# Assignment 03

**Name -** Sk Fardeen Hossain
**Enrollment No -** 2021CSB023
**Department -** Computer Science and Technology

## Question 01:-

1. A disaster has struck a **futuristic smart city**, and your task is to develop an **AI-powered drone navigation system** to **rescue stranded survivors**. The drone must **autonomously navigate through the city**, avoiding obstacles like collapsed buildings and fire zones while optimizing for the fastest route (equivalent to minimum energy consumption).

   Your objective is to implement the *A Search Algorithm** to guide the drone from its **starting position (S)** to **rescue multiple survivors (G1, G2, ... , Gn) (intermediate states) and return to base (B) (treated as goal state)** while minimizing energy consumption and ensuring safe traversal.

### Grid Representation of the City (State Space Representation of the City):

The smart city is represented as a **3D grid** where each $(x, y, z)$ coordinate defined as: $(x, y)$ in the 2D coordinate and z values is the depth of flying at different levels. Each of the depth '$z$' is represented by different notations as:

- **Roads & Open Spaces** (`0`) → The drone can fly here.
- **Buildings/Collapsed Structures** (`1`) → The drone **cannot** pass through.
- **Fire Zones** (`F`) → High-risk zones. Passing through **incurs an extra cost**.
- **Survivor Locations** ($G_1, G_2, \ldots, G_n$) → The drone must **visit and rescue them**.
- **Recharging Stations** (`R`) → The drone can **stop and recharge energy** if needed.
- **Base Station** (`B`) → The drone must **return here after completing the mission**.
- **Drone's Start Position** (`S`) → Where the drone begins.

### Drone Movement and Cost Considerations:

a) **The drone moves in 3D space**:
   - **Up** `(x, y, z+1)`, **Down** `(x, y, z-1)`,
   - **North** `(x-1, y, z)`, **South** `(x+1, y, z)`,
   - **East** `(x, y+1, z)`, **West** `(x, y-1, z)`.
   - **Diagonal movements** (like flying at an angle) **are not allowed** for simplicity.

b) **Energy Consumption Factor:**
   - Moving through **clear space (0)** costs **1 energy unit**.
   - Moving through **fire zones (F)** costs **3 energy units** due to turbulence.
   - Moving **upwards (z+1) costs extra 2 energy units**, while descending `(z-1)` costs **1 energy unit**.

c) **Objective:**
   - The drone **must rescue all survivors** before returning to base `(B)`.
   - The **optimal path minimizes total energy consumption**.

**Implementation Guidelines:**

a) *Use A search algorithm** to compute the most energy-efficient path.
b) **Priority Queue (Min-Heap)** should be used to store nodes with the lowest `f(n) = g(n) + h(n)`.
c) **Heuristic (h(n))**: Use a **3D Euclidean distance** to estimate cost to the goal:

$$h(n) = \sqrt{(x_{current} - x_{goal})^2 + (y_{current} - y_{goal})^2 + (z_{current} - z_{goal})^2}$$

d) **Account for terrain cost penalties** (e.g., fire zones, vertical movement, etc.).
e) **Recharging Logic**: If energy is too low to reach the next goal, navigate to the nearest `(R)` station.
f) **Ensure the algorithm supports multi-goal search**, as the drone must **visit multiple survivors** before returning.
g) **Handle Edge Cases**:
   - No valid path exists to all survivors.
   - Energy depletion before reaching a recharge station.
   - The base `(B)` is unreachable due to obstacles.

# Idea :

*It is important to note that I have not implemented the solution to the bonus challenges due to lack of specification, like wind cost, drone coordination cost, etc*

*It is also important to note that since there is no mention of time penalty for recharge when fuel exhausts (which never does), we have considered visiting recharge stations as merely fulfilling the fuel. Also since there is no input for initial fuel capacity, the concept refueling does not make sense here.*

Our target will be to use minimise the f_cost, where
f_cost=g_cost_h_cost,
g_cost -> Actual fuel consumed as per the constraints
h_cost -> This consists of 2 points,

1. When there are still people to rescue, the euclidean heuristic is between the current node position and the closest person position.

