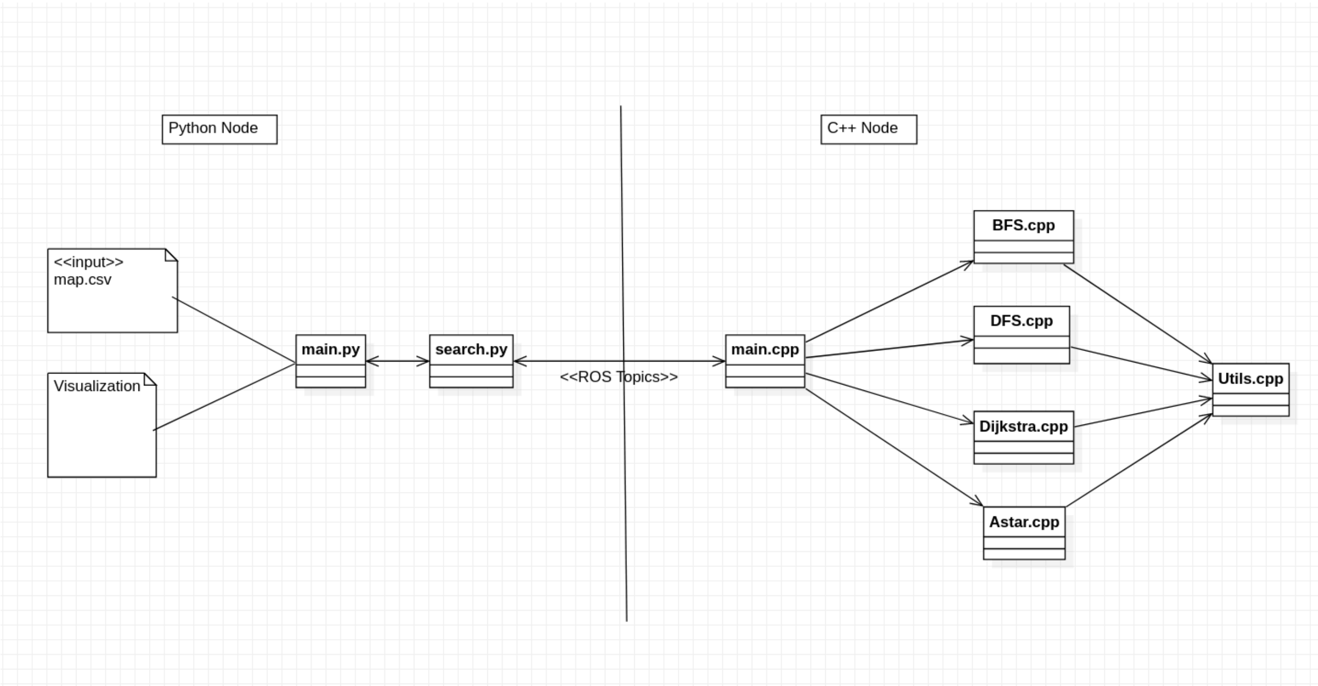
# RBE 550 – Motion Planning

# Programming Assignment -1

Submitted by:
Emmanuel Jayaraju
ejayaraju@wpi.edu

For this assignment, the algorithms are implemented using C++. Here's a high-level picture of how the files are organized and how the functions are grouped together to avoid duplicating the code.




- On the left-side
  - main.py: Reads the grid from the map.csv and visualizes the output as figures
  - search.py: This script acts as the client which obtains the output from C++ nodes on right side via ros topics

- On the right-side
  - main.cpp: The ROS node which uses the algorithm requested by search.py to search the given graph for a path and compute the nodes visited until goal is found.

  - BFS.cpp: This file contains the BFS algorithm implementation.
  - DFS.cpp: This file contains the DFS algorithm implementation.
  - Dijkstra.cpp: This file contains the Dijkstra algorithm implementation.
  - Astar.cpp: This file contains the Astar algorithm implementation.

  - Utils.cpp: This file contains the common functions used by all the above algorithms are grouped here to avoid code duplication. The common functions are:
    - `initialize_graph`: To create the graph and do initialization before starting the search process
    - `find_valid_neighbors`(): To find valid neighbours around a given node in a given graph
    - `generate_path`: Once the graph has been updated with information to generate path from start to goal, this function generates the path to return to the python script.

