AERSP 424: Advanced Computer Programming

Homework 2 Spring 2023

Submission Instructions:

- Submit a .zip files containing your .cpp and/or .h files.
- Your code needs to be compilable.
- If you upload an updated submission, please remove the previous submission as only the final submission will be graded.
- Using comments to explain your code is mandatory.
- Submission deadline is at 11:59 PM on Friday 3/3/2023 (The late policy mentioned in the syllabus will be applied).
- Explicitly declare variables with a proper name (easy to understand) and datatype (reflect the real-world scenario if possible).

Question 1 (40 points): *Containers and Algorithms*

Assuming you are operating three robots to move from one point to another inside a grid world. In this world, each grid is either wall, road, lake, or wood. Each robot has a cost (how difficult it is for the robot to move in each type of grid) associated with the movement over each grid type shown in the following table.

	Wall	Road	Lake	Wood
Robot 1	999	1	10	10
Robot 2	999	5	1	10
Robot 3	999	5	10	1

Write a code to find a path with a minimum cost/effort for each robot.

1. Pick an appropriate container to construct a 5x5 grid world that can contain a cost for each grid.
2. Make the initial point be (0,0) and the goal be (4,4). Both of these grids have a cost of zero. Note: an index starts from zero.
3. Randomly assign a type to the remaining grids as follows: 2 grids for walls, 4 grids for lakes, 4 grids for woods, and the rest are roads. Note: use a code for the random assignment. No manual assignment.

4. Print the grid world to visualize.
5. Each robot can only move left, right, up, down (no diagonal).
6. Implement the pseudocode of the A* algorithm to find the path.

Initialization:
to_be_search.add(grid=**initial_grid**, priority=0)
searched_grid = None
cumulative_cost[grid=**initial_grid**] = 0

Loop:
while **to_be_search** not empty:
  current_grid = **to_be_search**[grid=**lowest cost grid**] **// tricky part is how to get the lowest cost one.**

  if current_grid is **goal_grid**:
    searched_grid[grid**=goal_grid] = current_grid** **// implies moving to goal_grid from current_grid.**
    stop the iteration.

  for each **neighbor** in **neighbors_of_the_current_grid**: **// the number of neighbors is in between 2 to 4.**
    new_cost = **cummulative_cost**[grid=**current_grid**] + **cost_of_moving_to_neighbor_in_grid_world**
    // the following if statement check if the neighbor has already been visited.
    // If not, it will be added into to_be_search.
    // If it has but the new_cost is lower, it will be updated.
    if **neighbor** not in **cumulative_cost** OR **new_cost** < **cumulative_cost[**grid**=neighbor]**:
      cumulative_cost[grid**=neighbor]** = **new_cost**
      cost = **new_cost + abs (x goal - x neighbor) + abs (y goal - y neighbor)** **// cost + some kind of distance**
      to_be_search.add(**neighbor**, **cost**)
      searched_grid[grid=**neighbor**] = **current_grid** **// implies moving to neighbor from current_grid.**

Backward Reconstruction:
current_grid = **goal_grid**
path = None
while current_grid is not **initial_grid**: **// iteratively trace back from goal_grid to initial_grid.**
  path.add(**current_grid**)
  current_grid = **searched_grid**[grid=**current_grid**]
path.add(**initial_grid**)
path.reverse()

---

7. Find and print the path and the cost for each robot.

Below is the example result.

```
-----------------------------------
| INIT | road | wall | road | road |
-----------------------------------
| road | wall | lake | road | lake |
-----------------------------------
| road | lake | lake | wood | wood |
-----------------------------------
| road | road | road | wood | road |
-----------------------------------
| wood | road | road | road | GOAL |
-----------------------------------
Robot 1's path is (0,0)->(1,0)->(2,0)->(3,0)->(3,1)->(3,2)->(4,2)->(4,3)->(4,4) with a cummulative cost of 7
Robot 2's path is (0,0)->(1,0)->(2,0)->(2,1)->(2,2)->(3,2)->(4,2)->(4,3)->(4,4) with a cummulative cost of 27
Robot 3's path is (0,0)->(1,0)->(2,0)->(3,0)->(3,1)->(3,2)->(3,3)->(3,4)->(4,4) with a cummulative cost of 31
```

Question 2 (30 points): *Class and Operator Overloading*

Write two classes. The first one is a class of 3-dimensional column vector named **'Vector'**. The other one is a class of 3x3 matrix named **'Matrix'**. Initialize the value of their element as a real number. Then, implement the following features by overloading given operators in the parenthesis.

