A* Search for Pac-Man Instructions

Introduction

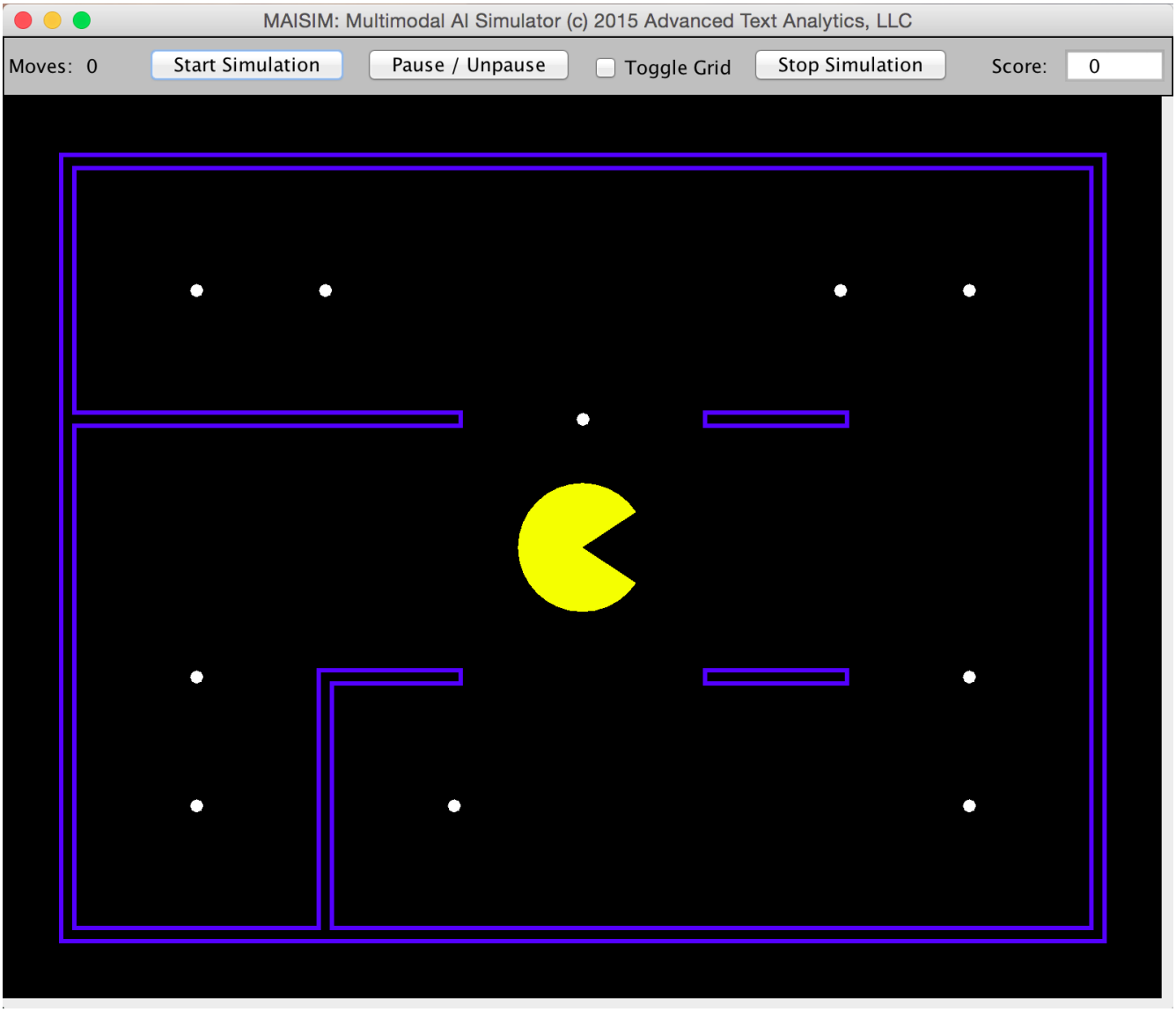
For this assignment you will develop your own heuristic and implement A* search in Java in the form of an agent program that will be used to drive Pac-Man through three mazes to collect food dots in an optimal way. You will be provided with simulation infrastructure in the form of a JAR file so you will 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.