# Common Functions

Here's the code snippets for each of the common functions explained above:

**`initialize_graph()`**

```
void Utils::initialize_graph(const Eigen::MatrixXi& grid,
                             MatrixXNode& graph)
{
    int rows = grid.rows();
    int cols = grid.cols();

    // Fill the graph matrix with Nodes corresponding to grid information
    for (int i=0; i<rows; ++i)
    {
        for (int j=0; j<cols; ++j)
        {
            graph(i,j) = make_shared<Node>(i, j, grid(i,j));
        }
    }
}
```

…

INPUTS:
grid

OUTPUT:
graph

EXPLANATION:
For a given grid, this function creates the graph for the algorithms to be applied on it.
I'm using Eigen C++ library to create the graph as a 2D matrix.

**find_valid_neighbors()**

```cpp
void Utils::find_valid_neighbors(const MatrixXNode& graph,
                                 const shared_ptr<Node> current_node,
                                 vector<pair<shared_ptr<Node>,int>>& neighbors_w_list)
{
    neighbors_w_list.clear();

    int total_rows = graph.rows();
    int total_cols = graph.cols();

    int row = current_node->row;
    int col = current_node->col;

    // Right neighbor
    if ( (col+1)< total_cols && !graph(row, col+1)->obs)
    {
        auto R = graph(row, col+1);
        neighbors_w_list.push_back(make_pair(R, current_node->edge_w[0]));
    }

    // Down neighbor
    if ( (row+1)< total_rows && !graph(row+1, col)->obs)
    {
        auto D = graph(row+1, col);
        neighbors_w_list.push_back(make_pair(D, current_node->edge_w[1]));
    }

    // Left neighbor
    if ( (col-1)> 0 && !graph(row, col-1)->obs)
    {
        auto L = graph(row, col-1);
        neighbors_w_list.push_back(make_pair(L, current_node->edge_w[2]));
    }

    // Top neighbor
    if ( (row-1)> 0 && !graph(row-1, col)->obs)
    {
        auto T = graph(row-1, col);
        neighbors_w_list.push_back(make_pair(T, current_node->edge_w[3]));
    }
}
```

...

INPUTS:
graph
current_node

OUTPUT:
neighbours_w_list

EXPLANATION:
For a any graph and a given node in it, this function finds the valid neighbours.
Valid neighbors are those nodes which are not obstacles and within the boundary of the graph.
The list also contains the associated costs to come to that particular node, which is useful for Dijsktra's and A* algorithm.

## generate_path()

```
bool Utils::generate_path(const MatrixXNode& graph,
                          const Eigen::Vector2i& start,
                          const Eigen::Vector2i& goal,
                          vector<Eigen::Vector2i>& path)
{
    path.clear();

    auto current_node = graph(goal(0), goal(1));

    // If parent doesn't exist for goal node, then path was not found
    if (current_node->parent == nullptr)
    {
        path.clear();
        return false;
    }
    else
    {
        // Starting from goal node, move to start node using parent info to find the path
        while (current_node != graph(start(0), start(1)))
        {
            path.push_back({current_node->row, current_node->col});
            current_node = graph(current_node->parent->row, current_node->parent->col);
        }
        path.push_back({current_node->row, current_node->col});     // push start node also

        reverse(path.begin(), path.end());                          // get path from start to goal
        return true;
    }
}
```

…

INPUTS:
graph
start
goal

OUTPUT:
path

EXPLANATION:
Every algorithm independently updates the nodes in the graph with cost, it’s parent, etc.
Then, given that updated graph to this function, this generates the list of paths required as output to plot the output path in the graph.

Now, let us look at the algorithms one-by-one.

