Introduction

This is a worksheet for the third COMP3071 lab.

In this session, you'll work on two tasks.

The objective of the first task is to update the existing code to enable a systematic comparison of various parameter settings against the average amount of dirt collected by the robot vacuum cleaner system. In the second task, the goal is to integrate a planning algorithm that can efficiently direct the movement of the robot towards the least costly path between two given points.

Today's activities are important for various reasons within the context of the robot vacuum cleaner system. By conducting a systematic comparison of different parameter settings, the system can be optimized to collect the maximum amount of dirt in the least amount of time. This can lead to a more effective cleaning process, ultimately saving time and energy while enhancing the system's overall performance. Moreover, by identifying the most optimal path between two points, the robot can save energy and time during its movement, resulting in faster and more efficient cleaning. This is especially crucial in large areas where the robot needs to cover a significant distance as it can substantially reduce the time and energy required to complete the task.

Task 1: Getting Started

To start, download the "`simpleBot2_withCounting.py`" file from the module Moodle page. Ensure that you can run the file. It's a solution for the counting task from last week. You can either run the file from the command line or import it into an IDE.

Task 1

The objective of the first task is to modify `simpleBot2_withCounting.py` to enable a methodical comparison of various parameter settings concerning the quantity of dirt collected by the robot vacuum cleaner system. Specifically, we want to analyze how the number of bots affects the total amount of dirt collected. For each configuration, the experiment will be repeated multiple times.

To achieve this goal, the following modifications can be implemented:

1. Rename the `main` method as `runOneExperiment`
2. Update the `runOneExperiment` method to take a parameter called `numberOfBots`.
3. Modify the "`register`" function and its call in the `runOneExperiment` method to enable the supply of different `numberOfBots`"values.
4. Instead of just printing the average amount of dirt collected, update the code to extract this information by modifying the `moveIt()` method.

A. Let's consider how we can achieve this requirement. Previously, the system was set to exit once the following `if-statement` was fulfilled.

```
if moves>numberOfMoves:
    sys.exit()
```

B. To accomplish this, comment out `sys.exit()` and replace it with `window.destroy()` to close the window instead of exiting.

C. After this modification, an element related to the `window` variable is missing. This element needs to be added to the `moveIt()` method.

D. The `runOneExperiment()` method also needs to be updated to return the total amount of dirt collected.

5. In the updated `runOneExperiment()` method, add the following argument at the end of the method to return the total amount of dirt collected

```
return count.dirtCollected
```

6. Test the code by running the `runOneExperiment(numberOfBots)` function at the end of the file. Assign any value to `numberOfBots`. This code should work if all the steps from 1 to 5 are followed.

   A. Alternatively, the line can be written as `print(runOneExperiment(5))` to print the total amount of dirt collected by `5` bots to the console.

7. Note that the line `runOneExperiment()` at the end of the file is only for development purposes. It should be commented out when implementing the actual code. Use this line for development from steps 1 to 5 only.

```
# runOneExperiment()
```

8. A new function can be introduced in a new `.py` file to examine how the number of bots affects the amount of dirt collected and run the experiment multiple times for each configuration. This function can be named `runAllExperiments.py` and should contain the following methods:

   A. Add a new function called `runSeveralTimes(numberOfBots, numberOfRuns)` that runs the `main_with_counting` function a specific number of times (determined by the `numberOfRuns` parameter) and returns the average amount of dirt collected.

   B. Add another function called `runSeveralExperiments()` that

      i. creates a list `[1,2,3…10]` representing the number of bots in each experiment. It loops through this list and calls the `runSeveralTimes()` function for each number.

ii. At the end of this process, a table of the number of bots versus the amount of dirt collected should be displayed. This table can be printed in text on the console, or
iii. visualized as a graph such as a barchart or line chart if you are familiar with a plotting package.
iv. Additionally, if you are familiar with statistical analysis, you can calculate whether there is a significant difference between the average dirt gathered for each successive pair of experiments.

Task 2: Getting Started

First, download the `simpleBot3.py` and `aStar.py` files from the module's Moodle page. Make sure both files is executable either through the command line or by importing them into an IDE.