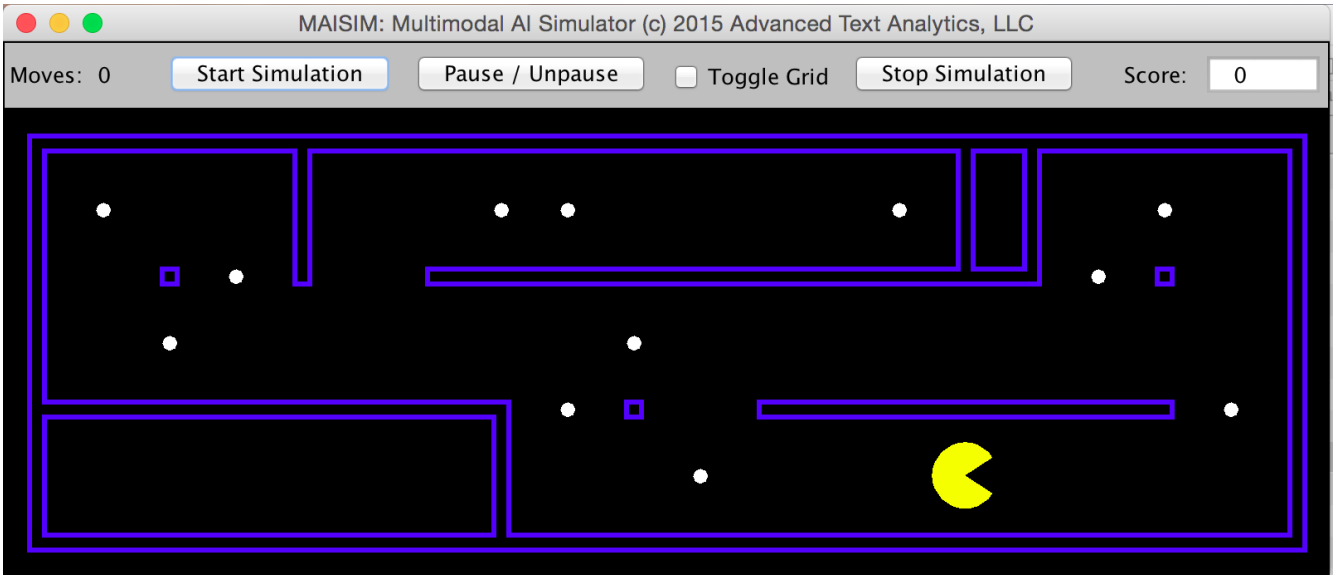
You will also be provided with the three mazes and a sample agent that you can compile and run. The sample agent is a simple replanning agent: it solves all three mazes, but it is not optimal.

