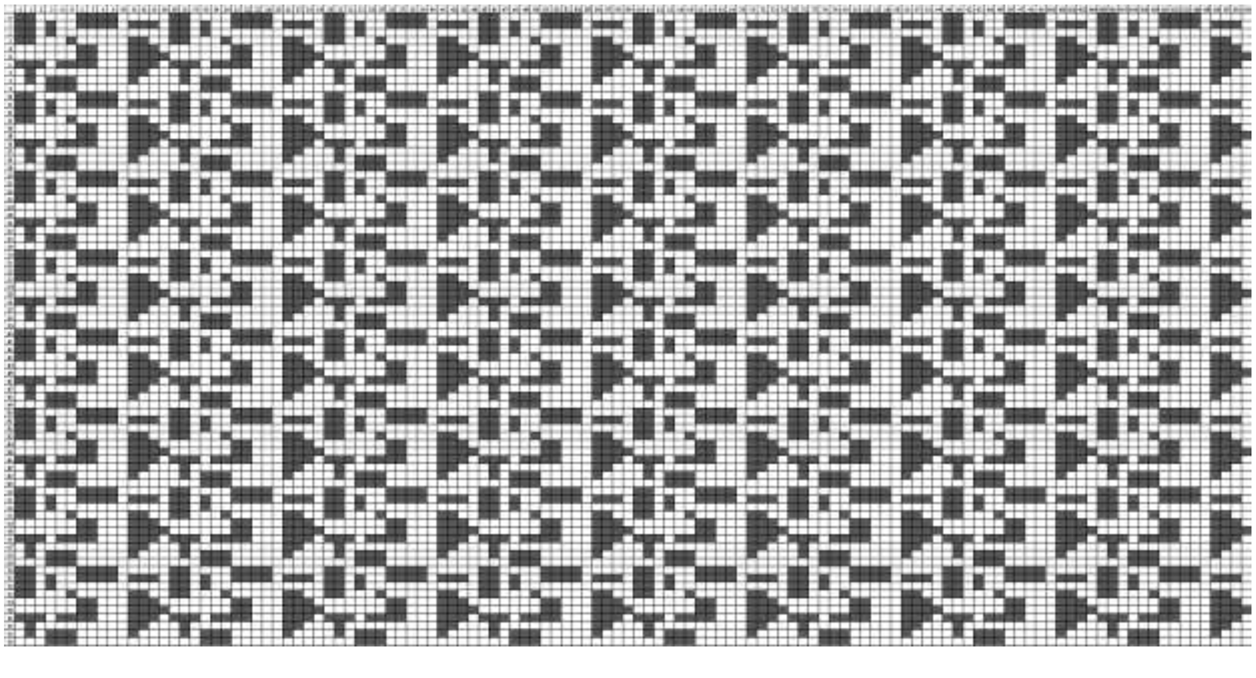


Fig. B.6. Map-11 repeated_x8

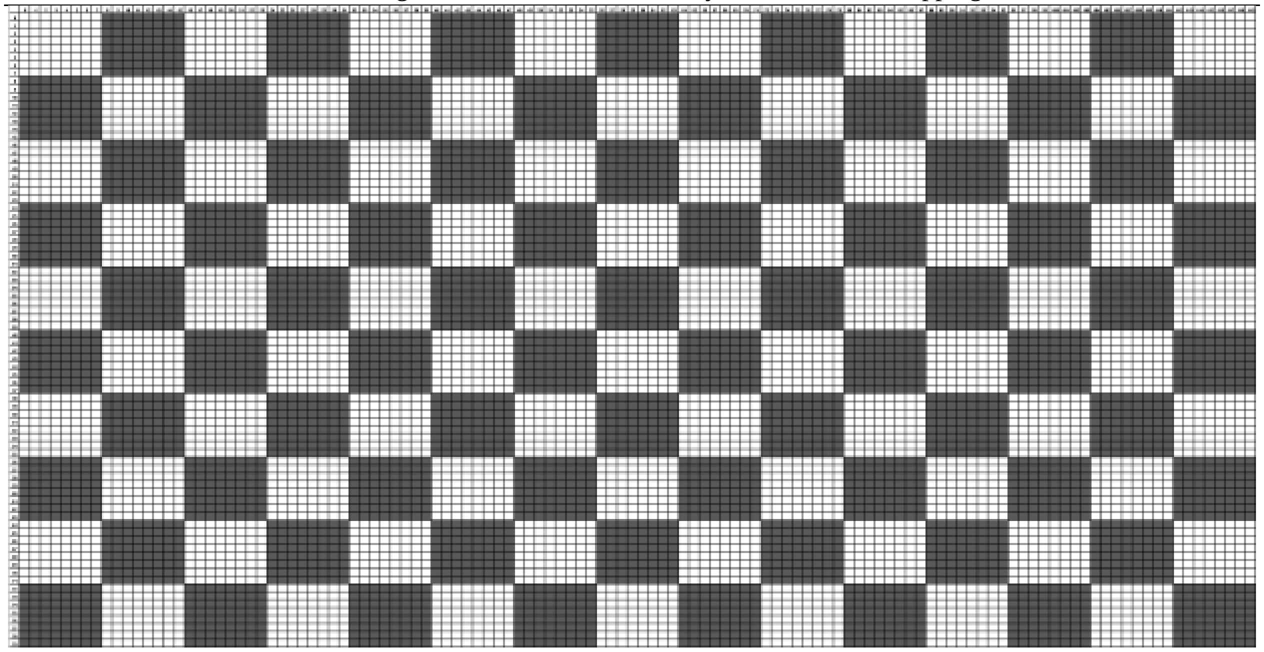


