POLITECNICO DI TORINO

MASTER THESIS

Network-based control of UAVs in presence of packet dropouts and delays

Author:
Xiaoying CHEN

Supervisor:
Prof. Alessandro RIZZO
PhD Carlos
Perez-MONTENEGRO

Automatica Research Group
DET - Department of Electronics and Telecommunications

July 14, 2020

Acknowledgements

To my parents and friends, thank you for your support and help during this difficult time.

Contents

Chapter 1

Introduction

1.1 Project context

The use of Unmanned Aerial Systems (UAS)s in urban environments is booming. The UAS has shown that they are useful for urban missions as inspection or delivery. To fulfill these missions several studies are carried out, for example, safety studies, path planning, coordination of vehicle fleets and automatic control, and these are current issues in the scientific community.

Due to this, there is a necessity to have a system that interacts with the user and automatically manages the coordination of the tasks required for the mission, guaranteeing the safety of the population and automating the tasks. This allows to extend the execution of urban missions to multiple zones at low cost and with high reliability. Additionally, in the near future, multiple urban missions can be executed simultaneously. Therefore, the management system also has to comply with the emerging standards of air traffic management, in particular the Unmanned Aerial System Traffic Management (UTM).

Unmanned Aerial Vehicles (UAV) have been widely used nowadays as platforms to work in various environments and applications. Their use is rapidly expanding to commercial, scientific, agricultural, and other applications such as urban surveillance, smart delivery and inspections in structures and natural environments.

Many UAVs can be remotely controlled via network. In order to perform various tasks and services, UAVs must be able to communicate with networking backbones and infrastructures as well as with each other. There are two communication architectures: UAV-to-UAV (U2U) and UAV-to-infrastructure (U2I)[11]. Application for the latter includes receiving control from the operator and sending image or sensor data and positioning to the ground.

The developed Human-Machine Interface (HMI) allows any users to easily configure and execute a delivery mission with multiple UAVs, the user is not required

of advanced knowledge of the vehicle's algorithms or virtual reality to set the missions. As a last issue, the system is hierarchical and therefore compatible with future UTM higher level systems.

When a goal is defined, path planning is the crucial element of the whole system. It's very important to ensure the proper behaviour of the UAV to prevent harm to population and other infrastructures. However, the wireless network is not always reliable, interference may happen due to large physical obstacles or other devices that emit an electro-magnetic signal. Safety measures must be carried out when the communication of remote control is interrupted or lost.

The objective of this project is to develop a control system in a cloud-based architecture that offers path-planning level control to the UAV, and also realize the communication between the cloud and the UAV as well as dealing with the network failures.

1.2 Proposed architecture

In the near future, UAS will maneuver in urban environments autonomously. To be more specific, multiple applications in so-called smart cities ([14]) and autonomous strategies that optimize and coordinate vehicles in cooperative missions are highly desirable. To achieve this objective, the use of cloud-based architectures ([7]) will be useful.

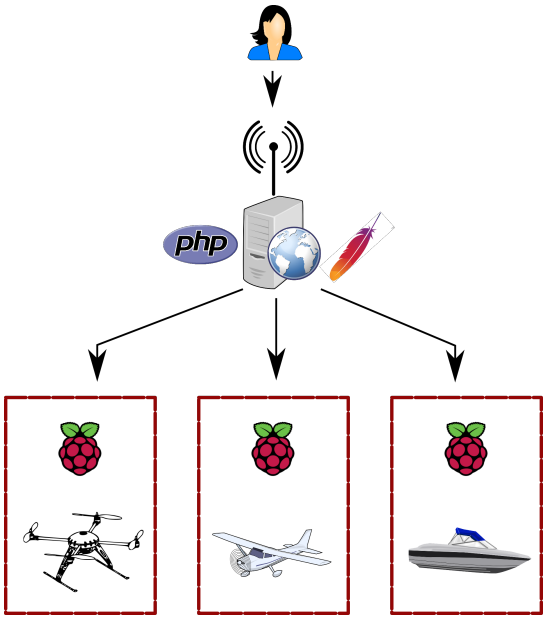


FIGURE 1.1: General Architecture

To solve these problems a cloud-based architecture like the one shown in Figure 1.1 is treated here.

This architecture has the following advantages

- Control data can be stored in a database,
  - allowing the engineer to obtain the parameters of the model of each vehicle and adjust the control strategies.
  - allowing the engineer to refine the models through identification methods.
- The control methodology can be modified easily without additional maneuvers such as retrieving unmanned vehicles.
- Control strategies can include high computational load, especially for those with online optimization process such as Model Predictive Control (MPC).
- Control methodologies can be adapted to different types of vehicles.

Here, two methodologies based on the general architecture in Figure 1.1 is presented, one in which the control unit is on-board and is updated or modified from the server and another in which the control unit is completely remote and it is located in the server.

Given the difficulties of using real vehicles in urban environments due to national and international standards, a Cloud-in-the-Loop (CIL) strategy is implemented for the purpose of validation of the two methodologies, where the server and the UAV are simulated.

The document is divided as follows, Section 1.2.1 illustrates the methodologies and Section 1.2.2 explains how to validate the methodologies through the CIL strategy and shows some results.

1.2.1 Cloud-based proposed strategies

Here two strategies to take advantage of an architecture in the cloud are exposed.

- **Exchangeable control unit**

  If vehicle information is in the server the engineer can test the laws of control in the updated model, develop simulations and load the control unit directly in the vehicle and the control unit suitable for the operating conditions could be selected.

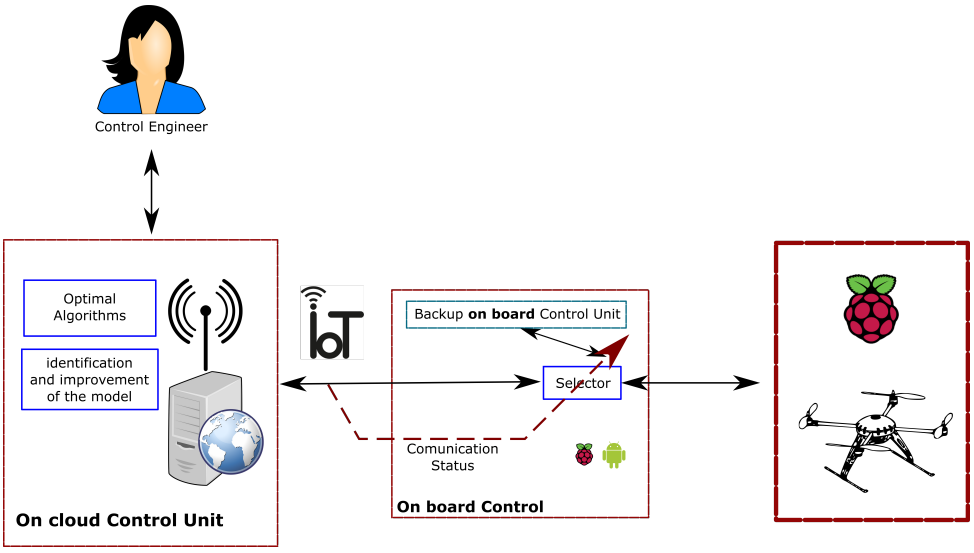


FIGURE 1.2: Multi slot control unit

Figure 1.2 shows the the exchangeable control unit strategy which allows multiple control units to be loaded into the vehicle and through a selector, which is controlled by the responsible engineer, the control unit to be used is determined.

The advantages of this methodology are:

- It allows to have several control units on the vehicle and these can change according to the mission guidelines and conditions of the environment (time of day, weather, among others)
- control unit is always on the vehicle and only communicates with the cloud to update its position and transfer sensor information. Therefore it is robust to problems in the communication network.

- **On-cloud control unit**

  Figure 1.3 shows a strategy where the control unit runs completely on the server and on the device there is only one backup control unit in case of failure in the communication network.

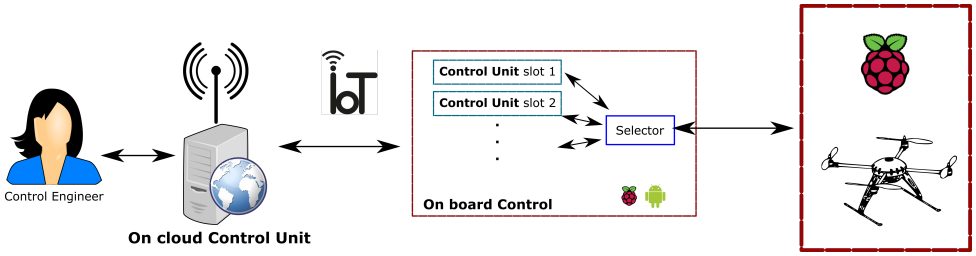


FIGURE 1.3: On cloud control

The advantages of this methodology are:

- It allows to execute computationally heavy algorithms like those of predictive model, this does not depend on the vehicle and therefore optimal processes for each vehicle can be considered.
- the engineer can modify the control law with greater flexibility.

This on-cloud architecture is adopted in this project. As a foundation of the on-cloud control, the network-based communication between the cloud and the UAV is implemented, both sides can detect the network connection failure. The on-cloud control unit and the on-board backup control unit on the UAV are also realized in this project.

The on-cloud control unit is a path planner which takes the initial position of the UAV and a goal defined by the user to generate a sequence of way points. Particularly, a sending buffer which can store 10 way points is implemented in the cloud. Each time after the cloud receives the position of UAV, the sending buffer is updated with the next way point for the UAV to reach together with the following 9 way points in path, and then sent to the UAV via network. The cloud then waits to receive the UAV's new position, and repeat the above procedure.

The on-board controller of the UAV receives a sequence of way points from the cloud and stores them in the receiving buffer, takes the first way point as its next position to reach, and send the current position of the UAV when the next position is reached. When the network is normal, only the first way point in the receiving buffer is used by the UAV.

When a network failure occurs, the on-cloud controller would set a timeout of 18s, and the controller on the UAV would follow the next way point in the receiving buffer instead of taking out the first one. Each time the next position is reached, it attempts to send its position to the cloud to test if the network has returned to normal.

If the network connection resumes in a short time, the cloud can detect the resumption before the timeout and get the number of way points passed by the UAV during the communication interrupt. Those way points will be skipped before loading the new sequence to the sending buffer. The UAV also stops following the path in the receiving buffer but only takes the first way point as long as the receiving buffer can be normally updated.

But if the communication interrupt lasts too long, when the UAV has almost reached the end of the sequence in the receiving buffer, say the 8th way point, it is not likely to keep up with the original path. At this time a local planner is launched to generate a path from the last way point in the receiving buffer to a nearest safe point.

The UAV then updates its destination and follows the local path. In the meanwhile the cloud terminates the connection with the UAV after the timeout.

1.2.2 Cloud in the Loop

Even with the technological advances available, the development of missions in urban environment are limited to the national regulations of each country. Tests with real vehicles require multiple approvals. To increase the speed of development and validation of the methodologies explained in 1.2.1, a CIL strategy is implemented.

In the CIL strategy the vehicle is simulated in the Linux OS as a client, and the cloud control strategy is implemented as the server in the Mac OS. The scenarios of network failure are also simulated to test if the backup control unit on the vehicle functions as expected and if the control works correctly after the network connection resumes.

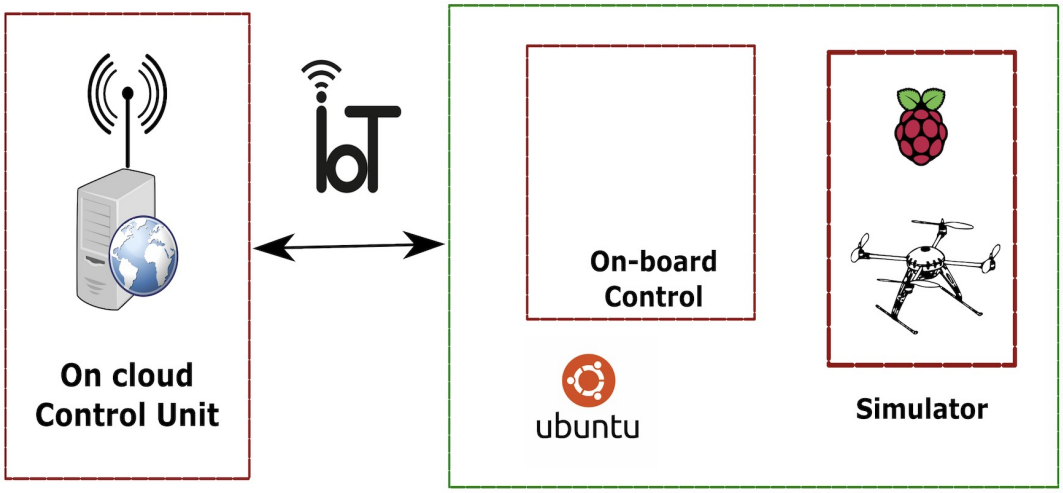


FIGURE 1.4: Cloud in the Loop

1.3 Project outline

This document is organised as follows:

- Chapter 1 describes the context of the project and its objective.
- Chapter 2 is all about path planning. First of all a research on the existing map representation approaches and path planning algorithms classified by the application scenarios are introduced, then it explains briefly the Dijkstra's algorithm that leads to the A* search algorithm. The A* algorithm is explained step by step using a grid map example, and its implementation in C code is illustrated in the end.

- Chapter 3 focuses on the communication aspect of the network-based server-client model proposed in Figure 1.4. In the first three sections the network model and two transport layer protocols: TCP and UDP are introduced as well as the concept of socket. The On-cloud control unit architecture in Figure 1.3 is selected to be implemented in this project with TCP as the communication protocol. In the last section it presents the implementation details of the control and communication between the server and the simulated vehicle using TCP socket in C code.

- Chapter 4 shows the test results of the project in four different scenarios: the communication is normal, short communication interrupt, long communication interrupt, and at last the case when the server socket is closed unexpectedly.

- Chapter 5 visualizes and extends the test results as the vehicle paths on maps of different dimensions and obstacle distributions. The performance of A* algorithm according to the test results is also discussed in this chapter.

- Chapter 6 draws a conclusion of the project and proposes the future work that needed to be done to complete the initiated work and improve the performance.

Chapter 2

Path Planning Algorithms

2.1 Background and Overview

A fundamental aspect of autonomous vehicle guidance is finding a path on the map combined with environment information in order that the vehicle can move from a source position to a destination position. Path planning is the determination of a sequence of way points in a given space that the vehicle should pass through[32]. The result of a successful path planning is that the vehicle could reach the goal in an optimal or sub-optimal path which conforms to it mechanical constraints while avoiding collision with either fixed or moving obstacles.

Path planning may be either local or global[2]. Local path planning is performed while the vehicle is moving, taking data from local sensors. In this case, the planner is able to generate a new path in response to the changes of the environment. Global path planning can be performed only if the environment is static and perfectly known to the planner. In this case, the path planning algorithm produces a complete path from the start point to the goal point before the vehicle starts its motion.

This chapter mainly focuses on the global path planning. A brief introduction of the existing map representation approaches and path planning algorithms is presented, and my implementation of A* search algorithm in C code is elaborated in the last section of the chapter. The vehicle is modeled as a point vehicle throughout this chapter for simplicity.

2.2 Stages of motion planning

Figure 2.1 shows an example of different stages to solve a motion planning problem in a decomposition approach[9]. In the first three stages the map is constructed and represented as a grid map. Then the graph search algorithm can be applied to the

map. There are various algorithms to choose from depending on the type of the vehicle and occasions of application. The algorithm produces a jagged path on the map. In the next stage the smoothing constraints are applied to form a trajectory which meets the dynamics of the vehicle and conforms to the path. At last, a control loop is added to guarantee that the vehicle always follows the trajectory.

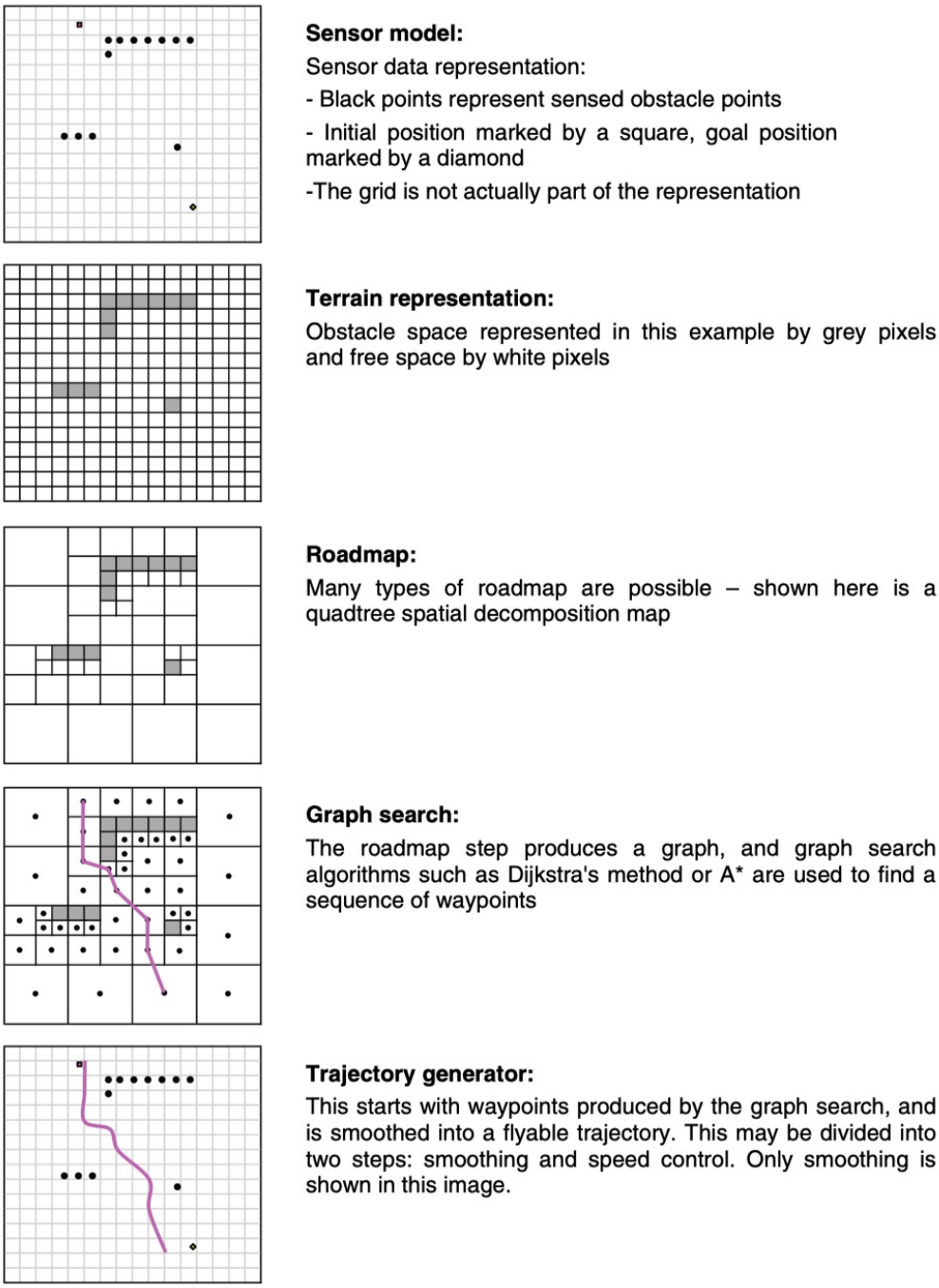


FIGURE 2.1: Stages in a typical example multi-level decoupled type planner taken from [9]

2.3 Map representation approaches

To begin with, the map should be processed to be converted to a certain form that is readable by the algorithm. The map representation can make a huge difference in the performance and path quality.

There exists many map representation approaches to be applied in different scenarios. Figure 2.2 shows three types of approaches that are commonly used in the path finding problems.

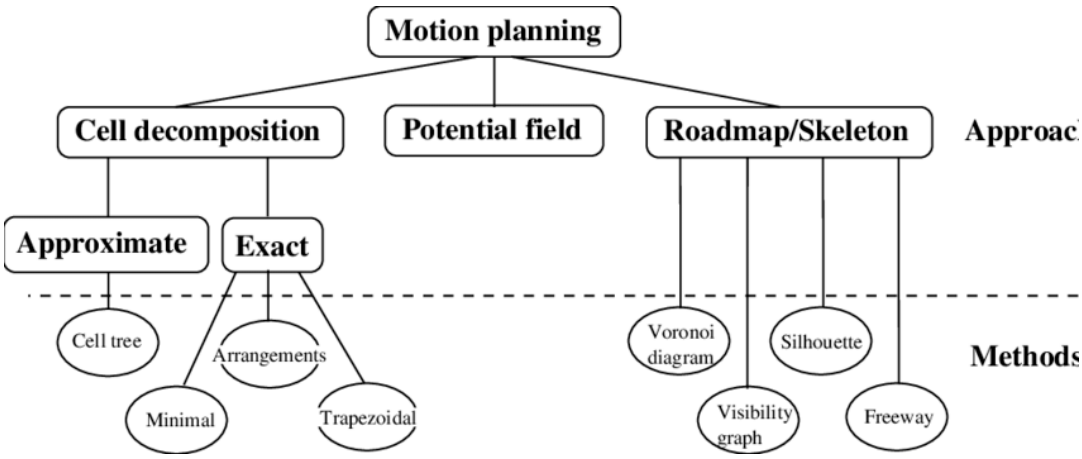


FIGURE 2.2: Motion planning methods taken from [13]

2.3.1 Roadmap Approach

Roadmap methods are based on the idea of capturing the connectivity of the free space in the form of a network of one-dimensional curves.[21] The roadmap is a set of one-dimensional curves in the free space, each of which connect two nodes of different polygonal obstacles. All the curves connect a vertex of one obstacle to a vertex of another obstacle without entering the interior of any polygonal obstacles.

If a continuous path can be found in the free space of the roadmap, the final solution is achieved by connecting the initial and goal points to this path. If more than one continuous path can be found and the number of nodes in the graph is relatively small, Dijkstra's algorithm (illustrated in Section 2.5) is often used to find the best path.

There are various types of roadmaps, including the visibility graph, the Voronoi diagram, the freeway net, and the silhouette. In this section the first two are illustrated.

- **Visibility graph**

As the name suggests, this approach builds a roadmap of lines connecting each vertex with all obstacle vertices visible from its position. An example of visibility graph is shown in Figure 2.3. The knowledge behind this approach is that the shortest path grazes polygonal obstacles at their vertices.

This is an optimal solution that applies to the point vehicle problem in only two dimensions. Since the curves are obtained by connecting the vertices of obstacles, the path may come arbitrarily close to the obstacles. As a result, this approach offers no safety buffer to prevent collisions in the case of systems with uncertainty in their position. Researchers have proposed a way to solve this problem by expanding the obstacle space by a ball larger than the vehicle's longest radius.[9]

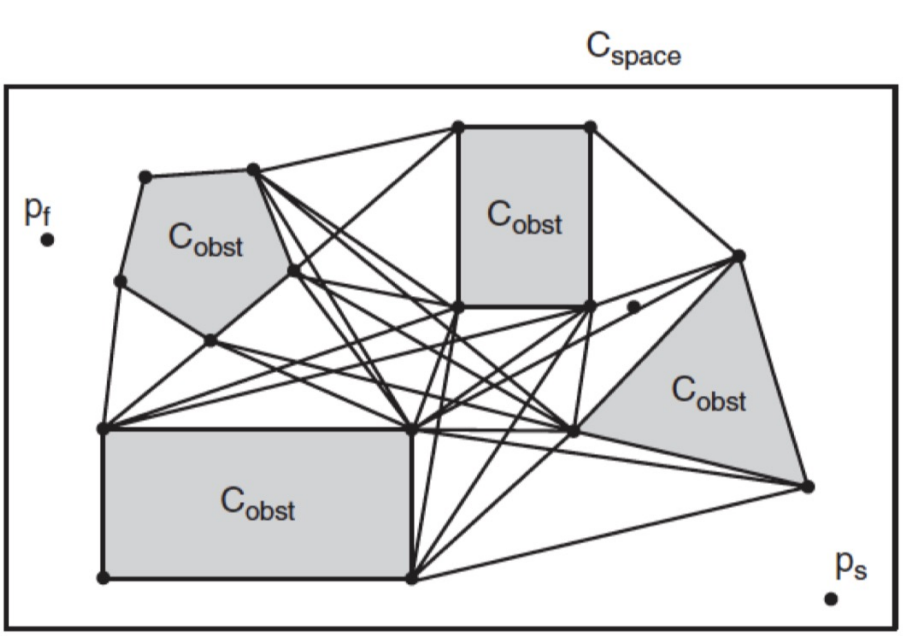
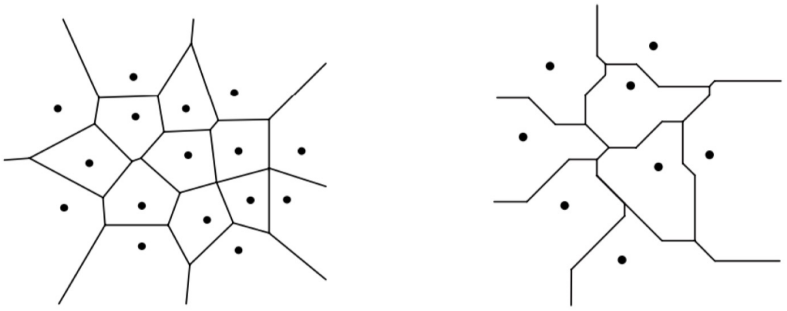


FIGURE 2.3: Visibility graph taken from [33]

- **Voronoi Diagram**

  The Voronoi diagram sections off a space into regions closest to a particular point, each region is called a site. For each site there is a corresponding region consisting of all points of the plane closer to that site than to any other.[4]. Contrary to the roadmap approach, it builds a skeleton that is maximally distant from the obstacles, and finds the shortest path that follows this skeleton[9]. An example of Voronoi diagram is shown in Figure 2.4. Voronoi diagrams can be identified into two classes based on the metrics to compute the distance between two points: Euclidean and Rectilinear.



