A sign with a black ant

AI-generated content may be incorrect. **Ant** **Simulation** A sign with a black ant

AI-generated content may be incorrect.

**Programming Tasks (Mark Scheme)**

*It should be noted that solutions to each problem may vary depending on the techniques selected by the candidate. Marks should be awarded for solutions which achieve the correct outcome but through different techniques than those shown here.*

# Task 1 (4 marks)

## Coding

 Prompt the user to confirm they would like to quit when they select option 9. **[1 mark]**

 Use of selection to check the user input and either quit or continue the simulation. **[1 mark]**

 Message to confirm that the simulation has finished. **[1 mark]**

### Teacher Notes

*Checking for different versions of “yes” is not required, but it is generally considered good practice for students to include this in applications.*

### Example Solution

Modification to the Main method:

break;

//CHANGE

case "9":

Console.WriteLine("Are you sure that you want to quit the simulation? yes / no");

string Confirm = Console.ReadLine().ToLower();

if (Confirm == "yes" || Confirm == "y")

{

break;

}

else

{

Choice = "";

}

break;

}

} while (Choice != "9");

Console.WriteLine("Simulation Complete, press any key to close");

//END CHANGE

Console.ReadLine();

## Testing

 Show the program displaying a message confirming that the simulation has ended. **[1 mark]**

A screenshot of a computer

AI-generated content may be incorrect.

# Task 2 (7 marks)

## Coding

 Suitable method to expose the Stages attribute in the Ant class. **[1 mark]**

 Use of iteration in the Nest class to loop through the Ants list. **[1 mark]**

 Check to confirm if an ant is a worker. **[1 mark]**

 Check to confirm if an ant is 15 stages “old”. **[1 mark]**

 Successfully remove the “old” ant from the Ants list. **[1 mark]**

 Confirmation message to the user that the ant has died. **[1 mark]**

### Teacher Notes

*The simulation generally uses foreach loops to iterate through the Ants list. Removal of items from the list while iterating through it using this technique can cause enumeration errors. Removing ants from the Ants list could be done in a number of different ways. Although this mark scheme demonstrates using the AdvanceStage method to do it, any suitable technique which meets the objectives should be accepted.*

### Example Solution

Exposing the Stages attribute in the Ant class:

//CHANGE

A screenshot of a computer program

AI-generated content may be incorrect. public int GetAge()

{

return Stages;

}

//END CHANGES

Modification of the Nest class to remove “old” ants:

//CHANGE

for (int i = Ants.Count - 1; i >= 0; i--)

{

if (Ants[i].GetTypeOfAnt() == "worker" && Ants[i].GetAge() > 14)

{

Ants.RemoveAt(i);

Console.WriteLine("Ant has died of old age");

}

}

//END CHANGES

## Testing

 Show the program displaying a message confirming that four ants have died of old age. **[1 mark]** ▶

# Task 3 (4 marks)

## Coding

 Use of iteration in the AdvanceStage method in the Simulation class to loop through the cells in the Grid. **[1 mark]**

 Reduction of the food level in each cell by 10%. **[1 mark]**

 Preventing the AmountOfFood attribute in the Cell class from dropping below zero. [1 mark]

### Teacher Notes

N/A

### Example Solution

Modification to the AdvanceStage method in the Simulation class:

public void AdvanceStage(int NumberOfStages)

{

for (int Count = 1; Count <= NumberOfStages; Count++)

{

//CHANGE

foreach (Cell C in Grid)

{

C.UpdateFoodInCell(-Convert.ToInt32(C.GetAmountOfFood() \* 0.1));

}

//END CHANGE

List<Pheromone> PheromonesToDelete = new List<Pheromone>();

Modification of the UpdateFoodInCell method in the Cell class to prevent the AmountOfFood attribute from dropping below zero:

public void UpdateFoodInCell(int Change)

{

//CHANGE

AmountOfFood += Change;

if (AmountOfFood < 0)

{

AmountOfFood = 0;

}

//END CHANGE

}

## Testing

 Show the program displaying that the food level in each cell containing food has reduced by 10%. **[1 mark]**

A black screen with white text

AI-generated content may be incorrect. A black screen with white text

AI-generated content may be incorrect.

# Task 4a (4 marks)

## Coding

 Checking that the user input is numerical and, if not, giving a suitable error message. **[1 mark]**

 Checking that the user input is in range and, if not, giving a suitable error message. **[1 mark]**

 Repeating the user input until the input is valid. [1 mark]

### Teacher Notes

*The example given below uses a list to check if the input is in range. Any valid solution is acceptable.*

### Example Solution

Modification of the Input mechanism to introduce a range:

//CHANGE

string Choice = "";

do

{

while (Choice == "")

{

DisplayMenu();

List<int> ValidMenuOptions = new List<int> { 1, 2, 3, 4, 5, 9 };

Choice = GetChoice(ValidMenuOptions);

}

//END CHANGE

switch (Choice)

Modification of the GetChoice method to perform a range check and format check:

//CHANGE

static string GetChoice(List<int> ValidMenuOptions)

{

bool ValidChoice = false;

while (!ValidChoice)

{

string Choice = Console.ReadLine();

int ChoiceAsInt = 0;

try

{

ChoiceAsInt = Convert.ToInt32(Choice);

if (ValidMenuOptions.Contains(ChoiceAsInt))

{

ValidChoice = true;

return Choice;

}

else

{

Console.WriteLine("That is not a valid menu option, please try again");

return "";

}

}

catch

{

Console.WriteLine("That is not a valid integer");

return "";

}

}

return null;

}

A screenshot of a computer program

AI-generated content may be incorrect. //END CHANGE

## Testing

 Show the program giving a suitable error message that the input is not a valid integer   
and error message showing that the input is not in range. **[1 mark]** ▶

# Task 4b (4 marks)

## Coding

 New method DisplaySimulationChoicesMenu which displays a summary of each of the four simulation settings in an effective way. **[1 mark]**

 Checking that the user input is in range and, if not, giving a suitable error message. **[1 mark]**

 Repeating the user input until the input is valid. [1 mark]

