Designing Intelligent Agents
Spring 2022
Worksheet 3

Colin Johnson

Introduction

The work in this session contains two ideas.

The first is writing code to do a more systematic comparison between different versions of the robot vacuum cleaner system, i.e. with different numbers of robots.

The work in this session is designed to explore how a planning algorithm can be used to guide an agent. You will be provided with some code for a search algorithm, A* search, and integrate this with the latest version of the *Bot* code from previous weeks. If there is time, you can implement another algorithm. You will then carry out experiments to compare the performance of these planning-based algorithm(s) against a random wandering.

Task 1: Getting Started

Download the file called *simpleBot2_withCounting.py* from the module Moodle page, and **make sure that you can run it.** This is a solution to the counting task from last week. You can run this from the command line, or import the file into an IDE.

Task 1

The aim of this task is to write code to systematically compare multiple parameter settings for the experiment. Edit *simpleBot2_withCounting.py* so that the *main* method now takes a parameter, called *numberOfBots*, and then uses this parameter to set the number of robots in the experiment (you will need to change the *register* function itself, and change the call to *register* within the *main* method. Remove the line *main()* at the end of the file. Change *moveIt* so that it returns the total dirt collected rather than printing it.

Now, add a new function, *runSeveralTimes()* that has parameters called *numberOfBots* and *numberOfRuns*. This runs *main* a number of times (specified by the "number of runs" parameter) and returns the average (mean) of the amount of dirt collected. If you are confident with statistics, you can also calculate the standard deviation of this amount. You might want to see if you can remove the animation of the bot so that the whole thing runs faster.

Finally, add another function, *runSeveralExperiments()*, which creates a list [1,2,3…10] representing the number of bots in each experiment. It loops through this vector, and calls *runSeveralTimes* for each number. At the end of this, display a table of the number of bots vs. the amount of dirt collected (this could just be printed in text on the console, or if you are familiar with the *pandas* library you could export an excel file). If you are familiar with a plotting package, visualise this as a graph such as a barchart or line chart. If you are familiar with statistical analysis, calculate whether there is a significant difference between the average dirt gathered for each successive pair of experiments.

Task 2: Getting Started

Download the files called *simpleBot3.py* and *aStar.py* from the module Moodle page, and **make sure that you can run them.** You can run this from the command line, or import the file into an IDE. If you are using Jupyter notebooks, there is an alternative. The *Bot* is a variant on what you have seen in previous weeks, except that there is also a matrix called *map* which shows, on a 10✕10 grid, where the dirt is. Imagine that there is a camera on the ceiling of the room, which shows where the dirt is, and passes this to the robot. This map is printed to the console when you run the program.

A* is a search algorithm, that finds a route through a network. You may be familiar with it if you have done a course on AI or algorithms; if not, it is described in the lecture about *Intelligence as Search*, or you can watch this video: https://www.youtube.com/watch?v=6TsL96NAZCo, which gives a very clear explanation of it. The version implemented in *aStar.py* conveniently finds a route through a 10✕10 grid. Run *aStar.py*. This displays (in the console) three things: the grid to be searched through (you can think of this as being the dirt in the map), the list of points to be visited in order, and the path taken (indicated by 1 for points visited). Note from looking at the first grid that there is a fairly obvious path to be taken, and that the search algorithm finds it reliably (run the code a few times). Note that the route starts in the (9,9) corner and progresses by single steps towards (0,0), not taking diagonals (this is to make the problem a little simpler).

