KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

T125B114
ROBOTŲ PROGRAMAMVIMO TECHNOLOGIJOS
Projektinio darbo ataskaita

Atliko:

Tomas Kašelynas IFF-7/5
Giedrius Rastauskas IFF-7/12
Lukas Žaromskis IFF-7/5

Priėmė:

Doc. Brūzgienė Rasa
Doc. Adomkus Tomas

KAUNAS 2020

Turinys

Užduotis

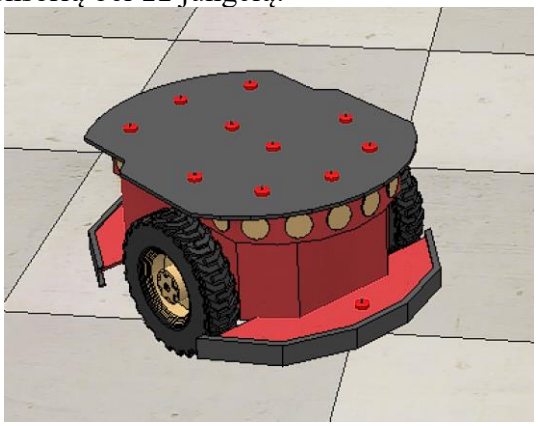
Robotas važiuoja link labirinto pradžios. Aptikęs kliūtis, jas apvažiuoja BUG0 algoritmu. Labirintą įveikia pagal dešinės rankos taisyklę. Toliau važiuoja link pabaigos taško ir kliūtis apvažiuoja BUG2 algoritmu.

Užduoties analizė

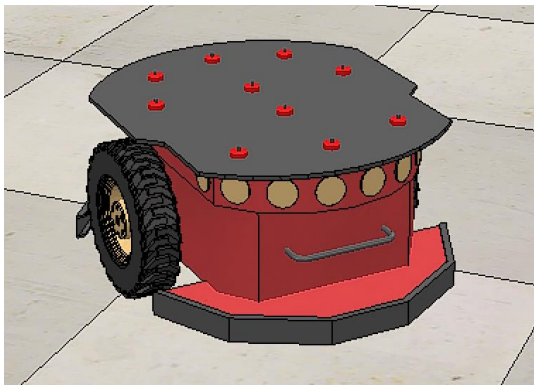
Roboto pradinis taškas yra toliau nuo labirinto ir jis turi pasiekti jo pradžią. Važiuoti tiesiai robotui nepavyks, nes tarp jo ir labirinto pradžios bus kliūčių. Kai robotas pasieks labirintą, jis jį turės įveikti ir pasiekti jo pabaigą. Po labirinto, robotas važiuodamas link pabaigos apvažiuoja kliūtis BUG2 algoritmu. Reikalinga, kad robotas mokėtų apvažiuoti kliūtis ir įveikti labirintą pagal dešinės rankos taisyklę.

Roboto aprašymas

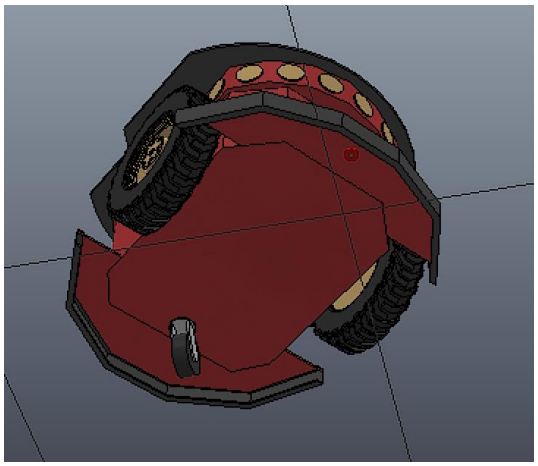
Robotas turi tris ratus: du pagrindiniai ratai su motorais ir vienas ratukas stabilizacijai. Taip pat robotas turi 16 ultragarsinių atstumo sensorių bei 11 jungčių.



1 pav. Roboto priekis



2 pav. Roboto galas



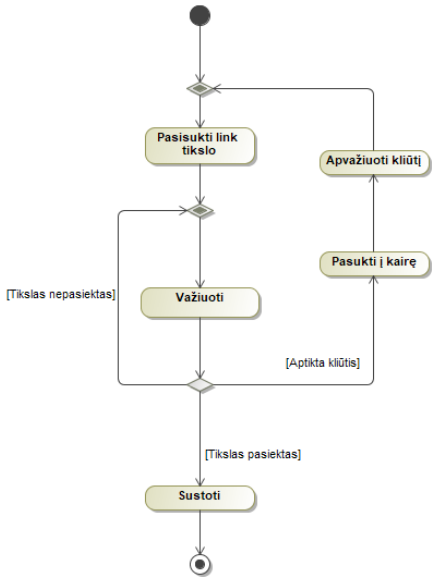
3 pav. Roboto apačia

Roboto valdymo architektūra

Robotas valdomas siunčiant jam komandas per Python „sim“ API [1]. Galima nustatyti kairiojo arba dešiniojo rato sukimosi greičius, gauti informaciją iš sensorių.