## BFS

*The logic is commented in the code. For ease of understanding, its broken down into steps.*

;

```
bool BFS::search(const Eigen::MatrixXi& grid,
                 const Eigen::Vector2i& start,
                 const Eigen::Vector2i& goal,
                 vector<Eigen::Vector2i>& path,
                 int& steps)
{
    cout << "searching . . ." << endl;

    // [0] Clearing garbage values in output variables
    path.clear();
    steps = 0;

    // [1] Converting the input grid into graph of Nodes and initializing each node
    m_total_rows = grid.rows();
    m_total_cols = grid.cols();
    m_graph = MatrixXNode(m_total_rows, m_total_cols);      // empty graph matrix is ready
    Utils::initialize_graph(grid, m_graph);


    // [2] Start & Goal node
    auto start_node = m_graph(start(0), start(1));
    auto goal_node  = m_graph(goal(0), goal(1));


    // [3] Queuing the nodes to visit in the loop
    queue <shared_ptr<Node>> Q;
    Q.push(start_node);


    // [4] Update each nodes' parents and distance from start using BFS
    auto current_node = Q.front();
    vector<pair<shared_ptr<Node>,int>> neighbors_w;
    int row=0, col=0;

    while ( (!Q.empty()) && (current_node != goal_node) )
    {
        current_node = Q.front();
        Q.pop();

        current_node->visited = true;
        steps += 1;

        row = current_node->row;
        col = current_node->col;

        neighbors_w.clear();
        Utils::find_valid_neighbors(m_graph, current_node, neighbors_w);

        for (auto& node_w : neighbors_w)
        {
            auto node = node_w.first;
            auto w = node_w.second;     // not useful for this algo

            if ( !node->visited )
            {
                node->cost      = current_node->cost + 1;
                node->parent    = current_node;
                node->visited   = true;
                Q.push(node);
            }
        }
    }

    // [5] Using updated graph's info, find the path from start to goal
    return Utils::generate_path(m_graph, start, goal, path);
}
```

# DFS

*The logic is commented in the code. For ease of understanding, its broken down into steps.*

```cpp
bool DFS::search(const Eigen::MatrixXi& grid,
                 const Eigen::Vector2i& start,
                 const Eigen::Vector2i& goal,
                 vector<Eigen::Vector2i>& path,
                 int& steps)
{
    cout << "searching . . ." << endl;

    // [0] Clearing garbage values in output variables
    path.clear();
    steps = 0;

    // [1] Converting the input grid into graph of Nodes and initializing each node
    m_total_rows = grid.rows();
    m_total_cols = grid.cols();
    m_graph = MatrixXNode(m_total_rows, m_total_cols);      // empty graph matrix is ready
    Utils::initialize_graph(grid, m_graph);


    // [2] Start DFS recursive logic
    auto start_node = m_graph(start(0), start(1));
    auto goal_node = m_graph(goal(0), goal(1));
    dfs_recursive_visit(start_node, goal_node, steps);


    // [3] Using updated graph's info, find the path from start to goal
    return Utils::generate_path(m_graph, start, goal, path);
}


void DFS::dfs_recursive_visit(shared_ptr<Node> current_node, shared_ptr<Node> goal_node, int& steps)
{
    current_node->visited = true;
    if (!m_goal_hit)
        steps += 1;

    vector<pair<shared_ptr<Node>,int>> neighbors_w;
    Utils::find_valid_neighbors(m_graph, current_node, neighbors_w);

    for (auto node_w : neighbors_w)
    {
        auto node = node_w.first;
        auto w = node_w.second;             // not useful for this algo

        if ( node && !node->visited )
        {
            node->parent = current_node;

            if (node == goal_node)
            {
                m_goal_hit = true;
                steps += 1;                 // since we need to count goal_node also
            }

            dfs_recursive_visit(node, goal_node, steps);
        }
    }
}

} // namespace motion_planning
```

;

Dijkstra

*The logic is commented in the code. For ease of understanding, its broken down into steps.*