Fig. B.7. Map-16 scaled_x8

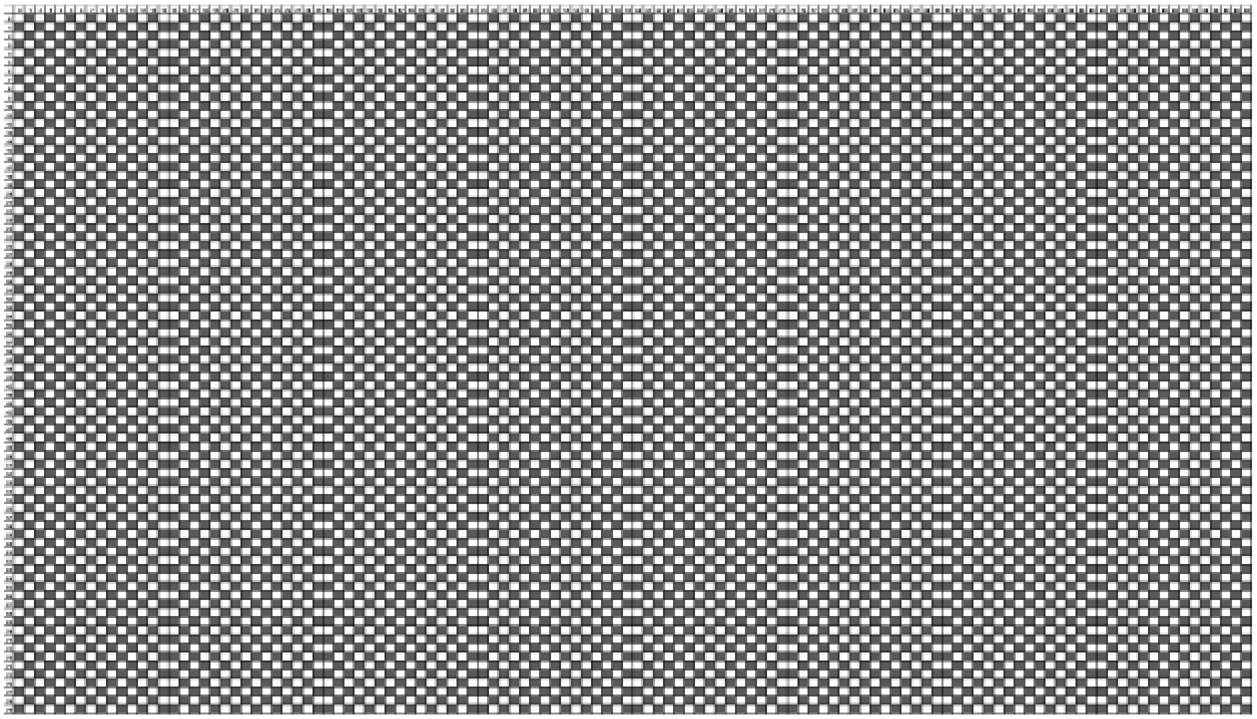


Fig. B.8. Map-16 repeated_x8