(a) Euclidean **(b)** Rectilinear

FIGURE 2.4: Voronoi diagram taken from [21]

2.3.2 Exact Cell Decomposition

This approach decompose the free space into trapezoids, which are then connected by a graph and searched using a graph search. An example is shown in Figure 2.5.

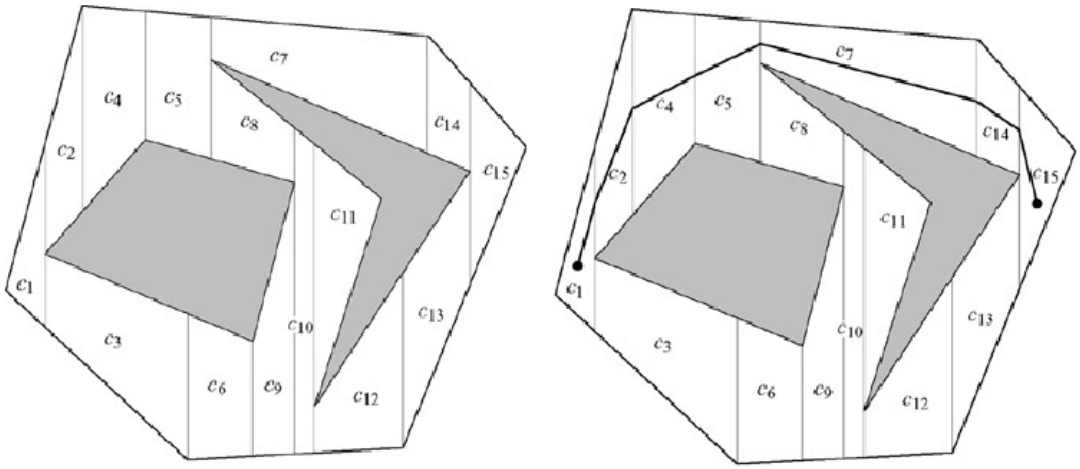


FIGURE 2.5: An example of path planning using Cell decomposition method taken from [22]

2.3.3 Approximate Cell Decomposition

- **Rectanguloid Cell Decomposition**

  This divides the entire configuration space into rectangular regions of the same size, and labels each rectangle as being completely filled (blocked), partially filled (risky), or completely empty (free). The resolution depends on how many rectangles are used to represent the configuration.

  The most common example is that of the A* or D* search over a square or cubic grid of occupied or unoccupied cells[5].

- **Quadtree Decomposition**

Quadtree decomposition uses cells of variable size to represent the environment. The cells are successively divided into four children cells recursively until a cell is located in a completely occupied or completely free zone or until a threshold resolution is achieved[26]. This approach reduces the number of points needed to represent obstacles as compared to a full grid representation. An example is shown in Figure 2.6.

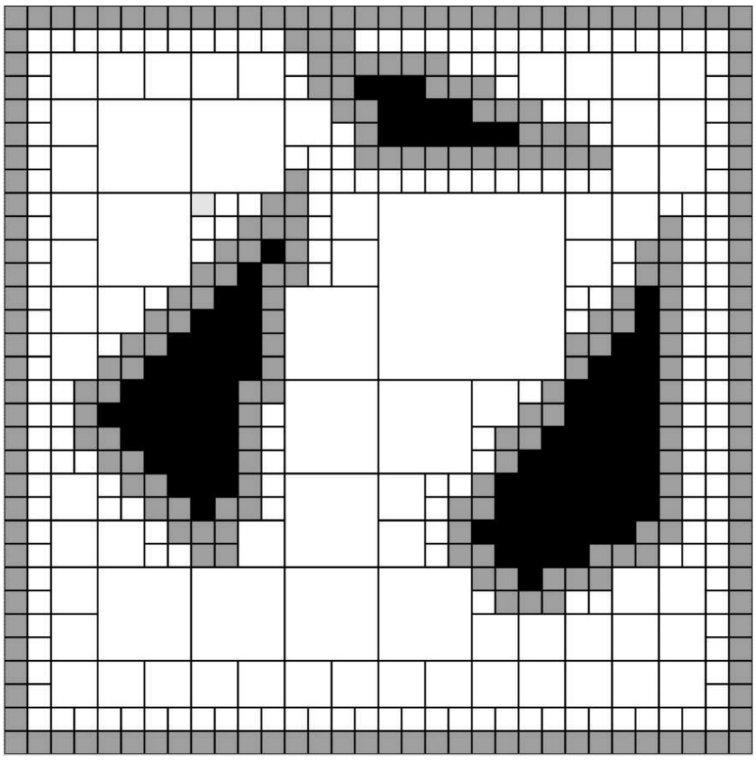


FIGURE 2.6: Quadtree decomposition of a 2D environment taken from [3]

2.3.4 Potential field

This method is based on the idea of assigning a potential function to the free space, and simulating the vehicle as a particle reacting to forces due to the potential field. The goal point has the lowest potential, and attracts the vehicle, while obstacles repel the vehicle. An example is shown in Figure 2.7.

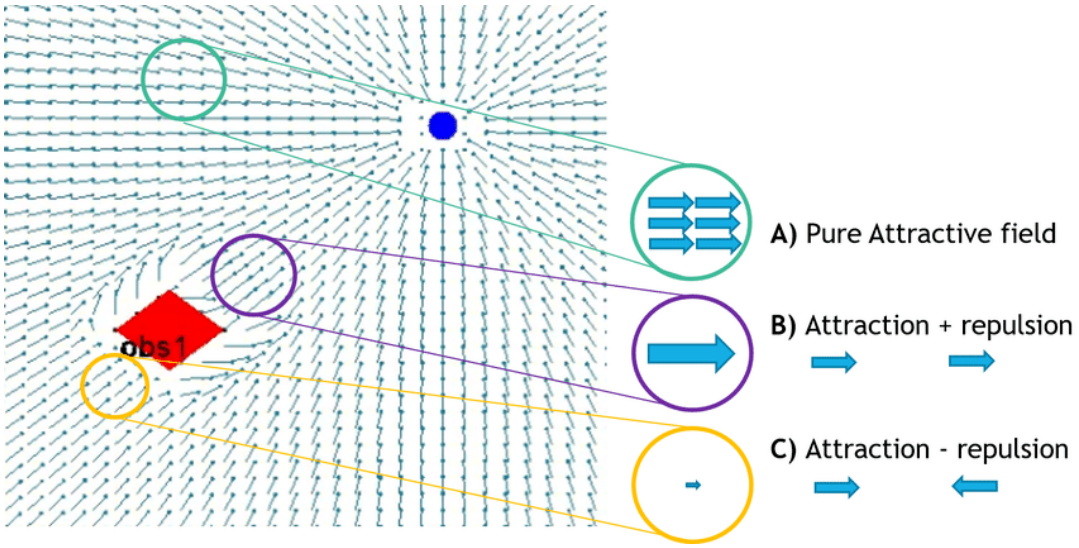


FIGURE 2.7: Potential field method example taken from [6]

2.3.5 Probabilistic Approach

- **Probabilistic Roadmap (PRM)**

  The basic idea of this approach is to take random samples from the configuration space, testing whether they are in the free space, and a local planner is used to connect these free configurations to other nearby free configurations to form a graph with starting and goal configurations added in. A graph search algorithm is then applied to the resulting graph to determine a path between the starting and goal configurations[25].

  This approach is generally used in manipulator problems, where the configuration space is typically of high dimension and complex[24]. However, due to its random nature, the rate of convergence is slow and the paths produced are not optimal. Besides, significant work is required to be done to transform the resultant paths to smooth trajectories in the case of two- and three-dimensional configuration space planning. The search efficiency is considerably low for long passageways.

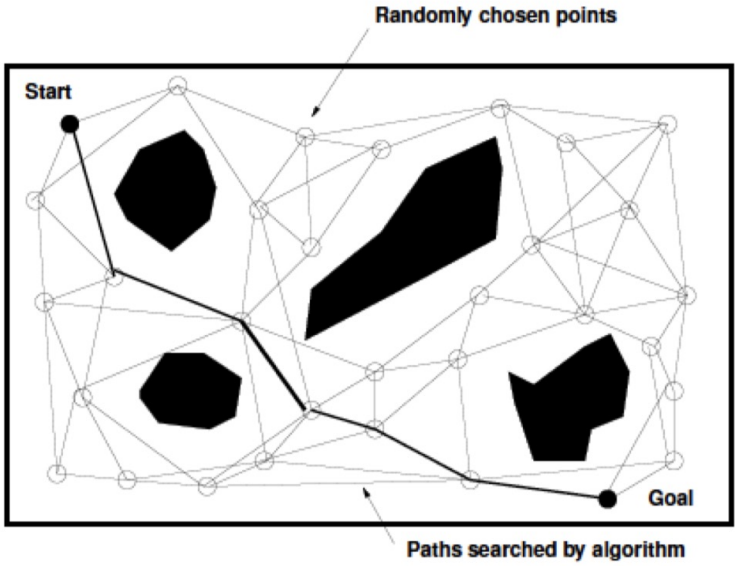


FIGURE 2.8: An example of PRM taken from [8]

- **Rapidly Exploring Random Tree (RRT)**

  Rapidly-exploring Random Tree (RRT) is opposite to the PRM approach, it is constructed in an iterative way that quickly expands to a randomly-chosen point in the configuration space. RRT is particularly suited for path planning problems that involve obstacles and non-holonomic constraints in non-convex high-dimensional spaces[34]. An example of RRT is shown in Figure 2.9.

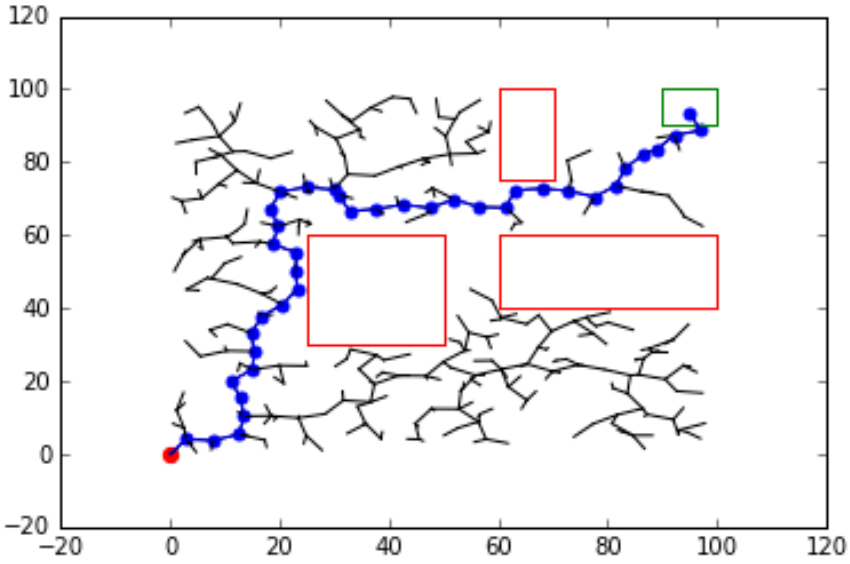


FIGURE 2.9: An example of RRT taken from [1]. Red point is the start point and the point in green block is the goal, the red blocks represent the obstacles

RRT expansion can be biased by increasing the probability of sampling states

in a specific zone. The search is then turned to a guided one with a higher probability of the states that lead the search towards the goal. This limits the expanding of RRT to the undesired directions and make it faster to reach the goal.

2.4 Search Algorithms

Once the environment is interpreted by applying one of the methods illustrated in the last section, a graph search algorithm should be chosen based on the specifications of the task in order to obtain the optimal path.

There are three main criteria for selecting the algorithm:

1. Completeness: the algorithm is always able to find a solution when it exits.

2. Optimality: it is guaranteed to obtain the path of the least cost.

3. Time/Space complexity: the running time of the algorithm to get a solution and the memory space occupied during its execution.

This section introduces several search methods that are widely used for path finding.

2.4.1 Uninformed Search Methods

- **Breadth-First Search (BFS)**

  BFS solves Single Source Shortest Path problem, i.e., it finds the shortest path between a source vertex and any other vertex in the graph. It starts at the tree root or some arbitrary node of a graph, and explores all of the neighbour nodes at the present depth before moving on to the first node at the next depth level. The algorithm is illustrated in Figure 2.10 and Figure 2.11 shows how the algorithm works.

```
1  procedure BFS(G, root) is
2      let Q be a queue
3      label root as discovered
4      Q.enqueue(root)
5      while Q is not empty do
6          v := Q.dequeue()
7          if v is the goal then
8              return v
9          for all edges from v to w in G.adjacentEdges(v) do
10             if w is not labeled as discovered then
11                 label w as discovered
12                 w.parent := v
13                 Q.enqueue(w)
```

FIGURE 2.10: BFS algorithm

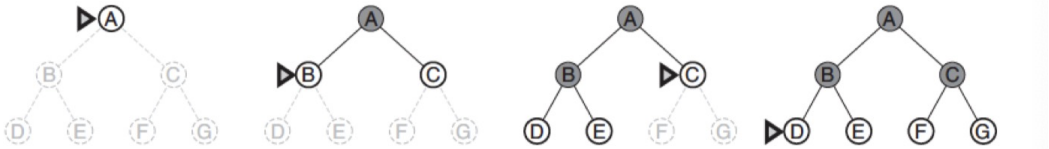


FIGURE 2.11: BFS on a binary tree taken from [27] p82

- **Depth-First Search (DFS)**

  DFS algorithm starts at the root node and explores along each branch as deep as possible until a leaf node is reached, then backtracking. Figure 2.12 shows how the algorithm works.

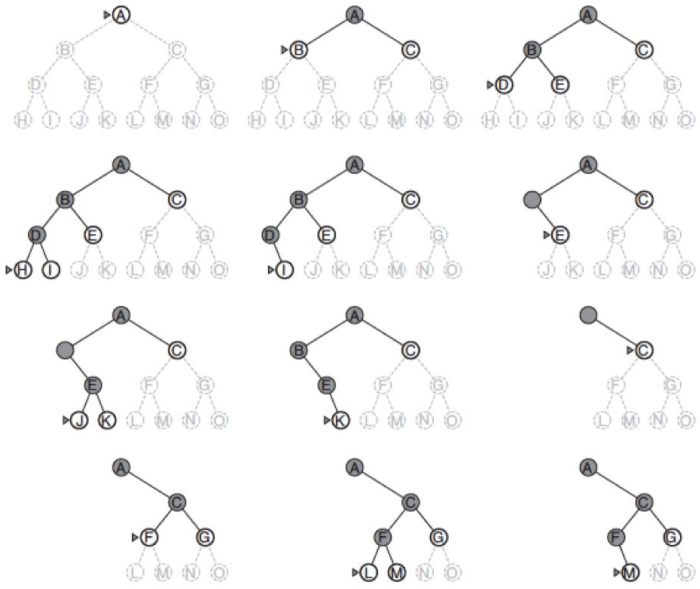


FIGURE 2.12: DFS on a binary tree taken from [27] p86

- BFS is complete on a finite graph, it is guaranteed to find a goal state if one exists. But DFS may get stuck in parts of the graph that have no goal state.

- BFS is optimal if all path costs are the same, because it always finds the shallowest node first. DFS is not optimal in nature.

- The time complexity of BFS is O(| V | + | E |), since in the worst case every vertex and every edge will be explored. Space complexity is the same as time complexity because every node has to be stored in memory. DFS can be better than BFS for dense solution space, and the space complexity is linear.

Uninformed search methods explore the search space in an ordered manner until the goal is reached. As a result, uninformed search methods are "blind" and the search orientation may often lead away from the goal. Even for some small problems the search can take unacceptable amounts of time and/or space.[10]

2.4.2 Informed Search Methods

Informed search gives the nodes that are likely to lead to a good solution a higher priority to be expanded. In this way the amount of search can be significantly reduced compared to the uninformed search especially for large search space. Heuristics is introduced in this method as a criterion to guide the direction of search. Heuristic function estimates how close a state is to the goal. However, the heuristic method might not always yield a solution of the least cost depending on the choice of the heuristic function, but it guarantees to find a good solution in acceptable time.

- **Best-first Search Algorithm (Greedy Search)**

  Greedy best-first search algorithm can be seen as the combination of DFS and BFS algorithms guided by a heuristic function. At each step, the node which estimated by the heuristic function to have the smallest cost to move to the goal is selected to be expanded.

  However, this algorithm also inherits the drawbacks of DFS and BFS, it proves to be both incomplete and non-optimal. The time and space complexity highly depends on the choice of the heuristic function.

- **A* Search Algorithm**

  This algorithm is illustrated in detail in Section 2.6

2.4.3 Search Methods in Partially Known Environments

The algorithms mentioned above assumes that the environment is completely known a priori and remains unchanged during the operation of the vehicle. In reality, unfortunately, this information is usually partially known because of the limited resolution, or the information has changed since it was measured. To achieve a path under this circumstance, a common solution is to generate a "global" path using the known information, and then make replans to "locally" circumvent obstacles on the route detected by the sensors on the vehicle. The replanning methods need to be fast enough so that the vehicle is able to get its next position in time while it is moving, sometimes the completeness and optimality may yield to timing.

Lifelong Planning A* (LPA*) algorithm and D* algorithm can be used for path finding in partially known environments.

- **LPA* Search Algorithm**

  LPA* is an incremental version of A* which can dynamically adjust to the changes on the map. Its first search is almost the same as that of A*, except that it breaks ties among vertices with the same f-value in favor of vertices with smaller g-values. Every time the edge costs of a graph change or vertices are added or deleted, a replan is performed to update the shortest path from a given start vertex to the goal vertex. LPA* reuses the parts of previous search tree that are identical to the new one, so the subsequent searches are potentially faster than performing a replan from scratch[15].

  LPA* introduces the rhs-value which is one-step lookahead value based on the g-values and thus potentially better informed than the g-values. The rhs-value of any other cell is the minimum over all of its neighbors of the g-value of the neighbor and the cost of moving from the neighbor to the cell in question[15]. If rhs(n) equals g(n), then n is called locally consistent. When edge costs change, local consistency needs to be re-established only for those nodes which are relevant for the route.

- **D* Search Algorithm**

  D* algorithm is used to find a path from a starting state to a goal state in a dynamically changing configuration space. In another word, to find a path on a graph where the edge cost parameters can change during the problem solving process[28]. D* algorithm is particularly fitted to robot applications. The robot is able to detect environment changes only within a limited scope, i.e. most edge cost corrections occur in the vicinity of the robot, as a result the

path needs only to be replanned out to the robot's current state to achieve the real-time property[29].

D* introduces the concept of LOWER state and RAISE state, the information about path cost increases (e.g. due to an increased edge cost) can be propagated to its neighbors.

A variant of D* is the D* Lite algorithm, it has similar functionality to D* but simpler.

2.5 Dijkstra's Algorithm

Dijkstra's Algorithm is performed upon the weighted graph in which each edge is assigned a weight that indicates the cost to move from one vertex to the other. It solves the problem of finding the lowest-cost path from a source vertex to all other vertices in the weighted graph. That is, finding the shortest path between any two vertices. This algorithm and its many variants are widely used in road network problems, network routing protocols and in the field of artificial intelligence.

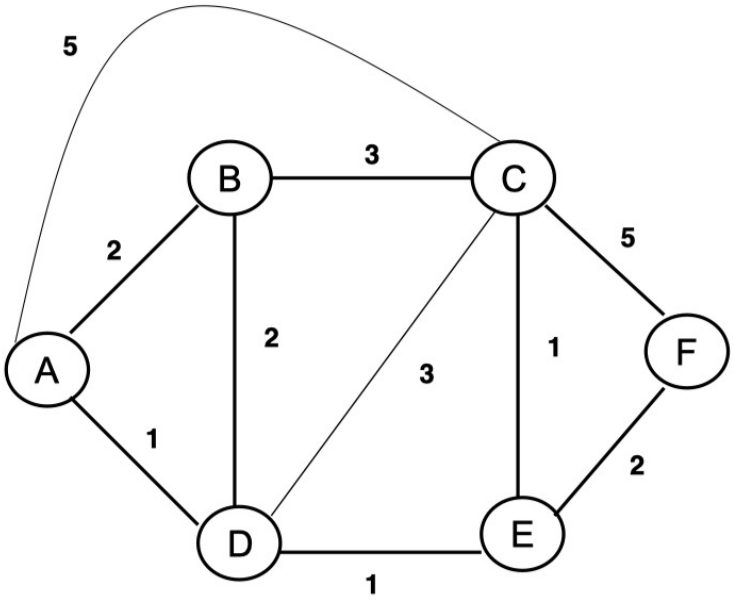


FIGURE 2.13: Weighted graph taken from [12]

Figure 2.13 shows a weighted graph labeled with the edge costs, vertex A is selected as the source vertex. The pseudo code of Dijkstra's algorithm is in Figure 2.14.

1. In the beginning two sets are created:

- Set Q, for all the unvisited vertices, initially contains all the vertices in the graph.

- Set S, for all the visited vertices, initially empty. The vertices in this set have already found their shortest path to the source and will not be examined as a neighbour.

2. Set the distance between the source vertex and A to zero, all the other distances between any two vertices to infinity.

3. Vertex A is the first to be examined because it has the minimal distance zero to the source while the others are initially set to infinity. So A is moved from set Q to set S. Then update the distance between A(the source vertex) and its neighbours B,C,D to 2,5,1 respectively, and record the last hop in the shortest path. In the first loop it is always the source vertex A. The distances of the other vertices in the graph from the source remain infinity.

4. In the next loop, the vertex D which has the minimal distance 1 from the source is taken from set Q to S, and all its neighbours which are not in the visited set S are examined(B,C,E). The currently shortest distances from the selected vertex D to its neighbours are

- updated if a shorter path is found: the distance between A and C is originally 5 for path A-C, but now a shorter path A-D-C of distance 1+3=4 is found; The distance from A to E is infinity, but now E can be reached via D and the distance is 1+1=2.

- otherwise remain the same as before: path A-D-B has a greater distance 3 than A-B of distance 2.

The last hop of the updated shortest distance is also updated: D for vertex C instead of A, and D for vertex E.

5. In the next loop vertex B is taken from set Q because it has the shortest distance 2 from the source. Its neighbours are examined as in the previous loops.

6. The loop continues until set Q becomes empty which means all the vertices in the graph have been visited. If a certain destination vertex is set, it is not necessary to wait until all the vertices in the graph have been visited, the algorithm stops once the destination vertex has the smallest distance from the source in set Q.