2. When all the people are rescued, the euclidean heuristic is between the current node position and the goal node position.

Then we apply conventional A* search algorithm, maintaining a priority_queue that monitors the f_cost, and chooses nodes having the smallest f_cost, and if a path is found we terminate the algorithm. Appropriate edge case handling has already been embedded in the code.

## Code :

```
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>
#include <unordered_map>
#include <sstream>
#include <limits>
#include <algorithm>

using namespace std;

struct Coord
{
   int x, y, z;
   bool operator==(const Coord &other) const
   {
       return x == other.x && y == other.y && z == other.z;
   }
};

string makeKey(const Coord &coord, int mask)
{
   ostringstream oss;
   oss << coord.x << "," << coord.y << "," << coord.z << "," << mask;
   return oss.str();
}
```

```
struct Node
{
   Coord coord;         // current coordinate (x, y, z)
   int mask;            // bitmask indicating which rescue goals have been
visited
   double g, h, f;      // g: cost so far, h: heuristic, f: total cost
   vector<Coord> path; // full 3D path (list of coordinates)

   bool operator>(const Node &other) const
   {
       return f > other.f;
   }
};

class AStarDrone
{
private:
   vector<vector<vector<char>>> grid;
   vector<Coord> rescueGoals;
   Coord base;
   int nGoals;

   int dx[6] = {-1, 1, 0, 0, 0, 0};
   int dy[6] = {0, 0, -1, 1, 0, 0};
   int dz[6] = {0, 0, 0, 0, -1, 1};

   double euclidean(const Coord &a, const Coord &b)
   {
       return sqrt(pow(a.x - b.x, 2) +
                   pow(a.y - b.y, 2) +
                   pow(a.z - b.z, 2));
   }

   // Multi-goal heuristic:
   // If not all rescue goals are visited, h = distance(current, nearest
unvisited goal) + distance(nearest goal, base)
   // Otherwise, h = distance(current, base)
   double heuristic(const Coord &current, int mask)
   {
       if (mask == (1 << nGoals) - 1)
```

```
            return euclidean(current, base);
        double best = numeric_limits<double>::max();
        double bestToBase = 0.0;
        for (int i = 0; i < nGoals; i++)
        {
            if (!(mask & (1 << i)))
            {
                double d = euclidean(current, rescueGoals[i]);
                if (d < best)
                {
                    best = d;
                    bestToBase = euclidean(rescueGoals[i], base);
                }
            }
        }
        return best + bestToBase;
    }

    // For free space ('0') cost is 1, fire ('F') cost is 3.
    // Additionally, moving upward (dz = 1) adds +2, moving downward (dz =
-1) adds +1.
    int getCost(char terrain, int dz_move)
    {
        int baseCost = (terrain == 'F' ? 3 : 1);
        if (dz_move == 1)
            return baseCost + 2;
        else if (dz_move == -1)
            return baseCost + 1;
        else
            return baseCost;
    }

public:
    AStarDrone(vector<vector<vector<char>>> cityGrid, vector<Coord>
rescueGoals, Coord base)
        : grid(cityGrid), rescueGoals(rescueGoals), base(base)
    {
        nGoals = rescueGoals.size();
    }
```

```
   // Returns a pair: (path, totalEnergyCost). An empty path indicates
failure.
   pair<vector<Coord>, double> findPath(Coord start)
   {
       priority_queue<Node, vector<Node>, greater<Node>> pq;
       unordered_map<string, double> visited; // state key -> best f cost

       // Start state: no rescue goals visited.
       Node startNode{start, 0, 0.0, heuristic(start, 0), heuristic(start,
0), {start}};
       pq.push(startNode);

       while (!pq.empty())
       {
           Node current = pq.top();
           pq.pop();

           string key = makeKey(current.coord, current.mask);
           if (visited.find(key) != visited.end() && visited[key] <=
current.f)
               continue;
           visited[key] = current.f;

           for (int i = 0; i < nGoals; i++)
           {
               if (!(current.mask & (1 << i)) && current.coord ==
rescueGoals[i])
               {
                   current.mask |= (1 << i);
               }
           }
           if (current.mask == (1 << nGoals) - 1 && current.coord == base)
               return {current.path, current.g};

           for (int i = 0; i < 6; i++)
           {
               Coord next{current.coord.x + dx[i],
                          current.coord.y + dy[i],
                          current.coord.z + dz[i]};
```

```
                if (next.x < 0 || next.y < 0 || next.z < 0 ||
                    next.x >= grid.size() ||
                    next.y >= grid[0].size() ||
                    next.z >= grid[0][0].size())
                    continue;
                if (grid[next.x][next.y][next.z] == '1')
                    continue;