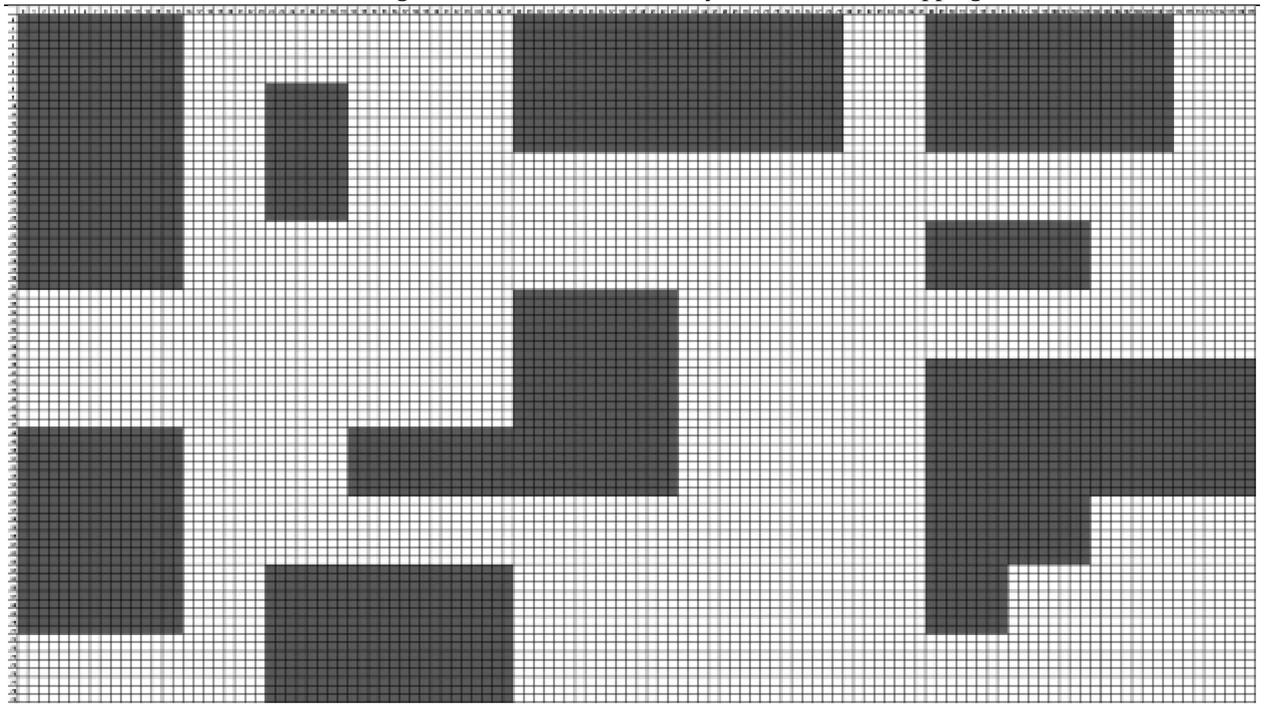


Fig. B.9. Map-17 scaled_x8

