SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

RMR

Bratislava 2023 **Bc. Péter Dvorák, Bc Gergely Göndör**

Úloha 1:

Ako prvý museli sme riešiť lokalizáciu robota s princípom odometire, vypočítať súradnice robota (x,y) a uhol natočenia. Mali sme prejdenú vzdialenosť ľavého a pravého kolesa z ktorých sme vedeli vypočítať žiadané hodnoty. Pretečenie enkóderov som vyriešil tak, že ich differenciu som uložil ako short. Potom s danými vzorcami vypočítal som súradnice a uhol natočenia robota.

Ako druhý museli sme riešiť polohovanie robota na želanú x a y. Kým robot nebol pri cieľa tak sa natočil do smeru cieľa a začal ísť rovno, bolo ošetrené aby robot sa netočil a neišiel naraz. Pri každom cikle bolo najprv ošetrené či robot smeruje na cieľ ak nie tak sa zastavil a začal sa točiť na cieľ po kratšej trase. Na nastavenia rýchlosti otáčanie a translácie som používal jednoduchý regulátor. Skladal som vektory so želanými hodnotami x a y a robot úspešne sa navigoval medzi tými bodmi. Pre vykonanie druhej časti úlohy treba kliknúť na pushbutton Navigate v GUI.

```
if (datacounter == 0)
{
    old_left_encounter = robotdata.EncoderLeft;
    old_right_encounter = robotdata.EncoderRight;
}
diff_in_left_encounter = robotdata.EncoderLeft - old_left_encounter;
diff_in_right_encounter = robotdata.EncoderRight - old_right_encounter;

left_wheel_distance = kobuki.tickToMeter * diff_in_left_encounter;
right_wheel_distance = kobuki.tickToMeter * diff_in_right_encounter;

double delta_fi = (right_wheel_distance - left_wheel_distance) / kobuki.b;

double delta_s = (right_wheel_distance + left_wheel_distance) / 2;

current_x += delta_s * cos(current_angle + (delta_fi / 2));
current_y += delta_s * sin(current_angle + (delta_fi / 2));

current_angle += delta_fi;

if (current_angle > PI)
    current_angle -= 2 * PI;
if (current_angle <= -PI)
    current_angle = current_angle + 2 * PI;
datacounter++;

old_left_encounter = robotdata.EncoderLeft;
old_right_encounter = robotdata.EncoderRight;
```

Obr. 1.: Prvá časť úlohy

```
if (position_index < (sizeof(y_goal) / sizeof(y_goal[0])) && is_navigating == true)
{
    Point2d current_position = {current_x, current_y};
    Point2d goal_position = {x_goal[position_index], y_goal[position_index]};
    if (distance(current_position, goal_position) > GOAL_THRESHOLD)
    {
        RobotState new_state = findBestTrajectory(current_x, current_y, current_angle, speed, rotation_speed, goal_position);
        speed = new_state.v;
        rotation_speed = new_state.w;
        if (rotation_speed < 0.1)
            goTranslate();
        else
            goRotate();
    }
    else
    {
        position_index++;
        std::cout << "At Goal" << std::endl;
    }
}
```

