

Repetitive Nearest Neighbor Search for Pac-Man

Introduction

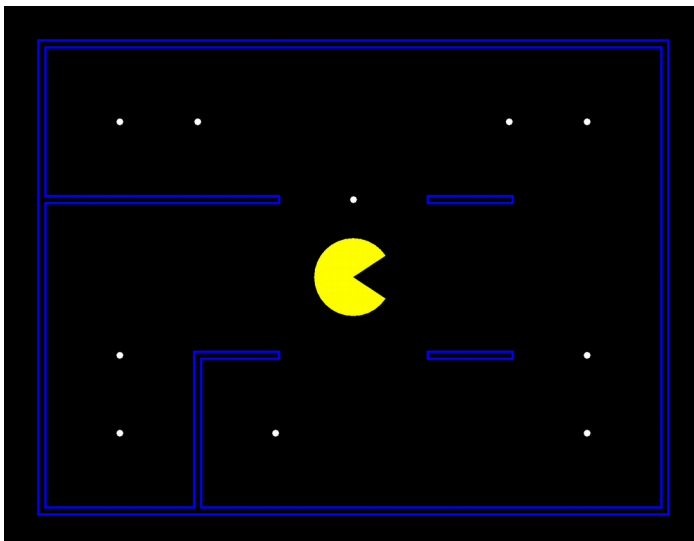
For this assignment you will implement the repetitive nearest neighbor search algorithm using Java as discussed in lecture to drive Pac-Man through mazes to collect food dots in an near-optimal way. You will be provided with the simulation infrastructure in the form of a JAR file, so you will not need to develop any graphics or simulation code. You will also be provided with API documentation in the form of PDF files for the classes in the JAR file that you are allowed to use in your agent program.

You will also be provided with a sample maze for development and a test maze for grading, plus a sample agent that you can compile and run. The sample agent is a simple replanning agent, as we have discussed in class.

Your task is to develop a replacement agent that implements the repetitive nearest neighbor algorithm (RNNA) that we discussed in lecture. In particular, your implementation must branch to explore multiple possibilities whenever there is more than one closest neighbor. Your implementation should be sufficiently general that it can also be run against additional mazes that have not been provided to you. In other words, your agent should not include any code that takes into account the peculiarities of any particular maze. It should be maze agnostic.

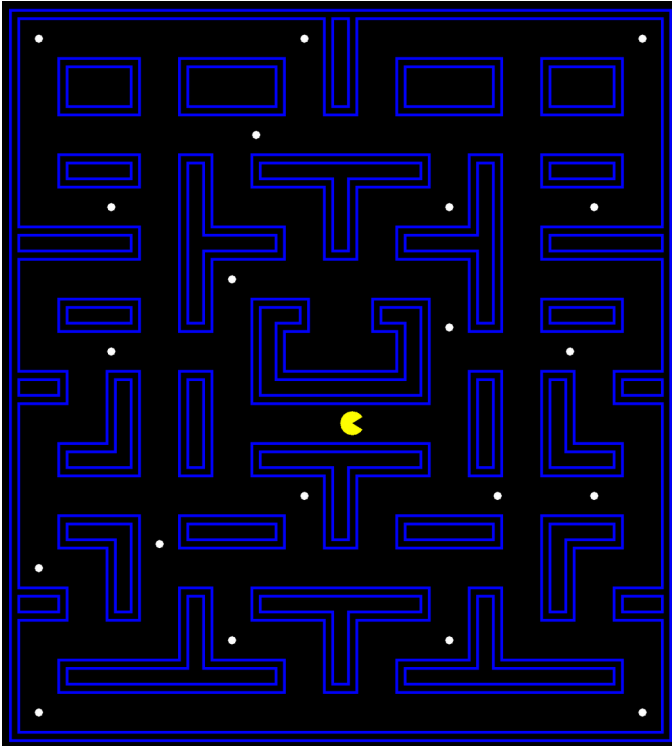
Development and Test Mazes

The following maze, which we call "tsp-tiny", is provided to you for development. It is sufficiently small that you may find it useful for debugging your program. The replanning agent solves this maze in 31 moves. Your RNNA agent should solve it in 27 moves, which also happens coincidentally to be the optimal answer.