### Teacher Notes

*The example given below uses a list to check if the input is in range. Any valid solution is acceptable.*

### Example Solution

Creation of new DisplaySimulationChoicesMenu method to display the four built-in simulation settings:

//CHANGE

static void DisplaySimulationChoicesMenu()

{

Console.WriteLine();

Console.WriteLine("Simulation choices:");

Console.WriteLine("1. Simulation 1: \tGrid: 5x5 \n\t\t\tNest: 1(with 500 food units)\n\t\t\tAnts: 5 (1 queen, 4 workers)\n\t\t\tFood Sources: 3\n\t\t\tPheromone Strength: 1000\n\t\t\tPheromone Decay Rate: 50 per stage\n");

Console.WriteLine("2. Simulation 2: \tGrid: 5x5 \n\t\t\tNest: 1(with 500 food units)\n\t\t\tAnts: 5 (1 queen, 4 workers)\n\t\t\tFood Sources: 3\n\t\t\tPheromone Strength: 1000\n\t\t\tPheromone Decay Rate: 100 per stage\n");

Console.WriteLine("3. Simulation 3: \tGrid: 10x10 \n\t\t\tNest: 1(with 500 food units)\n\t\t\tAnts: 9 (1 queen, 8 workers)\n\t\t\tFood Sources: 3\n\t\t\tPheromone Strength: 1000\n\t\t\tPheromone Decay Rate: 25 per stage\n");

Console.WriteLine("4. Simulation 4: \tGrid: 5x5 \n\t\t\tNest: 2(with 500 food units each)\n\t\t\tAnts: 6 per nest (1 queen, 5 workers)\n\t\t\tFood Sources: 3\n\t\t\tPheromone Strength: 1000\n\t\t\tPheromone Decay Rate: 25 per stage\n");

Console.WriteLine();

Console.WriteLine("> ");

}

//END CHANGE

Creation of new GetSimChoice method to perform a range check and format check:

//CHANGE

static string GetSimChoice()

{

bool ValidChoice = false;

while (!ValidChoice)

{

string Choice = Console.ReadLine();

int ChoiceAsInt = 0;

try

{

ChoiceAsInt = Convert.ToInt32(Choice);

if (ChoiceAsInt > 0 && ChoiceAsInt < 5)

{

ValidChoice = true;

return Choice;

}

else

{

Console.WriteLine("That is not a valid menu option, please try again");

DisplaySimulationChoicesMenu();

A screenshot of a computer program

AI-generated content may be incorrect. }

}

catch

{

Console.WriteLine("That is not a valid integer");

DisplaySimulationChoicesMenu();

}

}

return null;

}

## //END CHANGE

## Testing

 Show the program giving a suitable error message that the input is not a valid integer   
and error message showing that the input is not in range. **[1 mark]** ▶

# Task 4c (2 marks)

## Coding

 Additional check to see if there is already food in a cell. [1 mark]

### Teacher Notes

N/A

### A black screen with white text AI-generated content may be incorrect.Example Solution

Modification to the Constructor in the Simulation class:

for (int Count = 1; Count <= StartingNumberOfFoodCells; Count++)

{

bool Allowed;

do

{

Allowed = true;

Row = RGen.Next(1, NumberOfRows + 1);

Column = RGen.Next(1, NumberOfColumns + 1);

foreach (Nest N in Nests)

{

//CHANGE

if ((N.GetRow() == Row && N.GetColumn() == Column) || Grid[GetIndex(Row, Column)].GetAmountOfFood() > 0)

{

Allowed = false;

}

//END CHANGE

}

} while (!Allowed);

AddFoodToCell(Row, Column, 500);

## }

## Testing

 Show the program displaying three separate cells, each containing 500 units of food. **[1 mark]** ▶

# Task 4d (6 marks)

## Coding

 Use of a suitable range check to confirm that the user input is within the size of the grid. **[1 mark]**

 Range check which checks the Row and Column separately and gives separate suitable error messages. **[1 mark]**

 Format check to ensure the user input is an integer. [1 mark]

 Repeating the user input until the input is valid. [1 mark]

 Accessor methods in the Simulation class to expose the NumberOfColumns and NumberOfRows attributes. [1 mark]

### Teacher Notes

*Accessing the width and height data for the simulation could also be done directly within the GetCellReference method rather than the example solution given.   
The important aspect to assess is individual range and format checking of Row and Column.*

### Example Solution

Creation of new accessor methods in the Simulation class to expose the NumberOfColumns and NumberOfRows attributes:

//CHANGE

public int GetSimulationWidth()

{

return NumberOfColumns;

}

public int GetSimulationHeight()

{

return NumberOfRows;

}

//END CHANGE

Modification of the GetCellReference method to perform format check and range check:

//CHANGE

static void GetCellReference(ref int Row, ref int Column, int MaxWidth, int MaxHeight, string DisplayMessage)

{

Console.WriteLine();

bool ValidWidth = false;

bool ValidHeight = false;

while (!ValidHeight)

{

Console.WriteLine("Please enter in the cell reference for " + DisplayMessage);

Console.Write("Enter row number: ");

string RowAsString = Console.ReadLine();

try

{

Row = Convert.ToInt32(RowAsString);

if (Row > 0 && Row <= MaxHeight)

{

ValidHeight = true;

}

else

{

Console.WriteLine($"That is not a valid height. It must be between 1 and {MaxHeight} inc. Please try again");

Console.WriteLine();

continue;

}

}

catch

{

Console.WriteLine("That is not a valid integer");

Console.WriteLine();

continue;

}

}

while (!ValidWidth)

{

Console.WriteLine("Please enter in the cell reference for " + DisplayMessage);

Console.Write("Enter column number: ");

string ColAsString = Console.ReadLine();

try

{

Column = Convert.ToInt32(ColAsString);

if (Column > 0 && Column <= MaxWidth)

{

ValidWidth = true;

}

else

{

Console.WriteLine($"That is not a valid width. It must be between 1 and {MaxWidth} inc. Please try again");

Console.WriteLine();

continue;

}

}

catch

{

Console.WriteLine("That is not a valid integer");

Console.WriteLine();

continue;