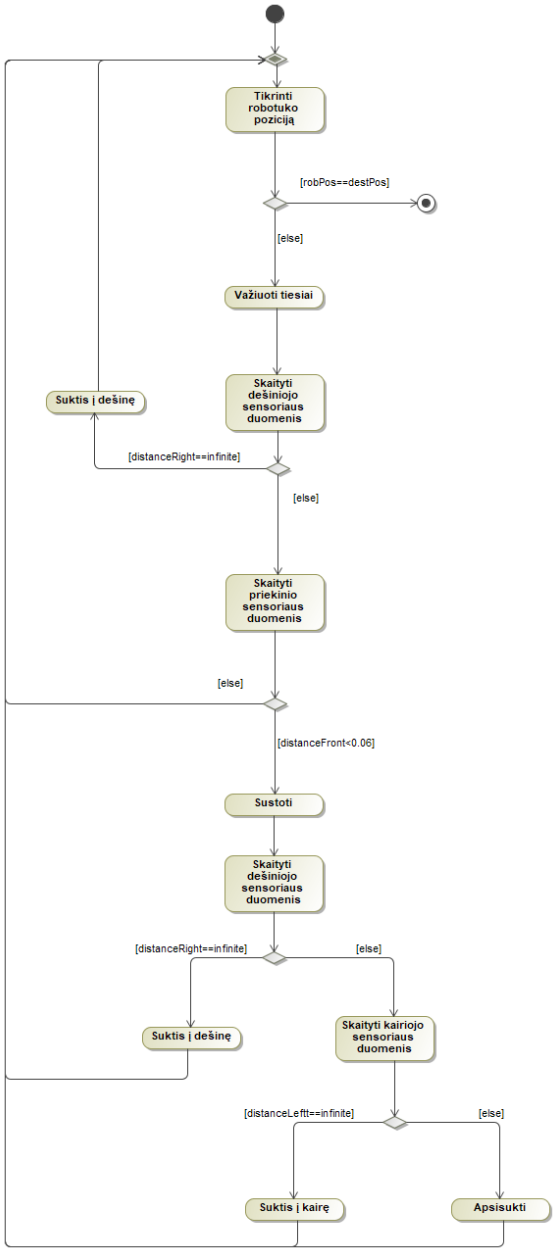
Roboto valdymo algoritmas

Pirmąsias kliūtis robotas įveikia naudojant BUG0 algoritmą. Jis važiuoja tiesiai link tikslo pozicijos, o kai yra aptinkama kliūtis yra sukama į kairę ir kliūtis apvažiuojama. Toliau vėl yra važiuojama tiesiai link kliūties ir jei yra aptinkama kliūtis, yra kartojamas apvažiavimas.