;

```
bool Dijkstra::search(const Eigen::MatrixXi& grid,
                      const Eigen::Vector2i& start,
                      const Eigen::Vector2i& goal,
                      vector<Eigen::Vector2i>& path,
                      int& steps)
{
    cout << "searching . . ." << endl;

    // [0] Clearing garbage values in output variables
    path.clear();
    steps = 0;

    // [1] Converting the input grid into graph of Nodes and initializing each node
    m_total_rows = grid.rows();
    m_total_cols = grid.cols();
    m_graph = MatrixXNode(m_total_rows, m_total_cols);      // empty graph matrix is ready
    Utils::initialize_graph(grid, m_graph);

    // [2] Start & Goal node
    auto start_node = m_graph(start(0), start(1));
    auto goal_node  = m_graph(goal(0), goal(1));


    // [3] Queuing the nodes to visit in the loop
    priority_queue <shared_ptr<Node>,
                    vector<shared_ptr<Node>>,
                    cmp> Q;
    Q.push(start_node);

    // [4] Update each nodes' parents and distance from start using Dijkstra
    auto current_node = Q.top();
    vector<pair<shared_ptr<Node>,int>> neighbors_w;
    int row=0, col=0;

    while ( (!Q.empty()) && (current_node != goal_node) )
    {
        current_node = Q.top();
        Q.pop();

        row = current_node->row;
        col = current_node->col;

        neighbors_w.clear();
        Utils::find_valid_neighbors(m_graph, current_node, neighbors_w);

        for (auto& node_w : neighbors_w)
        {
            auto node = node_w.first;
            auto w = node_w.second;

            auto d = current_node->cost + w;
            if ( !node->visited && d < node->cost )
            {
                node->cost      = d;
                node->parent    = current_node;
                node->visited   = true;
                Q.push(node);
            }
        }

        current_node->visited = true;
        steps += 1;
    }

    // [5] Using updated graph's info, find the path from start to goal
    return Utils::generate_path(m_graph, start, goal, path);
}
```

## *A**

```cpp
bool AStar::search(const Eigen::MatrixXi& grid, const Eigen::Vector2i& start, const Eigen::Vector2i& goal,
                   vector<Eigen::Vector2i>& path, int& steps)
{
    cout << "searching . . ." << endl;

    // [0] Clearing garbage values in output variables
    path.clear();
    steps = 0;

    // [1] Converting the input grid into graph of Nodes and initializing each node
    m_total_rows = grid.rows();
    m_total_cols = grid.cols();
    m_graph = MatrixXNode(m_total_rows, m_total_cols);      // empty graph matrix is ready
    Utils::initialize_graph(grid, m_graph);

    // [2] Start & Goal node
    auto start_node = m_graph(start(0), start(1));
    auto goal_node  = m_graph(goal(0), goal(1));

    // [3] Queuing the nodes to visit in the loop
    priority_queue <shared_ptr<Node>,
                    vector<shared_ptr<Node>>,
                    cmp> Q;
    Q.push(start_node);

    // [4] Update each nodes' parents and distance from start using AStar
    auto current_node = Q.top();
    current_node->cost = 0;
    vector<pair<shared_ptr<Node>,int>> neighbors_w;
    int row=0, col=0;

    while ( (!Q.empty()) && (current_node != goal_node) )
    {
        current_node = Q.top();
        Q.pop();

        if ( !current_node->visited )
        {
            current_node->visited = true;
            steps += 1;

            row = current_node->row;
            col = current_node->col;

            neighbors_w.clear();
            Utils::find_valid_neighbors(m_graph, current_node, neighbors_w);

            for (auto& node_w : neighbors_w)
            {
                auto node = node_w.first;
                auto w = node_w.second;
                auto h = abs(goal_node->row - node->row) + abs(goal_node->col - node->col);
                auto f = (current_node->cost - current_node->h) + w + h;
                current_node->h = h;

                if ( !node->visited && f < node->cost )
                {
                    node->cost      = f;
                    node->parent    = current_node;
                    Q.push(node);
                }
            }
        }
    }

    // [5] Using updated graph's info, find the path from start to goal
    return Utils::generate_path(m_graph, start, goal, path);
}
```

*The logic is commented in the code. For ease of understanding, its broken down into steps.;*

*main.cpp*

*This file contains the code to execute the algorithms with different grids. This is for time-being. As discussed, I will provide the ROS communication files ASAP to ease the grading task.*