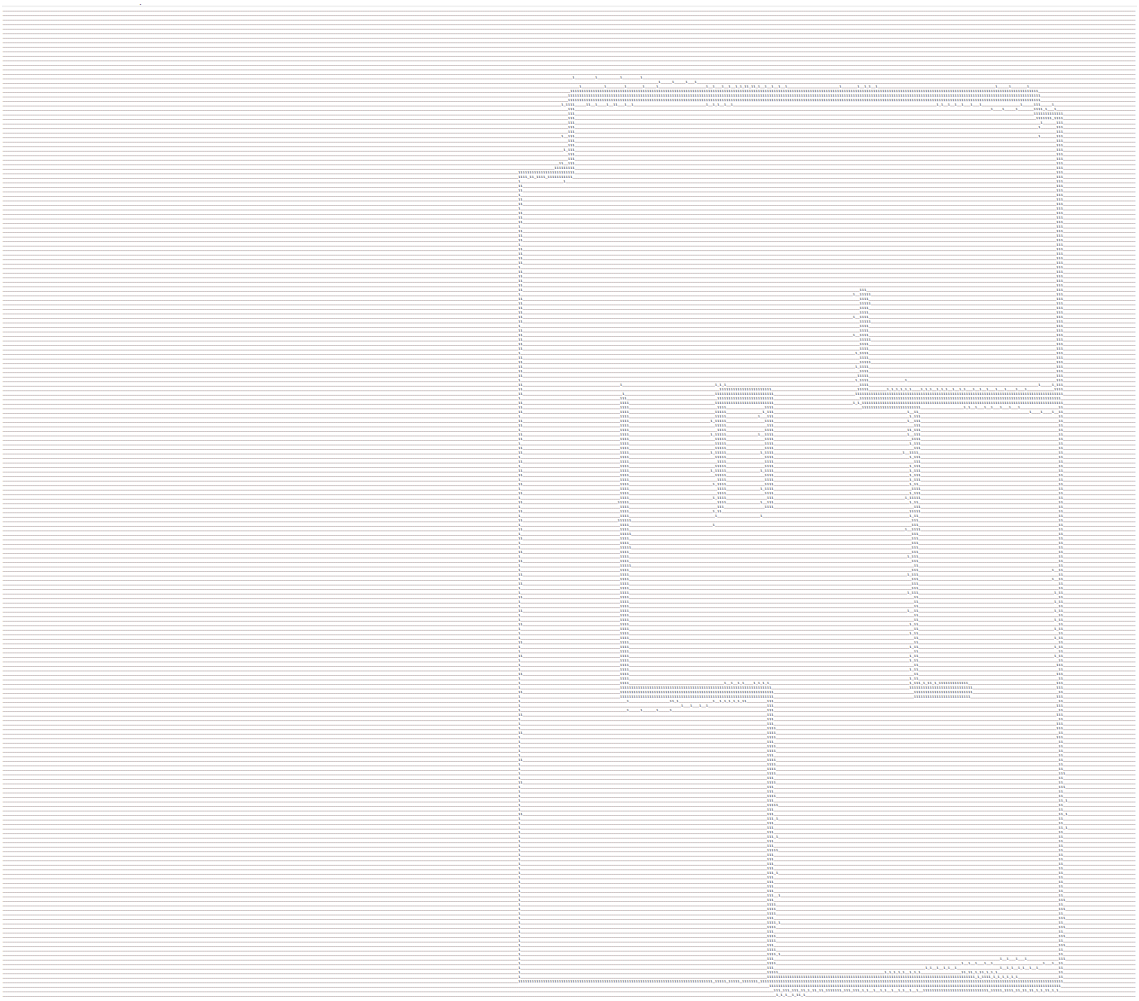
Obr. 2.: Druhá časť úlohy

Úloha 3:

Našou úlohou bolo mapovanie priestoru so zaznačenými prekážkami vo formáte okupačnej mriežky. Úlohu som vyriešil vo funkcií ProcessThisLidar(). Robot automaticky mapuje priestor nič nemusíme nato zapnúť. Ošetril som aby robot nemapoval keď sa točí. Veľkosť mapy je 500*500 a uložým do súboru mapa.txt. V úlohe 1 som počítal súradnice v metroch, tak aj pri tejto úlohy všetky dáta prepočítam na metre.

```
if (rotation_speed == 0 && mapping == true)
{
    for (int i = 0; i < copyOfLaserData.numberOfScans; i++)
    {

        if (copyOfLaserData.Data[i].scanDistance < 300 || copyOfLaserData.Data[i].scanDistance > 3000 || (copyOfLaserData.Data[i].scanDistance > 640 && copyOfLaserData.Data[i].scanDistance < 700))
        {
            continue;
        }
        else
        //            if (copyOfLaserData.Data[i].scanDistance > 230 || copyOfLaserData.Data[i].scanDistance < 3000)
        {
            double scan_distance = copyOfLaserData.Data[i].scanDistance / 1000;
            int point_y = -(current_y + scan_distance * sin((360 - copyOfLaserData.Data[i].scanAngle) * PI / 180.0 + current_angle)) / 12 * 500 + 500 / 2 - 1;
            int point_x = (current_x + scan_distance * cos((360 - copyOfLaserData.Data[i].scanAngle) * PI / 180.0 + current_angle)) / 12 * 500 + 500 / 2 - 1;
            created_map[point_x][point_y] = 1;
        }
    }

    FILE *fp;
    int u, v;
    fp = fopen("/home/pdvorak/rmr_school/School/RMR/all-lidar-robot/map.txt", "w");
    for (u = 0; u < 500; u++)
    {
        for (v = 0; v < 500; v++)
        {
            if (created_map[v][u] == 0)
                fprintf(fp, "_");
            else
                fprintf(fp, "%d", created_map[v][u]);
        }
        if (created_map[v][u] == 0)
            fprintf(fp, "_\n");
        else
            fprintf(fp, "%d\n", created_map[v][u]);
    }
    fclose(fp);
}
```