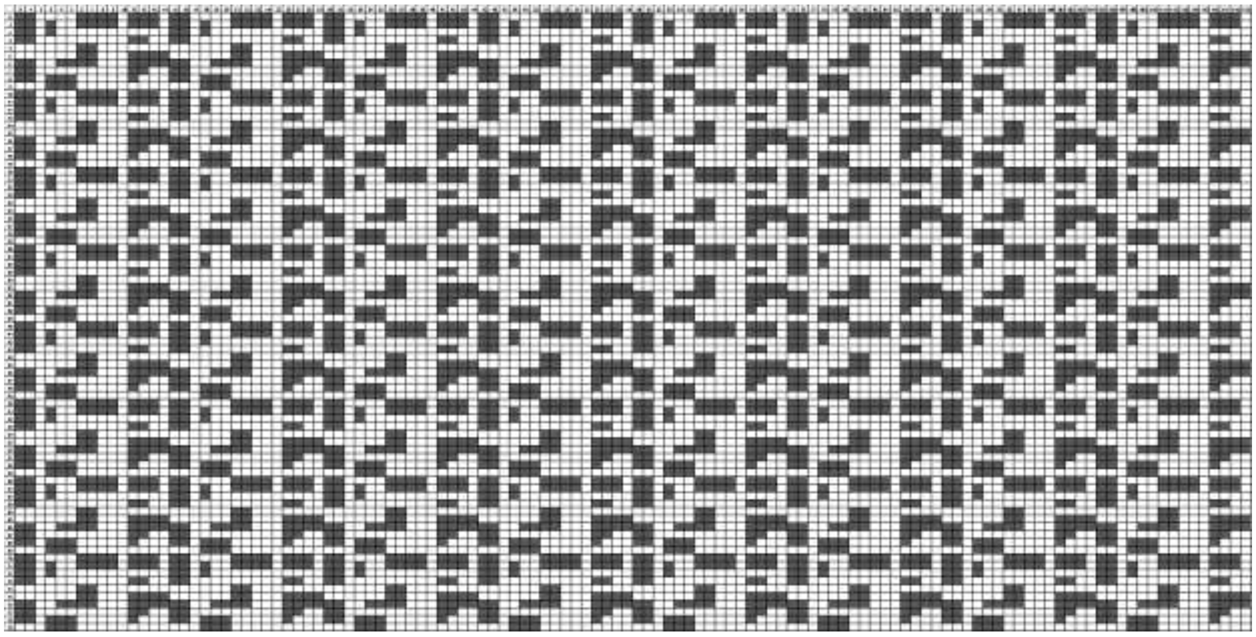


Fig. B.10. Map-17 repeated_x8

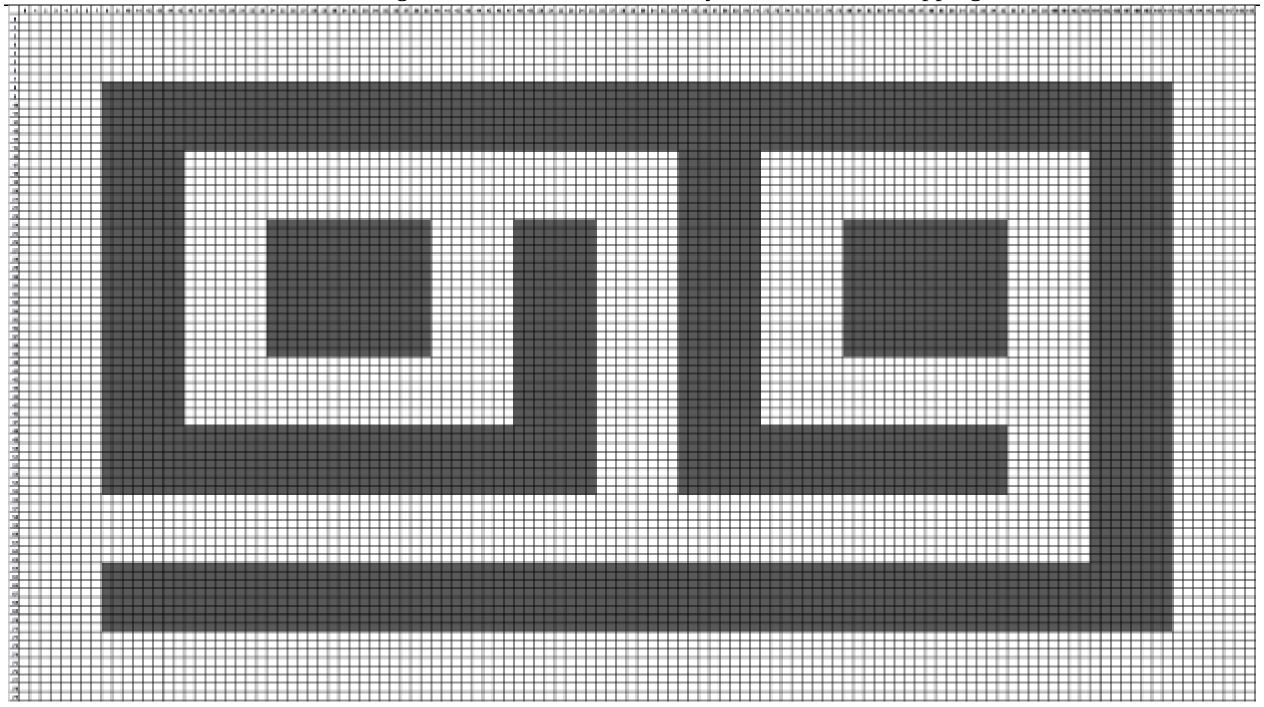