}

}

Console.WriteLine();

}

//END CHANGE

Modification of the Main static method to use the updated GetCellReference method:

case "2":

int StartRow = 0, StartColumn = 0, EndRow = 0, EndColumn = 0;

//CHANGE

GetCellReference(ref StartRow, ref StartColumn, ThisSimulation.GetSimulationWidth(), ThisSimulation.GetSimulationHeight(), "top left cell of the area you are interested in.");

GetCellReference(ref EndRow, ref EndColumn, ThisSimulation.GetSimulationWidth(), ThisSimulation.GetSimulationHeight(), "bottom right cell of the area you are interested in.");

Console.WriteLine(ThisSimulation.GetAreaDetails(StartRow, StartColumn, EndRow, EndColumn));

break;

case "3":

int Row = 0, Column = 0;

GetCellReference(ref Row, ref Column, ThisSimulation.GetSimulationWidth(), ThisSimulation.GetSimulationHeight(), "specific cell you are interested in.");

//END CHANGE

Console.WriteLine(ThisSimulation.GetCellDetails(Row, Column));

break;

## 

## A screenshot of a computer AI-generated content may be incorrect.Testing

 Show the program displaying a suitable out of range error message and a suitable format error message. **[1 mark]** ▶

# Task 5 (6 marks)

## Coding

 Suitable flag to hold the simulation in a loop. **[1 mark]**

 Test to check if all the ants have died. **[1 mark]**

 Test to check if all the food has been consumed. [1 mark]

 Selection to stop the simulation and quit the application if either of the criteria has been met. [1 mark]

 Suitable message to inform the user why the simulation has ended. [1 mark]

### Teacher Notes

*Multiple techniques could be used to hold the simulation in its loop.*

### Example Solution

Modification to the Main static method to quit the application when the simulation ends:

}

//CHANGE

if (ThisSimulation.HasSimulationEnded())

{

Console.WriteLine($"The simulation has ended because {ThisSimulation.GetEndReason()}");

break;

}

//END CHANGE

} while (Choice != "9");

Console.ReadLine();

}

Suitable mechanism for holding the simulation in a loop and testing that status:

class Simulation

{

protected List<Cell> Grid = new List<Cell>();

protected List<Ant> Ants = new List<Ant>();

protected List<Pheromone> Pheromones = new List<Pheromone>();

protected List<Nest> Nests = new List<Nest>();

protected int NumberOfRows, NumberOfColumns, StartingFoodInNest, StartingNumberOfFoodCells, StartingNumberOfNests;

protected int StartingAntsInNest, NewPheromoneStrength, PheromoneDecay;

//CHANGE

private bool SimulationEnded = false;

private string EndReason = "";

//END CHANGE

…

//CHANGE

public bool HasSimulationEnded()

{

return SimulationEnded;

}

public string GetEndReason()

{

return EndReason;

}

//END CHANGE

}

Modification of the AdvanceStage method in the Simulation class to test the two criteria for ending the simulation:

//CHANGE

if (Ants.Count == 0)

{

SimulationEnded = true;

EndReason = "all ants have perished.";

break;

}

bool FoodAvailable = false;

foreach (Cell C in Grid)

{

if (C.GetAmountOfFood() > 0)

{

FoodAvailable = true;

break;

}

}

A black screen with white text

AI-generated content may be incorrect. if (!FoodAvailable)

{

SimulationEnded = true;

EndReason = "all food has been collected.";

break;

}

//END CHANGE

foreach (Nest N in Nests)

{

N.AdvanceStage(Nests, Ants, Pheromones);

}

## Testing

 Show the program displaying a suitable message for why the simulation   
has finished. **[1 mark]** ▶

# Task 6 (10 marks)

## Coding

 Create a new FlyingAnt class which inherits Ant. **[1 mark]**

 Mechanism for the FlyingAnt to understand the bounds of the simulation rather than just its neighbouring cells like a normal worker ant. **[1 mark]**

 Mechanism for keeping the FlyingAnt still in the nest for three stages while it matures. [1 mark]

 Successfully selecting a new location for the FlyingAnt which is within the bounds of the simulation. [1 mark]

 Mechanism for keeping the FlyingAnt still in the nest for three stages while it matures. [1 mark]

 Modification of the AdvanceStage method in the Nest class to introduce a mechanism for a 20% chance of spawning a FlyingAnt. [1 mark]

 Modification of the AdvanceStage method in the Simulation class to inform the user when a FlyingAnt has flown the nest and where it is going to. [1 mark]

 Set up a new nest at the new location. [1 mark]

 Removal of the FlyingAnt once the nest has been created. [1 mark]

### Teacher Notes

*Although a flying ant does not have the functionality to return to the nest, it still needs a small food capacity in order to operate simply within AQA’s Version 2 of the   
pre-release code.*

### Example Solution

Creation of new FlyingAnt class:

//CHANGE

class FlyingAnt : Ant

{

private int SimulationWidth, SimulationHeight;

private bool HasFlown = false;

public FlyingAnt(int StartRow, int StartColumn, int NestInRow, int NestInColumn, int SimWidth, int SimHeight)

: base(StartRow, StartColumn, NestInRow, NestInColumn)

{

TypeOfAnt = "flying";

FoodCapacity = 5;

SimulationWidth = SimWidth;

SimulationHeight = SimHeight;

}

public override void AdvanceStage(List<Nest> Nests, List<Ant> Ants, List<Pheromone> Pheromones)

{

Stages++;

if (Stages == 3)

{

int RNoRow, RNoColumn;

bool ValidNewLocation;

do

{

ValidNewLocation = true;

RNoRow = RGen.Next(1, SimulationHeight);

RNoColumn = RGen.Next(1, SimulationWidth);

foreach (Nest N in Nests)

{

if (N.GetRow() == RNoRow && N.GetColumn() == RNoColumn)

{

ValidNewLocation = false;

}

}

} while (!ValidNewLocation);

Row = RNoRow;

Column = RNoColumn;

HasFlown = true;

Console.WriteLine("Flying Ant has flown the nest!");

}

}

public override bool IsAtOwnNest()