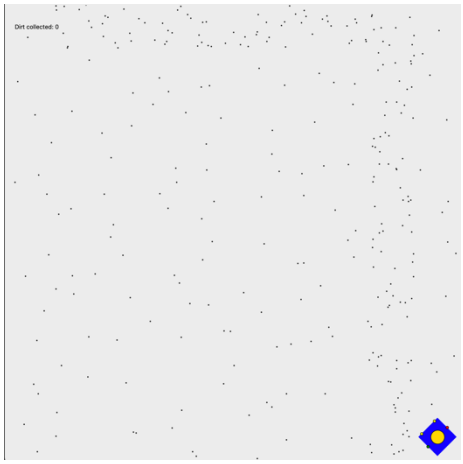
Task 2

The first part of this task is to integrate the two pieces of code. We will use *simpleBot3.py* as our main program, so let's pull in the functions from *aStar.py* into that; i.e. **include the line** *from aStar.py import ** at the beginning of *simpleBot3.py*. Now, we can apply the search algorithm to the map in *simpleBot3.py*. In the main method, before *moveIt*, **take the *map* and apply the *aStarSearch* function to it, storing the result in a variable called *path*.** Using the print function or otherwise, **have a look at what is in path.** If you have done this correctly, you should get a list like this:

```
[(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 represent the successive positions of the bot in the 10✕10 map. That is, the bot starts at 9,9, moves to 8,9, then 8,8, and so on.

Let's make the bot traverse these points. To start with, **place the bot in the corner**. That is, rather than placing it at a random position/angle, you will need to place it at a specific position/angle; let's say (950,950) with an angle of -3.0**math.pi/4.0*. You can either change the *init* method of the *Bot* class to do this, or you can write an additional method in *Bot* called something like *place*. **Run the code**. The bot should start in this position:



Now, let's follow the path that the A* algorithm provided. **Add code so that** the *transferFunction* can see the *path* variable. That is, you will need to pass *path* as a parameter into *moveIt*, and then as a parameter into *transferFunction*, so you will need to add it to the parameter list of both of those. Now, let's steer the robot towards each successive position in the *path*. **Comment out the code** in *transferFunction*. **Get the first item from the *path*** (don't remove it from the path yet), and call it something like *target*. This should initially be (9,9).

Note that the coordinates in *path* are like (8,9) whereas the grid is 1000✕1000. So, you will need to scale it up; **multiply each coordinate by 100, and add 50,** so that the first target, for example is (950,950).

Now, we are going to need to steer the robot towards this pair of coordinates. Remember—we have seen code to do this before! **Have a look back** at *simpleBot2.py* from last time. There is some code like this:

```
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: #approximately the same
        self.vl = 5.0
        self.vr = 5.0
```

that steers the robot towards the charger. We are not going towards the charger this time, but towards the current target point. So, **modify that code and place it in *transferFunction*** so that the robot goes towards the target. You will find the *distanceToLeftSensor* and *distanceToRightSensor* methods useful.

Now, **you will need to check whether the robot is close to the target** point. You can do this by using an *if* statement to check if the distance to either the left sensor or right sensor is below a threshold (that you will need to determine). If so, you should remove the current target point from the *path* so that the robot is now focused on the next one.

Finally, **you will need to check for when you have completed the path**. The simplest way to do this is to check if *path* is empty at the end of each call to *transferFunction*, and if so, return a value (e.g. return the string "finished") and if so, call the code in the "if moves>numberOfMoves…" condition (i.e. you add an *or* to the if statement).

I hope that everyone in the class will reach this stage.

Experiments

The next task is to **carry out experiments to see whether the two behaviours are different** (like we did in Task 1). Have a look at the lecture slides on experimentation and the ideas from Task 1, and write a similar framework that contrasts (1) the code you have just written; and, (2) the original "wandering" code from the transfer function that you commented out earlier. You will need to replace *main* with a function that takes a parameter (e.g. "wandering" or "planned"), and which returns the amount of dirt gathered. By adapting the code from the live coding session (or the slide from the lecture), run the code 100 times in each of the "wandering" or "planned" states and record the amount of dirt collected each time. Visualise this, calculate some descriptive statistics, and perhaps carry out a *t*-test to see if there is a difference.

If you wanted to do this particularly well, you should introduce some randomness to the setup—e.g. having a generator that generates different dirt distributions, and examines whether the A* planning algorithm can still find a solution.

Extensions

Here are some extensions to the tasks above. I am not anticipating that you will attempt these in the class, but they provide ideas that might form the beginnings of your coursework.

- 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.
- 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.