Obr. 3.: Script pre úlohu



Obr. 4.: map.txt

Úloha 4:

Našou úlohou bolo nájsť množinu bodov cez ktorých robot musí prejsť od štartu do cieľa pri najkratšom ceste. Skúsil som to urobiť pomocou súboru priestor.txt a nakoniec mal som chyby v tej mape. Potom som zhrubšil steny a prekážky o 3 buniek. Nastavil som koordináty pre cieľ a zavolal som funkciu pre flood algoritmus na mapu a to som uložil v súbore flood.txt. Po flood algoritme sme našli najkratšiu cestu pre mapu a to som uložil v súbore shortest_path.txt. Znaky "A" na updated_map sú zóny so zákazom vstupu.

```
float x1, x2, y1, y2;

// walls
for (int i = 0; i < map.wall.numofpoints - 1; ++i)
{
    if (i + 1 < map.wall.numofpoints)
    {
        if (map.wall.points[i].point.x < map.wall.points[i + 1].point.x)
        {
            x1 = map.wall.points[i].point.x;
            x2 = map.wall.points[i + 1].point.x;
        }
        else
        {
            x1 = map.wall.points[i + 1].point.x;
            x2 = map.wall.points[i].point.x;
        }
        if (map.wall.points[i].point.y < map.wall.points[i + 1].point.y)
        {
            y1 = map.wall.points[i].point.y;
            y2 = map.wall.points[i + 1].point.y;
        }
        else
        {
            y1 = map.wall.points[i + 1].point.y;
            y2 = map.wall.points[i].point.y;
        }
    }
    x1 = (int)(x1 / 4);
    x2 = (int)(x2 / 4);
    y1 = (int)(y1 / 4);
    y2 = (int)(y2 / 4);
    for (int j = x1; j <= x2; ++j)
    {
        for (int k = y1; k <= y2; ++k)
        {
            path_finding_map[j][k] = 1;
        }
    }
}
for (int i = 0; i < 109; ++i)
    path_finding_map[0][i] = 1;
```

Obr.5.: Načítavanie steny

```
// Obstacles
for (int i = 0; i < map.numofObjects; ++i)
{
    for (int j = 0; j < map.obstacle[i].numofpoints; ++j)
    {
        if (j + 1 < map.obstacle[i].numofpoints)
        {
            if (map.obstacle[i].points[j].point.x < map.obstacle[i].points[j + 1].point.x)
            {
                x1 = map.obstacle[i].points[j].point.x;
                x2 = map.obstacle[i].points[j + 1].point.x;
            }
            else
            {
                x1 = map.obstacle[i].points[j + 1].point.x;
                x2 = map.obstacle[i].points[j].point.x;
            }
            if (map.obstacle[i].points[j].point.y < map.obstacle[i].points[j + 1].point.y)
            {
                y1 = map.obstacle[i].points[j].point.y;
                y2 = map.obstacle[i].points[j + 1].point.y;
            }
            else
            {
                y1 = map.obstacle[i].points[j + 1].point.y;
                y2 = map.obstacle[i].points[j].point.y;
            }
        }
        x1 = (int)(x1 / 4);
        x2 = (int)(x2 / 4);
        y1 = (int)(y1 / 4);
        y2 = (int)(y2 / 4);

        for (int k = x1; k <= x2; ++k)
        {
            for (int l = y1; l <= y2; ++l)
                path_finding_map[k][l] = 1;
        }
    }
}
```