The test maze for this assignment is "tsp-original-20", shown below, which contains 20 food pellets. It is not feasible to solve this maze using either UCS or brute-force search. The replanning agent can

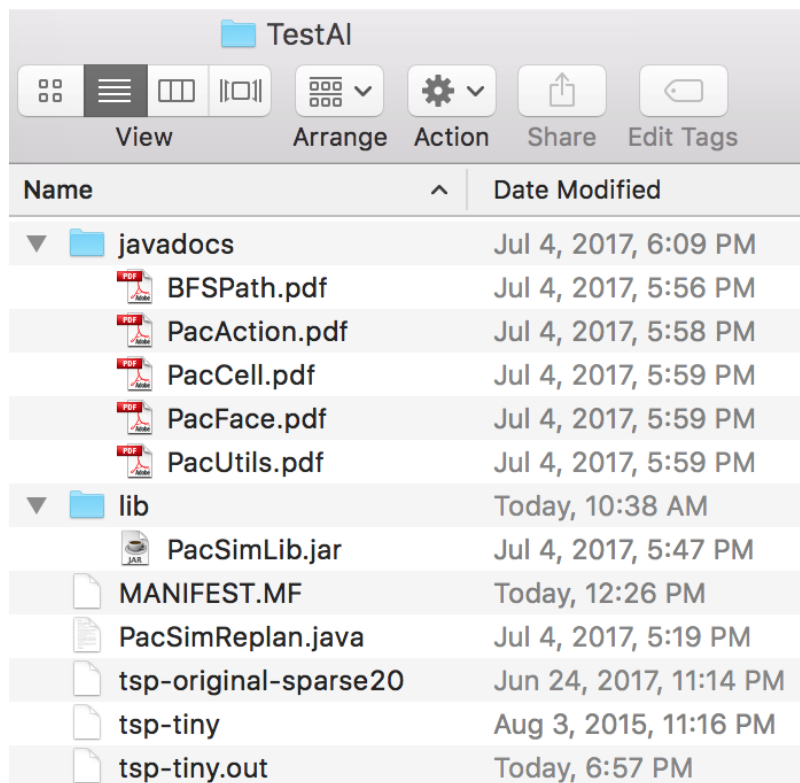
solve this maze in 246 moves. Your RNNA agent should be able to solve it in 217 moves. If your agent solves the maze in more than this number of moves, you will receive only partial credit for this assignment.



Files Provided

The files you will need are contained in the TestAI.zip file, which you can download [here](#).

Decompressing the file produces the following file structure:



Name	Date Modified
▼ javadocs	Jul 4, 2017, 6:09 PM
BFSPath.pdf	Jul 4, 2017, 5:56 PM
PacAction.pdf	Jul 4, 2017, 5:58 PM
PacCell.pdf	Jul 4, 2017, 5:59 PM
PacFace.pdf	Jul 4, 2017, 5:59 PM
PacUtils.pdf	Jul 4, 2017, 5:59 PM
▼ lib	Today, 10:38 AM
PacSimLib.jar	Jul 4, 2017, 5:47 PM
MANIFEST.MF	Today, 12:26 PM
PacSimReplan.java	Jul 4, 2017, 5:19 PM
tsp-original-sparse20	Jun 24, 2017, 11:14 PM
tsp-tiny	Aug 3, 2015, 11:16 PM
tsp-tiny.out	Today, 6:57 PM

The javadocs folder contains the API documentation for the classes that you may need for writing your program efficiently.

The lib folder contains PacSimLib.jar, which is the simulation engine that you will need for compiling and running your agent.

PacSimReplan.java is the simple replanning agent that is provided as a baseline for your development.

MANIFEST.MF is a sample Java manifest file for use in creating executable JAR files, as described in the next section.

The development and test mazes included in this distribution are also provided as binary files: tsp-tiny and tsp-original-sparse-20.

The file tsp-tiny.out is a text file that contains the required output format for an RNNA agent solving the tsp-tiny maze. The output for the test maze should be in the same format.

Compiling and Running the Sample Agent

Before beginning with this assignment, you should make sure that you have downloaded and installed Java on your system and that you can execute Java from the command line. You can check this by opening a command window and entering the command: "java -version". If your system responds with a Java SE version number, then your system is configured correctly.

You should download to your development system all of the files listed above and place them all in a folder on your desktop. Then, open a command window and navigate to the folder where you have placed the files.

To compile the sample agent, type the following command and then press the Enter key:

```
javac -cp lib/PacSimLib.jar PacSimReplan.java
```

To run the sample agent against the "Tiny" maze, type the following command and then press the enter key:

```
java -cp .:lib/PacSimLib.jar PacSimReplan tsp-tiny
```

Please note the period and colon before the library reference. They direct the program to look for files in the current directory. For Windows systems, the colon should be replaced by a semi-colon. The program should run and bring up the GUI screen for the tsp-tiny maze as shown above. Just press the "Start Simulation" button and you should see Pac-Man move about and eat all the food pellets. Notice the move counter in the upper left when the run finishes. You will observe that Pac-Man has eaten all the dots in 31 moves.

Your NNA agent should be able to do better. As described previously, it should be able to eat all the dots on this maze in 27 moves.

To run the against the test maze, use the same run command with the test maze file name.

