Student: Qiao Lin/ CS 6310/ Assignment #4

OsMowSis – Question and Answers

- **How do you track the current direction and location of each mower?**
  In class *Mower* there is an attribute **direction** which is used to track current direction of each mower. The data type of **direction** is **Direction**, which is an enumeration and consists of 8 possible directions {NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST}.
  In class *Mower* there is an attribute **location** which is used to track current location of each mower. The data type of **location** is **Location** which contains 2 attributes {xCoordinate, yCoordinate} specifying the exact location (x, y) on the lawn.
  In the beginning, attributes **direction** and **location** will be populated with initial values read from scenario file.
  During a Simulation Run, the attribute **direction** will be updated, if mower executes **SteerAction**. And the attribute **location** will be updated, if mower executes **MoveAction**.

- **How do you track the locations of gophers?**
  In class *SimulationState*, it has an attribute **gophers** which will track all gopher instances.
  In class *Gopher* the attribute **location** is used to keep track of the location of a gopher instance. The data type of **location** is **Location** which contains 2 attributes {xCoordinate, yCoordinate} specifying the exact location (x, y) on the lawn. The initial values of the attribute **location** are populated with the values read from scenario file. And the attribute **location** will be updated, if gopher instance executes **MoveAction**.

- **How do you track the how much of the grass has been cut so far?**

In class *Lawn*, it has an operation **getEmptySquares()** which will parse each square state in attribute **lawnSquares** and return all squares with *SquareState* {EMPTY}. The size of empty squares is how much of the grass has been cut so far.

- **How do you update the simulation state for each of the varied actions?**
  In class *SimulationState*, it has an attribute **lawn** which will track the state of lawn instance, an attribute **mowers** which will track states of all mower instances, and an attribute **gophers** which will track states of all gopher instances.
  1. In class *SimulationState*, the operation **validateMowerAction(actionProposed: Action, mower: Mower)** will update the state of mower instance and of lawn instance for different actions.
     1.1. If a PassAction is executed by a mower instance, the operation **setTrackAction(action:Action)** will update value of attribute **trackAction** to PassAction in mower instance.
     And the operation **reduceEnergy(amount: Integer)** will subtract 0 from the value of attribute **energy** in mower instance.
     1.2. If a SteerAction is executed by a mower instance, the operation **setTrackAction (action:Action)** will update value of attribute **trackAction** to SteerAction in mower instance.
     And the operation **setDirection(direction:Direction)** will update the value of attribute **direction** in mower instance.
     And the operation **reduceEnergy(amount: Integer)** will subtract 1 from the value of attribute **energy** in mower instance.
     1.3. If a CircularScanAction is executed by a mower instance, the operation **setTrackAction(action:Action)** will update value of attribute **trackAction** to CircularScanAction.
     And the operation **setTrackScanResults(scanResults1: Square[1..*])** will update the value of attribute **trackScanResults** in mower instance.
     And the operation **reduceEnergy(amount: Integer)** will subtract 1 from the value of attribute **energy** in mower instance.

1.4. If a LinearScanAction is executed by a mower instance, the operation **setLastAction(action:Action)** will update value of attribute **lastAction** to LinearScanAction.

And the operation **setTrackScanResults(scanResults1: Square[1..*])** will update the value of attribute **trackScanResults** in mower instance.

And the operation **reduceEnergy(amount: Integer)** will subtract 3 from the value of attribute **energy** in mower instance.

1.5. If a MoveAction is executed by a mower instance, the operation **setTrackAction(action:Action)** will update value of attribute **trackAction** to MoveAction in mower instance.

And the operation **setLocation(location: Location)** will update the value of attribute **location** in mower instance.

And the operation **reduceEnergy(amount: Integer)** will subtract 2 from the value of attribute **energy** in mower instance.

If the square at new location has fence, the operation **setMowerStatus(mowerStatus: MowerStatus)** will update the value of attribute **mowerStatus** to *MowerStatus* {CRASH} in mower instance.