1. Element-wise vector addition, e.g., $x + y$ (+).
2. Element-wise vector subtraction, e.g., $x - y$ (-).
3. Dot-product of two vectors, e.g., $x \cdot y$ or $x^T y$ (,).
4. Cross-product of two vectors, e.g., $x \times y$ (*).
5. Element-wise matrix addition, e.g., $A + B$ (+).
6. Element-wise matrix subtraction, e.g., $A - B$ (-).
7. Matrix-Matrix multiplication, e.g., AB (*).
8. Matrix-Vector multiplication, e.g., Ax (*).
9. Matrix transpose, e.g., A^T(~).
10. Vector-transpose-Matrix-Vector multiplication, e.g., $x^T Ax$ (A combination of 3. and 8.).

Pick your favorite matrices and vectors to test the results. Outside of these two classes, write operator overloading functions (<<) to print the results.

Question 3 (30 points): *Class and Inheritance*

To emulate a sensor, we usually use a simulated result from dynamical equations as a truth data. During the class lecture, we rewrite the first question of Homework 1 in a form of an OOP that generate a truth data. This question investigates further on how to emulate a gyroscope sensor. For simplicity, let the emulated data, ω_{gyr}, of the gyroscope depend on the following information.

1. Angular velocities of the vehicle (p, q, r variables in HW1Q1).
2. White/Gaussian noise vector from the standard normal distribution.
3. A constant bias error vector, b_ω.

4. The mounting orientation $(\phi_{gyr}, \theta_{gyr}, \psi_{gyr})$ of the sensor relative to the C.G. of the vehicle.

Mathematically, expressed as

$$\omega_{gyr} = R(\phi_{gyr}, \theta_{gyr}, \psi_{gyr}) \begin{bmatrix} p \\ q \\ r \end{bmatrix} + \begin{bmatrix} b_{\omega_x} \\ b_{\omega_y} \\ b_{\omega_z} \end{bmatrix} + \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \end{bmatrix}$$

where

$$R(\phi_{gyr}, \theta_{gyr}, \psi_{gyr}) = \begin{bmatrix} c_\theta c_\psi & c_\theta s_\psi & -s_\theta \\ -c_\phi s_\psi + s_\phi s_\theta c_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & s_\phi c_\theta \\ s_\phi s_\psi + c_\phi s_\theta c_\psi & -s_\phi c_\psi + c_\phi s_\theta s_\psi & c_\phi c_\theta \end{bmatrix}$$

and each ε is drawn from the standard normal distribution.

- Write a class **'Gyroscope'** class that inherits the **'FlightSim'** class from the lecture, **'Vector'** and **'Matrix'** classes from the Question2.
- Pick any real numbers for the bias vector, except all being zero.
- Pick any real numbers for the sensor mounting orientation vector, except all being zero.
- Perform the same iteration as in HW1Q1 using an object instantiated from the **'Gyroscope'** class by appropriately calling *dynamics*(), *integrate*(), and operator overloading functions.
- In each iteration, draw each ε from the standard normal distribution.
- Print out the final result.

(Optional) To result can be check by, setting all ε to be zero during the iteration process, compute

$$\begin{bmatrix} \hat{p} \\ \hat{q} \\ \hat{r} \end{bmatrix} = R^T(\phi_{gyr}, \theta_{gyr}, \psi_{gyr}) \left(\omega_{gyr} - \begin{bmatrix} b_{\omega_x} \\ b_{\omega_y} \\ b_{\omega_z} \end{bmatrix} \right)$$

and see if $\begin{bmatrix} \hat{p} \\ \hat{q} \\ \hat{r} \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$. The following code draw a random number from the standard normal distribution.

```cpp
#include <iostream>
#include <random>

double draw_from_standard_normal_dist()
{
  std::random_device rd{};
  std::mt19937 gen{ rd() };
  std::normal_distribution<> d{ 0, 1 };
  return d( gen );
}

int main()
{
  double epsilon = draw_from_standard_normal_dist();
  std::cout << epsilon << std::endl;
}
```