;

```
int main(int argc, char **argv)
{
//  ros::init(argc, argv, "hw1_basic_search_algo", ros::init_options::AnonymousName);

    Eigen::MatrixXi grid(10,10);
    grid << 0,  0,  0,  0,  0,  0,  0,  0,  1,  0,
            0,  1,  1,  1,  1,  0,  0,  0,  1,  0,
            0,  0,  0,  0,  0,  0,  0,  0,  1,  0,
            0,  0,  0,  1,  1,  1,  1,  1,  1,  0,
            0,  1,  0,  1,  0,  0,  0,  0,  0,  0,
            0,  1,  0,  1,  0,  1,  1,  1,  1,  0,
            0,  1,  0,  0,  0,  1,  1,  1,  1,  0,
            0,  1,  1,  1,  0,  0,  0,  1,  1,  0,
            0,  0,  0,  0,  0,  0,  0,  1,  0,  0,
            0,  0,  0,  0,  0,  0,  0,  0,  0,  0;

    const Eigen::Vector2i start {0,0};
    const Eigen::Vector2i goal {3,1};
    vector<Eigen::Vector2i> path;
    int steps;


    cout << "------------BFS------------" << endl;
    motion_planning::BFS bfs;
    bfs.search(grid, start, goal, path, steps);

    cout << "Steps: " << steps << endl;
    for (auto& i : path)
    {
        cout << "[" << i(0) << "," << i(1) << "], ";
    }
    cout << endl;

    cout << "------------DFS------------" << endl;
    motion_planning::DFS dfs;
    dfs.search(grid, start, goal, path, steps);

    cout << "Steps: " << steps << endl;
    for (auto& i : path)
    {
        cout << "[" << i(0) << "," << i(1) << "], ";
    }
    cout << endl;

    cout << "------------Dijkstra------------" << endl;
    motion_planning::Dijkstra dijkstra;
    dijkstra.search(grid, start, goal, path, steps);

    cout << "Steps: " << steps << endl;
    for (auto& i : path)
    {
        cout << "[" << i(0) << "," << i(1) << "], ";
    }
    cout << endl;

    cout << "------------A*------------" << endl;
    motion_planning::AStar astar;
    astar.search(grid, start, goal, path, steps);

    cout << "Steps: " << steps << endl;
    for (auto& i : path)
    {
        cout << "[" << i(0) << "," << i(1) << "], ";
    }
    cout << endl;

//  /* This starts the ROS server to serve different algo etc */
```

*How are the algorithms similar and different?*

As seen in the lectures and the code above, we can compare the algorithms based on

1. the cost function each uses to decide how to explore the graph to reach the goal.
2. The data structure used to implement the algorithm
3. BFS, Dijkstra and A* always gives the shortest path, whereas in DFS, it depends on the terrian

Here's a quick summary of how I understand them:

| Algorithm | Cost Function | Data Structure | Comment |
|---|---|---|---|
| BFS | None | Queue | Blindly explores all the surrounding neighbors around each node until goal is found. |
| DFS | None | Stack/Recursion | Blindly explore all neighbors, but first aims at exploring the children and grandchildren nodes and so on. |
| Dijkstra | $f(x) = g(x)$ | Priority Queue | Sorts the queue of nodes to visit based on the cost defined. Instead of blindly exploring all neighbors, it considers the paths which has the least cost to travel while deciding which node to visit next. |
| A* | $f(x) = g(x) + h(x)$ | Priority Queue | It builds on the logic of Dijkstra's algorithm, ie. it also considers the future cost to get to goal from each node in addition to the past cost calculated at each node. The future cost is calculated based on some heuiristic (in our HW, it is Manhattan distance). |

$g(x)$ → Cost to get to a current node from the start….ie remembering the past difficulty in moving from the start node to the current node in the graph.