The `simpleBot3.py` is similar to the program from the previous weeks, but with the addition of a matrix called `map`. The map shows the location of dirt on a `10x10` grid. Imagine that there is a camera on the room's ceiling that shows the location of the dirt and relays it to the robot. Upon execution of the program, the `map` (i.e., coordinate) is printed to the console.

The goal of this task is to integrate the A* search algorithm into `simpleBot3.py` to guide the robot in cleaning up the room.

It is beneficial to familiarize both `simpleBot3.py` and `aStar.py`.

`simpleBot3.py` contains the code for the robot and the map of the room, while `aStar.py`"contains the A* search algorithm that finds a path through a network.

A* is a search algorithm that you may have learned if you've taken a AI-related moddule or algorithms. If not, you can watch this video (https://www.youtube.com/watch?v=6TsL96NAZCo), which provides a clear explanation of it.

The version of A* implemented in `aStar.py`" conveniently finds a route through a 10 x 10 grid.

When you run `aStar.py`, it will display three things in the console:

i. the grid to be searched (which represents the dirt on the map),
ii. the list of points to be visited in order, and
iii. the path taken (indicated by 1 for points visited).

Note that the route starts in the location (`9,9`) which is located at the corner and progresses by single steps towards (`0,0`), not taking diagonals to simplify the problem.

To proceed further, try to modify the `simpleBot3.py`" to incorporate the A* algorithm. Subsequently, conduct experiments to compare the performance of the two methods. You may

also consider enhancing the program by visualizing the results or conducting statistical analysis to determine any significant differences between the two methods.

Task 2

The objective of the second task is to improve `simpleBot3.py` with aStar.py so that the robot can move toward the least costly path between two given points.

To achieve this goal, the following modifications can be implemented to the `simpleBot3.py`:

1. To integrate the aStar.py code into simpleBot3.py, add the following line at the beginning of `simpleBot3.py`:

   ```
   from aStar import *
   ```

   This imports all the functions from the `aStar.py` file into `simpleBot3.py`.

2. Now that the search algorithm is available in `simpleBot3.py`, you can apply it to the `map`.
   A. In the main method, call the `aStarSearch function` with the map as the argument, and store the result in a variable called path:
   B. Inspect the information of the path variable by using the `print` function. If you have made the changes correctly, the generated output should be a list of tuple as below:

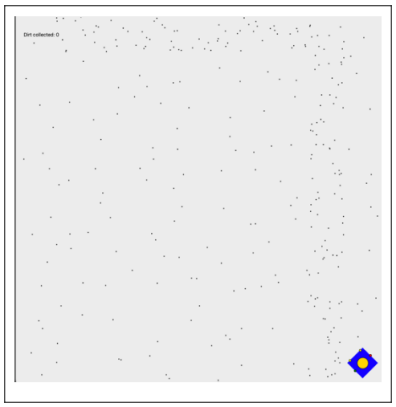
   ```
   [(9, 9), (8, 9), (8, 8), (8, 7), (8, 6), (8, 5),
   (8, 4), (8, 3), (8, 2), (8, 1), (8, 0), (7, 0),
   (6, 0), (5, 0), (4, 0), (3, 0), (2, 0), (1, 0), (0, 0)]
   ```

   These `tuples` represent the successive positions of the bot on the 10x10 map.

   For instance, the bot starts at position (`9, 9`) and moves to (`8, 9`), then to (`8, 8`), and so on, until it reaches the final position at (`0, 0`).

   The next step is to make the bot traverse through these points.

3. Next, place the bot in the corner of the map:
   A. Rather than placing the bot at a random position/angle, specify its position/angle as (`950, 950`) and `-3.0*math.pi/4.0`, respectively.
   B. You can either modify the `__init__` method of the `Bot class` to do this, or create a new method called place in the Bot class.
   C. Run the code to ensure that the bot starts in the specified position.



Tip: To freeze the bot movement, increase the duration of the after method by changing the value of `200` to a longer time, such as `2000`. After confirming that the robot is starting in the correct position, restore the duration to the original value

```
canvas.after(200,moveIt,canvas,registryActives,registryPassives,count,moves,window,this_path)
```

4. In order to enable the transfer function to follow the path generated by the A* algorithm, add the path variable as a parameter to both the `transferFunction` and `moveIt` methods.