Obr.6.: Načítávanie prekážky

```
for (int i = 0; i < 150; ++i)
{
    for (int j = 0; j < 150; ++j)
    {
        if (path_finding_map[i][j] == 1)
        {
            if ((j + 1) < 149 && path_finding_map[i][j + 1] == 0)
                path_finding_map[i][j + 1] = 900;
            if ((j + 2) < 149 && path_finding_map[i][j + 2] == 0)
                path_finding_map[i][j + 2] = 900;
            if ((j + 3) < 149 && path_finding_map[i][j + 3] == 0)
                path_finding_map[i][j + 3] = 900;
            if ((j - 1) > 0 && path_finding_map[i][j - 1] == 0)
                path_finding_map[i][j - 1] = 900;
            if ((j - 2) > 0 && path_finding_map[i][j - 2] == 0)
                path_finding_map[i][j - 2] = 900;
            if ((j - 3) > 0 && path_finding_map[i][j - 3] == 0)
                path_finding_map[i][j - 3] = 900;
            if ((i + 1) < 149 && path_finding_map[i + 1][j] == 0)
                path_finding_map[i + 1][j] = 900;
            if ((i + 2) < 149 && path_finding_map[i + 2][j] == 0)
                path_finding_map[i + 2][j] = 900;
            if ((i + 3) < 149 && path_finding_map[i + 3][j] == 0)
                path_finding_map[i + 3][j] = 900;
            if ((i - 1) > 0 && path_finding_map[i - 1][j] == 0)
                path_finding_map[i - 1][j] = 900;
            if ((i - 2) > 0 && path_finding_map[i - 2][j] == 0)
                path_finding_map[i - 2][j] = 900;
            if ((i - 3) > 0 && path_finding_map[i - 3][j] == 0)
                path_finding_map[i - 3][j] = 900;
        }
    }
}
```

Obr.7.: Hrubšie steny a prekážky

```
{
    path_finding_map[int(end_point.x)][int(end_point.y)] = 2;
    bool is_there = false;
    int map_constant = 150;
    while (1)
    {
        is_there = false;
        for (int i = 0; i < map_constant; ++i)
        {
            for (int j = 0; j < map_constant; ++j)
            {
                if ((path_finding_map[i][j] != 0) && (path_finding_map[i][j] != 1) && path_finding_map[i][j] != 900)
                {
                    if (i - 1 >= 0)
                    {
                        if (path_finding_map[i - 1][j] == 0)
                        {
                            path_finding_map[i - 1][j] = path_finding_map[i][j] + 1;
                            is_there = true;
                        }
                    }
                    if (i + 1 < map_constant)
                    {
                        if (path_finding_map[i + 1][j] == 0)
                        {
                            path_finding_map[i + 1][j] = path_finding_map[i][j] + 1;
                            is_there = true;
                        }
                    }
                    if (j - 1 >= 0)
                    {
                        if (path_finding_map[i][j - 1] == 0)
                        {
                            path_finding_map[i][j - 1] = path_finding_map[i][j] + 1;
                            is_there = true;
                        }
                    }
                    if (j + 1 < map_constant)
                    {
                        if (path_finding_map[i][j + 1] == 0)
                        {
                            path_finding_map[i][j + 1] = path_finding_map[i][j] + 1;
                            is_there = true;
                        }
                    }
                }
            }
        }
        if (is_there == false)
        {
            break;
        }
    }
}
```