                int moveCost = getCost(grid[next.x][next.y][next.z],
dz[i]);
                double gNew = current.g + moveCost;
                double hNew = heuristic(next, current.mask);
                double fNew = gNew + hNew;

                vector<Coord> newPath = current.path;
                newPath.push_back(next);

                Node neighbor{next, current.mask, gNew, hNew, fNew,
newPath};
                string nKey = makeKey(neighbor.coord, neighbor.mask);
                if (visited.find(nKey) == visited.end() || visited[nKey] >
neighbor.f)
                {
                    pq.push(neighbor);
                }
            }
        }
        return {{}, 0.0};
    }
};

string cellDescription(char cell)
{
    switch (cell)
    {
    case 'S':
        return "Start";
    case '0':
        return "Free space";
    case 'F':
```

```
        return "Fire zone";
    case 'G':
        return "Rescue";
    case 'R':
        return "Recharge";
    case 'B':
        return "Base";
    default:
        return "";
    }
}

int moveCost(const vector<vector<vector<char>>> &grid, const Coord &next,
int dz)
{
    char cell = grid[next.x][next.y][next.z];
    int baseCost = (cell == 'F' ? 3 : 1);
    if (dz == 1)
        return baseCost + 2;
    else if (dz == -1)
        return baseCost + 1;
    else
        return baseCost;
}

int main()
{
    int layers, n, m;
    cout << "Enter the number of layers:\n";
    cin >> layers;
    cout << "Enter the width:\n";
    cin >> m;
    cout << "Enter the length:\n";
    cin >> n;
    vector<vector<vector<char>>> grid(n, vector<vector<char>>(m,
vector<char>(layers)));
    vector<Coord> rescueGoals;
    Coord start = {-1, -1, -1};
    Coord base = {-1, -1, -1};
    for (int i = 0; i < layers; i++)
```

```
{
    cout << "Layer " << i << ":\n";
    for (int j = 0; j < n; j++)
    {
        for (int k = 0; k < m; k++)
        {
            cout << "Cell <" << j << ", " << k << "> : ";
            cin >> grid[j][k][i];
            if (grid[j][k][i] == 'B')
                base = {j, k, i};
            else if (grid[j][k][i] == 'S')
                start = {j, k, i};
            else if (grid[j][k][i] == 'G')
                rescueGoals.push_back({j, k, i});
        }
    }
}
cout << "The Grid is:\n";
for (int i = 0; i < layers; i++)
{
    cout << "Layers " << i << ": \n";
    for (int j = 0; j < n; j++)
    {
        for (int k = 0; k < m; k++)
        {
            cout << grid[j][k][i] << " ";
        }
        cout << "\n";
    }
}
if (start.x == -1)
{
    cout << "No path found." << "\n";
    return 0;
}
AStarDrone drone(grid, rescueGoals, base);

auto result = drone.findPath(start);
vector<Coord> path = result.first;
double totalEnergy = result.second;
```

```
    if (path.empty())
    {
        cout << "No path found." << "\n";
    }
    else
    {
        cout << "Optimal Path Using A*:" << "\n"
             << "\n";
        double energySum = 0.0;
        for (size_t i = 0; i < path.size(); i++)
        {
            if (i == 0)
            {
                cout << "Step " << i + 1 << ": Start at (" << path[i].x <<
", "
                     << path[i].y << ", " << path[i].z << ") -> "
                     <<
cellDescription(grid[path[i].x][path[i].y][path[i].z])
                     << " (Energy used: 0)" << "\n";
            }
            else
            {
                int dx = path[i].x - path[i - 1].x;
                int dy = path[i].y - path[i - 1].y;
                int dz = path[i].z - path[i - 1].z;
                int cost = moveCost(grid, path[i], dz);
                energySum += cost;
                string moveDesc = "";
                if (dz == 1)
                    moveDesc = "Climb to ";
                else if (dz == -1)
                    moveDesc = "Descend to ";
                else
                    moveDesc = "Move to ";

                string cellDesc =
cellDescription(grid[path[i].x][path[i].y][path[i].z]);
                cout << "Step " << i + 1 << ": " << moveDesc << "(" <<
path[i].x << ", "
```

```
                    << path[i].y << ", " << path[i].z << ") -> "
                    << cellDesc << " (Energy used: " << cost << ")" <<
"\n";
            }
        }
        cout << "\n";
        cout << "Total Energy Used: " << totalEnergy << " units" << "\n";
        cout << "Rescue Status: ";
        for (size_t i = 0; i < rescueGoals.size(); i++)
        {
            cout << "G" << i + 1 << " ";
            if (i < rescueGoals.size() - 1)
                cout << ", ";
        }
        cout << "\n";
        cout << "Mission Completion: Success! The drone returned to base ("
             << base.x << ", " << base.y << ", " << base.z << ")" << "\n";
    }

    return 0;
}
```