5. To steer the robot toward the next position in the path, retrieve the element in the `path`, and call it the `target` variable.

   A. IMPORTANT: Comment out the existing code under the `transferFunction`.
   B. Some notes about the `target` variable
      i. The initial coordinates should be (`9,9`). It's worth noting that a tuple in coordinate form is expressed as (`x,y`).
      ii. Keep in mind that the coordinates in the path appear in the format (8,9), while the grid size is 1000 by 1000.
      iii. To scale up the coordinates, multiply the `x` and `y` values of the `target` by 100, and then add 50 to each result. For instance, the first target's coordinates would be (`950, 950`).
   C. Use the `distanceToLeftSensor` and `distanceToRightSensor` methods to determine the distance from the robot to target.

   D. To steer the robot towards a pair of coordinates, we can modify code from `simpleBot2.py`:
      i. Look at the code that steers the robot towards the charger when the battery is low, there is simple if-else block as follow

```
if self.battery<600:
   if chargerR>chargerL:
         self.vl = 2.0
         self.vr = -2.0
   elif chargerR<chargerL:
         self.vl = -2.0
         self.vr = 2.0
   if abs(chargerR-chargerL)<chargerL*0.1:
         self.vl = 5.0
         self.vr = 5.0
```

The code adjusts the robot's velocities based on the readings from the left and right sensors.

ii. Update the `transferFunction` method to steer the robot towards the target. Modify the existing code (i.e., the `if-else` block) that steers the robot towards the charger to steer the robot towards the target instead.

iii. Add an if statement to check if the robot is close enough to the target. If the distance from either the left or right sensor is less than a specified threshold, remove the current target from the path list so that the robot can focus on the next one

iv. **Optional**: You can also add text and a circle to indicate the location of the target. Visualizations can be helpful for development purposes. To do this, use the `canvas.create_text()` method and `canvas.create_oval()` method under the draw method to draw the text and circle, respectively.

6. Check whether all coordinate pairs have been completely traversed. There are several ways to do this.
   - One way is to check if the path list is empty at the end of each call to the `transferFunction`. If the path list is empty, return the string `finished`. Additionally, if the `path` list is empty or if the number of moves exceeds a specified threshold, end the simulation, OR
   - Check if the number of `moves` doesn't exceed the `threshold` and if the current `path index` equals the `path size`. If both conditions are met, print the total amount of dirt collected and indices in the path, then destroy the window.

Experiments

The next task is to conduct experiments to determine whether the two behaviors differ, as we did in Task 1.

1. Using the same framework as Task 1, create a similar setup that compares:
   - The code you just wrote
   - The original 'wandering' code from the transfer function that you commented out earlier."
2. Replace 'main' with a function that takes a parameter (e.g., 'wandering' or 'planned') and returns the amount of dirt gathered.
3. Run the code 100 times in each of the 'wandering' or 'planned' states, and record the amount of dirt collected each time.
4. Visualize the results, calculate some descriptive statistics, and consider performing a t-test to determine whether there is a significant difference."

If you want to achieve exceptional results, consider introducing some randomness to the setup. For instance, you could use a generator that creates various distributions of dirt, and then test whether the A* planning algorithm can still find a solution.

Extensions

Here are some extensions to the tasks above. While I do not expect you to attempt these during the lab session, they can provide ideas that may serve as the basis for your coursework.

1. Using genetic algorithms rather than the planning algorithm. Genetic algorithms find a good route by taking a "population" of different, initially random solutions (i.e. paths), ranking them by the quality of the solution (i.e. amount of dirt collected), and then forming a new population by making small changes ("mutations") to the currently successful paths, or by crossing-over two of the currently successful paths. Find out about this, implement it, and contrast the wandering, planned and genetic solutions (both on amount of dirt collected, and computation time). If you want to do this well, you will need to think carefully about the number of moves allowed by each. If you were to do all of that, and analyse it clearly, it could be a decent coursework project.
2. Re-planning. Re-introduce the "battery" from last week. Make sure that the battery charge runs out before the path is complete, otherwise this task is pointless! Change the transferFunction so that it prioritises going back to the battery when the charge is low. Then, re-plan the best route from the charger to the corner.