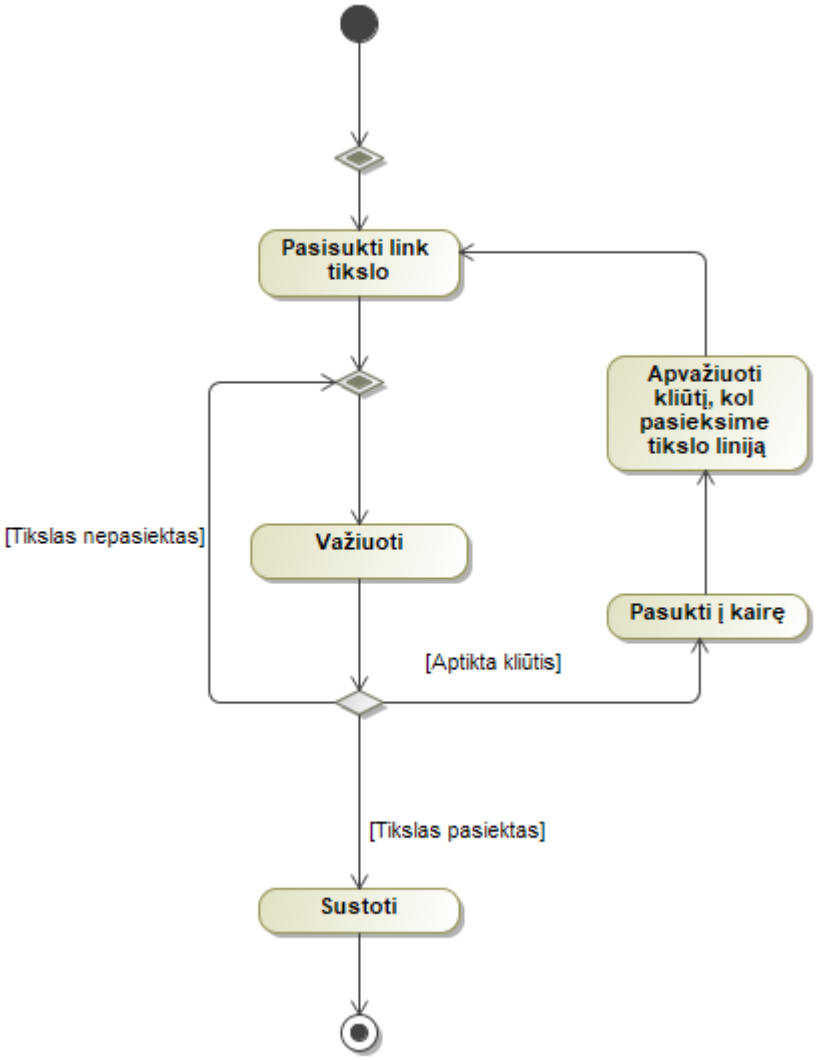
4 pav. BUG0 veiklos diagrama

Labirintą robotas įveikia naudojant dešinės rankos taisyklės algoritmą. Robotas važiuoja tiesiai tol, kol išvažiuoja iš labirinto. Aptikęs priešais sieną, pasižiūri, ar gali važiuoti į dešinę pusę. Jei gali, pasisuka į dešinę ir važiuoja tiesiai. Jei negali, tikrina kairę pusę. Jei ten nėra sienos, sukasi į kairę ir važiuoja tiesiai. Jei negali važiuoti nei į dešinę, nei į kairę, apsisuka ir važiuoja tiesiai. Jeigu važiuojant tiesiai aptinka, kad gali važiuoti į dešinę, sukasi į dešinę ir važiuoja tiesiai.



5 pav. MazeSolver veiklos diagrama

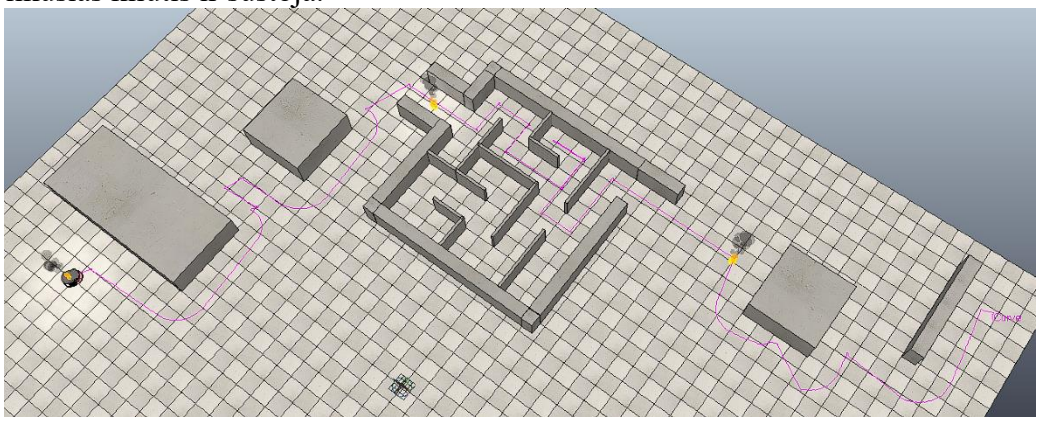
Įveikęs labirintą, robotas toliau kliūtis apvažiuoja BUG2 algoritmu. Panašiai kaip ir BUG0 algoritmas, pirmiausiai, robotas važiuoja tiesiai link tikslo pozicijos. Aptikęs kliūtį, ją apvažiuoja sukdamas į kairę ir važiuoja tol, kol priartėja prie tikslo linijos (tikslo linija - nuo pradžios taško iki tikslo taško įsivaizduojama linija). Toliau vėl kartojamas algoritmas.



pav. 6 BUG2 veiklos diagrama

Roboto modeliavimo rezultatai

Simuliacijos rezultatas pateiktas 7 pav. Robotukas yra pradinėje pozicijoje. Rožinė linija rodo jo judėjimo trajektoriją. Jo trajektorija rodo, kad jis juda link tikslo tol, kol pasiekia kliūtį. Tada ją apvažiuoja ir bando toliau judėti link tikslo. Kartoja tol, kol pasiekia tikslą. Tada važiuoja link labirinto, kurį įveikia pagal dešinės rankos taisyklę – seka dešinę sieną. Įveikęs labirintą juda link galutinio tikslo, apvažiuoja likusias kliūtis ir sustoja.