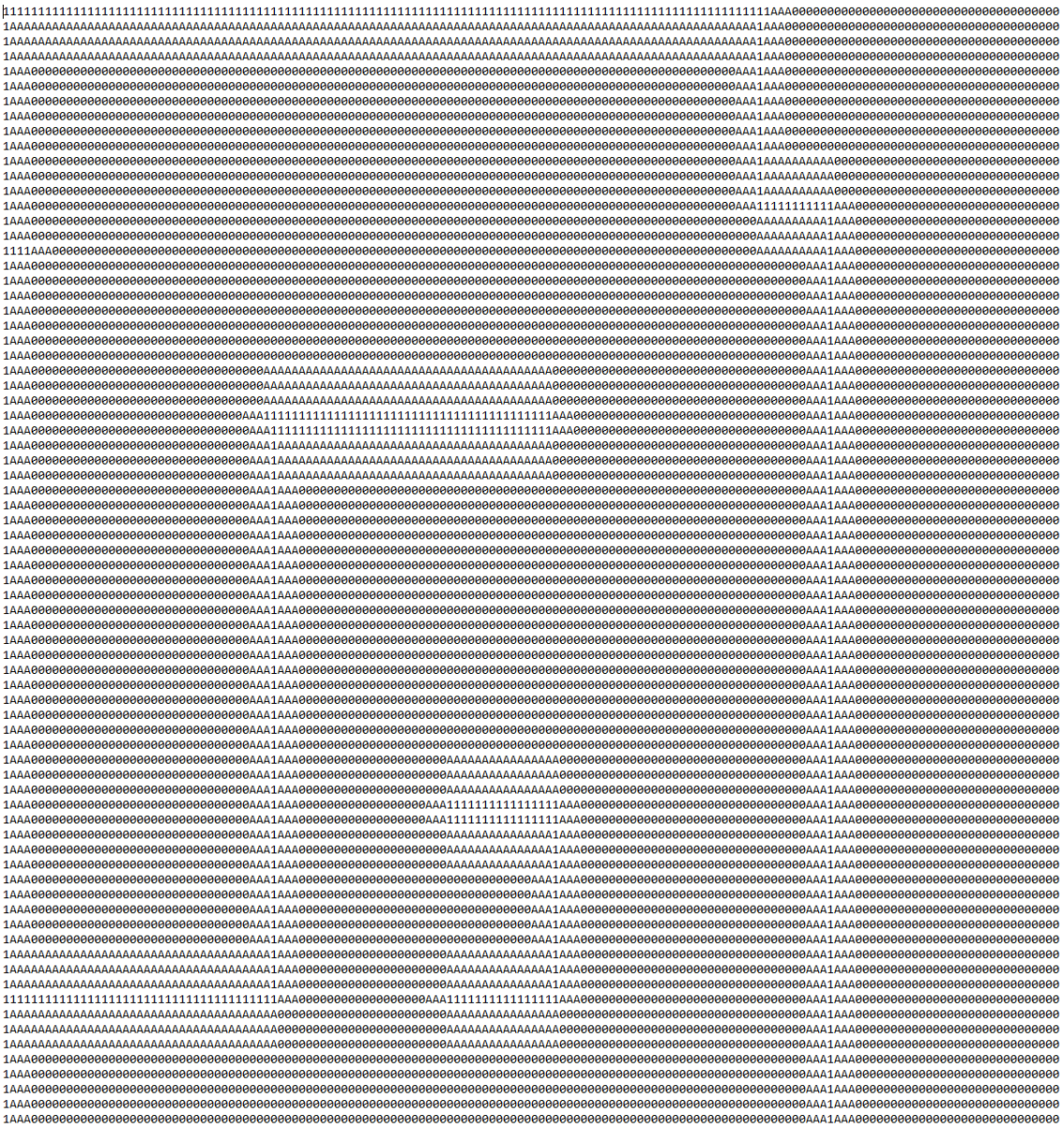
Obr.8.: Flood algoritmus, pred tým nastavil som koordináty cieľa na 2

```
{
    int counter = 1;
    int c_new = 1;
    int current_number = path_finding_map[start_x][start_y];
    while (current_number != 2)
    {
        for (int i = -1; i <= 1; i++)
        {
            for (int j = -1; j <= 1; j++)
            {
                if ((i == 0 && j == 0 )|| (i == -1 && j==-1) ||(i == 1 && j==-1) ||(i == -1 && j==1) ||(i == 1 && j==1))
                {
                    continue;
                }
                if (path_finding_map[start_x + i][start_y + j] < current_number)
                {
                    current_number = path_finding_map[start_x + i][start_y + j];
                    shortest_map[start_x + i][start_y + j] = counter;
                    Point2d temp_point;
                    temp_point.x = ((double)((start_x + i) * 4) / 100) - 0.4;
                    temp_point.y = ((double)((start_y + j) * 4) / 100) - 0.4;
                    path_points.push_back(temp_point);
                    c_new = counter++;
                    start_x += i;
                    start_y += j;
                }
                if (path_finding_map[start_x + i][start_y + j] == 2)
                {
                    current_number = path_finding_map[start_x + i][start_y + j];
                    shortest_map[start_x + i][start_y + j] = counter;
                    c_new = counter++;
                    Point2d temp_point;
                    temp_point.x = ((double)((start_x + i) * 4) / 100) - 0.4;
                    temp_point.y = ((double)((start_y + j) * 4) / 100) - 0.4;
                    path_points.push_back(temp_point);
                }
                if(c_new != counter)
                {
                    break;
                }
            }
            if(c_new != counter)
            {
                counter = c_new;
                break;
            }
        }
    }
}
```