## Input :

The (5x5x3) grid as mentioned in the question, has been given as an input.
The number indices for the recharge and rescue cells has not been provided to provide simplicity.

*Also the important thing to note is that the optimal path and cost for the given input in the question is given as 23 and the input moves are also not optimal, so maybe there is a possibility of miscommunication regarding the specifications mentioned in the question.*

# Output :

```
The Grid is:
Layers 0:
S 0 1 0 0
0 F 1 1 G
F 0 0 0 F
R 1 1 1 0
B 0 1 0 G
Layers 1:
0 0 1 0 0
0 1 1 0 0
F 0 0 0 1
0 1 1 1 0
0 0 0 0 0
Layers 2:
1 0 0 0 0
0 F 0 1 0
0 0 1 0 F
0 0 0 0 0
0 1 0 0 1
Optimal Path Using A*:

Step 1: Start at (0, 0, 0) -> Start (Energy used: 0)
Step 2: Move to (1, 0, 0) -> Free space (Energy used: 1)
Step 3: Move to (2, 0, 0) -> Fire zone (Energy used: 3)
Step 4: Move to (2, 1, 0) -> Free space (Energy used: 1)
Step 5: Move to (2, 2, 0) -> Free space (Energy used: 1)
Step 6: Move to (2, 3, 0) -> Free space (Energy used: 1)
Step 7: Move to (2, 4, 0) -> Fire zone (Energy used: 3)
Step 8: Move to (1, 4, 0) -> Rescue (Energy used: 1)
Step 9: Move to (2, 4, 0) -> Fire zone (Energy used: 3)
Step 10: Move to (3, 4, 0) -> Free space (Energy used: 1)
Step 11: Move to (4, 4, 0) -> Rescue (Energy used: 1)
Step 12: Climb to (4, 4, 1) -> Free space (Energy used: 3)
Step 13: Move to (4, 3, 1) -> Free space (Energy used: 1)
Step 14: Move to (4, 2, 1) -> Free space (Energy used: 1)
Step 15: Move to (4, 1, 1) -> Free space (Energy used: 1)
Step 16: Move to (4, 0, 1) -> Free space (Energy used: 1)
Step 17: Descend to (4, 0, 0) -> Base (Energy used: 2)

Total Energy Used: 25 units
Rescue Status: G1 , G2
Mission Completion: Success! The drone returned to base (4, 0, 0)
```