7 pav. Simuliacijos rezultatai

Roboto valdymo programa

Vienos svarbiausių funkcijų (7 pav.) yra *connect*, kuri prisijungia prie serverio, *disconnect*, kuri atsijungia, bei *getHandle*, kuri grąžina valdiklio objektą. Dažniausiai naudojamos funkcijos yra susijusios su judėjimu (8 pav.). *moveForward* funkcija abiem motorams duoda teigiamo greičio, kad važiuotų į priekį, *moveBackwards* duoda neigiamo greičio, kad važiuotų atgal, *turnLeft* ir *turnRight* duoda priešingus greičius, kad robotukas suktųsi į atitinkamą pusę. *Stop* funkcija sustabdo robotuko judėjimą. Orientavimosi funkcijos (9 pav.) skirtos darbui su robotuko pozicija aplinkoje. *getPosition* gauna robotuko poziciją aplinkoje, *getRotation* gauna robotuko pasisukimą, *getDistanceFromSensor* grąžina atstumą iki kliūties, *isApproximatePosition* tikrina, ar robotukas yra tam tikroje vietoje su nustatyta paklaida, *isApproximateRotation* grąžina, ar robotukas yra pasisukęs tam tikru kampu pagal nustatytą paklaidą. *normalizeAngle* (10 pav.) normalizuoja duotą kampą. *rotateUntilAngle* (11 pav.) verčia robotuką suktis tol, kol pasiekiamas nurodytas kampas su tam tikra paklaida, o *getDesiredRotation* (11 pav.) gauna kampą, iki kurio reikia suktis. Šios funkcijos yra naudojamos *rotateTowards* funkcijoje (12 pav.), kuri suka robotuką link nurodyto taško. *moveForwardFor* funkcija (13 pav.) verčia robotuką judėti nurodytu greičiu nurodytą laiko tarpą. *wallFollowRHS* algoritmas (14 pav.) seka kliūtį, laikydamas sau iš dešinės. Ji naudojama *bug0* algoritme, kuris parodytas 15 pav. *turn90Degrees* funkcija (16 pav.) naudojama labirinte, kad robotukas suktųsi pagal ašis ir nenukryptų nuo sienų. Labirinto įveikimo algoritmas pagal dešinės rankos taisyklę pateiktas 17 pav. *distanceToLine* funkcijoje yra skaičiuojamas atstumas iki tikslo linijos (18 pav.). Jis yra naudojamas bug2 algoritme, kuris parodytas 19 pav. Pagrindinis kodas, kuris yra atsakingas už algoritmų valdymą, pateiktas 20 pav.

```python
# Connects to the CoppeliaSim remote server
def connect():
    sim.simxFinish(-1)
    clientID = sim.simxStart('127.0.0.1', 19997, True, True, 5000, 5)
    if clientID != -1:
        print('Connected to remote API server')
        return clientID
    else:
        sys.exit('Failed connecting to remote API server')

# Get an object handle from the scene
def getHandle(client, objectID):
    retVal, handle = sim.simxGetObjectHandle(client, objectID, sim.simx_opmode_oneshot_wait)
    if retVal == 0:
        print('OK - ' + objectID + ' handle assigned')
        return handle
    else:
        sys.exit('Failed to get ' + objectID + ' handle')

# Disconnects from the CoppeliaSim remote server
def disconnect(client):
    sim.simxFinish(client)
```

8 pav. Prisijungimo, atsijungimo ir valdiklio paėmimas

```
# Sets the robot's wheel speeds
def move(client, leftMotor, leftSpeed, rightMotor, rightSpeed):
    retLeft = sim.simxSetJointTargetVelocity(client, leftMotor, leftSpeed, sim.simx opmode stream
ing)
    retRight = sim.simxSetJointTargetVelocity(client, rightMotor, rightSpeed, sim.simx_opmode_str
eaming)
    return retLeft | retRight

# Sets the robot to move forward
def moveForward(client, leftMotor, rightMotor, speed):
    retLeft = sim.simxSetJointTargetVelocity(client, leftMotor, speed, sim.simx_opmode_streaming)
    retRight = sim.simxSetJointTargetVelocity(client, rightMotor, speed, sim.simx_opmode_streamin
g)
    return retLeft | retRight

# Sets the robot to move backwards
def moveBackwards(client, leftMotor, rightMotor, speed):
    retLeft = sim.simxSetJointTargetVelocity(client, leftMotor, -
speed, sim.simx_opmode_streaming)
    retRight = sim.simxSetJointTargetVelocity(client, rightMotor, -
speed, sim.simx_opmode_streaming)
    return retLeft | retRight

# Sets the robot to turn left
def turnLeft(client, leftMotor, rightMotor, speed):
    retLeft = sim.simxSetJointTargetVelocity(client, leftMotor, -
speed, sim.simx_opmode_streaming)
    retRight = sim.simxSetJointTargetVelocity(client, rightMotor, speed, sim.simx_opmode_streamin
g)
    return retLeft | retRight

# Sets the robot to turn right
def turnRight(client, leftMotor, rightMotor, speed):
    retLeft = sim.simxSetJointTargetVelocity(client, leftMotor, speed, sim.simx_opmode_streaming)
    retRight = sim.simxSetJointTargetVelocity(client, rightMotor, -
speed, sim.simx_opmode_streaming)
    return retLeft | retRight

# Stops the robot
def stop(client, leftMotor, rightMotor):
    retLeft = sim.simxSetJointTargetVelocity(client, leftMotor, 0, sim.simx_opmode_oneshot_wait)
    retRight = sim.simxSetJointTargetVelocity(client, rightMotor, 0, sim.simx_opmode_oneshot_wait
)
    return retLeft | retRight
```