Obr.9.: Algoritmus na nájdenie optimálnej trasy

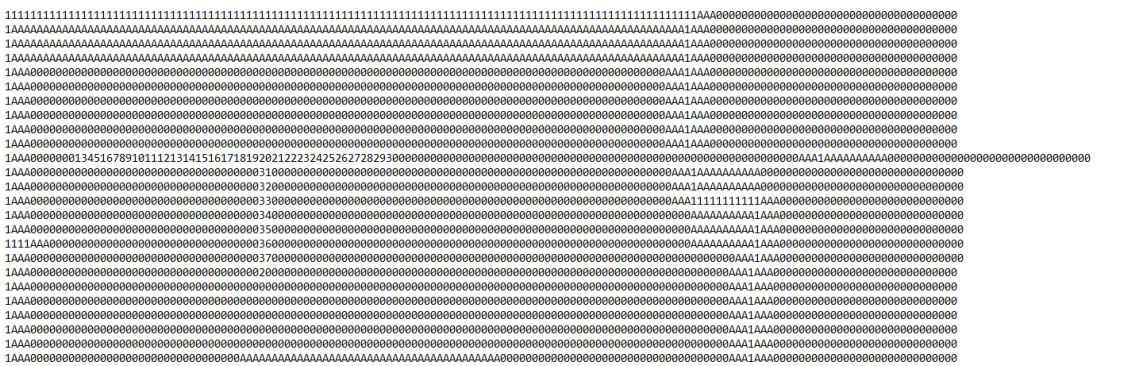
```
std::vector<Point2d> MainWindow::clearPath()
{
    double last_element_angle = atan2(path_points.at(0).y - path_points.at(1).y, path_points.at(0).x -path_points.at(1).x);
    std::vector<Point2d> points;
    points.push_back({0,0});
    for(int i = 1 ; i < path_points.size() - 1; ++i)
    {

        double new_angle = atan2(path_points.at(i).y - path_points.at(i+1).y, path_points.at(i).x - path_points.at(i+1).x);
            if(last_element_angle == new_angle)
            {
            }
            else
            {
                points.push_back(path_points.at(i+1));
            }
            last_element_angle = new_angle;

    }
    points.push_back(path_points.back());
    return points;
}
```

Obr.9.: Algoritmus na odstránenie bodov cesty, pri ktorých sa smer cesty nezmenil



Obr. 10.: updated_map.txt

Obr. 11.: flood.txt



Obr. 12.: shortest_path.txt