Fig. B.11. Map-22 scaled_x8

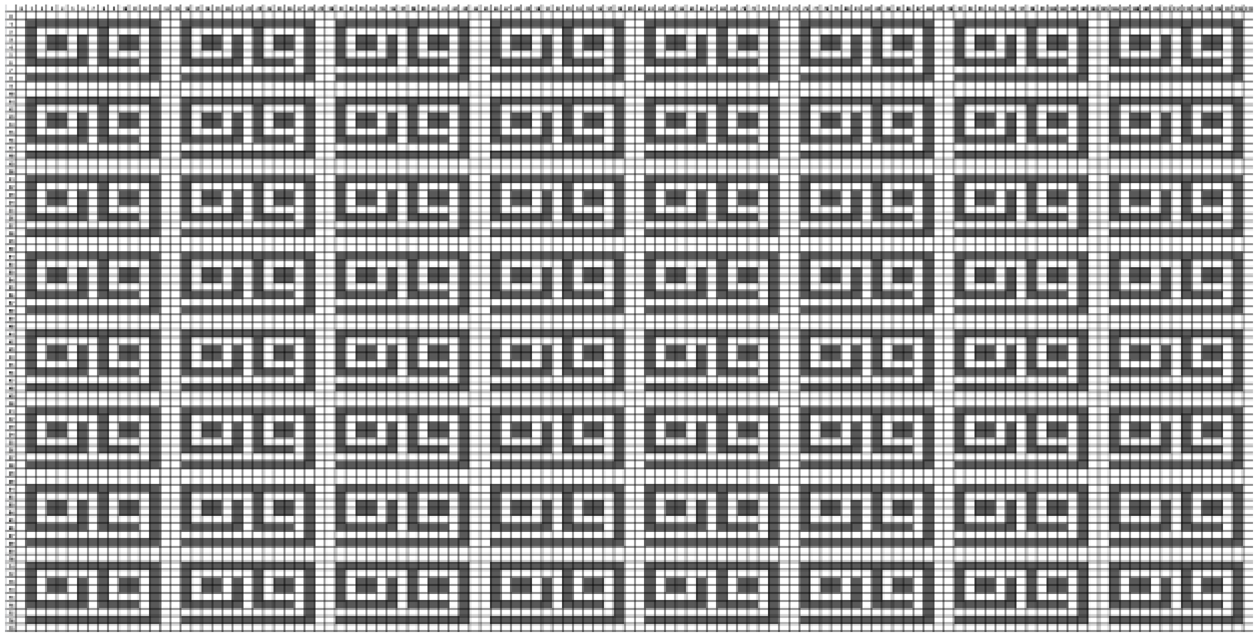


Fig. B.12. Map-22 repeated_x8