9 pav. Judėjimo funkcijos

```
# Gets the position of an object in the scene
def getPosition(client, handle):
    retVal, pos = sim.simxGetObjectPosition(client, handle, -1, sim.simx_opmode_oneshot_wait)
    return pos

# Gets the rotation of an object in the scene
def getRotation(client, handle):
    retVal, rot = sim.simxGetObjectOrientation(client, handle, -1, sim.simx_opmode_oneshot_wait)
    return rot

# Gets the distance from the sensor reading
def getDistanceFromSensor(client, sensor):
    return_code, detection_state, detected_point, detected_object_handle, detected_surface_normal
_vector = sim.simxReadProximitySensor(client, sensor, sim.simx_opmode_oneshot_wait)
    if detection_state:
        dist = np.sqrt(np.power(detected_point[0], 2) + np.power(detected_point[1], 2))
    else:
        dist = np.inf
    return dist

# Checks if the two point are close one another
def isApproximatePosition(source, dest, error):
    retVal = True;
    # We only care about XY surface position
    retVal = retVal & (abs(source[0] - dest[0]) < error)
    retVal = retVal & (abs(source[1] - dest[1]) < error)
    return retVal

# Checks if the two rotations are close one another
def isApproximateRotation(source, dest, error):
    return abs(source - dest) < error
```

10 pav. Orientavimosi funkcijos

```
# Normalizes the angle to be between 0 and 2pi
def normalizeAngle(angle):
    if (angle < 0):
        return angle + 2 * np.pi
    else:
        return angle
```

11 pav. Kampo normalizacija

```
# Rotates the robot until the desired angle is reached
def rotateUntilAngle(client, robot, leftMotor, rightMotor, angle, speed = 0.2, error = 0.01):
    rot = getRotation(client, robot)
    # Decide which direction to turn
    willTurnLeft = True
    normAngle = normalizeAngle(angle)
    normRot = normalizeAngle(rot[2])
    if normAngle > normRot:
        diff = normAngle - normRot
        if diff > np.pi:
            willTurnLeft = False
    else:
        diff = normRot - normAngle
        if diff < np.pi:
            willTurnLeft = False
    if willTurnLeft:
        while not isApproximateRotation(rot[2], angle, error):
            turnLeft(client, leftMotor, rightMotor, speed)
            rot = getRotation(client, robot)
    else:
        while not isApproximateRotation(rot[2], angle, error):
            turnRight(client, leftMotor, rightMotor, speed)
            rot = getRotation(client, robot)
    stop(client, leftMotor, rightMotor)

# Gets a desired angle that the source must be at to look at target
def getDesiredRotation(source, target):
    return np.arctan2(target[1] - source[1], target[0] - source[0])
```

12 pav. Sukimosi iki tam tikro kampobei norimo kampo gavimo funkcijos

```
# Rotates the robot toward a given target
def rotateTowards(client, robot, leftMotor, rightMotor, destinationHandle):
    destPos = getPosition(client, destinationHandle)
    # Repeat rotation 3 times for better accuracy
    # Because the center of the robot changes when rotating
    for i in range(3):
        if (i == 0):
            speed = 1.5
            error = 0.1
        else:
            speed = 0.2
            error = 0.01
        robPos = getPosition(client, robot)
        desAngle = getDesiredRotation(robPos, destPos)
        rotateUntilAngle(client, robot, leftMotor, rightMotor, desAngle, speed, error)
```

13 pav. Sukimosi link taško funkcija

```
# Moves the robot forward for N seconds
def moveForwardFor(client, leftMotor, rightMotor, speed, moveFor):
    time_end = time.time() + moveFor;
    while time.time() < time_end:
        moveForward(client, leftMotor, rightMotor, speed)
```