7. The shortest paths can be retrieved using the last hop information recorded in each loop. Figure 2.15 shows the shortest path marked in red from vertex A to C.

```
dist[s] ←o                          (distance to source vertex is zero)
for all v ∈ V–{s}
    do dist[v] ←∞                   (set all other distances to infinity)
S←∅                                 (S, the set of visited vertices is initially empty)
Q←V                                 (Q, the queue initially contains all vertices)
while Q ≠∅                          (while the queue is not empty)
do  u ← mindistance(Q,dist)         (select the element of Q with the min. distance)
    S←S∪{u}                         (add u to list of visited vertices)
    for all v ∈ neighbors[u]
        do if  dist[v] > dist[u] + w(u, v)      (if new shortest path found)
           then    d[v] ←d[u] + w(u, v)         (set new value of shortest path)
                                                (if desired, add traceback code)
return dist
```

FIGURE 2.14: Dijkstra's pseudo code

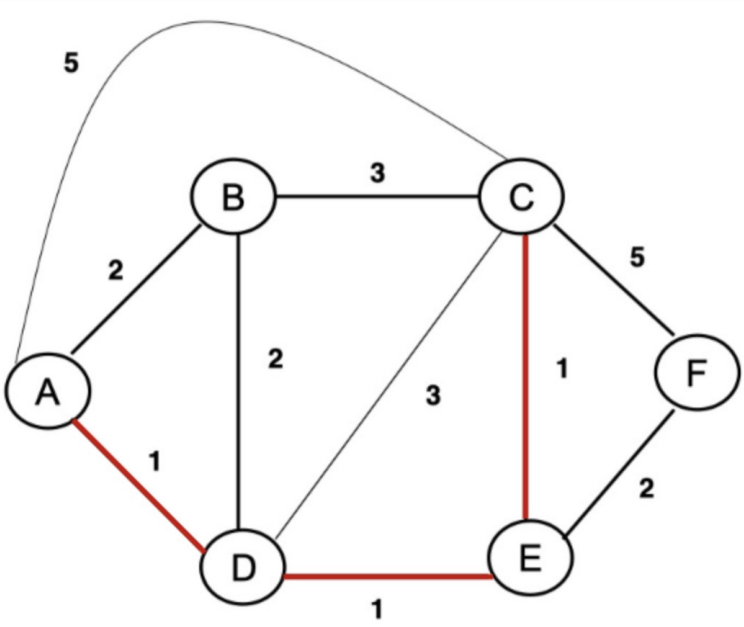


FIGURE 2.15: Shortest path between 2 vertices

2.6 A* Algorithm

When in a situation that requires to find an optimal path between a start node and a certain goal node on a map, Dijkstra's algorithm is non-directional and performs a lot of computations that are not going in the right direction. The nodes in the opposite direction of the goal are also expanded despite that they are not very possibly to be in the final path, expanding them is usually not very necessary. To solve this problem, a heuristic value is introduced. This heuristic indicates approximately how far it is to reach the goal for every node, and the node with the lowest estimated distance from the start to the goal gets the priority to be expanded. This extra information can be used to avoid visiting other unnecessary nodes and thus find the path faster.

This improved algorithm with a heuristic is known as A* Search Algorithm. It can be seen as an extension of Dijkstra's algorithm of which the direction of search is guided. Depending on the environment, A* might accomplish a search much faster than Dijkstra's. A* is applicable to the cases where the environment information is fully known and doesn't change during the operation of the vehicle. This section illustrates the A* algorithm on a grid map, but this algorithm also works with arbitrary graphs.

2.6.1 Grid map representation

The configuration space can be divided uniformly into small rectangular blocks. Those blocks are called "cells", and each cell is assigned a value to indicate if it is a free space or blocked. It is possible to choose cells, edges or vertices for movement. In this case the vehicle only moves to the center of a cell.

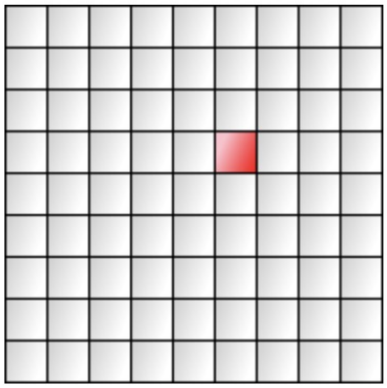


FIGURE 2.16: Grid representation of a map

2.6.2 The Heuristic function

The heuristic can be used to build a function f(n), that gives an estimation of the total path cost from the source cell to the goal, n represents the current cell on the path. We define it like this:

$$f(n) = g(n) + h(n)$$

where

- *f(n)* = total estimated path cost through node n.
- *g(n)* = actual path cost from the source cell to n.

- *h(n)* = estimated movement cost to move from n to the goal.

The h value can be calculated in three ways:

1) Manhattan Distance

It is the sum of absolute values of differences in the current cell's x and y coordinates and the goal's coordinates respectively, i.e.,

$$h = abs(current\_cell.x - goal.x) + abs(current\_cell.y - goal.y)$$

Figure 2.17 shows the Manhattan distance between the source cell (marked as a red ball) and the goal (marked as an orange star).

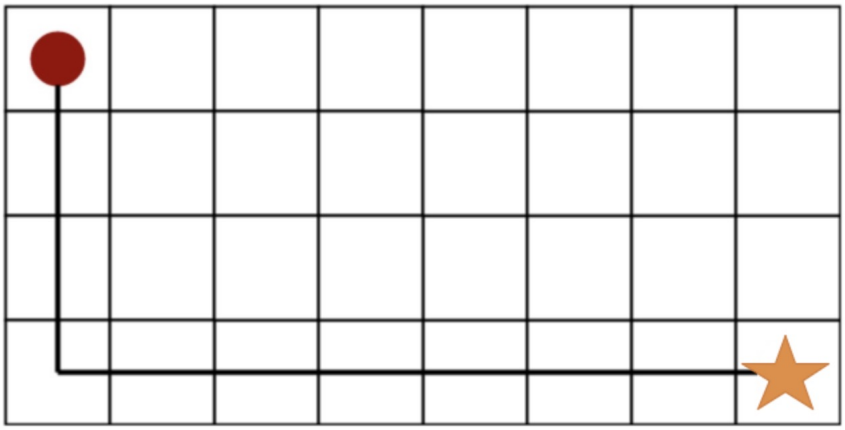


FIGURE 2.17: Manhattan distance

Manhattan distance is used when it is allowed to move in only four directions: up, down, left and right as in Figure 2.18.

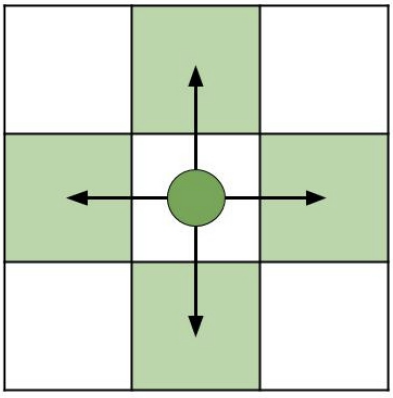


FIGURE 2.18: 4 directions move

2) Diagonal Distance

It is the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

$$h = max\{abs(current\_cell.x - goal.x), abs(current\_cell.y - goal.y)\}$$

Figure 2.19 shows the Diagonal distance between the source cell (marked as a red ball) and the goal (marked as an orange star).

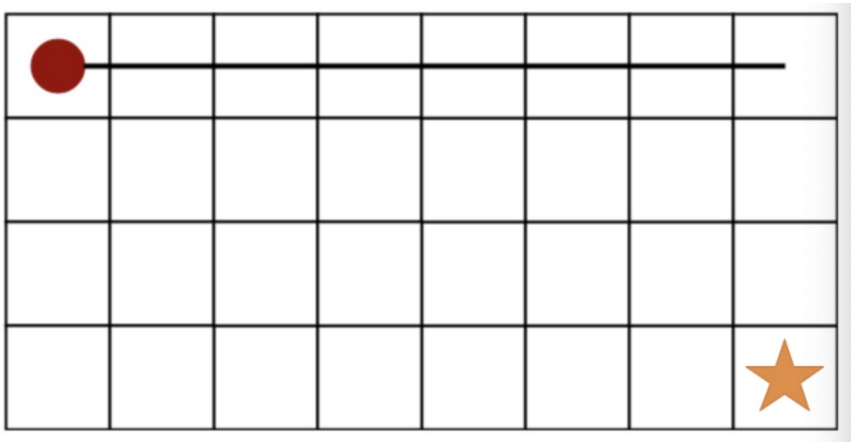


FIGURE 2.19: Diagonal distance

Diagonal distance is used when it is allowed to move in eight directions which is similar to a move of a King in Chess as in Figure 2.20.

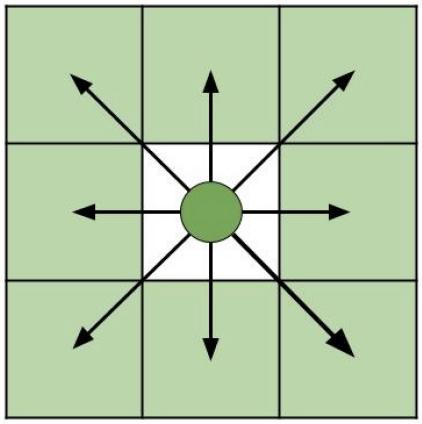


FIGURE 2.20: 8 directions move

3) Euclidean Distance

It is the straight line distance between the source and goal, i.e.,

$$h = sqrt((current\_cell.x - goal.x)^2 + (current\_cell.y - goal.y)^2)$$

This is used when it is allowed to move in any directions.

Figure 2.21 shows the Euclidean distance between the source cell (marked as a red ball) and the goal (marked as an orange star).

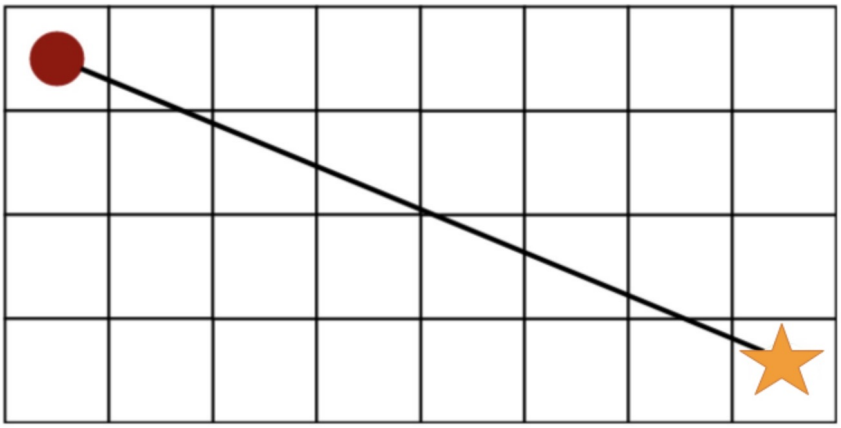


FIGURE 2.21: Euclidean distance

The heuristic function is problem-specific and has a significant impact on the performance of the algorithm. To guarantee that the path found is the shortest one, the heuristic function must be admissible that never overestimates the actual cost to get to the goal node, otherwise it could miss a path which is estimated to cost more but actually costs less than the "shortest" one found.

2.6.3 Algorithm Explained

Considering a 2D grid in Figure 2.22 having several obstacles, and we want to move from the source cell in the bottom left corner to the goal cell in the top left corner. An example is used in this section to illustrate how A* algorithm works.

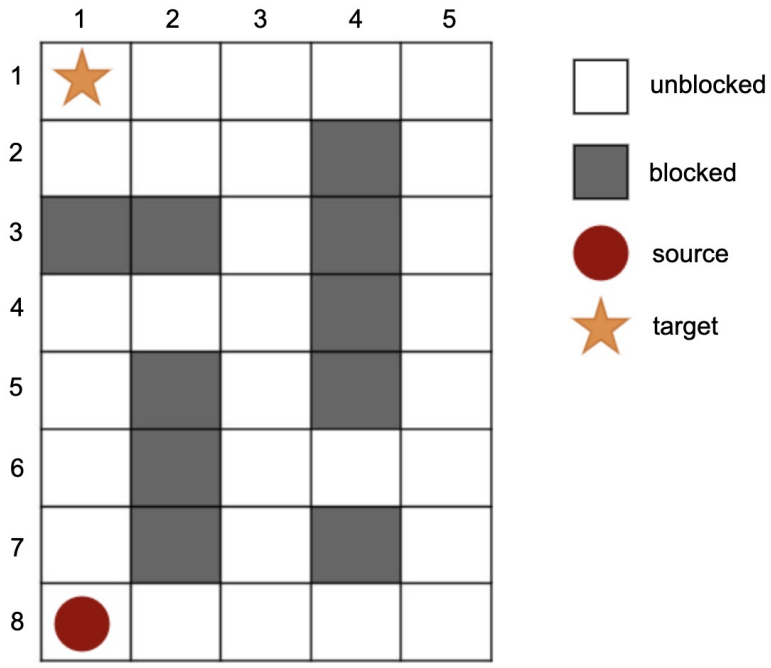


FIGURE 2.22: 2D map

First of all create two lists which are similar to Dijkstra's:

- **Open List** : stores the nodes that can be detected but not accessible yet.
- **Closed List** : stores the nodes that have been expanded.

The algorithm is described in Figure 2.23.

1. Initialize the OPEN list and the the CLOSED list.

2. Put the starting node on the OPEN list.

3. while the OPEN list is not empty

   a) find the node with the least *f* on the OPEN list, call it "q".

   b) pop q off the OPEN list.

   c) generate q's 4 or 8 successors and set their parents to q.

   d) for each successor

      i) if successor is the goal, stop search
      successor.*g* = q.*g* + distance between successor and q
      successor.*h* = distance from goal to successor
      successor.*f* = successor.*g* + successor.*h*

      ii) if a node with the same position as successor is in the OPEN list which has a lower *f* than successor, skip this successor. Otherwise update the node value *f* to the lower cost and update its parent to q.

      iii) if a node with the same position as successor is in the CLOSED list which has a lower *f* than successor, skip this successor. Otherwise, add the node to the OPEN list.

   end (for loop)

   e) push q on the closed list.

   end (while loop)

4. If the goal is not reached after the OPEN list becomes empty again, there is no path between the starting node and the goal.

FIGURE 2.23: A* algorithm

1) In the beginning, there is only the start node (8.1) on the OPEN list, and the path cost g is 0. Pop the node off and examine all its neighbours(assume it is allowed to move in 4 directions, the cost from the node to its reachable neighbours is always 1), two of them are out of the map, the other two are reachable with the path cost g=1. Using the Manhattan distance, the heuristics and f of node (7,1) and (8,2) are marked in Figure 2.24. Now the OPEN list and the CLOSED list are empty, set the parent of these two nodes to (8,1) and add them to the OPEN list. Add node (8,1) to the CLOSED list.

- OPEN list: { (7,1), (8,2) }

- CLOSED list: { (8,1) }

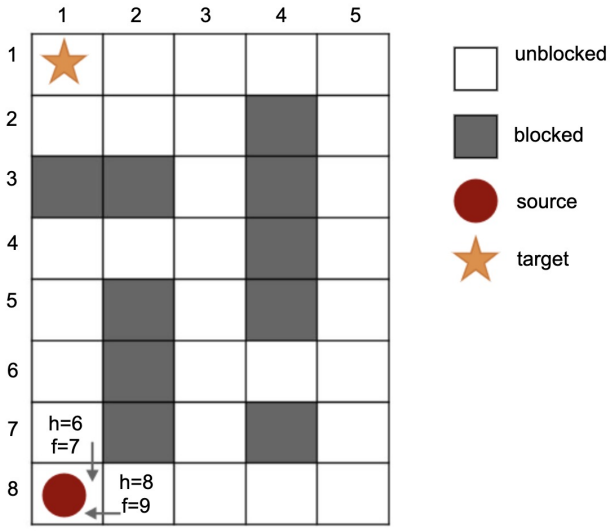


FIGURE 2.24: Step 1

2) Now there are two nodes on the OPEN list, pop (7,1) off the list because it has the least f. The node has only one successor (6,1) with g=2, update the node values as in in Figure 2.25 and add its successor (6,1) to OPEN list, add (7,1) to CLOSED list.

- OPEN list: { (8,2), (6,1) }

- CLOSED list: { (8,1), (7,1) }

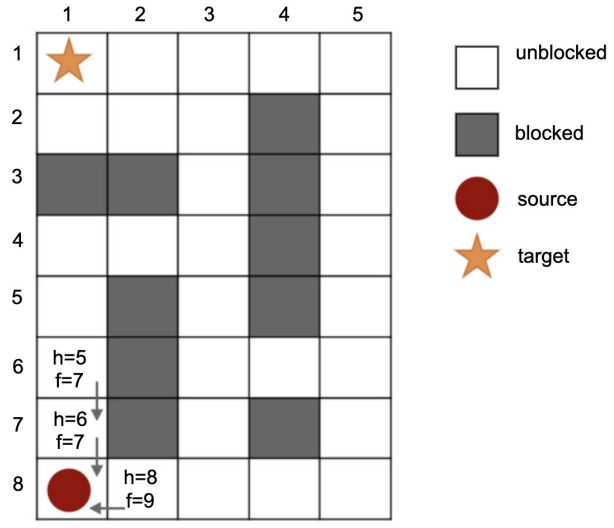


FIGURE 2.25: Step 2

3) Pop (6,1) off, it has two successors (5,1) and (7,1) as in in Figure 2.26. (7,1) is already in the CLOSED list, and in this grid representation of uniformed cost, it is not necessary to check if the successor has a lower f than the node of the same position in the CLOSED list, because once a node is added to the CLOSED list, it

always has the lowest f. This rule applies to all the following steps. Add (6,1) to the CLOSED list.

- OPEN list: { (8,2), (5,1) }

- CLOSED list: { (8,1), (7,1), (6,1) }

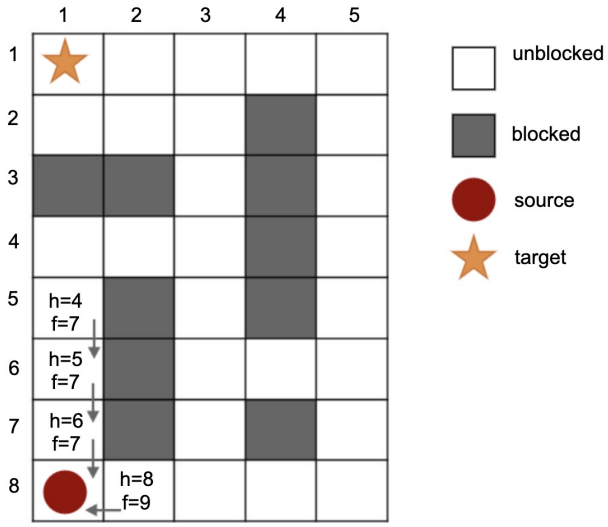


FIGURE 2.26: Step 3

4) Pop (5,1) off, and add its successor (4,1) to the OPEN list as in in Figure 2.27.

- OPEN list: { (8,2), (4,1) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1) }

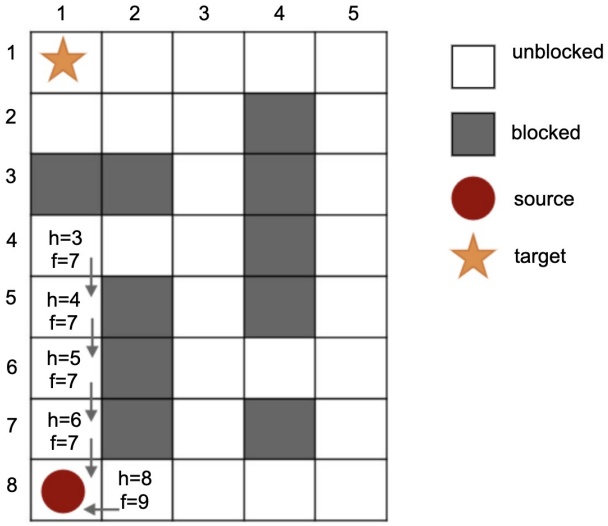


FIGURE 2.27: Step 4

5) Pop (4,1) off, and add its successor (4,2) to the OPEN list as in in Figure 2.28.

- OPEN list: { (8,2), (4,2) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1), (4,1) }

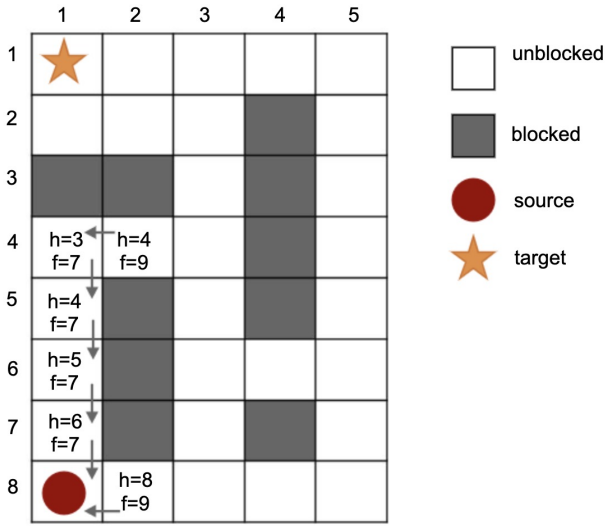


FIGURE 2.28: Step 5

6) In this step, there are two nodes in the OPEN list with the same f=9, assume (8,2) is popped off. Add its successor (8,3) to the OPEN list as in in Figure 2.29. This is an example to show that there could be some cases that the node popped off the OPEN list is not adjacent to the node examined in the last step. In the next steps the node that are most recently pushed on the OPEN list will be popped off.

- OPEN list: { (4,2), (8,3) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1), (4,1), (8,2) }

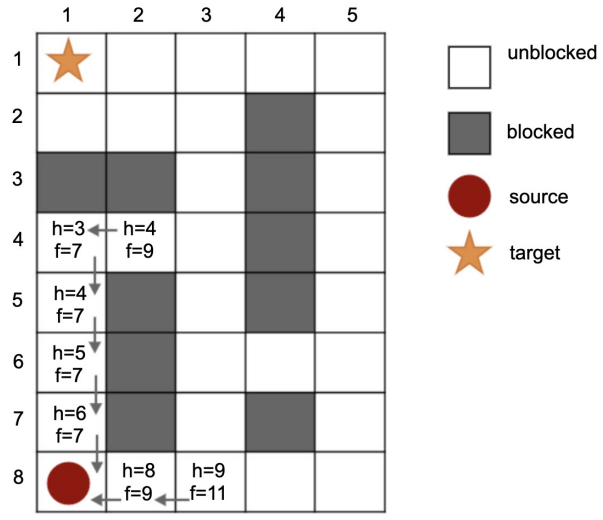


FIGURE 2.29: Step 6

7) Pop node (4,2) off, and add its successor (4,3) to the OPEN list as in in Figure 2.30.

- OPEN list: { (8,3), (4,3) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1), (4,1), (8,2), (4,2) }

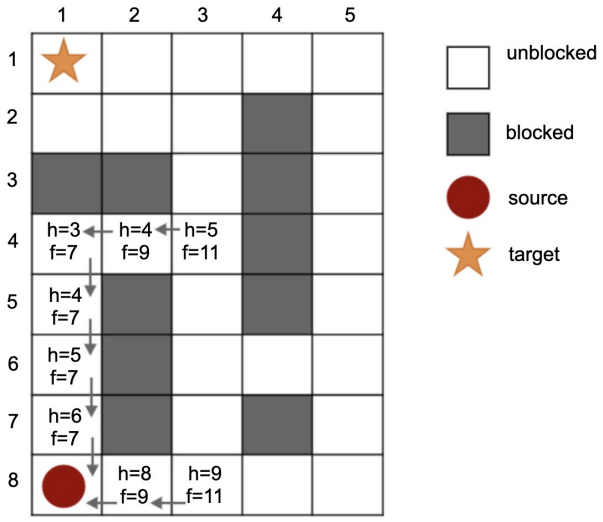


FIGURE 2.30: Step 7

8) Pop node (4,3) off, and add its successors (3,3) and (5,3) to the OPEN list as in in Figure 2.31.

- OPEN list: { (8,3), (3,3), (5,3) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1), (4,1), (8,2), (4,2), (4,3) }

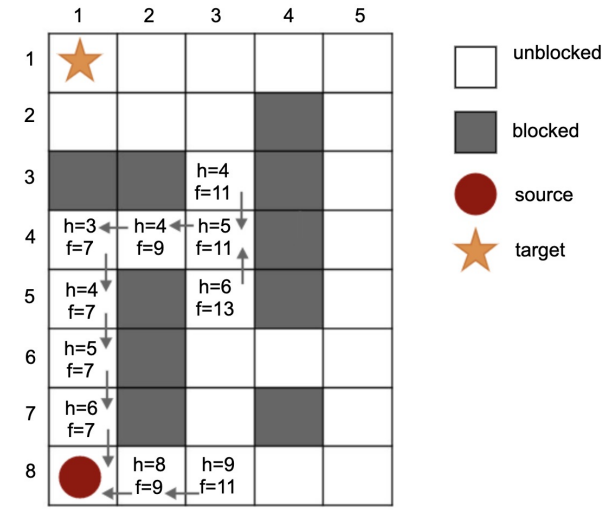


FIGURE 2.31: Step 8

9) Now there are two nodes (8,3) and (3,3) of the same f=11 on the OPEN list. Pop node (3,3) which is most recently added to the OPEN list. Add its successor (2,3) to the OPEN list as in in Figure 2.32.