{

if (HasFlown)

{

return false;

}

return Row == NestRow && Column == NestColumn;

}

}

//END CHANGE

Modification of the Nest class to introduce the concept of the simulation width and height. Modification of the AdvanceStage method in the Nest class to introduce a 20% chance of a FlyingAnt being born to a queen:

class Nest : Entity

{

protected int FoodLevel, NumberOfQueens;

protected static int NextNestID = 1;

//CHANGE

private int SimulationWidth, SimulationHeight;

public Nest(int StartRow, int StartColumn, int StartFood, int SimWidth, int SimHeight) : base(StartRow, StartColumn)

{

FoodLevel = StartFood;

NumberOfQueens = 1;

ID = NextNestID;

NextNestID++;

SimulationWidth = SimWidth;

SimulationHeight = SimHeight;

}

//END CHANGE

…

for (int A = 1; A <= NumberOfQueens; A++)

{

int RNo1 = RGen.Next(0, 100);

if (RNo1 < 50)

{

int RNo2 = RGen.Next(0, 100);

if (RNo2 < 2)

{

Ants.Add(new QueenAnt(Row, Column, Row, Column));

NumberOfQueens++;

}

else

{

Ants.Add(new WorkerAnt(Row, Column, Row, Column));

}

}

//CHANGE

int RNo3 = RGen.Next(0, 100);

if (RNo3 < 20)

{

Console.WriteLine("Flying Ant born into the nest!");

Ants.Add(new FlyingAnt(Row, Column, Row, Column, SimulationWidth, SimulationHeight));

}

//END CHANGE

}

Modification of the AdvanceStage method in the Simulation class to advise the user when the FlyingAnt has flown the nest and where it is setting up a new nest:

//CHANGE

else if (CurrentCell.GetAmountOfFood() > 0 && A.GetFoodCarried() == 0 && A.GetFoodCapacity() > 0 && A.GetTypeOfAnt() != "flying")

{

int FoodObtained;

do

{

FoodObtained = RGen.Next(1, A.GetFoodCapacity() + 1);

} while (FoodObtained > CurrentCell.GetAmountOfFood() || (A.GetFoodCarried() + FoodObtained) > A.GetFoodCapacity());

CurrentCell.UpdateFoodInCell(-FoodObtained);

A.UpdateFoodCarried(FoodObtained);

}

else

{

if (A.GetFoodCarried() > 0)

{

UpdateAntsPheromoneInCell(A);

}

A.ChooseCellToMoveTo(GetIndicesOfNeighbours(A.GetRow(), A.GetColumn()),

GetIndexOfNeighbourWithStrongestPheromone(A.GetRow(), A.GetColumn()));

}

if (A.GetTypeOfAnt() == "flying" && !A.IsAtOwnNest())

{

Console.WriteLine($"Flying ant has flown to cell {A.GetRow()}, {A.GetColumn()} to create a new nest");

SetUpANestAt(A.GetRow(), A.GetColumn());

}

//END CHANGE

Modification of the SetUpANestAt method in the Simulation class to remove the FlyingAnt once a new nest has been created:

public void SetUpANestAt(int Row, int Column)

{

//CHANGE

for (int i = 0; i < Ants.Count; i++)

{

if (Ants[i].GetTypeOfAnt() == "flying" && Ants[i].GetRow() == Row && Ants[i].GetColumn() == Column)

{

Ants.RemoveAt(i);

break;

}

}

Nests.Add(new Nest(Row, Column, StartingFoodInNest, NumberOfRows, NumberOfColumns));

//END CHANGE

Ants.Add(new QueenAnt(Row, Column, Row, Column));

for (int Worker = 2; Worker <= StartingAntsInNest; Worker++)

{

Ants.Add(new WorkerAnt(Row, Column, Row, Column));

}

}

## Testing

 Show the program displaying a suitable message that a flying ant has been created and where it is setting up a new nest. **[1 mark]**

A screen shot of a computer

AI-generated content may be incorrect.

 Show the program displaying more than one nest in the simulation. [1 mark]

A black screen with white text

AI-generated content may be incorrect.

# Task 7 (10 marks)

## Coding

 Create a new ForagerAnt class which inherits Ant. **[1 mark]**

 Required modifications to associated base classes to allow the ForagerAnt to operate correctly. **[1 mark]**

 Suitable data structure or other mechanism for storing the neighbouring cell matrix associations with the compass points. **[1 mark]**

 Modification of the ChooseCellToMoveTo method in the ForagerAnt class to allow it to move towards a stronger-smelling food cell rather than follow normal pheromones. [1 mark]

 Display messages to the user to give updates on what the ForagerAnt is currently doing. **[1 mark]**

 Modification to the Cell class to introduce the concept of smell 10 times the amount of food in the cell. Ensure the food levels cannot drop below zero. **[1 mark]**

 Modification of the AdvanceStage method in the Simulation class to call the modified ChooseCellToMoveTo method. **[1 mark]**

 Creation of new GetIndexOfNeighbourWithStrongestSmell method to identify the index of the neighbouring cell with the strongest smell. [1 mark]

 Adding a single ForagerAnt to a nest. **[1 mark]**

### Teacher Notes

N/A

### Example Solution

Creation of new ForagerAnt class:

//CHANGE

class ForagerAnt : Ant

{

private Dictionary<int, string> SmellDirection = new Dictionary<int, string>();

public ForagerAnt(int StartRow, int StartColumn, int NestInRow, int NestInColumn)

: base(StartRow, StartColumn, NestInRow, NestInColumn)

{

TypeOfAnt = "forager";

FoodCapacity = 30;

SmellDirection.Add(0, "North-West");

SmellDirection.Add(1, "North");

SmellDirection.Add(2, "North-East");

SmellDirection.Add(3, "West");

SmellDirection.Add(5, "East");

SmellDirection.Add(6, "South-West");

SmellDirection.Add(7, "South");

SmellDirection.Add(8, "South-East");

}

public override void ChooseCellToMoveTo(List<int> ListOfNeighbours, int IndexOfNeighbourWithStrongestPheromone, int IndexOfNeighbourWithStrongestSmell = -1)