14 pav. Judėjimo nurodytą laiką funkcija

```
# Makes the robot follow a wall until the robot has turned by 90 degrees
def wallFollowRHS(client, robot, leftMotor, rightMotor, sensor, speed):
    dist = getDistanceFromSensor(client, sensor)
    rotCoords = getRotation(client, robot)
    rot = normalizeAngle(rotCoords[2])
    prevRot = 0
    sumRotDelta = 0
    deg90 = np.pi / 2
    halfSpeed = speed / 3
    leftSpeed = speed
    rightSpeed = halfSpeed
    delta = 0
    prevDist = dist
    while sumRotDelta > -deg90 and sumRotDelta < deg90:
        if (dist < 0.08):
            leftSpeed = halfSpeed
            rightSpeed = speed
        elif (dist > 0.1):
            leftSpeed = speed
            rightSpeed = halfSpeed
        move(client, leftMotor, leftSpeed, rightMotor, rightSpeed)
        dist = getDistanceFromSensor(client, sensor)
        prevDist = dist
        rotCoords = getRotation(client, robot)
        rot = normalizeAngle(rotCoords[2])
        deltaRot = rot - prevRot
        prevRot = rot
        if (deltaRot < 0.1):
            sumRotDelta = sumRotDelta + deltaRot
    stop(client, leftMotor, rightMotor)

# Makes the robot follow a wall until the robot reached the destination line or has turned by 360
 degrees
def wallFollowRHS2(client, robot, leftMotor, rightMotor, sensor, speed, destPos, robPos):
    dist = getDistanceFromSensor(client, sensor)
    rotCoords = getRotation(client, robot)
    rot = normalizeAngle(rotCoords[2])
    prevRot = 0
    sumRotDelta = 0
    degrees = np.pi * 2
    halfSpeed = speed / 2
    leftSpeed = speed
    rightSpeed = halfSpeed
    delta = 0
    prevDist = dist
    times = 0
    while sumRotDelta > -degrees and sumRotDelta < degrees:
        if (dist < 0.04): # 0.08
            leftSpeed = halfSpeed
            rightSpeed = speed
        elif (dist > 0.06):
            leftSpeed = speed
            rightSpeed = halfSpeed
        if times > 20:
            if distanceToLine(getPosition(client, robot), destPos, robPos) < 0.01:
                print('Distance to line < 0.01')
                break
        times += 1
        move(client, leftMotor, leftSpeed, rightMotor, rightSpeed)
        dist = getDistanceFromSensor(client, sensor)
        prevDist = dist
        rotCoords = getRotation(client, robot)
        rot = normalizeAngle(rotCoords[2])
        deltaRot = rot - prevRot
        prevRot = rot
        if (deltaRot < 0.1):
            sumRotDelta = sumRotDelta + deltaRot
    stop(client, leftMotor, rightMotor)
```

15 pav. Sienos sekimo algoritmai

```
# Makes the robot reach destination1 by using the bug0 algorithm
def bug0(client, robot, leftMotor, rightMotor, sensors, minWallDist = 0.15):
    print('BUG0 - started.')
    destHandle = getHandle(client, 'destination1')
    criticalDist = minWallDist / 3
    destPos = getPosition(client, destHandle)
    robPos = getPosition(client, robot)
    rotateTowards(client, robot, leftMotor, rightMotor, destHandle)
    while not isApproximatePosition(robPos, destPos, 0.05):
        dist1 = getDistanceFromSensor(client, sensors[3])
        dist2 = getDistanceFromSensor(client, sensors[4])
        if (dist1 < minWallDist and dist2 < minWallDist) or dist1 < criticalDist or dist2 < criti
calDist:
            # rotate left
            prevDist1 = np.inf
            prevDist2 = np.inf
            sensDist1 = getDistanceFromSensor(client, sensors[7])
            sensDist2 = getDistanceFromSensor(client, sensors[8])
            print('rotating left')
            diff = np.abs(sensDist1 - sensDist2)
            while sensDist2 == np.inf or sensDist1 < prevDist1 or sensDist2 < prevDist2:
                turnLeft(client, leftMotor, rightMotor, 1)
                prevDist1 = sensDist1
                sensDist1 = getDistanceFromSensor(client, sensors[7])
                prevDist2 = sensDist2
                sensDist2 = getDistanceFromSensor(client, sensors[8])
                diff = np.abs(sensDist1 - sensDist2)
            print('moving until no obstacle')
            wallFollowRHS(client, robot, leftMotor, rightMotor, sensors[7], 1.5)
            stop(client, leftMotor, rightMotor)
            # rotate towards target
            print('rotating towards target')
            rotateTowards(client, robot, leftMotor, rightMotor, destHandle)
        moveForward(client, leftMotor, rightMotor, 1.5)
        robPos = getPosition(client, robot)
    stop(client, leftMotor, rightMotor)
    print('BUG0 - destination reached!')
```