- OPEN list: { (8,3), (5,3), (2,3) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1), (4,1), (8,2), (4,2), (4,3), (3,3) }

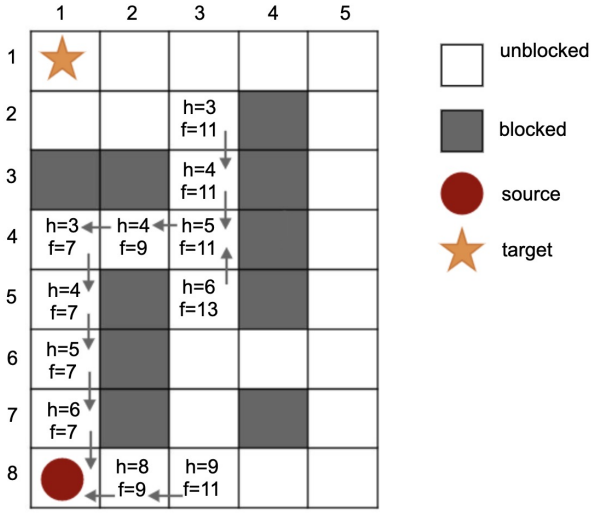


FIGURE 2.32: Step 9

10) Pop node (2,3), add its successors (2,2) and (1,3) to the OPEN list as in in Figure 2.33.

- OPEN list: { (8,3), (5,3), (2,2), (1,3) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1), (4,1), (8,2), (4,2), (4,3), (3,3), (2,3) }

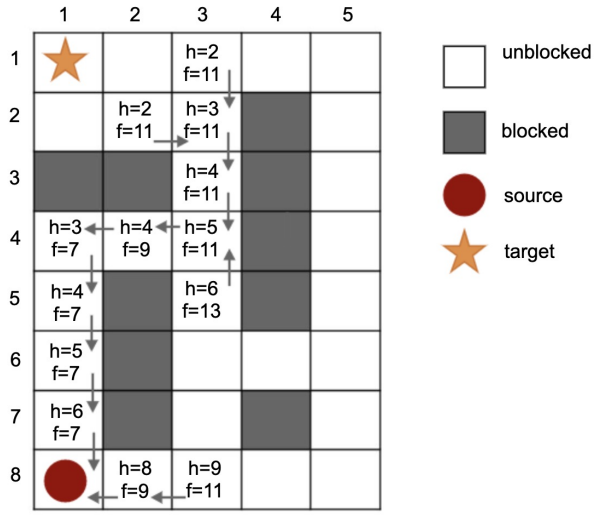


FIGURE 2.33: Step 10

11) Pop node (1,3), add its successors (1,2) and (1,4) to the OPEN list as in in Figure 2.34.

- OPEN list: { (8,3), (5,3), (2,2), (1,2), (1,4) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1), (4,1), (8,2), (4,2), (4,3), (3,3), (2,3), (1,3) }

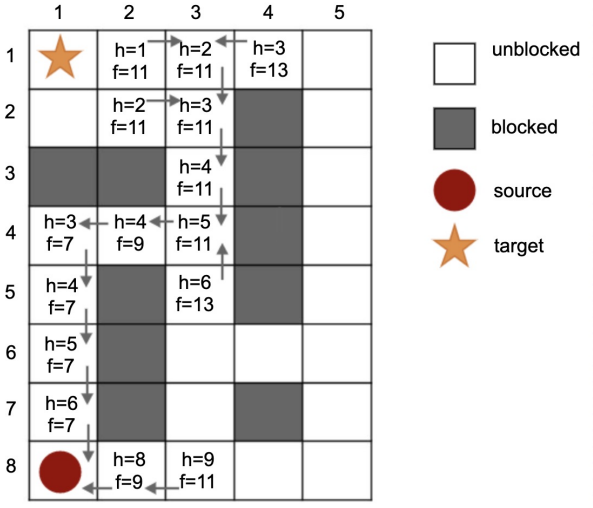


FIGURE 2.34: Step 11

12) Pop node (1,2), generate its successors (1,1), (1,3) and (2,2) as in in Figure 2.35. Since (1,1) is the goal, the search is stopped.

- OPEN list: { (8,3), (5,3), (2,2), (1,4) }

- CLOSED list: { (8,1), (7,1), (6,1), (5,1), (4,1), (8,2), (4,2), (4,3), (3,3), (2,3), (1,3) }

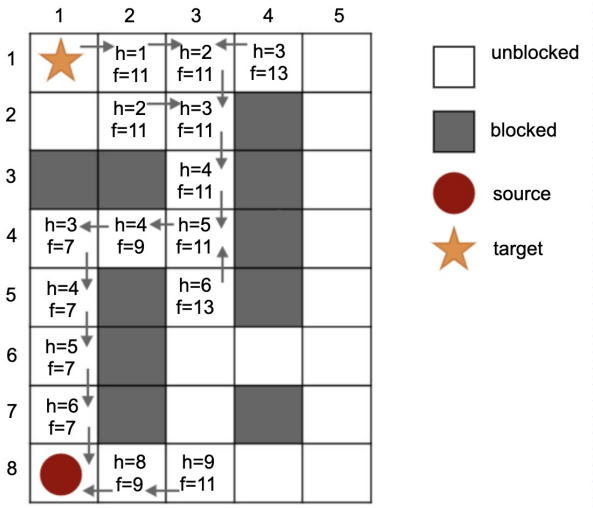
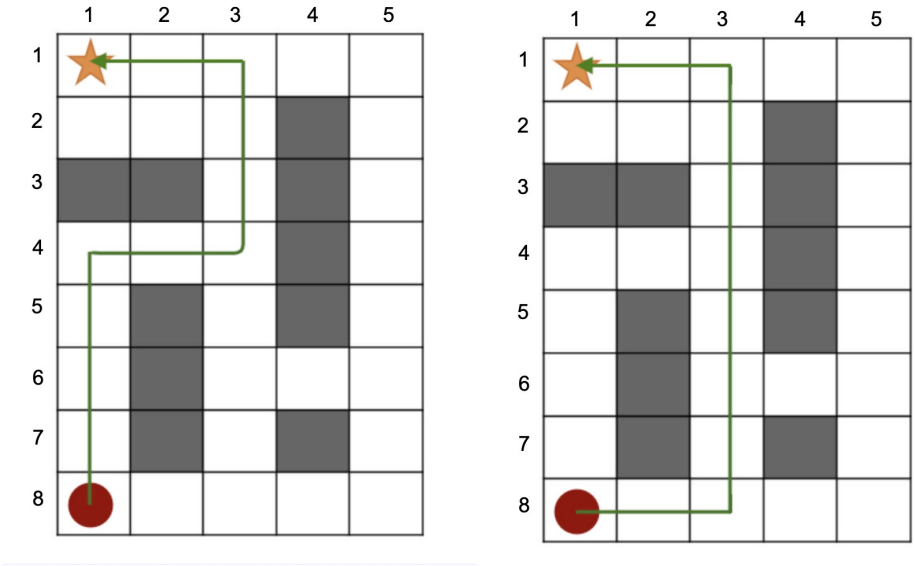


FIGURE 2.35: Step 12

In the end, the path can be traversed using the pointers set in each step. The path is (8,1)->(7,1)->(6,1)->(5,1)->(4,1)->(4,2)->(4,3)->(3,3)->(2,3)->(1,3)->(1,2)->(1,1) shown in Figure 2.36(a).

There can be more than one shortest path on a map depending on the sequence of nodes popped off the OPEN list. Figure 2.36(b) shows another path that could possibly be achieved by the algorithm.



(a) path1 (b) path2

FIGURE 2.36

2.6.4 Criteria of A* Search Algorithm

A* search algorithm is complete, it will always find a solution if one exists. It is also optimal as long as the heuristic function is admissible.

A* has a large memory requirement as it keeps all the generated nodes in memory. When the heuristic being used is admissible but not consistent, it is possible for a node to be expanded many times, an exponential number of times in the worst case, thus the exponential time complexity. As a result, it is not practical for various large-scale problems.

2.6.5 Implementation Details

The code is written in C language. The map is represented using a grid, implemented as a 2-dimension array, where 1 stands for an unblocked cell, and 0 for a blocked cell. Several data structures are also created to implement the OPEN and

CLOSED list. Since the standard C Library does not provide the stack data structure, a Stack library should be created in the first place.

1) The Stack library

The data structure of type Stack consists of 4 fields as in Figure 2.37. This is rather a header that contains the general information of the stack, the first field points to the memory address where the first data element is at. The data on the stack conforms to the principle that the last one pushed to the stack is to be the first to pop off.

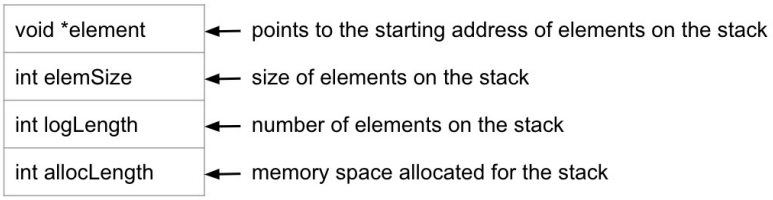


FIGURE 2.37: Stack data structure

The library also provides basic functions for the construction and dispose of the stack, as well as push and pop operations.

- **void stackNew (stack *s, int *elemSize*);**

  Creates a stack and initialize the elements of *s*. Allocate 1 unit of size *elemSize* in the memory for the data, i.e. *s*->allocLength = 1. Set *s*->logLength = 0 which indicates it is initially empty. *s*->element points to the memory space allocated before.

- **void stackDispose (stack *s);**

  Destroy the stack and free the memory space allocated for the data.

- **int stackEmpty (stack *s);**

  Check if the stack is empty. If yes return 1, otherwise return 0.

- **void stackGrow(stack *s);**

  Extend the stack to double of the previous size.

- **void stackPush (stack *s, void *elemAddr);**

  Add an element to the top of the stack at address *elemAddr*. First of all check if the stack is full, i.e. *s*->logLength = *s*->allocLength. If the stack is full, call StackGrow() to extend the memory space. Push the data pointed by *elemAddr* to the top of the stack, and increase *s*->logLength by 1.

- **void stackPop (stack *s, void *elemAddr);**

  Pop off the data on the top of the stack. Copy the data to the memory address pointed by *elemAddr*, and decrease *s*->logLength by 1.

- **void stackSwitchTop (stack *s, void *elemAddr);**

  Switch the data on the top of the stack and the data pointed by *elemAddr*.

2) Basic data structures

Three data structures in Figure 2.38 are created to hold the information about each cell on the map.

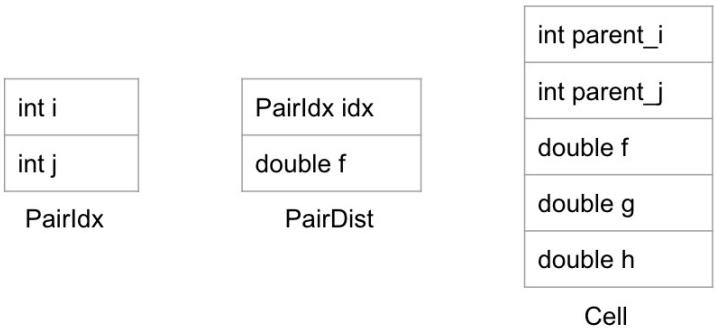


FIGURE 2.38: Basic data structures

- PairIdx : Represent the position of a cell on the grid.

- PairDist : A shortcut of a cell with its index and f value.

- Cell : The f,g,h value of a cell and its parent index.

3) OPEN list and CLOSED list

The OPEN list is implemented as a stack of PairDist type elements defined above. The CLOSED list is a 2-dimension array of _Bool type initialized to all 0s, when a cell is added to the CLOSED list, the value of the corresponding position in the array is updated to 1.

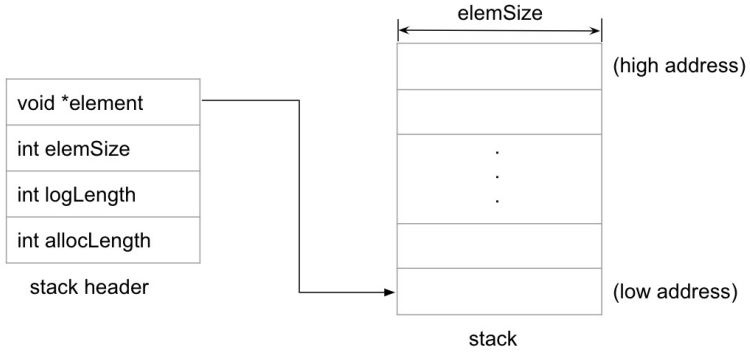


FIGURE 2.39: OPEN list implemented as a stack

Besides, a 2-dimension array cellDetails[][] of Cell type is also created to record the parameters of each cell on the grid. Each element of the array corresponds to the the cell on the grid of the same index. The parent fields are initialized to -1, and f,g,h to FLT_MAX. Those fields will be updated during the execution of the algorithm, and this array is used to backtrack the path after the algorithm is finished.

Although it is more concise and space saving to implement the CLOSED list as a Cell type stack instead of creating two arrays, the time complexity of finding a node of a certain index (i,j) on the stack is *O(n)*, while for the array is only *O(1)*. The array implementation is more efficient in general.

4) Algorithm functions

- **int aStarSearch (int max_row, int max_col, int grid[max_row][max_col], PairIdx src, PairIdx dest, _Bool isDiagonalEnabled, stack *ps)**

  Parameters:

  - **max_row, max_col** : dimension of the map

  - **grid[max_row][max_col]** : grid-represented map

  - **src** : starting point cell index

  - **dest** : goal point cell index

  - **isDiagonalEnabled** : flag to enable or disable the move to diagonal cells

  - **ps** : points to the stack that records the path found

  This is the function that implements the A* algorithm. First of all it checks if the start and goal cells are blocked or exceed the boundary of the map. If not, the A* algorithm described in Figure 2.23 starts with the provided parameters.

  A 2-dimension Cell type array cellDetails[][] of the same dimension as the map grid is also created within the function. The parent fields are initialized to -1,

and f,g,h to FLT_MAX. Those fields are updated during the execution of the algorithm, and this array will later be passed to function **tracePath()**(explained below) as a parameter.

If the goal is reached, function **tracePath()** is called and the path will be recorded on the stack pointed by **ps**, function **aStarSearch()** returns 1 indicating that the path from the starting cell to the goal cell exists.

When the OPEN list becomes empty but the goal is still not yet reached, it indicates that no path exists from the starting cell to the goal cell. The function returns 0.

- **void tracePath (int max_col, Cell cellDetails[][max_col], PairIdx dest, stack* ps)** :

  Parameters:

  - **max_col** : horizontal dimension of the map

  - **cellDetails[][max_col]** : array that keeps the cell details and used for backtracking the path

  - **dest** : goal point cell index

  - **ps** : points to the stack that records the path found

  This function is called by **aStarSearch()** when the search algorithm is finished. Using the parent pointers kept in the array **cellDetails**, it is able to trace the path from the destination to the starting cell. The strating cell index is not in the parameter list, but its parent pointer points to itself, which marks the end of the backtracking. The path is recorded on the stack pointed by **ps**.

- **double calculateHValue(int row, int col, PairIdx dest)** :

  Parameters:

  - **row** : vertical index of the cell(starts from 0)

  - **col** : horizontal index of the cell(starts from 0)

  - **dest** : destination cell

  This is the heuristic function to calculate the estimated distance from a certain cell indexed (**row,col**) to the cell **dest**. The function returns the h(n) of cell (**row,col**).

Chapter 3

Network Functions

This chapter briefly introduces the structure of the network and the network communication protocols, as well as the socket functions used in a connection-oriented communication.

Also the detailed implementation of the system in Figure 3.1 is illustrated in Section 3.4.

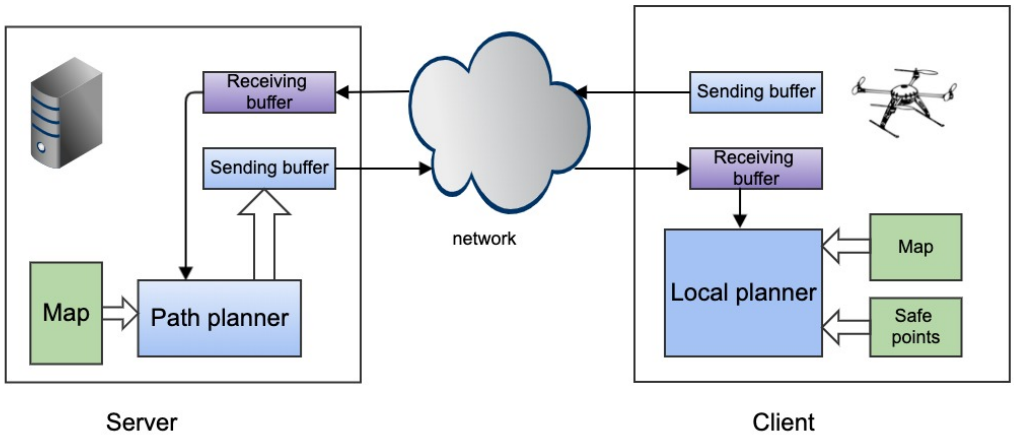


FIGURE 3.1: System architecture

3.1 Network model

3.1.1 OSI model

The Open Systems Interconnection(OSI) model is a conceptual model that defines the characteristics and standards which enable communication systems of different underlying internal structures and technologies to communicate using standard protocols.

The OSI model splits up the communication system into seven abstract layers:[16]

1) Physical Layer

This layer includes the physical equipment that connects computers in a network such as the cables, switches and satellite connections. The data is converted into a bit stream of 0s and 1s for being actually transmitted in this layer, and the physical layer of both devices should agree on some conventions of data rate, synchronization of bits and transmission mode in order that the data gets transmitted correctly.

2) Data Link Layer

Data Link Layer facilitates the data transfer on the same network segment across the physical layer. Data is converted into frames in this layer, and the frames are transferred hop to hop with a flow control and error control.

3) Network Layer

This network layer deals with organizing the data for transfer and reassembly between two different networks. It extracts IP address from TCP/IP data units, performs host resolution and routing. The data transferred on this layer is in the form of packet.

4) Transport Layer

The transport layer manages the end-to-end communication of variable-length data sequences between two hosts, while maintaining the quality of services by error control on the receiving end and flow control .

5) Session Layer

The session layer enables two end-user application processes to hold an ongoing communications long enough to transfer all the data being exchanged across a network. This communication is called a session. It handles session establishments, data exchanges, and closure of a session when it ends. The session layer also provides synchronization checkpoints which allow the transfer to resume at a checkpoint instead of from the beginning when a network failure during the transfer occurs.

6) Presentation Layer

This layer is primarily responsible for formatting the data so that it can be used by the application layer. It translates the data between communicating devices that use different encoding methods, adds encryption on the sender's end and decodes on the receiver's end over an encrypted connection, and also compresses the data before delievering it to session layer.

7) Application Layer

This is the layer closest to the end user that directly interacts with data from the user. Everything at this layer is application-specific, and it provides many protocols such as FTP, DNS, HTTP, SMTP to enable software applications that needs to transfer data between devices on the network.

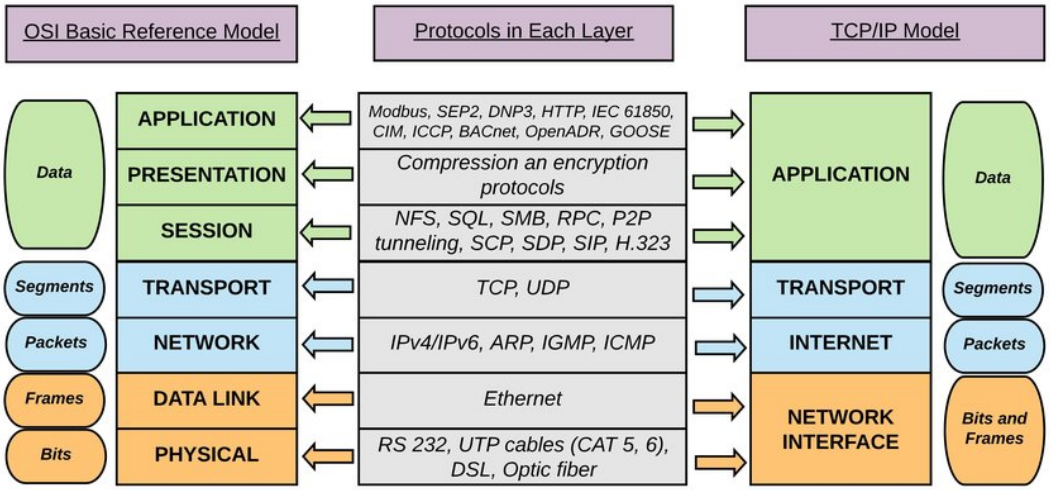


FIGURE 3.2: OSI vs TCP/IP model taken from [30]

3.1.2 TCP/IP model vs OSI model

The foundational protocols in the Internet protocol suite are the Transmission Control Protocol (TCP) and the Internet Protocol (IP). The Internet protocol suite only requires that a hardware and a software layer exist that are capable of transferring packets on a computer network, no matter how they are implemented. The services of protocols are provided by the operating system, application programmers can access those services via interfaces in the application layer and the transport layer.

The TCP/IP model can be seen as a concise version of the OSI model of fewer layers. The three top layers: Application Layer, Presentation Layer and Session Layer in the OSI model are merged to be the Application layer in the TCP/IP model. Besides, the lowest two layers of the OSI model are combined to be the lowest layer of TCP/IP model. Figure 3.2 shows the correspondence of layers of the two different models.

3.2 TCP and UDP protocols

TCP and UDP are transport layer protocols that most commonly used in internet for transmitting data between one host to another.

3.2.1 TCP protocol

Transmission Control Protocol (TCP) is a connection-oriented Internet communication protocol that enables the sending node to send data packets as a byte stream to the receiving node in an ordered sequence. The sending nodes is informed of the delivery of packets to a destination node by the use of sequence numbers and acknowledgment from the receiving node.

TCP ensures reliability, resequencing and retransmitting of data until a timeout condition is reached or acknowledgment of data packets is received. It guarantees to deliver data in the same order as sent from server to user and vice versa. Also, TCP offers flow control as well as error detection, and lost data features by trigging retransmission until an error-free acknowledgment is received.

Figure 3.3 shows the three-way handshake mechanism used by the TCP protocol during the establishment and four-way handshake mechanism for termination of a connection. In the data transfer phase, each data segment sent from the sender is assigned a sequence number. After a data segment is sent, the sender waits for an acknowledgment sent from the receiver. If the acknowledgment is not arrived before timeout, the data will be retransmitted. These mechanisms ensure the reliability of the data transferred.

TCP protocol is best suited for applications that require high reliability, and transmission time is relatively less critical.