{

if (AmountOfFoodCarried > 0)

{

Console.WriteLine("Forager has food and is heading back to the nest");

if (Row > NestRow)

{

Row--;

}

else if (Row < NestRow)

{

Row++;

}

if (Column > NestColumn)

{

Column--;

}

else if (Column < NestColumn)

{

Column++;

}

}

else if (IndexOfNeighbourWithStrongestSmell != -1)

{

int IndexToUse = ListOfNeighbours.IndexOf(IndexOfNeighbourWithStrongestSmell);

Console.WriteLine($"Forager Ant is moving towards the strongest smell coming from the {SmellDirection[IndexToUse]}");

ChangeCell(IndexToUse, ref Row, ref Column);

}

else

{

Console.WriteLine("No smell detected by forager, moving randomly...");

int IndexToUse = ChooseRandomNeighbour(ListOfNeighbours);

ChangeCell(IndexToUse, ref Row, ref Column);

}

}

}

## //END CHANGE

Modification of **any required base classes** to enable the new ForagerAnt class to operate:

//CHANGE

public override void ChooseCellToMoveTo(List<int> ListOfNeighbours, int IndexOfNeighbourWithStrongestPheromone, int IndexOfNeighbourWithStrongestSmell = -1)

{

//END CHANGE

Modification of the Cell class to introduce the concept of smell:

//CHANGE

class Cell : Entity

{

protected int AmountOfFood;

protected int Smell;

public Cell(int StartRow, int StartColumn) : base(StartRow, StartColumn)

{

AmountOfFood = 0;

Smell = 0;

}

public int GetAmountOfFood()

{

return AmountOfFood;

}

public int GetSmell()

{

return Smell;

}

public override string GetDetails()

{

return $"{base.GetDetails()}{AmountOfFood} food present, smell: {Smell}{Environment.NewLine}{Environment.NewLine}";

}

public void UpdateFoodInCell(int Change)

{

AmountOfFood += Change;

if (AmountOfFood < 0)

{

AmountOfFood = 0; //Ensure food doesn't go negative

}

//Update smell based on the amount of food

Smell = AmountOfFood \* 10;

}

//END CHANGE

}

Modification of the AdvanceStage method in the Simulation class to call the updated ChooseCellToMoveTo method:

else

{

//CHANGE

if (A.GetFoodCarried() > 0)

{

UpdateAntsPheromoneInCell(A);

}

A.ChooseCellToMoveTo(GetIndicesOfNeighbours(A.GetRow(), A.GetColumn()),

GetIndexOfNeighbourWithStrongestPheromone(A.GetRow(), A.GetColumn()), GetIndexOfNeighbourWithStrongestSmell(A.GetRow(), A.GetColumn()));

//END CHANGE

}

}

foreach (Nest N in Nests)

{

N.AdvanceStage(Nests, Ants, Pheromones);

}

Create new GetIndexOfNeighbourWithStrongestSmell method:

//CHANGE

private int GetIndexOfNeighbourWithStrongestSmell(int Row, int Column)

{

int StrongestSmell = 0, IndexOfStrongestSmell = -1;

foreach (int Index in GetIndicesOfNeighbours(Row, Column))

{

if (Index != -1 && Grid[Index].GetSmell() > StrongestSmell)

{

IndexOfStrongestSmell = Index;

StrongestSmell = Grid[Index].GetSmell();

}

}

return IndexOfStrongestSmell;

}

//END CHANGE

Modify the SetUpANestAt method to add a single forager ant:

public void SetUpANestAt(int Row, int Column)

{

Nests.Add(new Nest(Row, Column, StartingFoodInNest));

Ants.Add(new QueenAnt(Row, Column, Row, Column));

//CHANGE

for (int Worker = 2; Worker <= StartingAntsInNest - 1; Worker++)

{

Ants.Add(new WorkerAnt(Row, Column, Row, Column));

}

Ants.Add(new ForagerAnt(Row, Column, Row, Column)); //Adds a single forager ant to each nest

//END CHANGE

}

## Testing

A screen shot of a computer

AI-generated content may be incorrect.● Show the program displaying suitable messages from the forager ant as it progresses and then heads back   
to the nest with food. **[1 mark]** ▶

# Task 8 (5 marks)

## Coding

 New option 6 to get the location to move an ant to – including appropriate modification of the DisplayMenu method. **[1 mark]**

 Create new SetNewLocation method in the WorkerAnt class to change the Row and Column of an ant. Additionally show required modification in the Ant class to allow correct operation. **[1 mark]**

 Create new RelocateAnt method in the Simulation class which will only operate on a cell which contains a worker ant. [1 mark]

 Suitable messages when an ant is moved successfully or the user attempts to select a cell which does not contain a worker ant. [1 mark]

### Teacher Notes

N/A

### Example Solution

Creation of new option 6 and modification of the DisplayMenu to match:

//CHANGE

case "6":

int AntOriginalRow = 0, AntOriginalColumn = 0, AntRelocationRow = 0, AntRelocationColumn = 0;

Console.WriteLine("Give the location of the ant that you want to relocate: ");

GetCellReference(ref AntOriginalRow, ref AntOriginalColumn);

Console.WriteLine("Give the location that you would like to relocate the ant to: ");

GetCellReference(ref AntRelocationRow, ref AntRelocationColumn);

Console.WriteLine(ThisSimulation.RelocateAnt(AntOriginalRow, AntOriginalColumn, AntRelocationRow, AntRelocationColumn));

break;

//END CHANGE

}

} while (Choice != "9");

Console.ReadLine();

}

//CHANGE

static void DisplayMenu()

{

Console.WriteLine();

Console.WriteLine("1. Display overall details");

Console.WriteLine("2. Display area details");

Console.WriteLine("3. Inspect cell");

Console.WriteLine("4. Advance one stage");

Console.WriteLine("5. Advance X stages");

Console.WriteLine("6: Relocate an ant to a new cell on the grid");

Console.WriteLine("9. Quit");

Console.WriteLine();

Console.Write("> ");

}

## //END CHANGE

Creation of new SetNewLocation method in the WorkerAnt class:

//CHANGE

public override void SetNewLocation(int NewRow, int NewColumn)

{

Row = NewRow;

Column = NewColumn;

}

//END CHANGE

\*\*\*\*\*\*\*\*Virtual Method in the Ant class

//CHANGE

public virtual void SetNewLocation(int NewRow, int NewColumn)

{

}

//END CHANGE

Create new RelocateAnt method in the Simulation class with required error checking:

//CHANGE

public string RelocateAnt(int AntOriginalRow, int AntOriginalColumn, int AntRelocationRow, int AntRelocationColumn)

{

foreach (Ant A in Ants)

{

if (A.GetRow() == AntOriginalRow && A.GetColumn() == AntOriginalColumn && A.GetTypeOfAnt() == "worker")

{

A.SetNewLocation(AntRelocationRow, AntRelocationColumn);

return $"Ant relocated successfully from {AntOriginalColumn},{AntOriginalRow} to {AntRelocationColumn},{AntRelocationRow}.";

}

}

return "No ant found at the original location.";

}

//END CHANGE

## Testing

A screenshot of a computer program

AI-generated content may be incorrect. Show the programing giving a suitable error message and then an ant successfully moved. Show a second screenshot showing the ant in a new location. **[1 mark]**

A black screen with white text

AI-generated content may be incorrect.

# Task 9 (9 marks)

## Coding

 Creation of new Bird class which inherits Entity. **[1 mark]**

 Mechanism to remove up to five ants at each stage if it is in the same cell. **[1 mark]**

 Suitable message to inform the user how many ants the bird has eaten in a single stage. [1 mark]

 Mechanism for the bird to be able to move to any random neighbouring cell that is within the bounds of the simulation “world”, including any required virtual methods in the base class to allow correct option. [1 mark]

 Modification of the AdvanceStage method in the Simulation class to correctly advance and instruct the bird to move. **[1 mark]**

 Correct modification of the GetDetails and GetAreaDetails methods to indicate the bird’s location when the simulation details are displayed. **[1 mark]**

 Suitable helper methods as required to be able to display the bird’s location when the simulation details are displayed. **[1 mark]**

 Randomly positioning the bird when the simulation starts. **[1 mark]**

### Teacher Notes

*The question requires that the candidate shows the location of the bird. This could be done using a “Birds” list but with only one element in it, following the same structure as the “Ants” or “Pheromones” lists that the original solution uses. That example is shown here so that candidates understand this concept for a question which might require a new list of multiple objects in the simulation. Since the question only specifies one bird to be added to the simulation, a single bird object in the Simulation class would also be acceptable.*

### Example Solution

Creation of new Bird class which inherits Entity:

//CHANGE

class Bird : Entity

{

public Bird(int StartRow, int StartColumn) : base(StartRow, StartColumn)

{

}

public override void AdvanceStage(List<Nest> Nests, List<Ant> Ants, List<Pheromone> Pheromones)

{

List<Ant> AntsToRemove = new List<Ant>();

foreach (Ant A in Ants)

{

if (A.GetRow() == Row && A.GetColumn() == Column)

{

AntsToRemove.Add(A);

if (AntsToRemove.Count == 5)

{

break;

}

}

}

if (AntsToRemove.Count > 0)

{

Console.WriteLine($"Bird at ({Row}, {Column}) has eaten {AntsToRemove.Count} ant(s)!");

}

foreach (Ant A in AntsToRemove)

{

Ants.Remove(A);

}

}

private void ChangeCell(int Direction, ref int RowToChange, ref int ColumnToChange)

{

if (Direction > 5)

{

RowToChange++;

}

else if (Direction < 3)

{

RowToChange--;

}

if (new int[] { 0, 3, 6 }.Contains(Direction))

{

ColumnToChange--;

}

else if (new int[] { 2, 5, 8 }.Contains(Direction))

{

ColumnToChange++;

}

}

public override void ChooseCellToMoveTo(int NumberOfRows, int NumberOfColumns)

{

int NewRow = Row, NewColumn = Column;

int Direction = 0;

bool ValidLocation = false;

while (!ValidLocation)

{

Direction = RGen.Next(0, 9);

ChangeCell(Direction, ref NewRow, ref NewColumn);

if (NewRow >= 1 && NewRow <= NumberOfRows && NewColumn >= 1 && NewColumn <= NumberOfColumns)

{

Row = NewRow;

Column = NewColumn;

ValidLocation = true;

}

}

}

}

## //END CHANGE

Required modification to the Entity class:

//CHANGE

public virtual void ChooseCellToMoveTo(int NumberOfRows, int NumberOfColumns)

{

}

//END CHANGE

Modification to the AdvanceStage method in the Simulation class to enable the bird to advance its stages and move:

//CHANGE

foreach (Bird B in Birds)

{

B.AdvanceStage(Nests, Ants, Pheromones);

B.ChooseCellToMoveTo(NumberOfRows + 1, NumberOfColumns + 1);

}

//END CHANGE

Modification to the GetDetails and GetAreaDetails methods *(typically this will be the same code in both methods, therefore only one is shown here)*:

//CHANGE

int AmountOfBirds = GetNumberOfBirdsInCell(TempCell);

if (AmountOfBirds > 0)

{

Details += $"| Birds: {AmountOfBirds} | ";

}

//END CHANGE

Details += Environment.NewLine;

Helper method to identify when a bird is in a cell:

//CHANGE

public int GetNumberOfBirdsInCell(Cell C)

{

int Count = 0;

foreach (Bird B in Birds)

{

if (B.InSameLocation(C))

{

Count++;

}

}

return Count;

}

//END CHANGE

Randomly positioning the bird at the start of the simulation:

class Simulation

{

protected List<Cell> Grid = new List<Cell>();

protected List<Ant> Ants = new List<Ant>();

protected List<Pheromone> Pheromones = new List<Pheromone>();

protected List<Nest> Nests = new List<Nest>();

//CHANGE

protected List<Bird> Birds = new List<Bird>();

//END CHANGE

…

//CHANGE

int BirdRow = RGen.Next(1, NumberOfRows + 1);

int BirdColumn = RGen.Next(1, NumberOfColumns + 1);

Birds.Add(new Bird(BirdRow, BirdColumn));

//END CHANGE

## Testing

 Show the program displaying a suitable message of how many ants the bird has eaten. **[1 mark]**

A black screen with white text

AI-generated content may be incorrect.