16 pav. bug0 algoritmas

```
# Checks if the two given values are approximate to one another
def isApproximate(val1, val2, error = 0.1):
    return (np.abs(val1 - val2)) < error

# Turns the robot by 90 degrees or turns the robot to face these angles (0, 90, 180, 270)
def turn90Degrees(client, robot, leftMotor, rightMotor, direction, speed = 0.2):
    stop(client, leftMotor, rightMotor)
    rot = getRotation(client, robot)
    deg90 = np.pi / 2
    if direction == 'right':
        if isApproximate(rot[2], 0):
            rotateUntilAngle(client, robot, leftMotor, rightMotor, -deg90)
        elif isApproximate(rot[2], deg90):
            rotateUntilAngle(client, robot, leftMotor, rightMotor, 0)
        elif isApproximate(rot[2], -deg90):
            rotateUntilAngle(client, robot, leftMotor, rightMotor, np.pi)
        else:
            rotateUntilAngle(client, robot, leftMotor, rightMotor, deg90)
    elif direction == 'left':
        if isApproximate(rot[2], 0):
            rotateUntilAngle(client, robot, leftMotor, rightMotor, deg90)
        elif isApproximate(rot[2], deg90):
            rotateUntilAngle(client, robot, leftMotor, rightMotor, np.pi)
        elif isApproximate(rot[2], -deg90):
            rotateUntilAngle(client, robot, leftMotor, rightMotor, 0)
        else:
            rotateUntilAngle(client, robot, leftMotor, rightMotor, -deg90)
```

17 pav. Pasisukimas 90 laipsnių kampu

```
# Makes the robot to complete the maze using the right hand side algorithm
def maze(client, robot, leftMotor, rightMotor, frontSensor, rightSensor, leftSensor):
    print('Maze by right hand rule - started')
    stop(client, leftMotor, rightMotor)
    destHandle = getHandle(client, 'destination2')
    robPos = getPosition(client, robot)
    destPos = getPosition(client, destHandle)
    while not isApproximatePosition(robPos, destPos, 0.2):
        distanceFront = getDistanceFromSensor(client, frontSensor)
        distanceRight = getDistanceFromSensor(client, rightSensor)
        distanceLeft = getDistanceFromSensor(client, leftSensor)
        moveForward(client, leftMotor, rightMotor, 2)
        if distanceFront < 0.06:
            print('Wall infront')
            stop(client, leftMotor, rightMotor)
            if distanceRight == np.inf:
                print('No wall on the right - turning 90 degrees right')
                turn90Degrees(client, robot, leftMotor, rightMotor, 'right')
            elif distanceLeft == np.inf:
                print('No wall on the left - turning 90 degrees left')
                turn90Degrees(client, robot, leftMotor, rightMotor, 'left')
            else:
                print('Cannot turn right or left - turning around')
                turn90Degrees(client, robot, leftMotor, rightMotor, 'right')
                turn90Degrees(client, robot, leftMotor, rightMotor, 'right')
        if distanceRight == np.inf:
            print('I can go right')
            stop(client, leftMotor, rightMotor)
            moveForwardFor(client, leftMotor, rightMotor, 2, 2)
            turn90Degrees(client, robot, leftMotor, rightMotor, 'right')
            moveForwardFor(client, leftMotor, rightMotor, 2, 2.5)
        robPos = getPosition(client, robot)
    print('Maze completed')
```

18 pav. Labirinto algoritmas

```
# Calculates the distance towards the target line
def distanceToLine(p0, initialPosition_, desiredPosition_):
    # p0 is the current position
    # p1 and p2 points define the line
    # initialPosition_ pradine pos
    # desiredPosition_ galutine pos
    p1 = initialPosition_
    p2 = desiredPosition_
    # here goes the equation
    up_eq = math.fabs((p2[1] - p1[1]) * p0[0] - (p2[0] - p1[0]) * p0[1] + (p2[0] * p1[1]) - (p2[1
] * p1[0]))
    lo_eq = math.sqrt(pow(p2[1] - p1[1], 2) + pow(p2[0] - p1[0], 2))
    distance = up_eq / lo_eq
    return distance
```

19 pav. Skaičiuoja atstumą iki tikslo linijos

```
# Makes the robot reach destination1 by using the bug2 algorithm
def bug2(client, robot, leftMotor, rightMotor, sensors, minWallDist = 0.15):
    print('BUG2 - started.')
    destHandle = getHandle(client, 'destination3')
    criticalDist = minWallDist / 3
    destPos = getPosition(client, destHandle)
    robPos = getPosition(client, robot)
    rotateTowards(client, robot, leftMotor, rightMotor, destHandle)
    while not isApproximatePosition(robPos, destPos, 0.05):
        dist1 = getDistanceFromSensor(client, sensors[3])
        dist2 = getDistanceFromSensor(client, sensors[4])
        if (dist1 < minWallDist and dist2 < minWallDist) or dist1 < criticalDist or dist2 < criti
calDist:
            # rotate left
            prevDist1 = np.inf
            prevDist2 = np.inf
            sensDist1 = getDistanceFromSensor(client, sensors[7])
            sensDist2 = getDistanceFromSensor(client, sensors[8])
            print('rotating left')
            diff = np.abs(sensDist1 - sensDist2)
            while sensDist2 == np.inf or sensDist1 < prevDist1 or sensDist2 < prevDist2:
                turnLeft(client, leftMotor, rightMotor, 1)
                prevDist1 = sensDist1
                sensDist1 = getDistanceFromSensor(client, sensors[7])
                prevDist2 = sensDist2
                sensDist2 = getDistanceFromSensor(client, sensors[8])
                diff = np.abs(sensDist1 - sensDist2)
            print('moving until no obstacle')
            wallFollowRHS2(client, robot, leftMotor, rightMotor, sensors[7], 1.5, destPos, robPos
)
            stop(client, leftMotor, rightMotor)
            # rotate towards target
            print('rotating towards target')
            rotateTowards(client, robot, leftMotor, rightMotor, destHandle)
        moveForward(client, leftMotor, rightMotor, 1.5)
        robPos = getPosition(client, robot)
    stop(client, leftMotor, rightMotor)
    print('BUG2 - destination reached!')
```