If the square at new location has another mower instance, the operation **setMowerStatus(mowerStatus: MowerStatus)** will update the value of attribute **mowerStatus** to *MowerStatus* {CRASH} in both mower instances.

If the square at new location has a gopher instance, the operation **setMowerStatus(mowerStatus: MowerStatus)** will update the value of attribute **mowerStatus** to *MowerStatus* {REMOVED} in mower instance.

If the square at new location has a state of *SquareState* {GRASS}, the operation **setSquareState(squareState: SquareState)** will update the square state to *SquareState* {EMPTY} in square instance at new location.

1.6. If any other action is proposed other than the actions above, the operation **setMowerStatus(mowerStatus: MowerStatus)** will update the value of attribute **mowerStatus** to *MowerStatus* { CRASH} in mower instance.

1.7. If the attribute **mowerStatus** of mower instance is *MowerStatus* {OK} and there is a rechargingPad instance at location of mower instance, the operation **recharge()** will recover the value of attribute **energy** to be the same as static attribute **energyCapacity** in mower instance.

1.8. If the attribute **mowerStatus** of mower instance is *MowerStatus* {OK}, the operation **isOutOfEnergy()** will check if the value of attribute **energy** is 0, if it's 0, the operation **setMowerStatus(mowerStatus: MowerStatus)** will update the value of attribute **mowerStatus** to *MowerStatus* { STALL} in mower instance.

2. In class *SimulationState*, the operation **validateGopherAction(action: MoveAction, gopher: Gopher)** will update state of gopher instance and of mower instances if needed.

   2.1. If a MoveAction is executed by a gopher instance, the operation **setLocation(location: Location)** will update the value of attribute **location** in gopher instance. If the square at new location has a mower instance, then the operation **setMowerStatus(mowerStatus: MowerStatus)** will update the value of attribute **mowerStatus** to *MowerStatus* { REMOVED } in mower instance.

- **How do you determine the appropriate output for a cscan() action?**

  In class *CircularScanAction* the operation **scan(simulationState: SimulationState): SquareState [8]** is used to determine the appropriate output. The implementation is as follows.

  1. It will calculate the locations of 8 surrounding squares based on attribute **curLocation: Location** in *CircularScanAction* instance.
  2. In order to get states of surrounding squares, it will iterate each location from the North-most and proceed in a clockwise direction, and calculate the square state at that location.

     2.1. If the location has a fence, then add *SquareState* {FENCE} into output list.

2.2. If the location has another mower instance, then add *SquareState* {MOWER} into output list.

2.3. If location has a gopher instance,

2.3.1. If square state at location is SquareState {GRASS}, then add SquareState {GOPHER_GRASS} into output list.

2.3.2. If square state at location is SquareState {EMPTY}, then add SquareState {GOPHER_EMPTY} into output list.

2.4. Otherwise,

2.4.1. If square state at location is SquareState {GRASS}, then add SquareState {GRASS} into output list.

2.4.2. If square state at location is SquareState {EMPTY}, then add SquareState {EMPTY} into output list.

- **How do you determine the appropriate output for an lscan() action?**

  In class *LinearScanAction* the operation **scan(simulationState: SimulationState): SquareState[1..*]** is used to determine the appropriate output. The implementation is as follows.