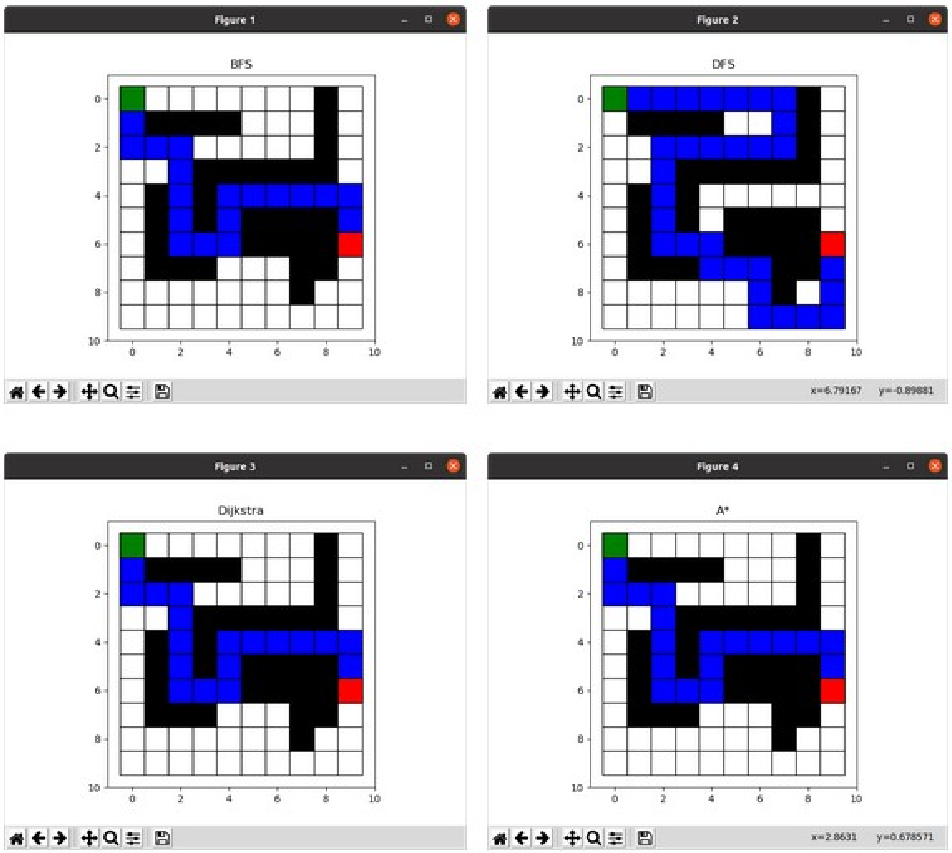
$h(x)$ → Cost to get to goal node from the current node… ie estimating the future cost in moving from the current node towards the goal in the graph.

Other similarities:

1. Completeness → All the above algorithms are complete.
2. Considering edge weigths → Both A* and Dijkstra considers this as well in their decisions.
3. Logic → Dijkstra is essentially weighted BFS

The outputs for the given map.
(The output was generated usnig C++. The outpus were hard-coded in the Python for visualization. This is for time-being. I will provide the ROS based communication from Python grader scripts to my C++ implementation as soon as possible as discussed with Preyash, our TA.)

### Output paths in the graph




```
Here's the number of cells each algorithm visits before hitting the goal:

  * It takes 64 steps to find a path using BFS
  * It takes 33 steps to find a path using DFS
  * It takes 64 steps to find a path using Dijkstra
  * It takes 60 steps to find a path using A*
```

The output of C++ code on the terminal for the given graph:

```
emmanuel@msi:~/workspace/ros-workspace/RBE550_ws$ rosrun hw1_BasicSearchAlgo hw1_basic_search_algo
-------------BFS-------------
searching . . .
Steps: 64
[0,0], [1,0], [2,0], [2,1], [2,2], [3,2], [4,2], [5,2], [6,2], [6,3], [6,4], [5,4], [4,4], [4,5], [4,6], [4,7],
 [4,8], [4,9], [5,9], [6,9],
-------------DFS-------------
searching . . .
Steps: 33
[0,0], [0,1], [0,2], [0,3], [0,4], [0,5], [0,6], [0,7], [1,7], [2,7], [2,6], [2,5], [2,4], [2,3], [2,2], [3,2],
 [4,2], [5,2], [6,2], [6,3], [6,4], [7,4], [7,5], [7,6], [8,6], [9,6], [9,7], [9,8], [9,9], [8,9], [7,9], [6,9]
,
-------------Dijkstra-------------
searching . . .
Steps: 64
[0,0], [1,0], [2,0], [2,1], [2,2], [3,2], [4,2], [5,2], [6,2], [6,3], [6,4], [5,4], [4,4], [4,5], [4,6], [4,7],
 [4,8], [4,9], [5,9], [6,9],
-------------A*-------------
searching . . .
Steps: 60
[0,0], [1,0], [2,0], [3,0], [3,1], [3,2], [4,2], [5,2], [6,2], [6,3], [6,4], [5,4], [4,4], [4,5], [4,6], [4,7],
 [4,8], [4,9], [5,9], [6,9],
emmanuel@msi:~/workspace/ros-workspace/RBE550_ws$
```

How to build the C++ code for verification and grading?

## Building

To build from source, clone the latest version from this repository into your catkin workspace and compile the package using

````
```bash
$ cd catkin_workspace/src
$ git clone https://github.com/emmanuel-logy/motion_planning_algorithms.git
$ cd ..
$ catkin_make
```
````

Instead of cloning, please feel free to use the zip file I've submitted. Thank you.

**- - - - - END - - - - -**