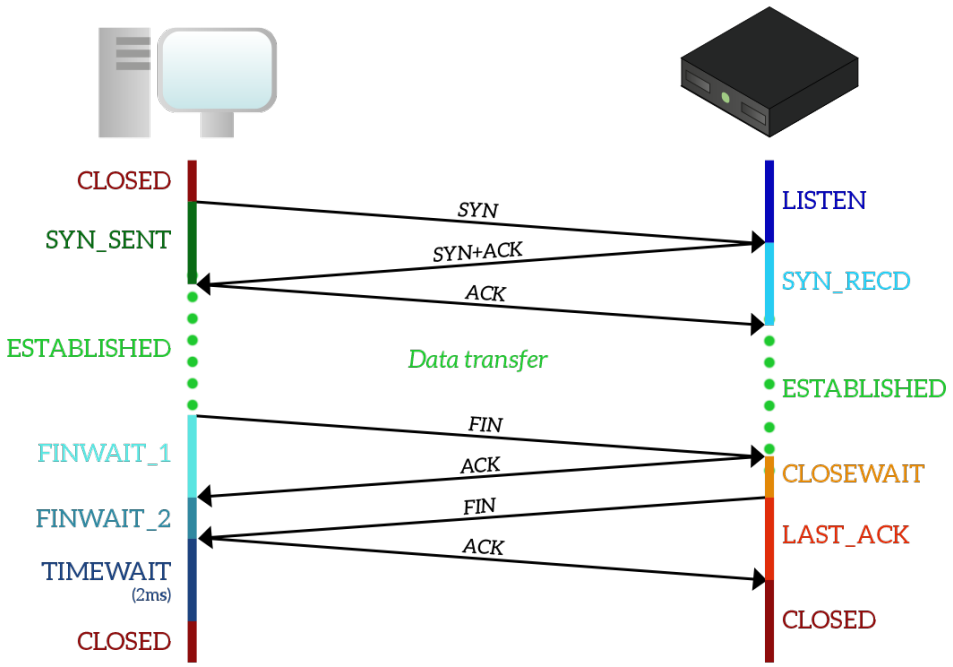


FIGURE 3.3: TCP session establishment and termination taken from [23]

3.2.2 UDP protocol

User Datagram Protocol (UDP) is especially used for time-sensitive transmissions such as video playback or DNS lookups. Unlike TCP, it does not require the receiving party to agree to the communication before the data transfer, and does not require an acknowledgment of the data transferred. Figure 3.4 shows the different fashions in the communication between UDP and TCP. Besides, UDP can broadcast datagram from a single source to all computers in a LAN due to its connection-free feature.

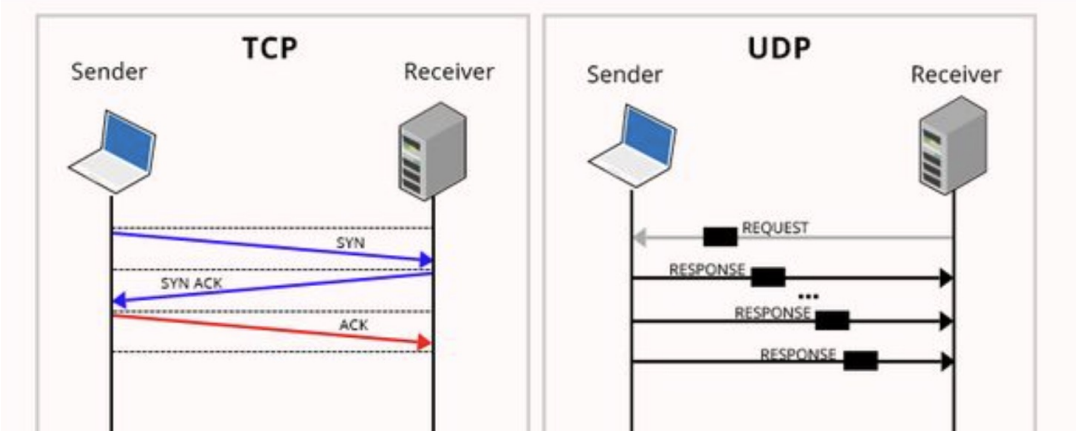


FIGURE 3.4: TCP vs UDP taken from [31]

UDP sends data without confirmation, and doesn't have the error checking and ordering functionality of TCP. Dropping packets is acceptable in some time-sensitive applications rather than waiting for delayed packets due to retransmission, which is usually not an option in a real-time system.

3.3 Sockets

A socket is a generalized inter-process communication channel. It supports communication between unrelated processes, and even between processes running on different machines that communicate over a network.

A socket is a logical endpoint for communication represented as a file descriptor. Once a socket has been created for communication via network, it must be specific to a protocol, machine, and port, those information is addressed in the header of a packet. Two sockets can communicate only if they specify the same protocol. Packets to be sent out are written to the socket, and data received from the network can be read from the socket.

The GNU C Library supports the sockets by including the header file *sys/socket.h*. The functions below are used to implement the server-client connection.

- **int socket(int domain, int type, int protocol);**

  Creates an endpoint for communication and returns a file descriptor that refers to that endpoint.

- **int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);**

  This function assigns the address specified by **addr** to the socket referred to by the file descriptor **sockfd**.

- **int listen(int sockfd, int backlog);**

  Marks the socket referred to by **sockfd** as a passive socket, which will be used to accept incoming connection requests using accept().

- **int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);**

  "The accept() system call is used with connection-based socket types. It extracts the first connection request on the queue of pending connections for the listening socket **sockfd**, creates a new connected socket, and returns a new file descriptor referring to that socket. It returns a file descriptor for the accepted socket on success."[18]

- **int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);**

The connect() system call connects the socket referred to by the file descriptor **sockfd** to the address specified by **addr**.

- **ssize_t send(int sockfd, const void *buf, size_t len, int flags);**

  Send() is used to transmit a message to another socket only when the socket is in a connected state. It returns the number of bytes sent on success.

- **ssize_t recv(int sockfd, void *buf, size_t len, int flags);**

  "Used to receive messages from a socket. If no messages are available at the socket, the recv() call waits for a message to arrive unless the socket is set to non-blocking. It returns the number of bytes received on success."[17]

- **int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);**

  "Select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become ready for some class of I/O operation. The **timeout** argument is a *timeval* structure that specifies the interval that select() should block waiting for a file descriptor to become ready.

  On success, it returns the number of file descriptors; The return value may be zero if the timeout expired before any file descriptors became ready. On error, -1 is returned, and errno is set to indicate the error."[19]

  This function is used here to test if any new messages arrive on the socket.

- **int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);**

  "Setsockopt() manipulate options for the socket referred to by the file descriptor **sockfd**. By specifying **level** field to SOL_SOCKET, **optname** to SO_SNDTIMEO and SO_RCVTIMEO, and **optval** to a *timeval* structure that specifies the timeout, the socket is set to be non-blocking."[20]

- **int close(int fd);**

  Used to close a file closes a file descriptor, which refers to a socket here.

3.4 Implementation details

The Server-Client model is essential for the communication via network between the remote server and the vehicle, where the server is a computer and the vehicle works as the client. After the connection is established, the server sends a sequence of way points of the path to the vehicle, while the vehicle sends its current position to the

server. Reliability is crucial for this application, it should guarantee that the data received is not corrupted or timeout, and the sequence of way points in its original order.

As a result, the TCP protocol is chosen to implement the application. Figure 3.5 shows the Sever-Client model using TCP/IP protocols. Besides, an additional exception handling procedure is also included in the application to deal with the communication interrupt due to poor network connection and unexpected closure of the server.

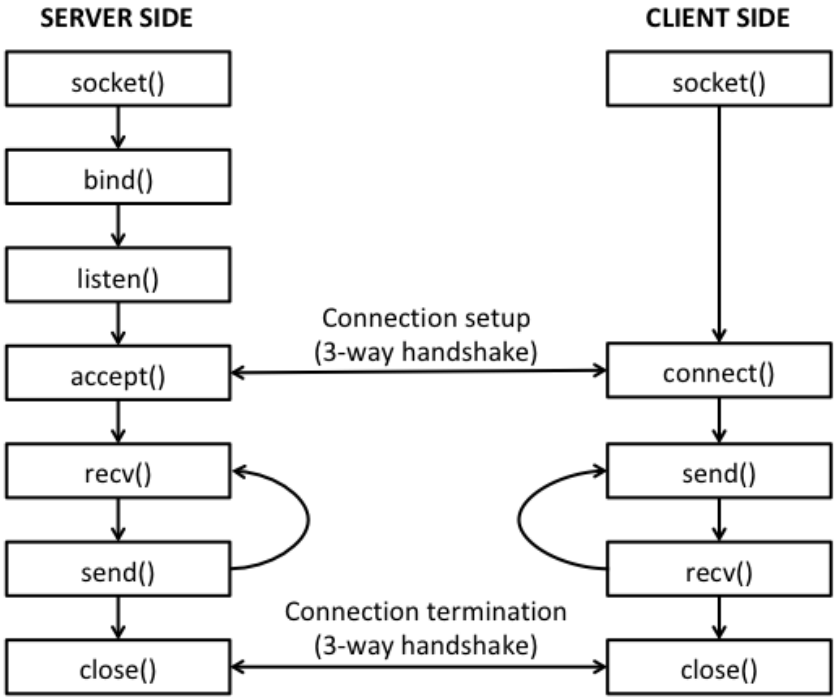


FIGURE 3.5: Sever-Client model using TCP/IP protocols

3.4.1 Sending and receiving buffers

Both the server and the client have their own sending buffer and receiving buffer. On the server side, the sending buffer holds the index of 10 way points to be sent to the client, and the receiving buffer holds a pair of index of the current client position. On the client side, the sending buffer holds the current position of the client, and the receiving buffer holds the index of 10 way points sent from the server.

When the connection is normal, the client only takes the first way point in its receiving buffer as the next position to reach, the other 9 are not used at all. But this redundancy is necessary when a connection interrupt occurs, the client is not able to updated its receiving buffer. In this case the second way point in its receiving buffer would be the next position to reach, i.e. the client can temporarily follow the path in

its receiving buffer. This makes it possible for the client to stick to the original path in spite of a short connection interrupt.

3.4.2 Server workflow

The map is stored as a grid map on the server. The server program takes as input the index of the goal, and is able to find a path starting from the position of the client and the goal. After the path is generated, the server sends the way points on the path to the goal one by one with some redundancy, it waits for the client to send its position with a timeout before sending the next way point on the path until the goal is reached. The sever is able to detect the connection interrupt and handle the circumstances when the connection is resumed or completely lost.

In this section the workflow of the server and its interaction with client is illustrated.

1) Create a socket and listen

First of all, the server creates a socket for listening and bind it to a local address, then the program starts listening on port 5000 to wait for connection requests. When a connection request from a client arrives, the server accepts the request and creates another socket dedicated for the communication between the server and the client.

2) Get the current position of the client (vehicle).

After the connection is established, the server first receives the current position sent from the client, and then send the goal to the client.

3) Run path planning algorithm.

Now the server knows the position of the client as the starting point, and the goal has been passed to the server program as a parameter. The server runs the path planning algorithm implemented in the last section.

If no path is found, the server informs the client and closes the socket. Otherwise the path is stored on the path stack, and the first 10 way points on the stack is popped to the sending buffer in its original order. The server then sends the data in the sending buffer to the client. Figure 3.6 shows the stack and sending buffer in this step.

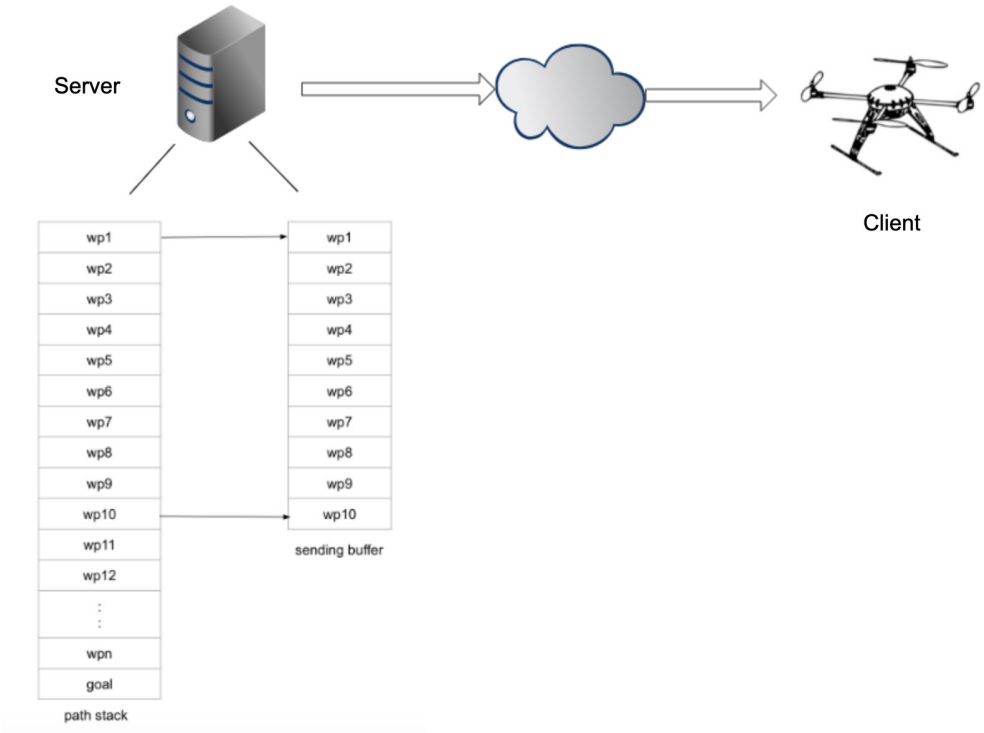


FIGURE 3.6: Server sending buffer after the path is found

4) Communication between server and client

The server waits to accept the current position of the client with a 3s timeout.

4-i) If the data arrives within timeout, first of all check if the data is equal to the goal. If the goal has been reached, the server closes the socket and the communication is terminated. If the goal has not been reached, the server then sends the new data in the sending buffer which is shifted one element as in Figure 3.7, and waits to accept the next data from the client with a 3s timeout. (jump to the beginning of step 4)

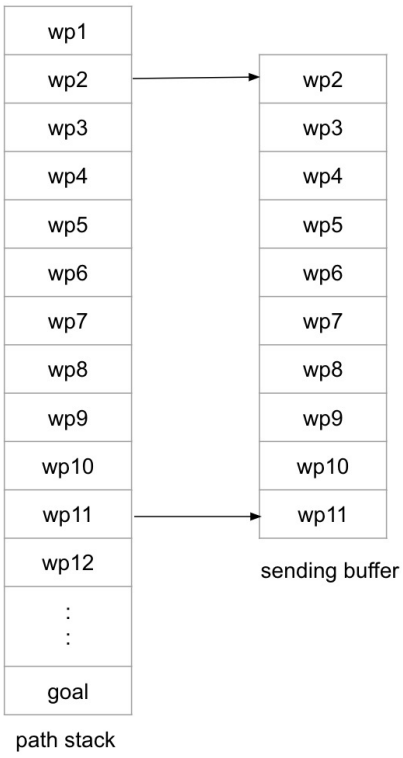


FIGURE 3.7: Updated server sending buffer

4-ii) If there is timeout, a timeout flag is set to indicate that the connection to the client is temporary lost. The server then continues waiting for the data from the client, but with a 18s timeout. If the connection is not resumed within the 18s timeout, the socket is closed and the communication is terminated. Otherwise, the first data received from the client is the number of way points it has passed during the connection interrupt. The server can use this number to update the sending buffer. An example is shown in Figure 3.8, the client has passed 4 way points during the connection interrupt, and it is now moving towards wp9. The client sends the number in the client sending buffer to the server, and the server then uses it to update the sending buffer to skip the way points that has been passed by the client during the connection interrupt. The server then waits to accept the next data from the client with a 3s timeout.(jump to the beginning of step 4)

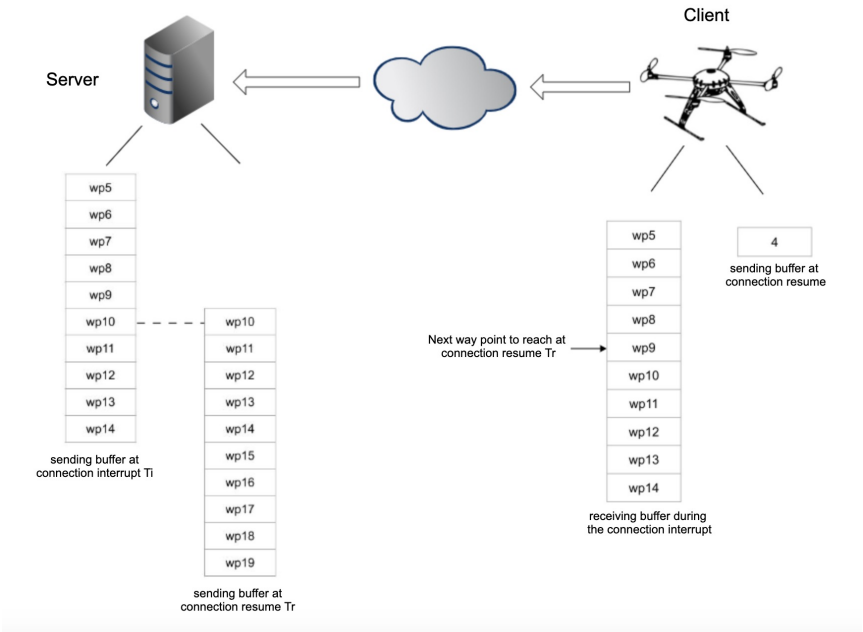


FIGURE 3.8: Case at the resume of network connection

3.4.3 Client workflow

The client can be a vehicle of any type that having a microcomputer on it. The client also keeps a copy of the map on the server, and a list of safe points where the vehicle can land, dock or stop in case of unforeseen circumstances. It is able to detect network connection interrupt and allows the vehicle to keep up with the original path during a short communication interrupt. If the communication interrupt lasts too long, the local planner will be launched to find a new path. The client implements the same path finding algorithm as the server, which will be called when the client is not able to follow the original path any more. In this case a new path is generated starting from the current position of the vehicle to the nearest safe point. The safe points are usually evenly distributed on the map so that the new path is not too long and so as the running time of the algorithm.

In this section the workflow of the client and its interaction with server is illustrated.

1) Create a socket and connect it with the server socket.

First of all, the client creates a socket, server's IP address and port 5000 is used as arguments of the connection.

2) Receive the first data from server

After the connection is established, the client waits for the first data from the server to arrive. The first data is the index of goal point that the vehicle is going to reach.

3) Send the current position to server

The client then sends the current position of the vehicle to server, this is used as the starting point of the path to be planned on the server.

4) Communication between server and client

The client waits for the data from server with a 1s timeout.

4-i) If the data arrives within timeout, the data is stored in the client's receiving buffer. The first way point in the buffer is taken out as the next position for the vehicle to reach. The client then sleeps for 2s which simulates the time needed for the vehicle to move in the real environment. At last, the client sends the updated position of the vehicle to the server, and waits for the next data from server(Jump to the beginning of step4).

4-2) If there is timeout, a communication interrupt occurs. The client is not able to get the next position to reach from the server.

It first checks if the goal is in its receiving buffer which is updated before the communication interrupt. If the goal is in it, the client does not have to wait for the server data any more. It will follow the way points in the receiving buffer until the goal is reached. Then the socked is closed and the program ends.

Otherwise the client take the second way point in the receiving buffer, and wait to receive data from the server again(Jump to the beginning of step 4).

If there is still timeout, the third way point in the receiving buffer will be taken out, so on and so forth. It should be noted that when the communication resumes, the data sent by the client to the server is not the current position of the vehicle but the number of points that has been passed by the vehicle during the interrupt(illustrated in Figure 3.8).

When the 8th way point is taken out, at this moment the communication interrupt has been too long and the client will no longer wait for the server data. The local planner is launched(Jump to step 5).

4-3) If the server socket is unexpectedly closed, the client launches the local planner immediately(Jump to step 5).

5) Launch local planner(if necessary)

The local planner takes the current position of the vehicle, and find the safe point that is nearest to the vehicle. Then the path planning algorithm is performed to generate a path from the current position of the vehicle to the nearest safe point as the new destination. The vehicle will run offline and follow the new path. When the vehicle arrives at the new destination, the client closes the socket and the program ends.

3.4.4 Server-Client communication

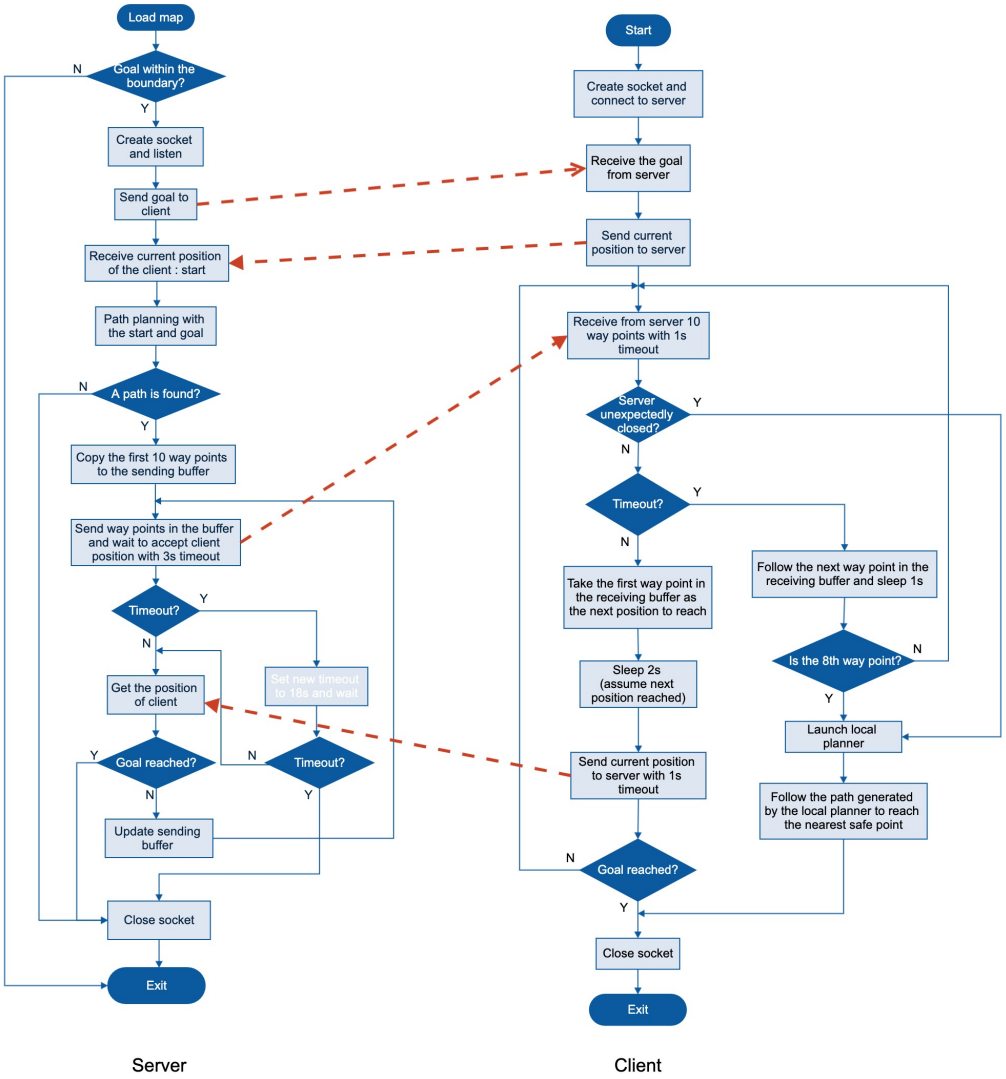


FIGURE 3.9: Server-Client flowchart and communication

Figure 3.9 shows the flowchart of the server and client described in section 3.4.2 and 3.4.3, and the communication between them. Some details are omitted in the flowchart.

Chapter 4

Test Results

This chapter shows the test result of the project using the map in Figure 5.17 under five circumstances. All the programs are implemented in C code and developed on Xcode. The server and the client(UAV) are running on the same pc but different OS. The server and client are connected to the virtual network provided by the VirtualBox virtual networking mechanism.

Server: macOS Mojave

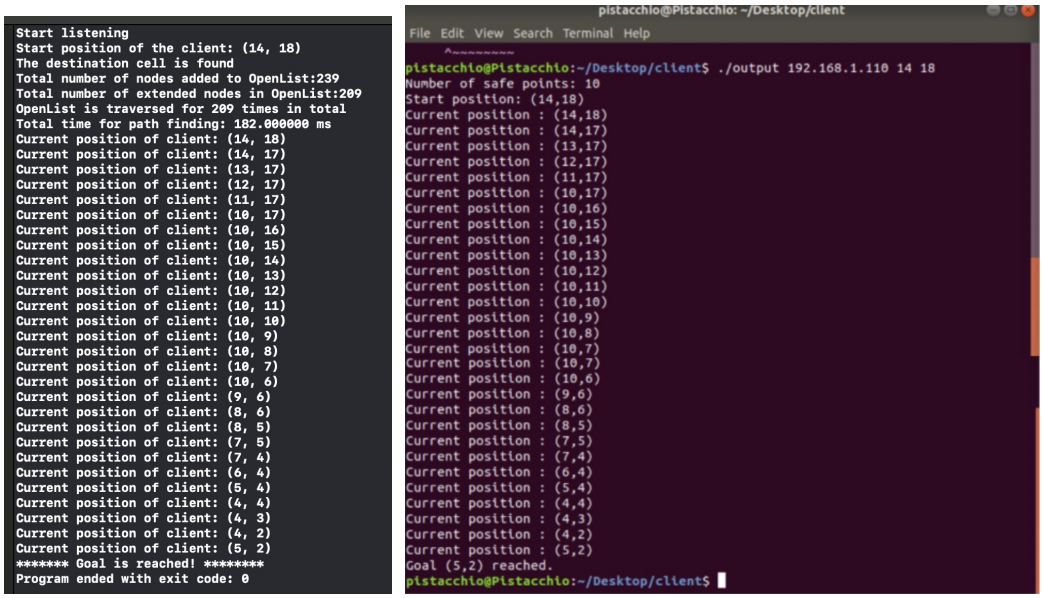
Client: Ubuntu on VirtualBox

Inputs: Input of the server program is the index of the goal : (5,2) set as the parameter; Input of the client program is the server ip and its current position.