Creating an Executable JAR File from the command line

To create an executable JAR file from the command line for the replanning agent, enter:

```
jar cfm PacSimReplan.jar MANIFEST.MF *.class *.java
```

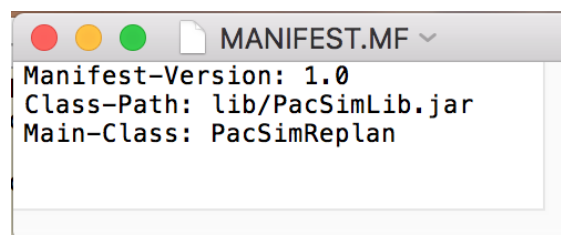
This will create the JAR file PacSimReplan.jar using the replanning agent. It uses the manifest file that is included in this distribution.

To run this executable JAR file against the tsp-tiny maze, enter:

```
java -jar PacSimReplan.jar tsp-tiny
```

You can run against the test maze by simply substituting the test maze name in the command above.

You can also create an executable JAR file for your NNA agent by editing the manifest file and changing the "Main-Class" property to name your agent's class. The manifest file that has been supplied looks like:



A suggested class name for your agent is PacSimRNNA, which (if you use that class name) must be specified in the file PacSimRNNA.java.

Please note that there should be a newline in the file after the name of the main class, so that the jar program can read the main class name.

Implementing Your Agent

Your agent class must interface properly with the simulation engine in order to drive Pac-Man. The sample agent program shows most but not all of the necessary elements. Here are the key elements:

1. Implement the PacAction interface:

By implementing this interface, the sim engine will know that your class contains the action() method, which will be called every time Pac-Man moves one square in the maze.

2. Constructor:

Whatever you name your class, the constructor must be in the same form as what you see in the sample program, for example:

```
public PacSimRNNA( String fname ) {  
    PacSim sim = new PacSim( fname );  
    sim.init(this);  
}
```

3. Main method:

The main method of your class should simply invoke your constructor using the passed-in maze name, for example:

```
public static void main( String[] args ) {  
    new PacSimRNNA( args[ 0 ] );  
}
```

4. Method init() :

The GUI supports running your agent program multiple times. Use this method to reset any variables that must be re-initialized between runs.

5. Method action() :

This method is where you will implement RNNA and drive Pac-Man. You will probably also write a number of utility methods and even some additional classes to support what you do here, but this method will direct their operation. All Java source files must contain the required header, which is described in the submittal section below

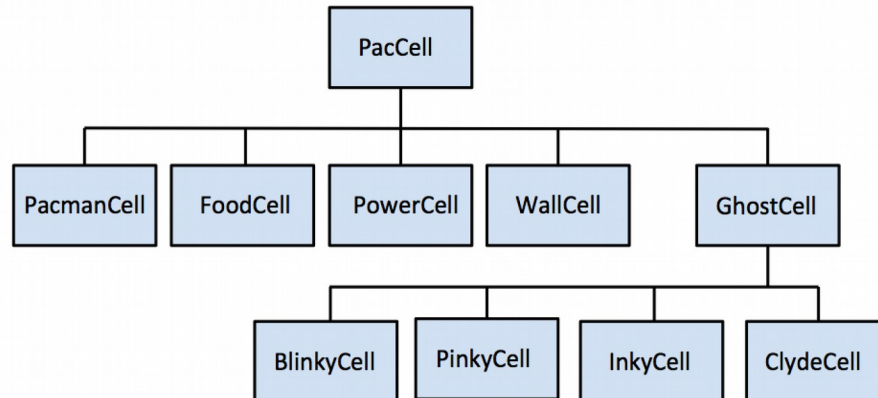
Your action() method must do two things:

(1) *Compute the RNNA solution path through the input maze:*

This is something that the `action()` method must do *exactly one time*, when this method is first called. The easy way to know when it's the first time is to have your solution be some kind of list member of the class, for example a `List<Point>`, which will be initialized to null in the `init()` method. Then, after you have determined the solution path, this member will not be null since it will be the list of all point locations on the solution path.

The input to the `action()` method will be an Object that will in fact be a two-dimensional array of `PacCell` cells, in other words: `PacCell[][]`. This array will represent the starting position for the maze. You must extract from this array the starting location of Pac-Man and the locations of all the food dots. You must also use this array to determine the valid successors of a location by taking into account which cells are walls and which are not.

All cells in the input maze array are `PacCells`, but you can use Java's *instanceof* operator and the inheritance hierarchy below to tell which are wall cells, which are food cells, and which is the Pac-Man cell.



Please note that to determine that a cell is empty, you must rule out all other possibilities. The mazes for this assignment will not have any ghosts or power pellets, so an empty cell will be a cell that is not an instance of a `PacmanCell`, `FoodCell`, or `WallCell`.