  1. It will calculate the locations of linear squares based on attribute **curLocation: Location** and **curDirection: Direction** in *LinearScanAction* instance.
     1.1. Calculate the location next to **curLocation** along **curDirection**.
     1.2. Keep calculating further locations along **curDirection** until location has a fence.
  2. In order to get states, it will iterate each location from the nearest to farthest, and calculate the square state at that location.
     1.1. If the location has a fence, then add *SquareState* {FENCE} into output list.
     1.2. If the location has another mower instance, then add *SquareState* {MOWER} into output list.
     1.3. If the location has a gopher instance,

1.3.1. If square state at location is SquareState {GRASS}, then add SquareState {GOPHER_GRASS} into output list.

1.3.2. If square state at location is SquareState {EMPTY}, then add SquareState {GOPHER_EMPTY} into output list.

1.4. Otherwise,

1.4.1. If square state at location is SquareState {GRASS}, then add SquareState {GRASS} into output list.

1.4.2. If square state at location is SquareState {EMPTY}, then add SquareState {EMPTY} into output list.

- **How do you determine when the simulation should be halted?**

  In class *SimulationMonitor*, the operation **isHalted()** is used to determine if simulation should be halted. The simulation will be halted, when one of the following three conditions has been met.

  1. In SimulationState instance, the operation **mowersAllRemoved()** returns true, which means for each mower instance, the value of attribute **mowerStatus** is *MowerStatus* {CRASH} or {REMOVED}. In another word, all mowers have been crashed or removed;
  2. In lawn instance, the operation **grassAllCut()** returns true, which means all grass has been cut. Its implementation is to parse the lawn squares and to return true if each square state has a state of SquareState {EMPTY}.
  3. In SimulationMonitor instance, the operation **isSimulationDuration()** returns false, which means the maximum number of turns has been reached. Its implementation is to check if the value of attribute **turnCompletedNumber** is less than the value of attribute **maxTurnNumber**.

- **How do you keep track of the knowledge needed to display the final report?**

  In order to keep track knowledge needed to display final report,

  1. In class *Lawn*, the operation **getArea()** is used to keep track the area of the lawn. The return value will be calculated as follows.

     Area =width*height

2. The value of attribute **originalGrassSquareNumber** in class *SummaryReport* is the same as the value of attribute **lawnArea**, because there is no crater in the lawn.
3. In class *Lawn*, the operation **getEmptySquares(): Square[0..*]** is used to keep track the square which has a state of *SquareState* {EMPTY}. The size of empty squares is the actual number of grass squares that were cut.
4. In class *SimulationMonitor*, the attribute **turnCompletedNumber** is used to keep track the total number of fully completed turns that were taken. Its initial value is zero and will be increased by 1 after one turn is completed.

- **How do you keep track of the partial knowledge collected by each mower?**
  In Mower class, attribute **knowledge: Knowledge** is used to keep track partial knowledge collected.
  In attribute *knowledge* instance,
  1. The value of attribute **id** is used to keep track which mower instance owns this knowledge instance. It will be populated the same value as the attribute **id** in mower instance.
  2. The attribute **partialSquares** is used to save square states as mower instance moves and scans. It's initially created as a 2 dimensional array of square instances with *SquareState* {UNKNOWN} and will be updated to actual square states as mower instance moves and scans.
  3. The attribute **relativeLocation** is used to keep track relative location of mower instance in 2D array **partialSquares** of knowledge instance. The initial value of **relativeLocation** is at the middle of 2D array **partialSquares**. The initial value of **xCoordinate** is the size of one of arrays of **partialSquares** divided by 2. The initial value of **yCoordinate** is the size of arrays of **partialSquares** divided by 2. The value of attribute **relativeLocation** will be updated as mower instance moves.
  4. The attribute **absoluteLocation** is used to keep track actual location of mower instance in real lawn instance. It will be calculated by