Outputs: Output of the server program is a txt file recording the path generated by the algorithm, and a txt file of nodes that has been added in the Open List; Output of the client is a txt file recording the path which it followed in the real case. The path of the client may not be the same as the one generated by server. Information about the path planning algorithm is also printed in the command window.

4.1 Normal case : No Communication Interrupt

The test result is shown in Figure 4.1. The paths on both sides are the same.

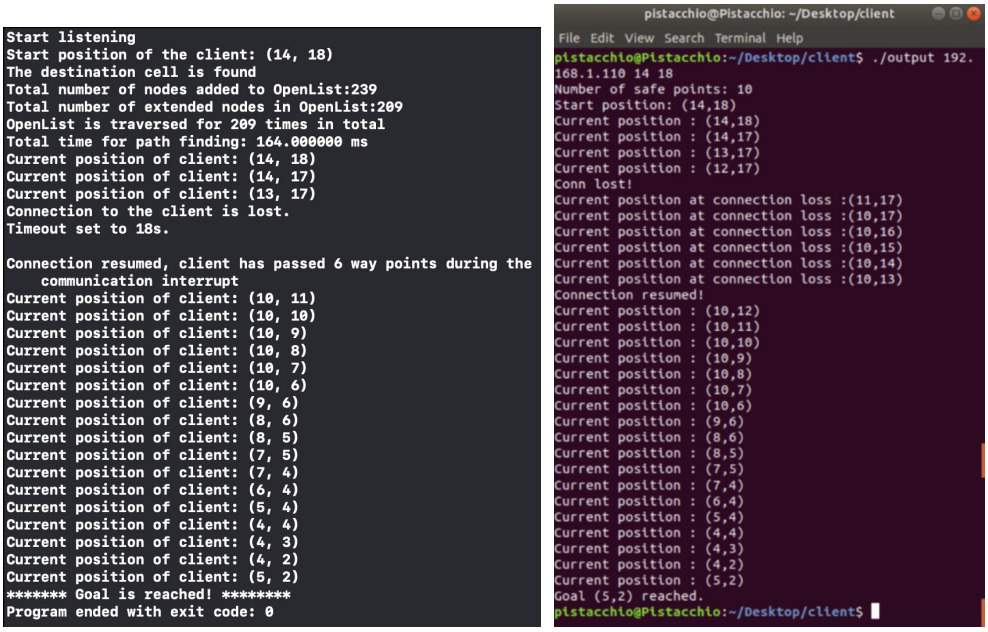


(a) server (b) client

FIGURE 4.1

4.2 Short Communication Interrupt

The test result is shown in Figure 4.2. The interrupt information is shown, but paths on both sides are the same. The client keeps on the path all the way and the server is able to send the right sequence after the short communication interrupt.

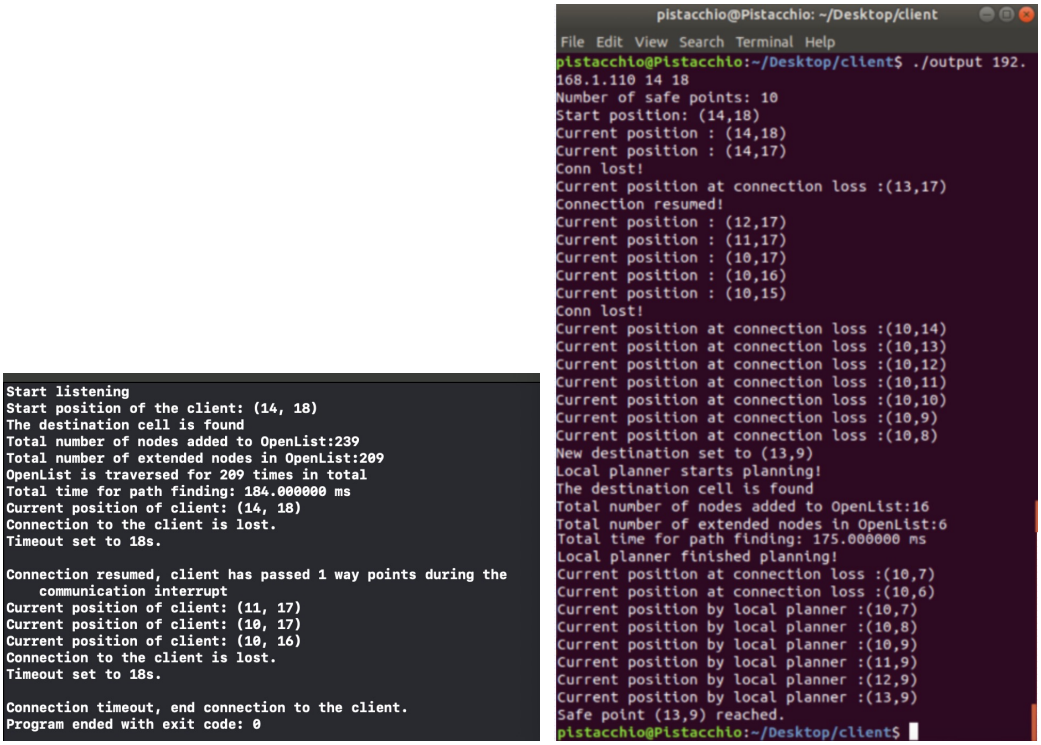


(a) server (b) client

FIGURE 4.2

4.3 Short Communication Interrupt Followed by a Long Communication interrupt

The test result is shown in Figure 4.3. The first interrupt is a short one, the client is able to keep the original path after the communication resumption. The second interrupt is too long so that the client launches its local planner when the way points in its receiving buffer has almost reached the end. The path generated by the local planner leads the client to the nearest safe point, while the server closes the socket after the 18s timeout.

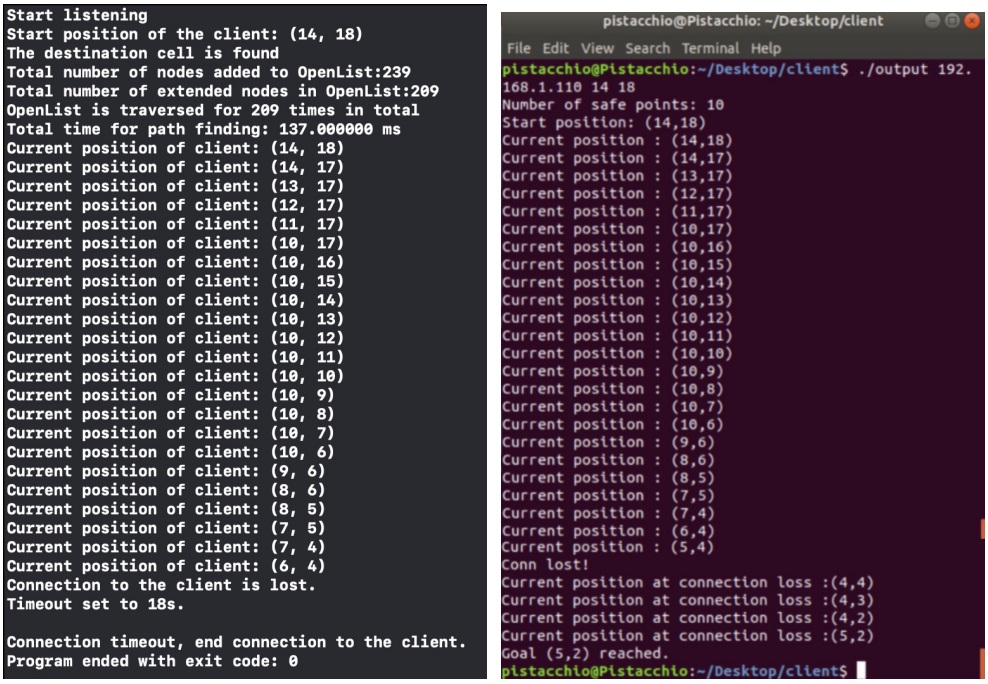


(a) server (b) client

FIGURE 4.3

4.4 Communication Interrupt Occurs When the Goal is in the Client's Receiving Buffer

The test result is shown in Figure 4.4. In this case whether the interrupt resumes or not, the client is able to reach the goal with the remaining way points in the receiving buffer, the local buffer will not be launched.

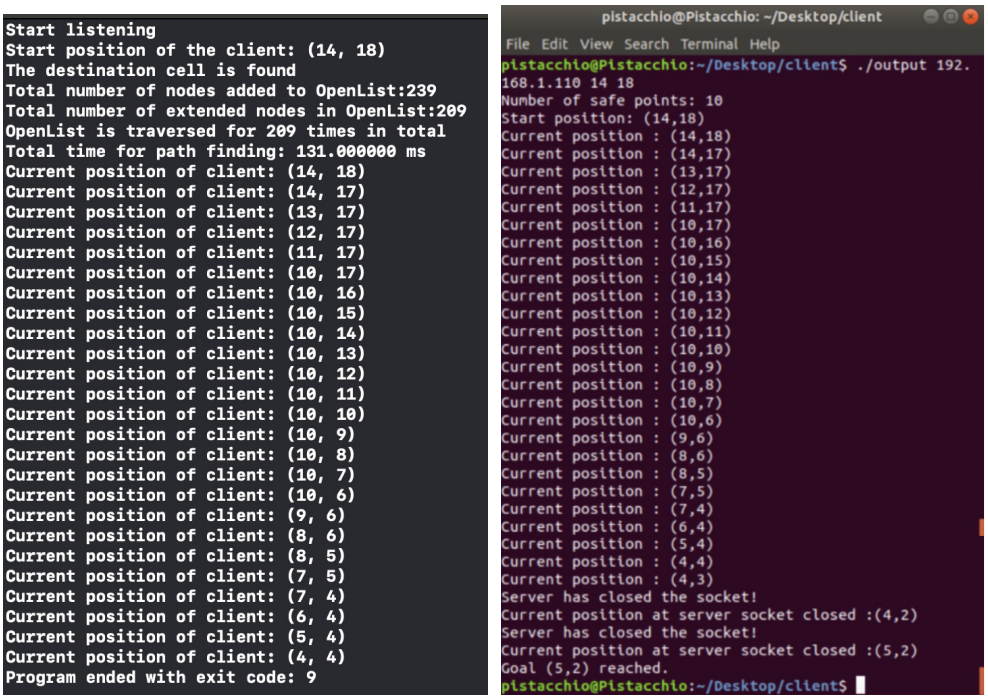


(a) server (b) client

FIGURE 4.4

4.5 Server Socket Exits Unexpectedly

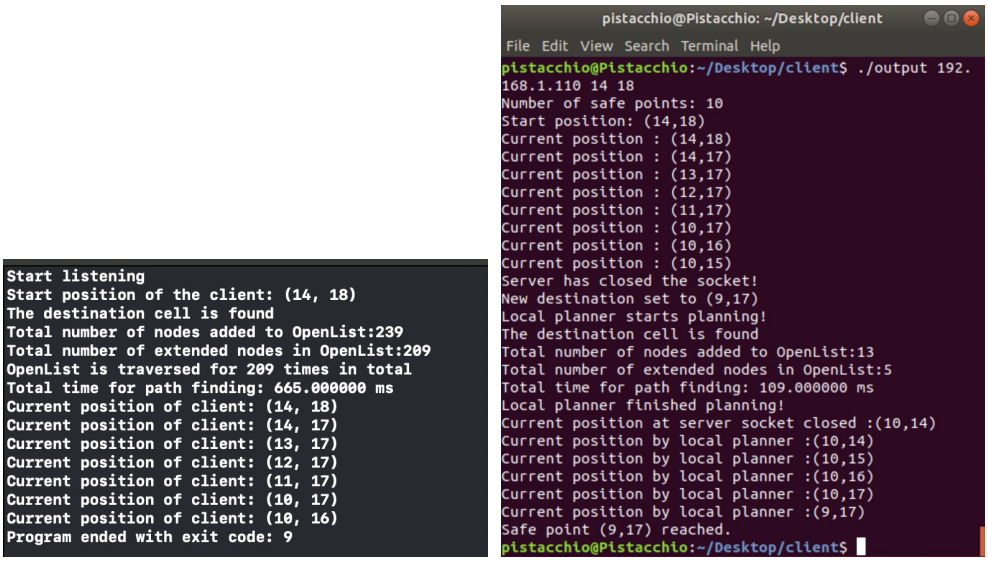
In this case the client first checks if the Goal is in its receiving buffer, if yes the client acts as in Figure 4.5(b), the same as in Figure 4.4(b).



(a) server (b) client

FIGURE 4.5

Otherwise the local planner is launched immediately to generate a new path to the nearest safe point as shown in Figure 4.6(b).



(a) server (b) client

FIGURE 4.6

In the next chapter more results are shown on the map to provide a straightforward view of the path.

Chapter 5

Simulation Results

This chapter shows the results obtained with a server-client model implementing the A* algorithm and network communication and fault management routines illustrated above.

The simulation is made on 3 maps of different complexities. For each map, 3 scenarios are tested:

- normal case without communication interrupt;
- case with short communication interrupts;
- case with a timed-out communication interrupt.

The execution time is also compared between the 3 maps.

The figures are drawn by Matlab/Simulink, using the path that is written to the output files of the program.

5.1 10x9 map size

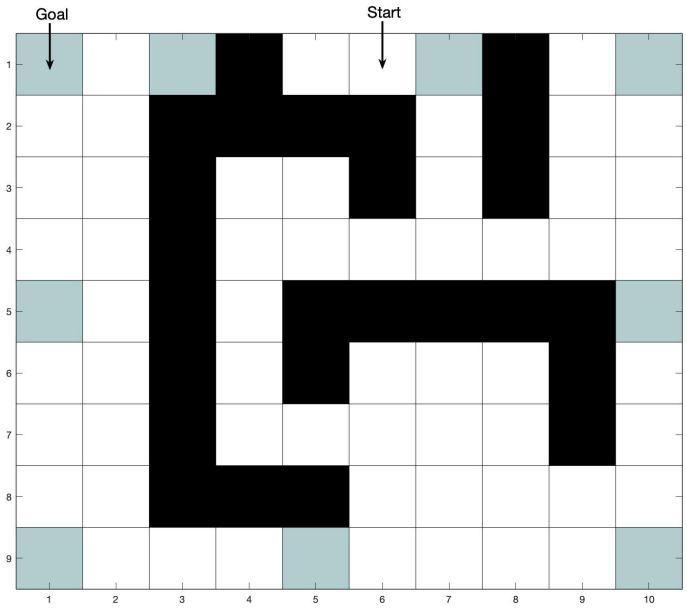
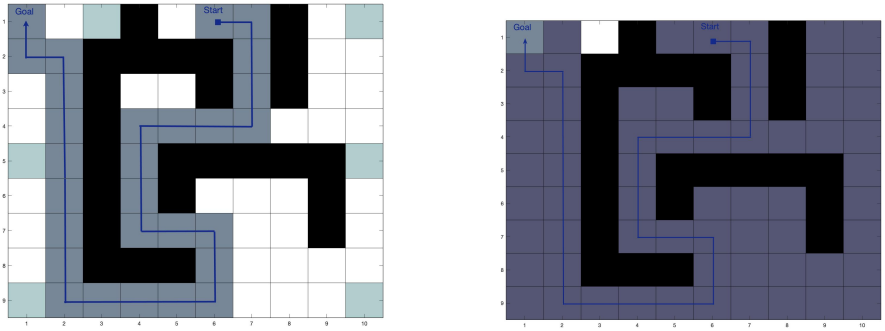


FIGURE 5.1: 10x9 map.

The map in Figure 5.1 is a small map with concave obstacles that is divided into 90 cells. The white colored cells represent the traversable free space, black colored for immobile obstacles, and green for safe points in case of timed-out communication interrupt. The start point and the goal is marked on the map.

5.1.1 Normal case



(a) path (b) path with cells in openlist

FIGURE 5.2: path-normal

Figure 5.2(a) shows an optimal path planned by A* algorithm from start point to the goal without any communication interrupt. Figure 5.2(b) marks dark grey all the grid cells that have been put on the openlist. It can be seen that almost all the free space cells have been on the openlist, these operations contribute a great deal to the total execution time of the algorithm.

5.1.2 Case with short communication interrupts

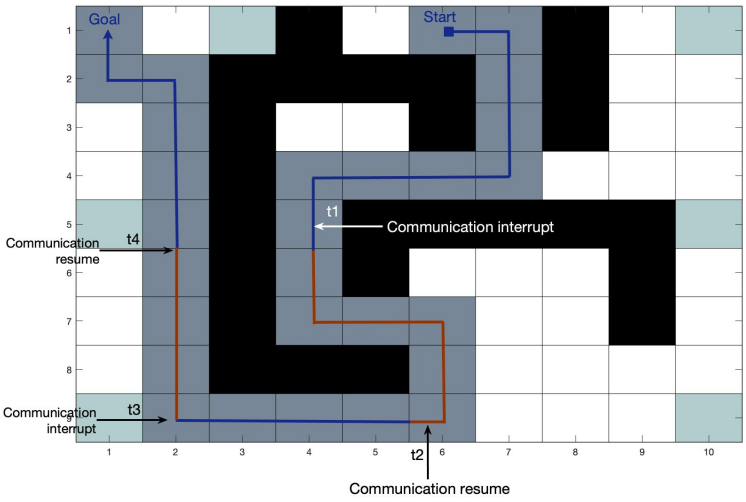


FIGURE 5.3: path with 2 interrupts

Figure 5.3 shows the case of 2 short communication interrupts that resumed within time-out. The first interrupt happens at t1, and at time t2 the communication is resumed. The second interrupt happens at t3 and resumes at t4. The path marked in red is obtained from the client local buffer which keeps the latest way points sent from the server before the communication interrupt. The overall path is the as with the normal case in Figure 5.2(a), which means that the short communication interrupts doesn't affect the path.

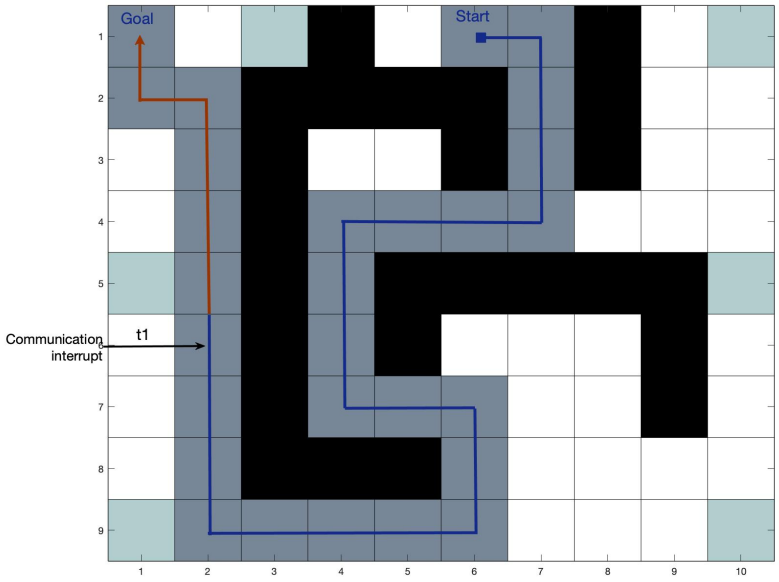


FIGURE 5.4: goal in the client local buffer

Figure 5.4 shows the case of a communication interrupt that occures at time t1. When the interrupt is detected, the program first checks if the goal is in the client local buffer. At this moment the goal(1,1) is already in the client local buffer, so the client does not need to check for the resumption of the communication or call the local planner, it runs off-line following the way points in the local buffer until the goal is reached. The overall path is the as with the normal case in Figure 5.2(a).

5.1.3 case with a timed-out communication interrupt

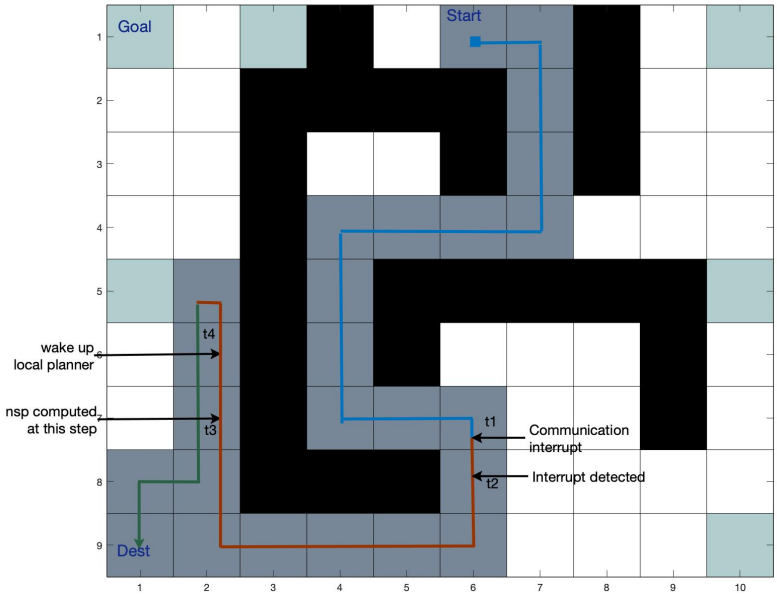


FIGURE 5.5: path with timed-out communication interrupt

Figure 5.5 shows the case of a timed-out communication interrupt. At time t2 the interrupt is detected and the client starts to follow the path in the local buffer which is marked red in the figure. When the time comes to t4 but the communication is not yet resumed, the client's local planner is waked up. It finds a path starting from the last way point in the local buffer to the nearest safe point as the new destination(which is computed based on the position at time t3). The path planned by the local planner is marked in green.

5.2 30x30 map

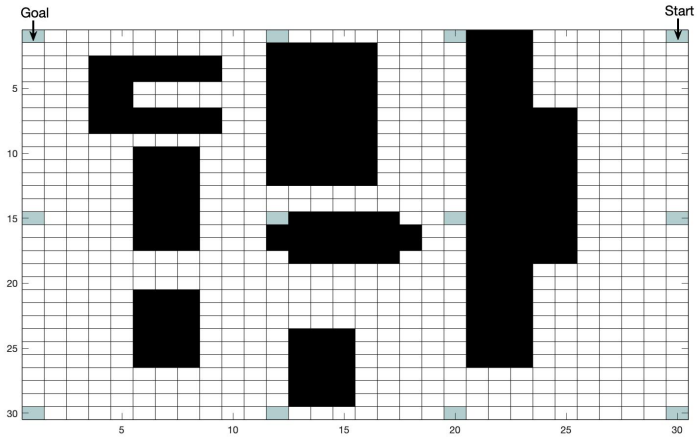


FIGURE 5.6: 30x30 map

Figure 5.6 is a 30x30 map with a relatively intensive obstacle distribution. The free space is the cells colored in white, obstacles colored in black, and safe points marked in light green.

