## Theoretical part:

### Task-1:

For the self delivery robot , I have decided to use SBCs to construct the brain of robot instead of microcontrollers. SBC is built with microcontrollers ,I/O and memory and functions like a normal computer .

I can make a simple robot with help of microcontroller . But in self delivery robot ,the robot needs AI algorithms , machine learning , face recognition , computer vision , motion control etc . To implement this I need SBC to run these complex functions . It can run a full operating system like Linux in Raspberry Pi . Also I can program SBC with any programming language . SBC can do most of the functions of a microcontroller in a robot . It connects through the GPI/O pins with other parts of the robots of different functions . that’s why I choose to use SBC to construct the brain of the robot .

### Task-2:

I will choose I2C protocol over UART and SPI to eastablish communication between arduino nano and Arduino mega .

I2C system has two pins SDA and SCL where SPI has four pins and UART has digital two pins .Comparing to SPI , I2C needs fewer wires and  transmits data in a single wire so the pin count remains low . Between the two arduinos I2C uses two wire communication protocol which is less complex than SPI . UART cannot transmit data in two devices at the same time and is slower than I2C as it doesn’t have clock frequency . I2C is unique because each arduino will have it’s own unique address .This feature doesn’t avail in SPI and UART .Though SPI has made possible to connect a master device with maximum four slave devices but I2C has ability to connect multiple masters with multiple slaves through unique byte addresses. Here some basic code snippets of I2C Arduino :

(Ref: docs.arduino.cc)

```
//Arduino mega

#include<Wire.h>

Void setup()
```

```
{
Wire.begin();
Serial.begin(9600);
}
Void loop(){
Wire.beginTransmission();
Wire.write();
Wire.endtransmission();
delay(500);
}
//Arduino nano
#include<Wire.h>
Void setup()
{
Wire,begin();
Wire.onRecieve(recieveevent);
Serial.begin();
}
Void recieveevent(int n)
{
While(Wire.available()){
Char c=Wire.read();
Serial.print(c);
}
```

}

**<u>Task-3:</u>**

A. DC motors are two types .They are brushed DC motors and brushless DC motors. DC motors, brushed, are used in computers,toys,home appliaces,lifts,smartphones and other power tools .Stepper motors, brushless, are used in robotics,printers,cameras,scanners etc. DC motors can run continuously for long time and don't need microcontrollers to handle but need good maintenance . Stepper motors cannot run continuously for long duration and need microcontrollers to handle but do not need constant maintenance as they are brushless .

B. H-bridge motor driver chip controls a bipolar stepper motor or single DC motor . It is used for applications which need bidirectional and different speed control of motors .
Half bridge driver chip is designed to control brushless DC motor or unipolar stepper motor .It is required to control simple speeds and torques of the motors.
Servo motor driver chip is used in servo motors to control  position of the application like robot .
Isolated gate driver chip is used in ac motors to maintain high voltage and high current power switches .
Stepper motor driver chip is used in stepper motors for current control and discrete position detection

C. PWM modifies the average power of a device without any waste by changing the duty cycle . PWM uses high frequency square signals to power on and off to the motor . The average voltage depends on the duty cycle . By changing the duty cycle , the average voltage and current  can be changed and the motor changes its speed and torque .

D. I have to connect encoder outputs to the input digital pins of microcontroller to implement motor feedback . Then write program in microcontroller so than the encoder signals can read . Using PWM I can control the speed and position of the rover .I will use PID controller to synchronize motors in velocity or position .

## Task-4:

1.Motor control: STM32 would be the best choice to control a robot having multiple motors . STM32 is ARM Cortex-M based microcontroller with high performance and low energy consumption . It has various peripherals and debugging interfaces to handle multiple motors .

2.Communication: ESP32 should be used for the long range communication as it has integrated Wifi and dual-mode Bluetooth . Also it has built-in antenna switches and low power system .

3.Real-time operation: STM32 has 32 bit Arm Cortex-M processing core which means high performance, memory and power . It is fast , runs in low power and has peripherals to interface with all kinds of electrical components .

4. Sensor Integration: For home automation project Arduino mega should be used as it is easy to program interface and has many pins to connect with different sensors.

5.Power Efficiency: ESP32 has low power consumption than STP32 and Arduino mega .For battery oriented remote sensor node , ESP32 has wireless communication system and long battery duration which makes the best option for it .

## Task-5:

For a food delivery rover robot, I think IMU, LIDAR, Odometer sensor should be used to control the movement of the robot .

IMU consists of gyroscopes , accelerometers, magnetometers . It provides the angular velocity and specific gravity of the robot . That helps to understand how fast the robot should be rotate during angular turns which causes drift slowly .

Lidar guides the robot by measuring multiple directions on the surface of mars with a combination of 3D and laser scanning . It detects objects in 3D which helps the robot to avoid obstacles and navigates smoothly .