LinearScanAction executions in Direction{ NORTH, EAST, SOUTH, WEST}

5. The attribute **mowerStatus** is used to keep track attribute **mowerStatus** in mower instance. Its purpose is to share attribute **mowerStatus** with other mower instances.
6. The attribute **rechargingPadLocation** is used to keep track the absolute location of mower's rechargingPad in real lawn instance. It will be calculated by the calculation of attribute **absoluteLocation**. Its purpose is to share attribute **rechargingPadLocation** with other mower instances.
7. The operation **saveCScanResults(trackScanResults: Square [8])** is used to save scan results from CircularScanAction and update square states in attribute **partialSquares** in knowledge instance.
8. The operation **saveLScanResults(trackScanResults: Square[1..*], direction:Direction)** is used to save scan results from LinearScanAction and update square states in attribute **partialSquares** in knowledge instance.
9. The operation **saveMoveResult(direction: Direction)** is used to save states after MoveAction. It will update square states in attribute **partialSquares** and attribute **relativeLocation** and attribute **absoluteLocation** if calculated in knowledge instance.
10. The operation **calculateAbsoluteLocation(): Location** is used to calculate the absolute location in real lawn instance by scanning.

- **How do you manage collaboration between the mowers?**

  The class *CommunicationChannel* is used to manage collaboration between the mowers.

  11. The attribute **sharedSquares** is a shared a data structure to save square states among mower instances. It's initially created as a 2 dimensional array of square instances with *SquareState* {UNKNOWN}. Its size will be calculated after the first mower instance figure out its attribute **absoluteLocation**.

12. The attribute **knowledgeList** is a list of knowledge instances referencing each mower instance's knowledge. After mower instances calculate their attribute **absoluteLocation** and **rechargingPadLocation**, other mower instances could access this information by parsing attribute **knowledgeList** in CommunicationChannel instance. Also other mower instances could access attribute **mowerStatus** of other mower instances by parsing attribute **knowledgeList** in CommunicationChannel instance.
13. The operation **shareKnowledge(knowledge: Knowledge)** is used to add new knowledge instance into attribute **knowledgeList**. Its purpose is to share attributes **absoluteLocation** calculated, **mowerStatus** and **rechargingPadLocation**.
14. The operation **updateSharedSquares(knowledge: Knowledge)** is used to update attribute **sharedSquares** by updated knowledge instance. The purpose of attribute **sharedSquares** is to share square states at absolute locations after calculation. It is the combined information of squares collected by each mower instance.

- **How do you determine the next action for a mower?**

  In class *Mower* the operations **determineNextAction**(), **getRandomAction()** and **getActionByStrategy()** are used to determine the next action for a mower.

  The determination process of **determineNextAction**() is as follows.

  1. If the value of attribute **strategy** in mower instance is 0, then use **getRandomAction()**. The determination process of **getRandomAction()** is as follows.
     1.1. Generate a random number between 0-99
     1.2. Decide the next action by range of number generated.
  2. If the value of attribute **strategy** in mower instance is 1, then use **getActionByStrategy()**. The determination process of **getActionByStrategy()** is as follows.
     2.1. If the attribute **trackAction** is **SteerAction** in the mower instance, then decide **MoveAction** to be the next Action.

2.2. If the attribute **trackAction** is **MoveAction** in the mower instance, then decide **ScanAction** to be the next Action.

2.3. Otherwise check squares nearby based on attribute **location**, attribute **knowledgeList** and attribute **sharedSquares** in CommunicationChannel instance.

2.3.1. If there is any squares with *SquareState* {GOPHER_GRASS} or {GOPHER_EMPTY} within 2 steps, then decide **SteerAction** to be the next Action in order to be ready to move away from gopher.

2.3.2. If there is any surrounding squares with *SquareState* {UNKNOWN}, then decide **ScanAction** to be the next Action.

2.3.3. If there is any surrounding squares with *SquareState* {MOWER}, then decide **SteerAction** to be the next Action in order to be ready to move away from other mower instances.

2.3.4. If there is any surrounding squares with *SquareState* {GRASS}, then decide **SteerAction** to be the next Action in order to be ready to move to grass square.

2.3.5. If there is any surrounding squares with *SquareState* {EMPTY}, then decide **SteerAction** to be the next Action in order to be ready to move to an empty square without gopher or mower around and with the most grass around if possible.

2.3.6. Otherwise, decide **ScanAction** to be the next Action.