5.2.1 Normal case

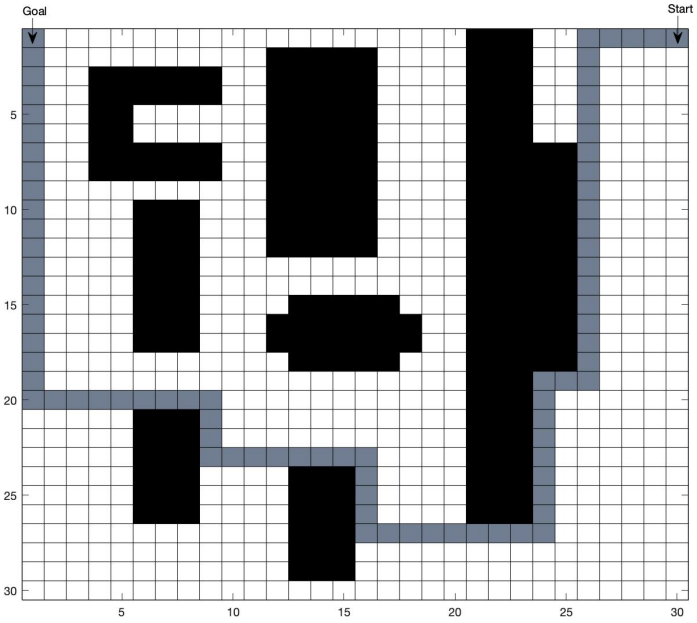


FIGURE 5.7: path in normal case

Figure 5.7 shows the path marked in grey from the start point to the goal without communication interrupt. Figure 5.8 marks all the cells that have been added to the openlist.

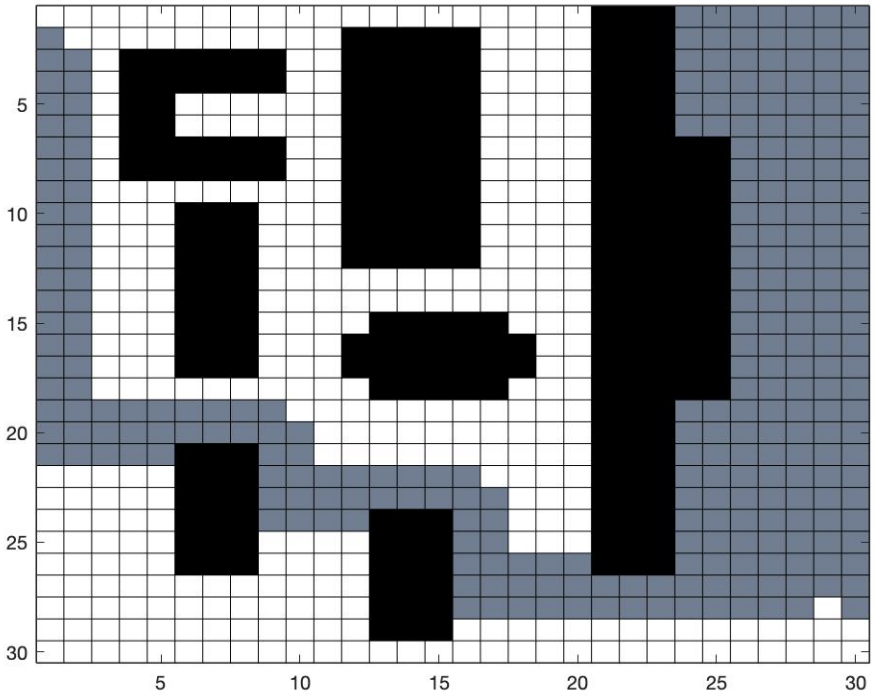


FIGURE 5.8: cells in openlist

5.2.2 Case with short communication interrupts

The result is the same as in Figure 5.7.

5.2.3 case with a timed-out communication interrupt

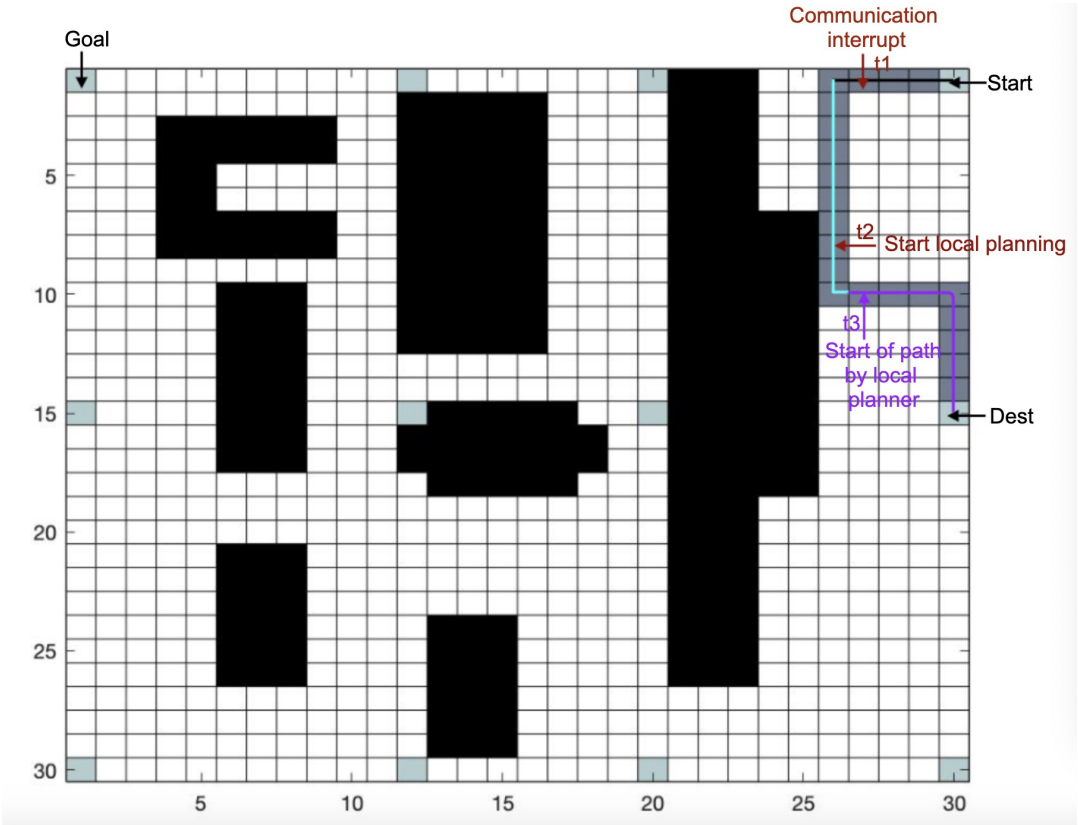


FIGURE 5.9: path with timed-out communication interrupt

Figure 5.9 and Figure 5.10 illustrates the case when there is a timed-out interrupt. The path colored in light blue is obtained from the client local buffer after the communication interrupt, and the path colored in purple is obtained by the local planner activated at time t2. The new destination is set to the nearest safe point instead of the original goal.

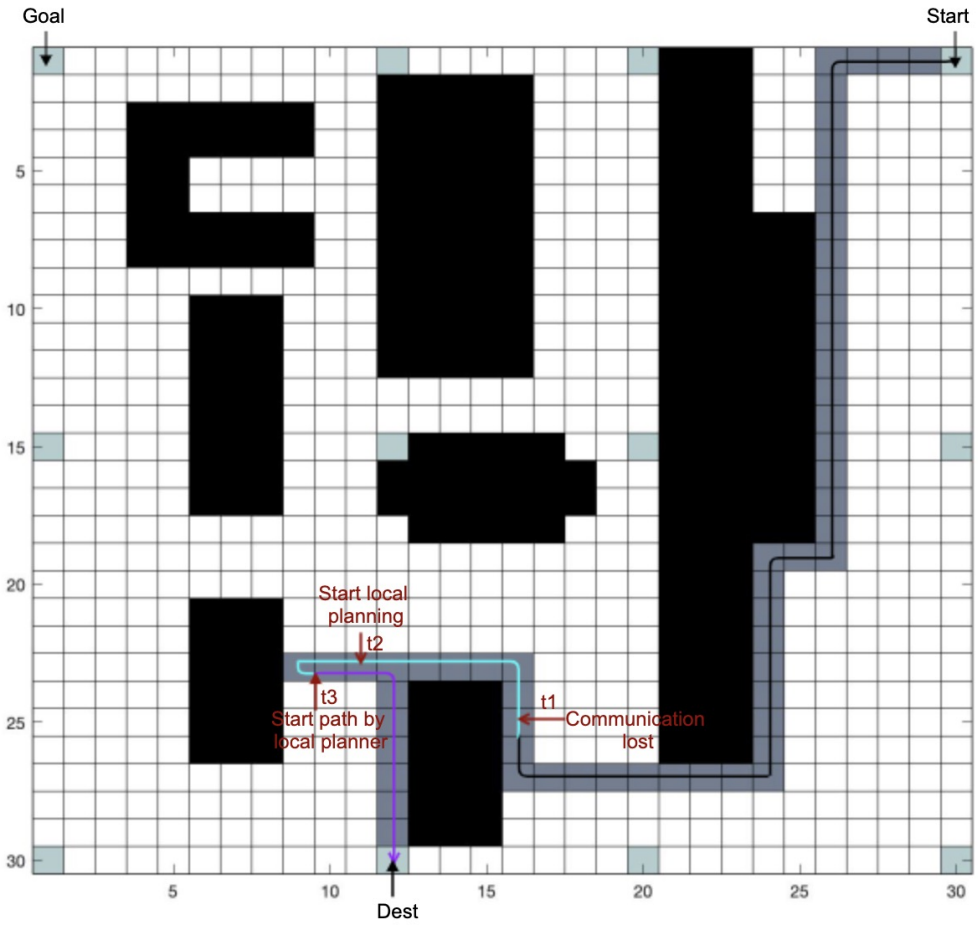


FIGURE 5.10: path with timed-out communication interrupt

5.3 50x50 map with fewer obstacles

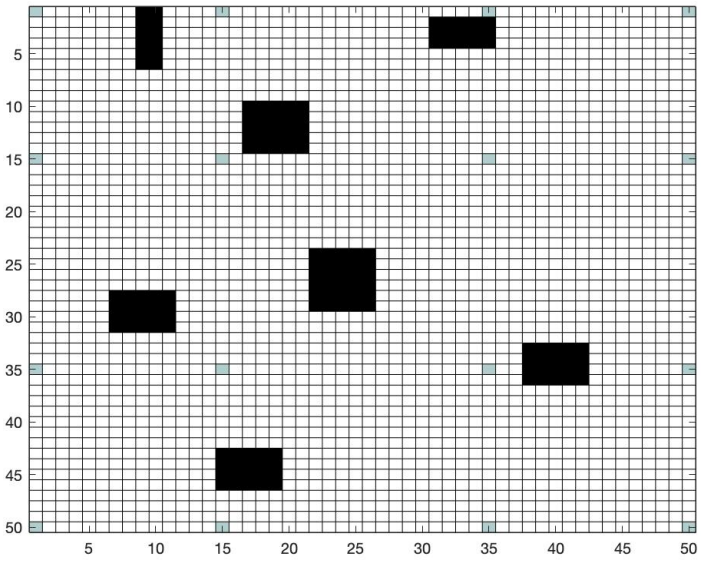
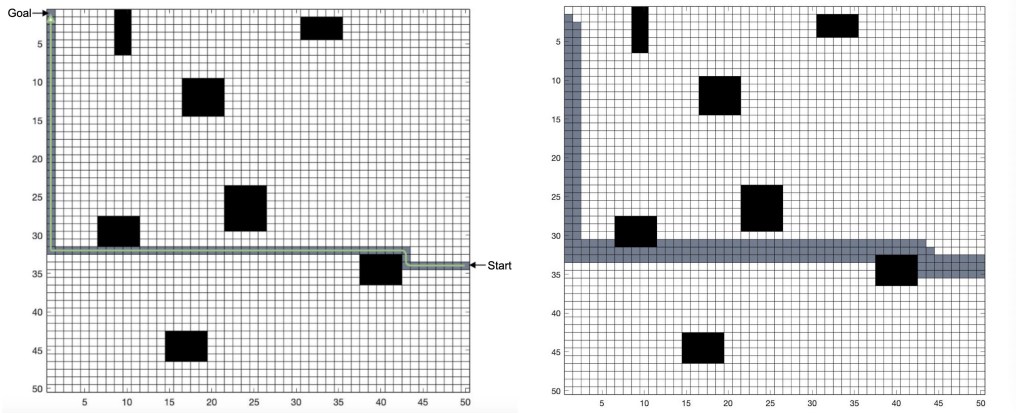


FIGURE 5.11: 50x50 map

Figure 5.11 is a map with fewer obstacles and 16 safe points distributed evenly within the grid. For this map 2 scenarios of paths are illustrated, a comparison between the 2 paths is illustrated in the next chapter.

5.3.1 Normal case



(a) path (b) cells in open list

FIGURE 5.12: path-normal

Figure 5.12(a) shows the path starting from the right of the map to the top left corner. The nodes in the open list are marked in grey in Figure 5.12(b). The open list does not get too large when the obstacles are relatively small and sparsely distributed.

5.3.2 Case with short communication interrupts

The result is the same as in Figure 5.12.

5.3.3 Server socket closed unexpectedly

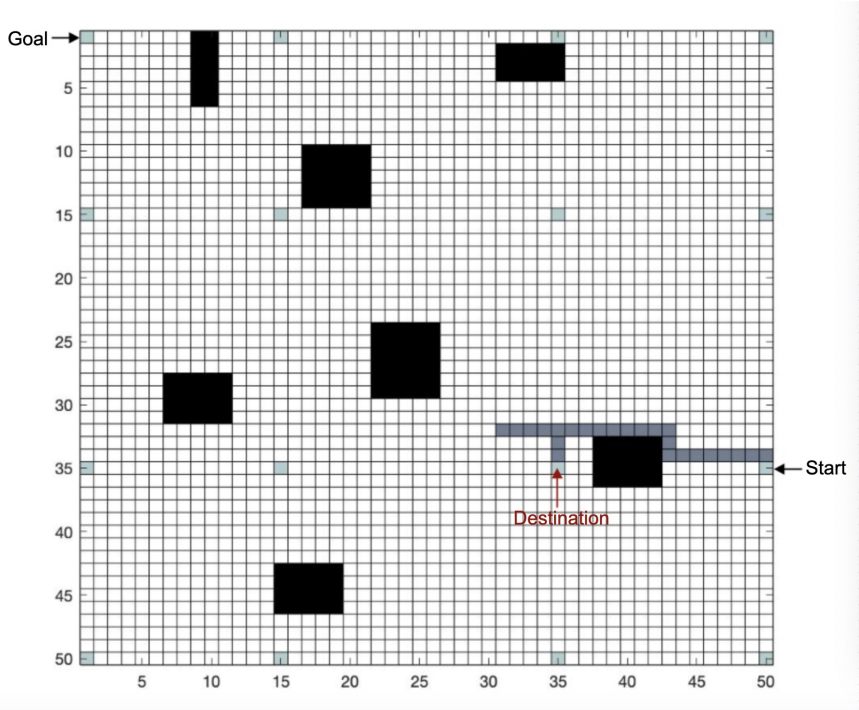


FIGURE 5.13: path with timed-out communication interrupt

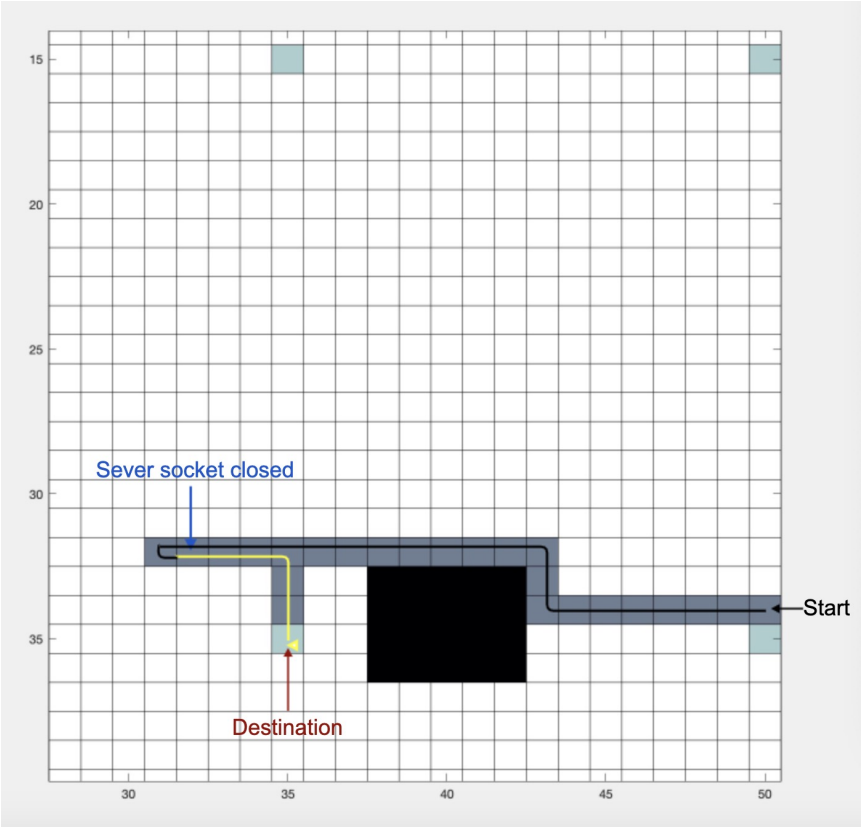
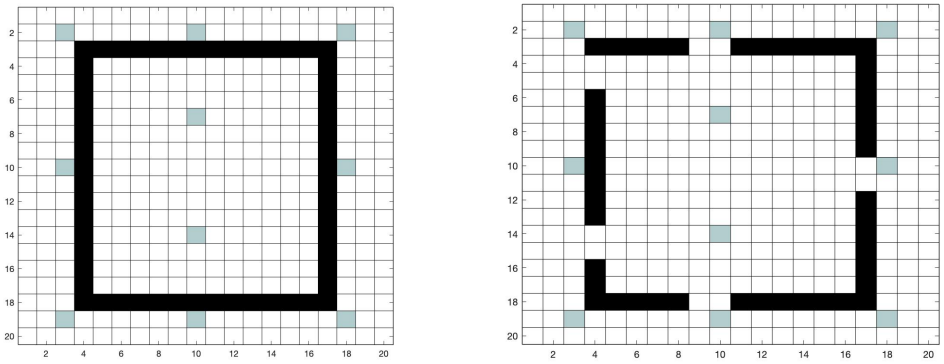


FIGURE 5.14: zoomed-in path

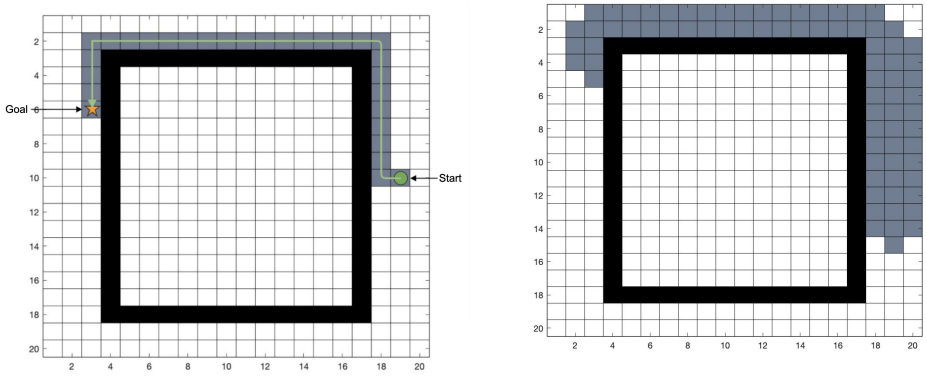
In this case, the communication is lost due to the unexpected shut-down of the server socket. When the exception is detected by the client, the local planner is called immediately, and the client would follow the path marked in yellow in Figure 5.14 to the nearest safe point as the new destination.

5.4 20x20 map with a ring-shaped obstacle



(a) ring-shaped obstacle (b) ring-shaped obstacle with holes

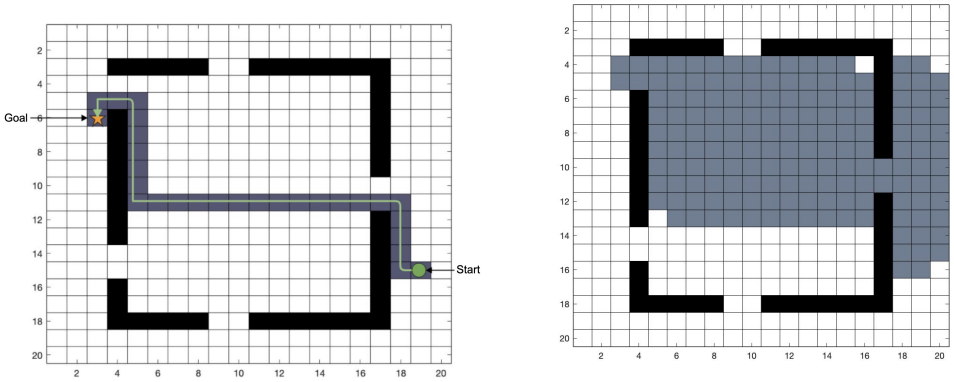
FIGURE 5.15: Maps with ring-shaped obstacle



(a) path1 (b) cells in open list

FIGURE 5.16: Map of a ring-shaped obstacle

Figure 5.16 shows a map with a ring-shaped obstacle in the center. Starting from the right side of the ring, the path bypasses the obstacle and reaches the goal successfully. The expanded nodes in the open list are marked in grey in Figure 5.16(b).



(a) path2 (b) cells in open list

FIGURE 5.17: Map of a ring-shaped obstacle with holes on the ring

Figure 5.17 shows a map of a ring-shaped obstacle with several holes. The algorithm is able to find a shortest path by crossing through the holes. The expanded nodes in the open list are marked in grey in Figure 5.17(b), in this case the open list is much larger than the above one in Figure 5.16(b). Figure 5.18 shows the path when a communication interrupt occurs.

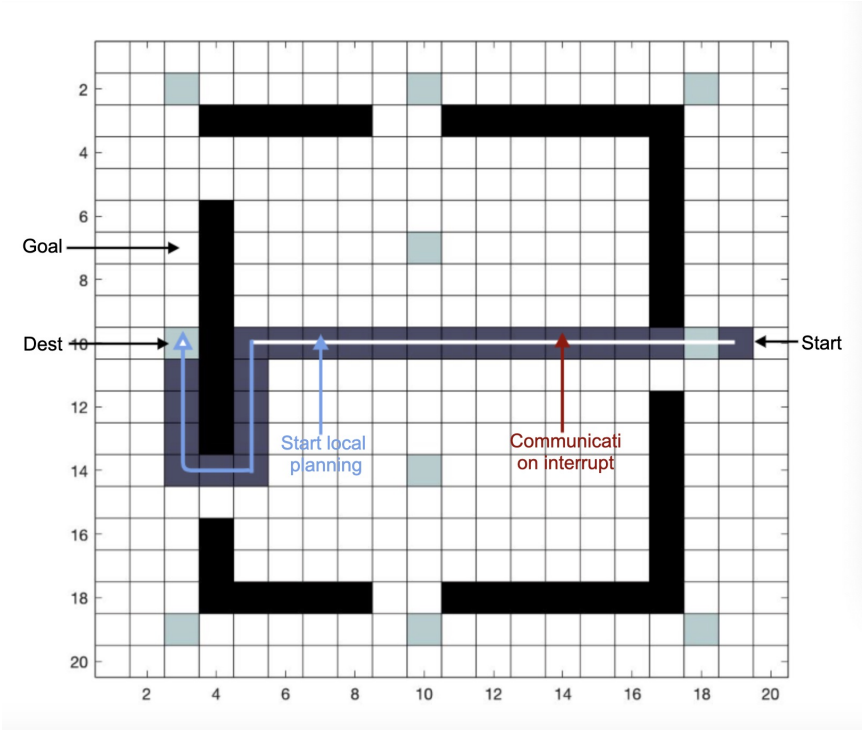


FIGURE 5.18: path with timed-out communication interrupt

5.5 Path using unmatched heuristics

This section is an example of path obtained using the unmatched heuristics on map in Figure 5.11. The movement is constrained in 4 directions, which should use Manhattan distance as the heuristics.