Odometer will be used to detect the change of position of the robot and the new position after a period of time  computing the distance from the starting point . Odometry is error sensitive due to uncertainty in the components of the robot and the surface . These sensors are the best one to choose to use on the robot .

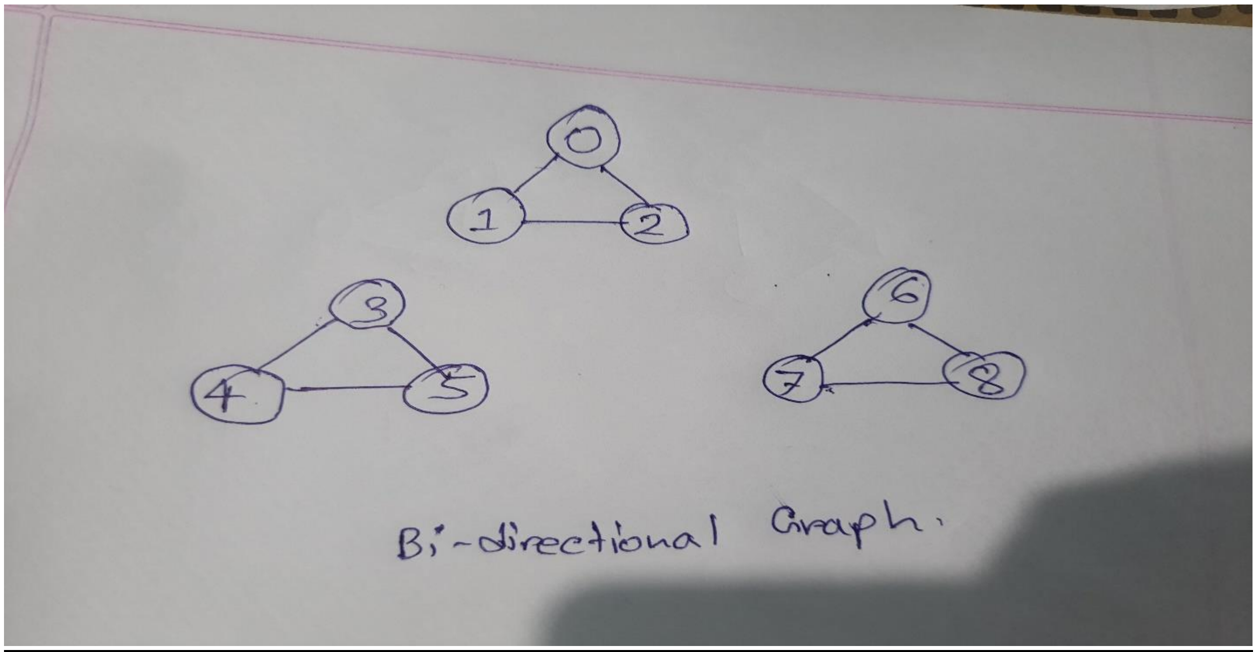
## **<u>Task-6:</u>**

## **<u>Logical Part:</u>**

### **<u>Task1:</u>**

**<u>A.</u>** To check the path if the robot can visit all the location or not we use the concept of Hamiltonian path using DFS  algorithm . This algorithm starts from a point and checks the branches of the path if there is no more branch left then it goes back to the intersection point using recursion . In this way it checks all the path if they are connected to a single path or not .
Githhub link: [https://github.com/nuhb25/altair](https://github.com/nuhb25/altair)

**<u>B.</u>**



## Task-2:

## Dijkstra Algorithm:

This algorithm is best for determining the shortest path of graphs with non-negative weighted edges .It also works in unweighted edges. It finds the node with minimum distance from the source and expanding the adjacent nodes updating the distance values. But this algorithm doesn’t work in negative cycles ,often stuck in infinite loops .It follows greedy approach. The graph size remains medium to large usually . Here the code snippet :

ref: https://www.programiz.com/dsa/dijkstra-algorithm

```
void Dijkstra(int Graph[MAX][MAX], int n, int start) {
 int cost[MAX][MAX], distance[MAX], pred[MAX];
 int visited[MAX], count, mindistance, nextnode, i, j;
 // Creating cost matrix
 for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
```

```
      if (Graph[i][j] == 0)
        cost[i][j] = INFINITY;
      else
        cost[i][j] = Graph[i][j];
  for (i = 0; i < n; i++) {
    distance[i] = cost[start][i];
    pred[i] = start;
    visited[i] = 0;
  }
  distance[start] = 0;
  visited[start] = 1;
  count = 1;
  while (count < n - 1) {
    mindistance = INFINITY;

    for (i = 0; i < n; i++)
      if (distance[i] < mindistance && !visited[i]) {
        mindistance = distance[i];
        nextnode = i;
      }
    visited[nextnode] = 1;
    for (i = 0; i < n; i++)
      if (!visited[i])
        if (mindistance + cost[nextnode][i] < distance[i]) {
          distance[i] = mindistance + cost[nextnode][i];
          pred[i] = nextnode;
        }
    count++;
  }
  // Printing the distance
  for (i = 0; i < n; i++)
    if (i != start) {
      printf("\nDistance from source to %d: %d", i, distance[i]);
    }
}
```