# Task 10 (9 marks)

## Coding

 Create new Weather class which inherits Entity. [1 mark]

 Mechanism to have only one weather event. [1 mark]

 Mechanism to remove all Ants and Pheromones in the same location as the weather for three stages. [1 mark]

 Suitable message to inform the user how many ants, pheromones and nests have been destroyed, if any, in a stage. [1 mark]

 Weather is only active for three stages. [1 mark]

 Ants avoid the weather when selecting which cell to move to (but do not if they are carrying food back to the nest). [1 mark]

 Modification to the AdvanceStage method in the Simulation class to give a 75% chance of creating a weather event and placing it randomly within the   
simulation “world”. [1 mark]

 Correct modification of the GetDetails and GetAreaDetails methods to indicate the weather’s location when the simulation details are displayed. **[1 mark]**

### Teacher Notes

*The question requires only one weather event and the candidate show its location. This could be done using a “Weather” list but with only one element in it, following the same structure as the “Ants” or “Pheromones” lists that the original solution uses. The example shown here gives an alternative approach using a single “Weather” object which stores its own location in the simulation “world” and can be moved / asked for that location. Both are acceptable.*

### Example Solution

Creation of Weather attribute:

protected int StartingAntsInNest, NewPheromoneStrength, PheromoneDecay;

private Weather CurrentWeather; // Only one weather event can exist at a time

public Simulation(List<int> SimulationParameters)

Creation of new Weather class which inherits Entity:

//CHANGE

class Weather : Entity

{

private int StagesRemaining;

public Weather(int StartRow, int StartColumn) : base(StartRow, StartColumn)

{

StagesRemaining = 3;

}

public override void AdvanceStage(List<Nest> Nests, List<Ant> Ants, List<Pheromone> Pheromones)

{

int AntsRemoved = 0;

int PheromonesRemoved = 0;

for (int i = Ants.Count - 1; i >= 0; i--)

{

if (Ants[i].GetRow() == Row && Ants[i].GetColumn() == Column)

{

Ants.RemoveAt(i);

AntsRemoved++;

}

}

for (int i = Pheromones.Count - 1; i >= 0; i--)

{

if (Pheromones[i].GetRow() == Row && Pheromones[i].GetColumn() == Column)

{

Pheromones.RemoveAt(i);

}

}

if (AntsRemoved > 0)

{

Console.WriteLine($"The weather has removed {AntsRemoved} ants");

}

if (PheromonesRemoved > 0)

{

Console.WriteLine($"The weather has removed {PheromonesRemoved} pheromones");

}

StagesRemaining--;

}

public bool IsExpired()

{

return StagesRemaining <= 0;

}

}

//END CHANGE

Modification of the ChooseCellToMoveTo method in the Ant class so that ants avoid the weather cell unless they are carrying food back to the nest:

//CHANGE

public override void ChooseCellToMoveTo(List<int> ListOfNeighbours, int IndexOfNeighbourWithStrongestPheromone, int IndexOfBadWeather)

{

//The ants move back towards the nest even if there is weather there...

if (AmountOfFoodCarried > 0)

{

if (Row > NestRow)

{

Row--;

}

else if (Row < NestRow)

{

Row++;

}

if (Column > NestColumn)

{

Column--;

}

else if (Column < NestColumn)

{

Column++;

}

}

else

{

int IndexToUse = -1;

do

{

if (IndexOfNeighbourWithStrongestPheromone != -1)

{

IndexToUse = ListOfNeighbours.IndexOf(IndexOfNeighbourWithStrongestPheromone);

}

else

{

IndexToUse = ChooseRandomNeighbour(ListOfNeighbours);

if (IndexToUse == IndexOfBadWeather)

{

continue;

}

}

int NewRow = Row, NewColumn = Column;

ChangeCell(IndexToUse, ref NewRow, ref NewColumn);

} while (IndexToUse == -1);

ChangeCell(IndexToUse, ref Row, ref Column);

}

//END CHANGE

}

Modification to the ChooseCellToMoveTo method in the base class to allow the Ant class to operate:

//CHANGE

public virtual void ChooseCellToMoveTo(List<int> ListOfNeighbours, int IndexOfNeighbourWithStrongestPheromone, int IndexOfBadWeather)

{

}

//END CHANGE

Modification of the AdvanceStage method in the Simulation class as required to make the ChooseCellToMoveTo method operate correctly:

else

{

if (A.GetFoodCarried() > 0)

{

UpdateAntsPheromoneInCell(A);

}

//CHANGE

A.ChooseCellToMoveTo(GetIndicesOfNeighbours(A.GetRow(), A.GetColumn()),

GetIndexOfNeighbourWithStrongestPheromone(A.GetRow(), A.GetColumn()),

GetIndexOfNeighbourWithBadWeather(A.GetRow(), A.GetColumn()));

}

//END CHANGE

Modification of the AdvanceStage method in the Simulation class to give a 75% chance of a weather event occurring and placing it randomly in the simulation “world”:

public void AdvanceStage(int NumberOfStages)

{

for (int Count = 1; Count <= NumberOfStages; Count++)

{

//CHANGE

List<int> Chances = new List<int>() { 0, 1, 2 };

if (CurrentWeather == null && Chances.Contains(RGen.Next(0, 4)))

{

int WeatherRow = RGen.Next(1, NumberOfRows + 1);

int WeatherColumn = RGen.Next(1, NumberOfColumns + 1);

CurrentWeather = new Weather(WeatherRow, WeatherColumn);

Console.WriteLine($"A weather event has occurred at: {WeatherRow},{WeatherColumn}");

}

//END CHANGE

…

foreach (Nest N in Nests)

{

N.AdvanceStage(Nests, Ants, Pheromones);

}

if (CurrentWeather != null)

{

//The impact of the weather needs to happen AFTER ants have moved around and been born / culled for this stage.