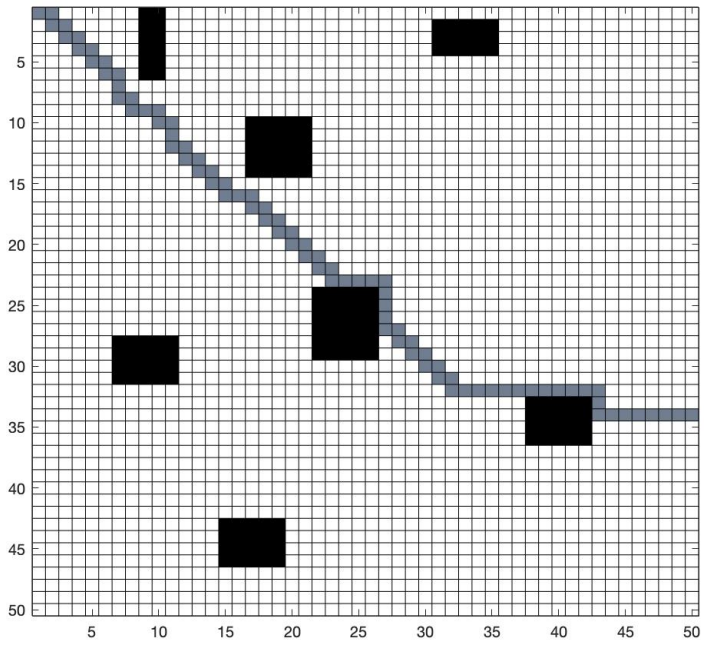


FIGURE 5.19: path with Euclidean distance as heuristics

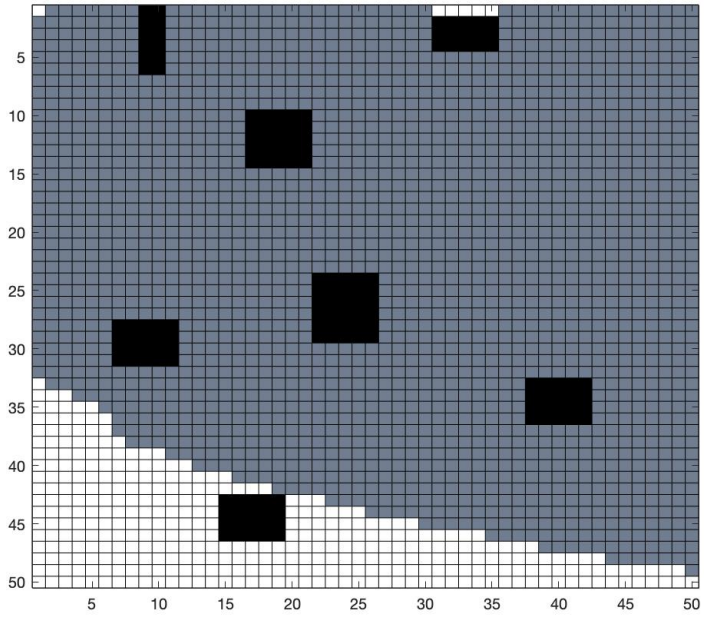


FIGURE 5.20: open list with Euclidean distance as heuristics

Figure 5.19 and Figure 5.20 shows the path and open list when using the Euclidean distance as the heuristics instead of Manhattan distance. Although it meets the constraint that the heuristics never exceeds the actual cost, it doesn't conform to the rules illustrated in Section 2.6.2. Compared with the results in Figure 5.12, it is able to find a shortest path of the same length, but creates a much larger open list due to the improper choice of the heuristic function, which results in larger time and space consumption to find the path.

5.6 Result of 8-direction movement

The previous results are under the configuration of 4-direction movement with Manhattan distance as heuristics. In this section the algorithm is configured to allow movement in 8 directions as in Figure 2.20 with the Diagonal distance as heuristics. The figures below shows the results on the maps of Figure 5.6, Figure 5.11 and Figure 5.6 of the same start and goal cells. The path is marked as a light green broken line, and the open list consists of cells that marked in light grey and the cells on the path. This configuration creates shorter paths, but a larger open list.

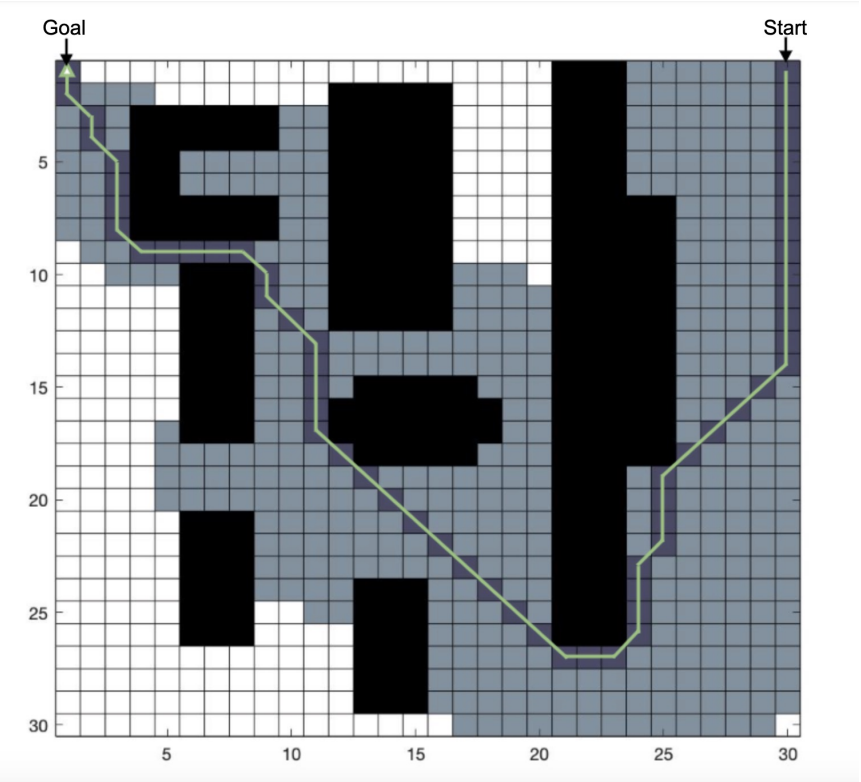


FIGURE 5.21: path and open list

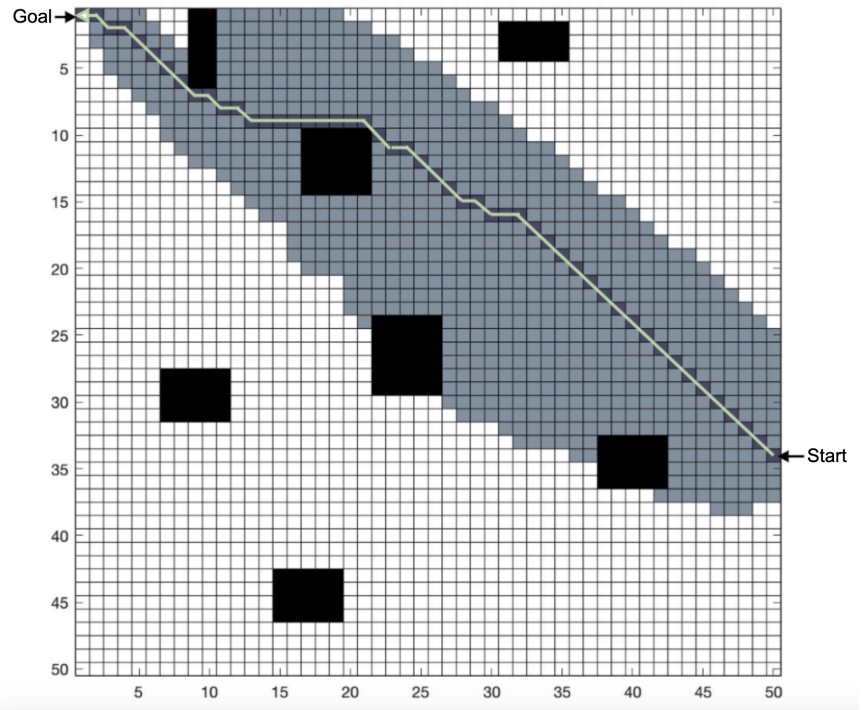


FIGURE 5.22: path and open list

TABLE 5.1: 4-direction movement

Map Size	Open List	Num of Traversal	Search Time(ms)
20x20-ring	76	55	110
20x20-ring-holes	135	137	120
30x30	276	212	158
50x50	200	82	124

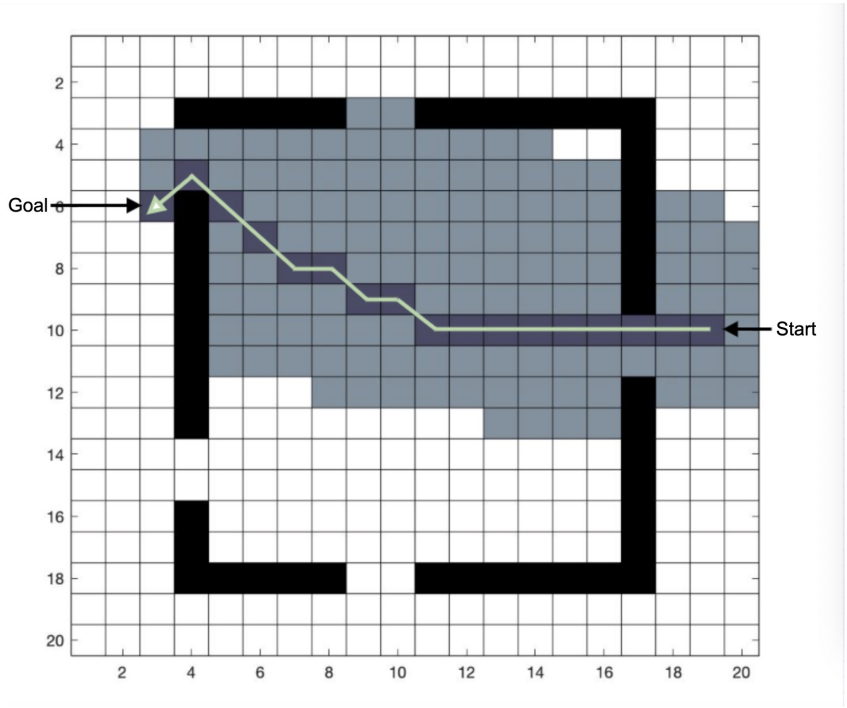


FIGURE 5.23: path and open list

5.7 Time and space consumption

This section analyses the execution time and the space occupation of the path-finding algorithm on 4 maps of 2 different configurations. The columns in the table below shows the maximum size of the open list, how many times the open list is traversed in total, and the time consumption to find the path for each map.

The heuristic function has a major effect on the practical performance of the search algorithm. Table 5.2 shows the result using the Diagonal distance as the heuristics, which is usually less accurate than the Manhattan distance used in Table 5.1. When there is no obstacle on the map, Manhattan distance is always equal to the actual distance of the path, while the Diagonal distance is equal to the actual distance only when the path is horizontal or vertical. This inaccuracy results in a larger open list and more traversals of the open list, thus longer time to find the path.

TABLE 5.2: 8-direction movement

Map Size	Open List	Num of Traversal	Search Time(ms)
20x20-ring	107	120	122
20x20-ring-holes	170	264	194
30x30	466	687	404
50x50	1332	2043	947

Although in general the execution time of the algorithm increases as the size of the map grows, it is also closely related to the distribution of the obstacles and the start and goal position. In Table 5.1 for instance, the execution time on 50x50 map is less than the 30x30 one, because the distribution of obstacles on the 50x50 map is much more sparse and less complex than on the 30x30 map.

Chapter 6

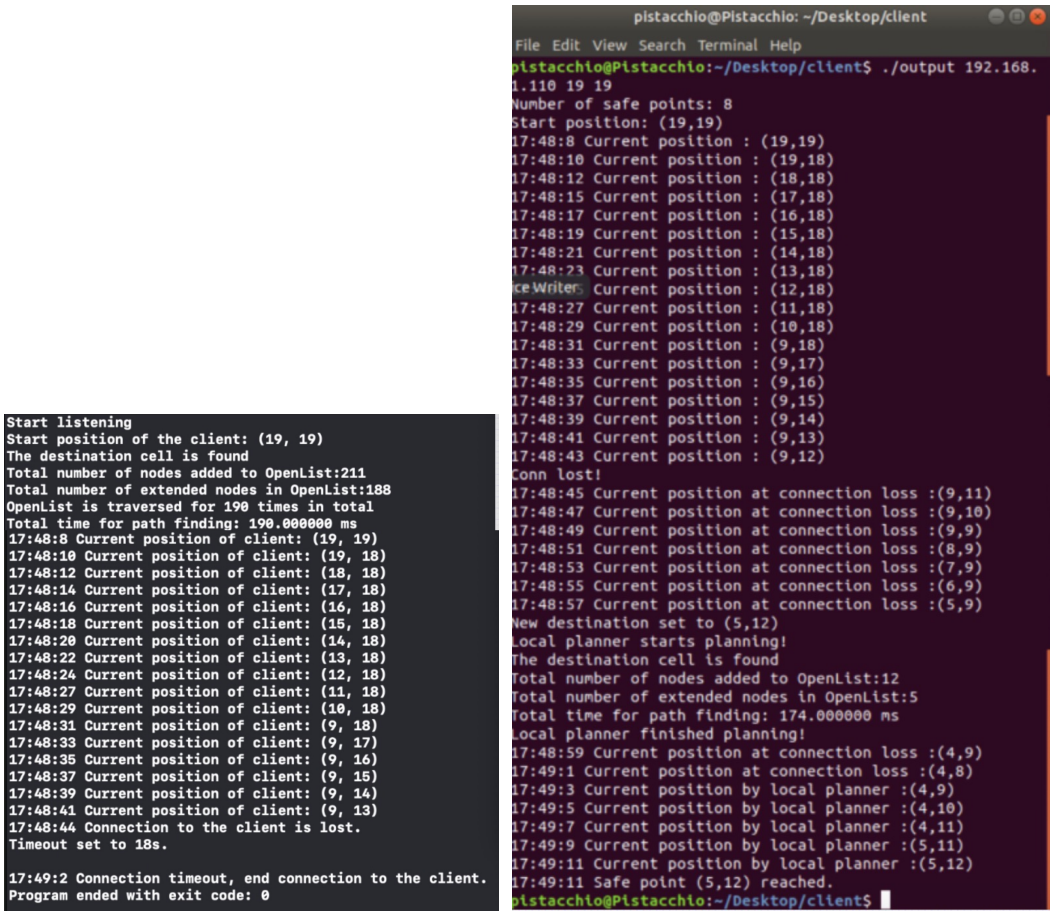
Conclusions

The simulation and network test results show that the path planning algorithm is able to find an optimal path on the grid maps, and the server-client architecture works as expected that the client receives the way points on the path one by one sent by the server. Both sides can detect the network failure, for a short communication interrupt the client is able to make the vehicle keep up with the original path, and both sides deal properly with the communication resumption. While under the circumstance of a long communication interrupt, the client turns off-line, either the local planner is launched to schedule a path to a new destination so that the vehicle can go to a nearest safe point, or the vehicle can still reach the original goal when it is already close to the goal.

Figure 6.1 shows an example of test result of a long communication interrupt on both the server and the simulated vehicle side. A 18s timeout is set when the server detects a network failure. After 18s the connection is not yet resumed, the server closes the socket thus terminates the connection with the vehicle.

In the meanwhile, the vehicle would first of all follow the sequence of the latest way points in its receiving buffer received from the server before the network failure, when the sequence in the receiving buffer has almost reaches the end but the connection is not yet resumed, the backup local planner is launched and a new path to the nearest safe point is generated. The vehicle then turns to the new path and stops at the new destination.

Figure 6.2 visualizes the test result in Figure 6.1 to provide a straightforward representation of the vehicle behaviour.



(a) Test result on server (b) Test result on simulated vehicle

FIGURE 6.1: Test result

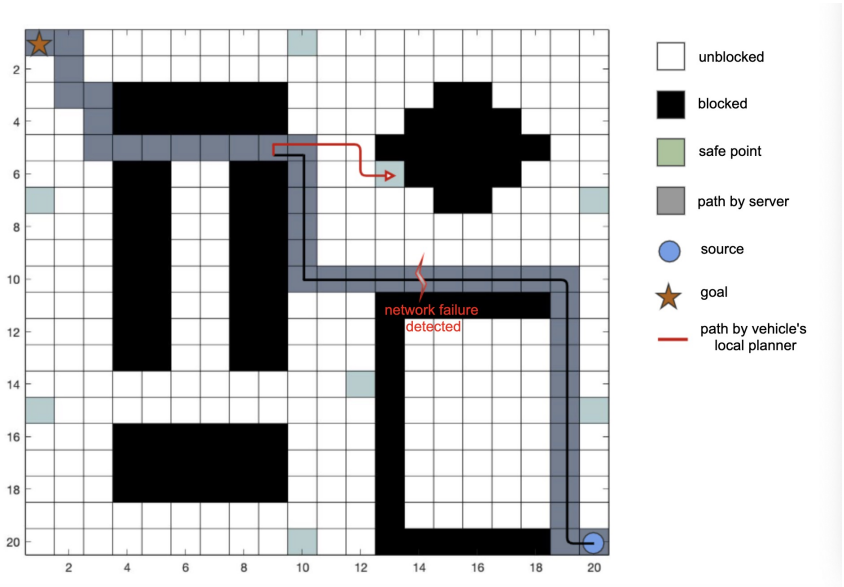


FIGURE 6.2: Visualized test result on 20x20 map with a long communication interrupt

There are also a few additional works that can be done in the future:

- **Enhance searching efficiency.** The time complexity of A* depends on the heuristic. It is important to properly choosing the heuristic function so that the heuristics is as close to the actual distance as possible. The other way is optimizing the data structure of the open list, for example adding a hash table for the open list, or implementing the open list as a Fibonacci heap, which could greatly reduce the time of the find-minimum operation.
- **Trajectory planning.** The path obtained by the A* algorithm is not a smooth curve, it needs to be post-processed to remove the jagged movement. Also the dynamics of the vehicle should be taken into account to transform the path to a smooth and flyable trajectory.
- **Introduction of Database.**The grid maps and safe points list are stored as a file on the server and client, and another file is created by the program to record the path. It is possible to add a database to make the data more organized if simultaneous control of multiple vehicles is to be implemented in the future.

Appendix A

List of files

On Server:

- a_star.h : header file of a_star.c.
- a_star.c : implementation of A* algorithm.
- stack.h : header file of stack.c.
- stack.c : implementation of stack.
- main.c : implementation of server workflow.
- map.txt : grid map represented in 0s and 1s.

On Client:

- a_star.h : header file of a_star.c.
- a_star.c : implementation of A* algorithm.
- stack.h : header file of stack.c.
- stack.c : implementation of stack.
- main.c : implementation of client workflow.
- map.txt : grid map represented in 0s and 1s.
- safe_points.txt : safe points list represented in the index of grid.

Bibliography

[1] Mario Arzamendia Lopez. *Ph.D. Dissertation - Reactive Evolutionary Path Planning for Autonomous Surface Vehicles in Lake Environments*. Mar. 2019.

[2] Norlida Buniyamin et al. "A simple local path planning algorithm for autonomous mobile robots". In: *International Journal of Systems Applications, Engineering Development* 5 (Jan. 2011), pp. 151–159.

[3] J. Burlet, O. Aycard, and T. Fraichard. "Robust motion planning using Markov decision processes and quadtree decomposition". In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*. Vol. 3. 2004, 2820–2825 Vol.3.

[4] Howie Choset. "Incremental Construction of the Generalized Voronoi Diagram, the Generalized Voronoi Graph, and the Hierarchical Generalized Voronoi Graph". In: *Choset-1997-14497* (Sept. 1997).

[5] David Ferguson, Maxim Likhachev, and Anthony (Tony) Stentz. "A Guide to heuristic-based path planning". In: *International Conference on Automated Planning and Scheduling (ICAPS)* (May 2005).

[6] Giovanni Filomeno et al. "Construction site pedestrian simulation with moving obstacles". In: Sept. 2016.

[7] Giancarlo Fortino et al. "Cloud-assisted body area networks: state-of-the-art and future challenges". In: *Wireless Networks* 20.7 (Oct. 2014), pp. 1925–1938. ISSN: 1022-0038. DOI: 10.1007/s11276-014-0714-1. URL: http://link.springer.com/10.1007/s11276-014-0714-1.

[8] Jared Giesbrecht. "Global Path Planning for Unmanned Ground Vehicles". In: (Dec. 2004), p. 56.

[9] C. Goerzen, Z. Kong, and B Mettler. "A Survey of Motion Planning Algorithms from the Perspective of Autonomous UAV Guidance". In: *Journal of Intelligent and Robotic Systems* 65.57 (Nov. 2010). DOI: 10.1007/s10846-009-9383-1. URL: https://link.springer.com/article/10.1007/s10846-009-9383-1.

[10] Crina Grosan and Ajith Abraham. "Informed (Heuristic) Search". In: *Part of the Intelligent Systems Reference Library book series* 17.ISRL ().

[11] Jawhar Imad et al. “Communication and Networking of UAV-Based Systems: Classification and Associated Architectures”. In: *Journal of Network and Computer Applications* 84.10.1016/j.jnca.2017.02.008 (Feb. 2017).

[12] Adeel Javaid. “Understanding Dijkstra Algorithm”. In: *SSRN Electronic Journal* (Jan. 2013). DOI: 10.2139/ssrn.2340905.

[13] Ronnie Johansson. “Intelligent Motion Planning for a Multi-Robot System”. In: (Nov. 2003).

[14] Arpan Kumar Kar et al. *Advances in Smart Cities: Smarter People, Governance, and Solutions*. Ed. by CRC Press. 2017, p. 217. ISBN: 1351652117.

[15] Sven Koenig, Maxim Likhachev, and David Furcy. “Lifelong Planning A*”. In: *Artificial Intelligence Journal* 155 (Jan. 2004).

[16] Sumit Kumar, Sumit Dalal, and Vivek Dixit. “The OSI model: Overview on the seven layers of computer networks”. In: *International Journal of Computer Science and Information Technology Research* 2.3 (July 2014). ISSN: 2348-120X.

[17] *Linux manual page*. https://man7.org/linux/man-pages/man2/recv.2.html, recv(2).

[18] *Linux manual page*. https://man7.org/linux/man-pages/man2/accept.2.html, accept(2).

[19] *Linux manual page*. https://man7.org/linux/man-pages/man2/select.2.html, select(2).

[20] *Linux manual page*. https://man7.org/linux/man-pages/man2/setsockopt.2.html, setsockopt(2).

[21] Seda Milos. “Roadmap methods vs. cell decomposition in robot motion planning”. In: Feb. 2007, pp. 127–132.

[22] Rasoul Mojtahedzadeh. “Robot Obstacle Avoidance using the Kinect”. PhD thesis. Aug. 2011.

[23] Kousik Nath. “Understanding TCP internals step by step for Software Engineers and System Designers — Part 1”. In: (2019). URL: https://codeburst.io/understanding-tcp-internals-step-by-step-for-software-engineers-system-designers-part-1-df0c10b86449.

[24] Jung Jun Park, Ji Hun Kim, and Jae-Bok Song. “Path Planning for a Robot Manipulator based on Probabilistic Roadmap and Reinforcement Learning”. In: *International Journal of Control, Automation and Systems* 5.6 (Dec. 2007).

[25] Geraerts R. and Overmars M.H. *A Comparative Study of Probabilistic Roadmap Planners*. 2004. ISBN: 978-3-540-45058-0.

[26] Ana Rodrigues, Pedro Costa, and José Lima. "The K-Framed Quadtrees approach for path planning through a known environment". In: *Advances in Intelligent Systems and Computing* (Nov. 2017).

[27] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. 2002. ISBN: 978-0137903955.

[28] A. Stentz. "Optimal and Efficient Path Planning for Partially-Known Environments". In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation* 4824407 (May 1994).

[29] Anthony Stentz. "The D* Algorithm for Real-Time Planning of Optimal Traverses". In: (Apr. 2011).

[30] Aditya Sundararajan et al. "A Survey of Protocol-Level Challenges and Solutions for Distributed Energy Resource Cyber-Physical Security". In: *Energies* 11 (Sept. 2018), p. 2360. DOI: 10.3390/en11092360.

[31] "TCP/IP VS UDP: WHAT'S THE DIFFERENCE?" In: (). URL: https://www.colocationamerica.com/blog/tcp-ip-vs-udp.

[32] Felipe Gonzalez Toro and Antonios Tsourdos. *UAV or Drones for Remote Sensing Applications*. 2018, p. 260. ISBN: 3038971111.

[33] A. Tsourdos, B. White, and M. Shanmugavel. *Cooperative Path Planning of Unmanned Aerial Vehicles*. 2010. ISBN: 9780470741290.

[34] Vojtěch Vonásek et al. "Rapidly-Exploring Random Trees: A New Tool for Path Planning". In: *Part of the Lecture Notes in Control and Information Sciences book series* 396.LNCIS ().