## **Bellman Ford Algorithm:**

Bellman ford algorithm works in both negative and non-negative weighted graphs to find the shortest path from single source. It is easily implemented and taken from dynamic programming approach . In non-negative weighted graph it has more higher time complexity [0(VE)] than Dijkstra’s algorithm . It is used best to detect negative weight cycles . Code:

Ref: https://www.programiz.com/dsa/bellman-ford-algorithm

```
void BellmanFord(struct Graph* graph, int u) {
 int V = graph->V;
 int E = graph->E;
 int dist[V];
 // Step 1: fill the distance array and predecessor array
 for (int i = 0; i < V; i++)
  dist[i] = INT_MAX;

 // Mark the source vertex
 dist[u] = 0;
 // Step 2: relax edges |V| - 1 times
 for (int i = 1; i <= V - 1; i++) {
  for (int j = 0; j < E; j++) {
   // Get the edge data
   int u = graph->edge[j].u;
   int v = graph->edge[j].v;
   int w = graph->edge[j].w;
   if (dist[u] != INT_MAX && dist[u] + w < dist[v])
    dist[v] = dist[u] + w;
  }
 }
 // Step 3: detect negative cycle
 // if value changes then we have a negative cycle in the graph
 // and we cannot find the shortest distances
 for (int i = 0; i < E; i++) {
  int u = graph->edge[i].u;
  int v = graph->edge[i].v;
```

```
    int w = graph->edge[i].w;
    if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
      printf("Graph contains negative w cycle");
      return;
    }
  }
  // No negative weight cycle found!
  // Print the distance and predecessor array
  printArr(dist, V);
  return;
}
```

## Floyd Warshall Algorithm:

Floyd's algorithm is most efficient computing the shortest path between all the pair of vertices . It works on negative edge and can detect negative cycles but fails to work . It has the highest time complexity than the other shortest path algorithms but has smaller graphs . It follows dynamic programming approach .The main formula is Shortestpath(i,j,k)=min(Shortestpath(i,j,k-1), Shortestpath(i,k,k-1)+ Shortestpath(k,j,k-1))

Code: https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

```
void floydwarshall()
{
int cost [N][N];
int i, j, k;
for(i=0; i<N; i++)
for(j=0; j<N; j++)
cost [i][j]= cost Mat [i] [i];
for(k=0; k<N; k++)
{
for(i=0; i<N; i++)
for(j=0; j<N; j++)
if(cost [i][j]> cost [i] [k] + cost [k][j];
cost [i][j]=cost [i] [k]+'cost [k] [i]:
```

}

## A* Algorithm:

This algorithm is best for finding the lowest cost function distance from the source in a 2D or grid based system . It follows the formula: f(n)=g(n)+h(n) . Here f(n) is the heuristic function that calculates the cost of the cheapest path from a specific node to goal and g(n) is from the source to a specific node . It is faster than Dijkstra as it depends on a heuristic function to guide towards goal .

It doesn’t work on negative weighted graphs and negative cycles .

Code:

Ref: https://en.wikipedia.org/wiki/A*_search_algorithm#Pseudocode

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from the start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
```

```
gScore[start] := 0


// For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
// how cheap a path could be from start to finish if it goes through n.
fScore := map with default value of Infinity
fScore[start] := h(start)


while openSet is not empty
    // This operation can occur in O(Log(N)) time if openSet is a min-heap or a priority queue
    current := the node in openSet having the lowest fScore[] value
    if current = goal
        return reconstruct_path(cameFrom, current)


    openSet.Remove(current)
    for each neighbor of current
        // d(current,neighbor) is the weight of the edge from current to neighbor
        // tentative_gScore is the distance from start to the neighbor through current
        tentative_gScore := gScore[current] + d(current, neighbor)
        if tentative_gScore < gScore[neighbor]
            // This path to neighbor is better than any previous one. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := tentative_gScore + h(neighbor)
            if neighbor not in openSet
                openSet.add(neighbor)


// Open set is empty but goal was never reached
return failure
```

## Microcontroller Part :

### Task-1:

**https://www.tinkercad.com/things/7hMRXUqbZ4G**

To connect two Arduino uno through I2C protocol I have to connect them with wire in SDA and SCL pins which is A4 and A5 pin headers on the boards . Arduino 1 is the message sender and Arduino 2 is the receiver . Arduino 1 sends the message to arduino 2 through the unique peripherals . For repeating the message I have to add “Wire.read()” show in serial monitor else it prints once .

### Task-2:

**https://www.tinkercad.com/things/fDnsjRpxT4v**