20 pav. Bug2 algoritmas

```
def main():
    clientID = connect()
    robot = getHandle(clientID, 'Pioneer_p3dx')
    leftMotor = getHandle(clientID, 'Pioneer_p3dx_leftMotor')
    rightMotor = getHandle(clientID, 'Pioneer_p3dx_rightMotor')
    sensors = []
    for i in range(16):
        sensor = getHandle(clientID, 'Pioneer_p3dx_ultrasonicSensor' + str(i + 1))
        sensors.append(sensor)
    stop(clientID, leftMotor, rightMotor)
    bug0(clientID, robot, leftMotor, rightMotor, sensors)
    stop(clientID, leftMotor, rightMotor)
    rotateUntilAngle(clientID, robot, leftMotor, rightMotor, np.pi, 0.1, 0.005)
    destPos = [2.5, 8, 0]
    robPos = getPosition(clientID, robot)
    while not isApproximatePosition(robPos, destPos, 0.1):
        moveForward(clientID, leftMotor, rightMotor, 1)
        robPos = getPosition(clientID, robot)
    stop(clientID, leftMotor, rightMotor)
    maze(clientID, robot, leftMotor, rightMotor, sensors[4], sensors[7], sensors[0])
    stop(clientID, leftMotor, rightMotor)
    moveForwardFor(clientID, leftMotor, rightMotor, 2, 8)
    stop(clientID, leftMotor, rightMotor)
    bug2(clientID, robot, leftMotor, rightMotor, sensors)
    stop(clientID, leftMotor, rightMotor)
    disconnect(clientID)

if __name__ == "__main__":
    main()
```

21 pav. Pagrindinis kodas

Roboto valdymo eksperimentinis tyrimas

Robotas gali įveikti bet kokį įveikiamą labirintą pagal dešinės rankos taisyklę, tačiau labirinto sienos turi būti lygiagrečios x ir y ašims, kad robotukas tiksliai pasisuktų 90 laipsnių kampu. Užduotis yra įveikiama šiek tiek greičiau nei per 8 minutes. Daugiausiai laiko užtrunka 90 laipsnių pasisukimai dėl duoto mažo sukimosi greičio, kad pasisukimai butų kuo tikslesni.

Išvados

1. Tomo Kašelyno (labirinto įveikimas pagal dešinės rankos taisyklę, labirinto pastatymas simuliacijoje, testavimas, ataskaitos pildymas) išvados:
   1.1. Ne visus labirintus galima įveikti pagal dešinės rankos taisyklę.
   1.2. Svarbu užtikrinti, kad robotukas judėtų kuo lygiagrečiau labirinto sienoms, nes kitaip gali aptikti sieną ir pradėti suktis netinkamu laiku.
   1.3. Norint įveikti labirintą pagal dešinės rankos taisyklę, robotukas turi teikti prioritetą pasisukimams į dešinę.
   1.4. Aptikęs galimą pasisukimą į dešinę, robotukas turi iš karto suktis į dešinę.
2. Luko Žaromskio (bug0 algoritmo realizacija, 90 laipsnių pasisukimo realizacija, kliūčių pridėjimas iki labirinto, testavimas, ataskaitos pildymas) išvados:
   2.1. Ne visas kliūtis galima apvažiuoti naudojant bug0 algoritmą.
   2.2. Reikia nustatyti tinkamą atstumą iki kurio gali privažiuoti robotas, nes kitu atveju jis gali atsitrenkti į sieną.
   2.3. Reikia nuspręsti, kada kliūtis yra laikoma apvažiuota, o tai gali paveikti kaip greitai jis įveikia kliūtį.
3. Giedriaus Rastausko (bug2 algoritmo realizacija, kliūčių pridėjimas iki labirinto, testavimas, ataskaitos pildymas) išvados:
   3.1. Testuojant ant didesnio simuliacijos greičio buvo gaunami grubūs įverčiai, dėl ko buvo manoma, kad algoritmas veikia netinkamai.

Naudota literatūra

[1] „Coppelia Robotics Remote API functions (Python),“ [Tinkle]. Available: https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm. [Kreiptasi 8 December 2020].