(2) *Determine the direction to move Pac-Man for the next move:*

This is how we simulate using a joystick to control Pac-Man's movements. Every time Pac-Man moves one square or cell in the maze, the simulation will ask your agent class in what direction to move next. It will do this by calling the `action()` method in your agent class. This should be easy once you have your solution path, since the solution path will be a sequence of cell locations including both empty and food cells, starting from Pac-Man's initial position. So, given your solution, just note Pac-Man's current position and the next step in the solution (which should be a cell immediately next to Pac-Man's current position), and simply determine the NSEW direction in which the solution cell lies. Then return the `PacFace` enum value corresponding to the direction that you have determined.

Please note that you must express the direction as a `PacFace` enum, since that is the required output of the `action()` method. You must return a `PacFace` value that should not be null even for the very first turn, when you compute the solution path, so you do not lose a step in the solution path.

Required Program Output

Your RNNA agent program must present output to System.out in the format shown in the file tsp-tiny.out, which is in the distribution. You will observe that the output contains a number of sections. Most of the output should be generated prior to the first move of the simulation. Only the last section is generated line-by-line with each call to your agent's action() method, including the first.

The required output sections are:

1. Title, identification of algorithm, authors, and identification of maze
2. Cost table - an $(n+1)$ by $(n+1)$ symmetric matrix of UCS distances from Pac-Man's initial position to each food dot in the initial configuration, where n is the number of food dots (10, for tsp-tiny), and where the first row and column represent Pac-Man's initial location. You should use the class BFSPath to generate the entries in this table. These values are the input data to the program.
3. Food array - a one-dimensional array of the x-y locations of all the food dots in the initial configuration. Please note that $+x$ is to the right and $+y$ is down, starting from the upper left of the PacCell array. This array is zero based, so there is an index shift of 1 when using these indices with the cost table. For example, the "cost" value 5 from Pac-Man's original position to food array dot 0 may be found in the (0,1) entry in the cost table (assuming rows and columns are also zero-based). The entries in this array should be ordered according to increasing values of the x-value, and for each value of x , according to increasing values of the y-value
4. Population entries that show the candidate solutions at each step of the NNA algorithm. Since there are 10 food dots for the tsp-tiny maze, not including Pac-Man's initial location, there are 10 steps in the algorithm. Also, since you must use RNNA, there must be 10 candidate solutions for the first step, representing moving from Pac-Man's initial position to each of the food dots first. Please note how branching occurs at steps 2 and subsequent steps by adding alternate solution paths to the population when there are multiple successors that are equally close to the current last location in one of the partial paths. Also note how each partial path entry reports its zero-based index, the total cost of the path so far, and a listing of the food dot locations in the current path. Finally, please also note that the number within a location's bracket is the cost (distance) from the previous location in the path, not the food mapping or food array index value.
5. Time to generate plan - the time to generate the RNNA plan, which is the minimal cost plan after all steps of the algorithm, i.e., after all food dots have been eaten. Use System.currentTimeMillis() to get long values at the start and end of the solution generation, and simply subtract one from the other to get the difference.
6. The words "Solution moves:" - which signals that the plan will now be followed.
7. The solution moves themselves. Each time your agent's action() method is called (including the first time, when outputs 1 - 7 are also generated), the method should report how many times the method has been called, Pac-Man's current location, and the PacFace enum direction that the method returns to the simulation for the move. You must use the format shown, for example:

"1 : From [4, 3] going N". Exact (pretty) spacing is not required, but all of the information must be as shown.

8. The statement "Game over: Pacman has eaten all the food" is generated by the simulation after the last food dot is eaten. Your action() method should not output this statement.

Development Teams

Teams of two students in the class are allowed, but are not required. You may develop the program by yourself if you like, with no reduction in requirements.

If you choose to work as a team, both team members must be students in the class, and:

(a) both team members must send a Webcourses message to our GTA with copy to the instructor at least 5 days prior to the due date stating your intention to work as a team and identifying the team members;

(b) both team members must submit the same program on Webcourses for grading;

(c) both submitted programs must include the names of both students in a code comment at the top of the program.

If submitting as a team, both team members will receive the same grade. If only one member of the team submits, then the team member who does not submit will receive a grade of zero for the assignment.

Program Testing

We will test your program by running your executable JAR from the command line against the test maze, so we suggest that you do the same to verify proper program operation and output prior to submitting.

What to Submit

You should submit a single Java executable JAR file containing your RNNA agent. The JAR file must contain both the source (.java) and the corresponding compiled (.class) files for all Java source files in your implementation. Of course, it is possible to implement the RNNA agent using only one Java source file.

All of your source files should include a header identifying UCF, this course and semester, and identifying the program author or authors, in the following format:

```
/**
 * University of Central Florida
 * CAP4630 – Fall 2017
 * Author: < Your_Name or Names >
 */
```


The program must be submitted on Webcourses by all authors. Email submissions will not be graded.
The submission will be graded using the grading rubric below.