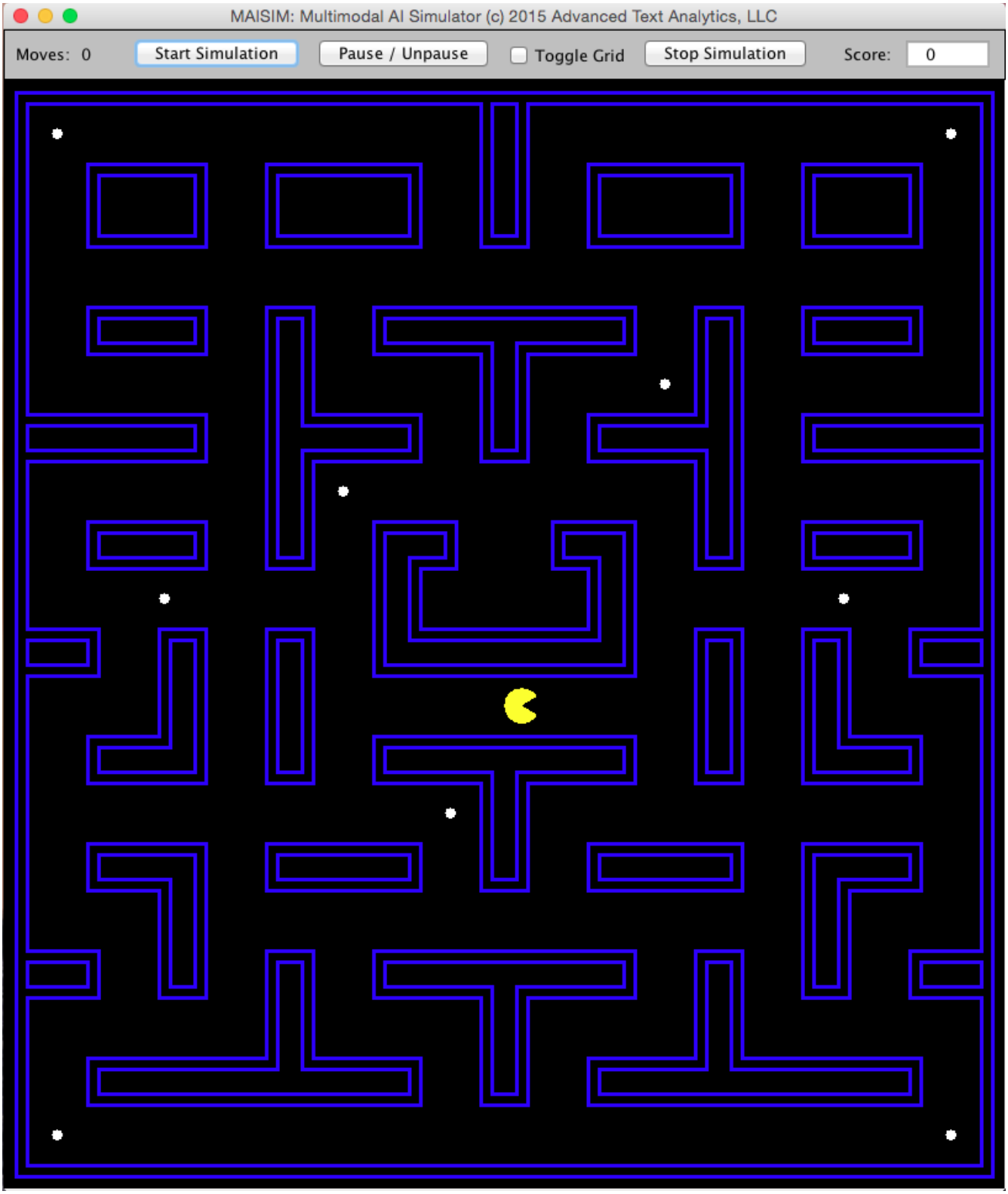
Your task will be to develop a replacement agent that implements the A* search using your own heuristic function, which you will describe in the header file for your program. This agent should be able to solve all three mazes optimally. It should also be sufficiently general that it can also be run against 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.

Test Mazes

Your agent must solve the following three mazes, which we call "Tiny", "Alley", and "Large", respectively:


MAISIM: Multimodal AI Simulator (c) 2015 Advanced Text Analytics, LLC
Moves: 0
Start Simulation
Pause / Unpause
Toggle Grid
Stop Simulation
Score:
0


MAISIM: Multimodal AI Simulator (c) 2015 Advanced Text Analytics, LLC
Moves: 0
Start Simulation
Pause / Unpause
Toggle Grid
Stop Simulation
Score:
0


MAISIM: Multimodal AI Simulator (c) 2015 Advanced Text Analytics, LLC
Moves: 0
Start Simulation
Pause / Unpause
Toggle Grid
Stop Simulation
Score:
0

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 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 .:PacSimLib.jar PacSimReplan tsp-tiny

Please note the period and colon before the library reference. The program should run and bring up the GUI screen 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 A* agent should be able to do better. In fact, it should be able to eat all the dots on this maze in 27 moves.

The "Alley" maze presents a more challenging situation. To run it, simply enter the same run command as above, only substitute "tsp-alley-sparse" for "tsp-tiny". The sample replanning agent solves this maze in 64 moves. Your A* agent should be able to solve it in 51 moves.

The "Large" maze presents an even more difficult problem due to the size of the maze, even though there are only 9 food pellets in the maze. The sample replan agent solves this maze in 213 moves. Your A* agent should be able to solve it in 153 moves.

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 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 .:PacSimLib.jar PacSimReplan tsp-tiny
```

Please note the period and colon before the library reference. The program should run and bring up the GUI screen 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 A* agent should be able to do better. In fact, it should be able to eat all the dots on this maze in 27 moves.

The "Alley" maze presents a more challenging situation. To run it, simply enter the same run command as above, only substitute "tsp-alley-sparse" for "tsp-tiny". The sample replanning agent solves this maze in 64 moves. Your A* agent should be able to solve it in 51 moves.

The "Large" maze presents an even more difficult problem due to the size of the maze, even though there are only 9 food pellets in the maze. The sample replan agent solves this maze in 213 moves. Your A* agent should be able to solve it in 153 moves.

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 look like what you see in the sample program:

```
public PacSimReplan( 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:

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

}

4. Method init() :

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

5. Method action() :

This method is where you will implement A* 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. See also the submittal section below in which we require that all classes and methods be included in one source Java file.

Your action() method must do two things:

(1) Compute the UCS 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 in 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.

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 look like what you see in the sample program:

```
public PacSimReplan( 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:

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

4. Method init() :

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

5. Method action() :

This method is where you will implement A* 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. See also the submittal section below in which we require that all classes and methods be included in one source Java file.

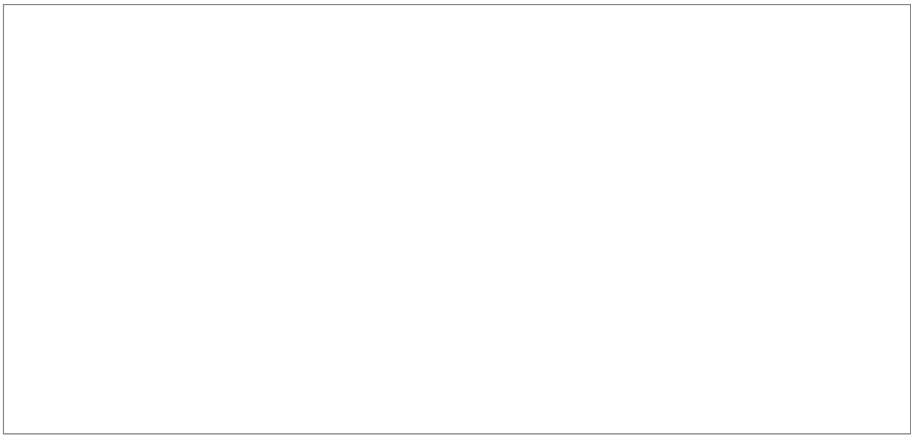
Your action() method must do two things:

(1) Compute the UCS 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 in 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. Also, when you have completed the entire solution path, simply return null. This should happen only when Pac-Man has eaten the last food dot. Also please remember that you must return a PacFace value that should not be null in the very first turn, because Pac-Man must move to start eating the food dots.

Programming Tip

You should use graph search to implement your A* search algorithm. This will drastically reduce the total number of nodes that your algorithm will need to expand. It may make a difference in being able to solve the "Large" maze since we allow a maximum of only 5 minutes wall clock time to solve that maze.

Required Program Output

Before driving Pac-Man through the maze, your program must provide output to System.out showing the progress and results of your A* search algorithm. Your program must output: (a) a message for every 1000 nodes expanded; (b) the total nodes expanded; and (c) the cell locations for the entire solution path.

Your output should be in the format of the following sample output for the "Tiny" maze. This output is for a UCS algorithm to show how the messages for every thousand node expansions should look.

```
$ java -cp .:PacSimLib.jar PacSimUCS tsp-tiny
Nodes expanded: 1000
Nodes expanded: 2000
Nodes expanded: 3000
Nodes expanded: 4000
Nodes expanded: 5000

Nodes expanded: 5206

Solution path:
( 4, 3 )
( 4, 2 )
( 4, 1 )
( 3, 1 )
( 2, 1 )
( 1, 1 )
( 2, 1 )
( 3, 1 )
( 4, 1 )
( 5, 1 )
( 6, 1 )
( 7, 1 )
( 7, 2 )
( 7, 3 )
( 7, 4 )
( 7, 5 )
( 6, 5 )
( 5, 5 )
( 4, 5 )
( 3, 5 )
( 4, 5 )
( 4, 4 )
( 4, 3 )
( 3, 3 )
( 2, 3 )
( 1, 3 )
( 1, 4 )
( 1, 5 )
```

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. Also, when you have completed the entire solution path, simply return null. This should happen only when Pac-Man has eaten the last food dot. Also please remember that you must return a PacFace value that should not be null in the very first turn, because Pac-Man must move to start eating the food dots.

Programming Tip

You should use graph search to implement your A* search algorithm. This will drastically reduce the total number of nodes that your algorithm will need to expand. It may make a difference in being able to solve the "Large" maze since we allow a maximum of only 5 minutes wall clock time to solve that maze.

Required Program Output

Before driving Pac-Man through the maze, your program must provide output to System.out showing the progress and results of your A* search algorithm. Your program must output: (a) a message for every 1000 nodes expanded; (b) the total nodes expanded; and (c) the cell locations for the entire solution path.

Your output should be in the format of the following sample output for the "Tiny" maze. This output is for a UCS algorithm to show how the messages for every thousand node expansions should look.