CurrentWeather.AdvanceStage(Nests, Ants, Pheromones);

if (CurrentWeather.IsExpired())

{

CurrentWeather = null; //Remove the weather event if it has expired

Console.WriteLine("The weather has passed...");

}

}

//END CHANGE

Modification to the GetDetails and GetAreaDetails methods *(typically this will be the same code in both methods, therefore only one is shown here)*:

//CHANGE

if (CurrentWeather != null)

{

if (CurrentWeather.GetRow() == Row && CurrentWeather.GetColumn() == Column)

{

Details += "| Weather |";

}

}

//END CHANGE

## Testing

 Show the program displaying a suitable message of how many ants/pheromones have been removed. **[1 mark]**

A screen shot of a computer

AI-generated content may be incorrect.

# Task 11 (5 marks)

## Coding

 Suitable data structure to store the food delivery data. [1 mark]

 New menu option 6 to display the food delivery data. [1 mark]

 Modification of the AdvanceStage method in the Simulation class to add delivery data to the data structure. [1 mark]

 Create new DisplayFoodDeliveryData method which displays details of the food collected by each ant together with an average. [1 mark]

### Teacher Notes

*The example solution given uses a dictionary to store the food delivery data. This could also be done with a two-dimensional array. Storing the delivery data as an attribute in the Ant class is not acceptable because ants are randomly culled from the simulation, which would destroy their data.*

### Example Solution

Modification to the Constructor in the Simulation class to store food delivery data:

class Simulation

{

protected List<Cell> Grid = new List<Cell>();

protected List<Ant> Ants = new List<Ant>();

protected List<Pheromone> Pheromones = new List<Pheromone>();

protected List<Nest> Nests = new List<Nest>();

protected int NumberOfRows, NumberOfColumns, StartingFoodInNest, StartingNumberOfFoodCells, StartingNumberOfNests;

protected int StartingAntsInNest, NewPheromoneStrength, PheromoneDecay;

//CHANGE

// Dictionary to store the amount of food delivered by each ant

private Dictionary<int, int> FoodDeliveredByAnt = new Dictionary<int, int>();

## //END CHANGE

Modification to the menu in the main static method in the Simulation class and the DisplayMenu method to give the new menu option:

//CHANGE

case "6":

ThisSimulation.DisplayFoodDeliveryData();

break;

//END CHANGE

}

} while (Choice != "9");

Console.ReadLine();

}

static void DisplayMenu()

{

Console.WriteLine();

Console.WriteLine("1. Display overall details");

Console.WriteLine("2. Display area details");

Console.WriteLine("3. Inspect cell");

Console.WriteLine("4. Advance one stage");

Console.WriteLine("5. Advance X stages");

//CHANGE

Console.WriteLine("6. Display food delivery data");

//END CHANGE

Console.WriteLine("9. Quit");

Console.WriteLine();

Console.Write("> ");

}

Modification of the AdvanceStage method in the Simulation class to correctly record which ant has delivered food to the nest, and how much:

foreach (Ant A in Ants)

{

A.AdvanceStage(Nests, Ants, Pheromones);

Cell CurrentCell = Grid[GetIndex(A.GetRow(), A.GetColumn())];

if (A.GetFoodCarried() > 0 && A.IsAtOwnNest())

{

AddFoodToNest(A.GetFoodCarried(), A.GetRow(), A.GetColumn());

//CHANGE

if (!FoodDeliveredByAnt.ContainsKey(A.GetID()))

{

FoodDeliveredByAnt[A.GetID()] = A.GetFoodCarried();//If the ant ID is not in the dictionary

}

else

{

FoodDeliveredByAnt[A.GetID()] += A.GetFoodCarried();

}

//END CHANGE

A.UpdateFoodCarried(-A.GetFoodCarried());

}

New DisplayFoodDeliveryData method:

//CHANGE

public void DisplayFoodDeliveryData()

{

Console.WriteLine("Ant ID\tFood Delivered");

Console.WriteLine("-----------------------");

int TotalFood = 0;

foreach (KeyValuePair<int, int> Entry in FoodDeliveredByAnt)

{

Console.WriteLine($"{Entry.Key}\t{Entry.Value}");

TotalFood += Entry.Value;

}

if (FoodDeliveredByAnt.Count > 0)

{

Console.WriteLine("-----------------------");

double AverageFood = TotalFood / FoodDeliveredByAnt.Count;

Console.WriteLine($"Average Food Delivered: {AverageFood:F2} units per ant");

}

Console.WriteLine("-----------------------");

}

//END CHANGE

## A screenshot of a computer AI-generated content may be incorrect.Testing

● Show the program displaying correct data for each ant in the simulation together with the average delivery per ant for all the ants in the colony. **[1 mark]** ▶

# Task 12 (7 marks)

## Coding

 New case 5 option in the Main method in the Simulation class to allow a custom simulation option. [1 mark]

 Creation of new method GetCustomSimulation. [1 mark]

 Suitable format and range checks which meet the requirements outlined. [1 mark]

 Suitable error messages if the input is not valid. [1 mark]

 Repeat the opportunity to input data until it is in a valid format for the specific parameter being inputted. [1 mark]

### Teacher Notes

*This example solution uses a helper method to do the format and range checking. This could all be included within the GetCustomSimulation method.*

### Example Solution

Modification to the Main method in the Simulation class:

static void Main()

{

List<int> SimulationParameters = new List<int>();

//CHANGE

int SimNo = GetInputInRange("Enter the simulation you want to run (1-4) enter 5 to create a custom simulation: ", 1, 5);

switch (SimNo)

{

case 1:

SimulationParameters = new List<int> { 1, 5, 5, 500, 3, 5, 1000, 50 };

break;

case 2:

SimulationParameters = new List<int> { 1, 5, 5, 500, 3, 5, 1000, 100 };

break;

case 3:

SimulationParameters = new List<int> { 1, 10, 10, 500, 3, 9, 1000, 25 };

break;

case 4:

SimulationParameters = new List<int> { 2, 10, 10, 500, 3, 6, 1000, 25 };

break;

case 5:

SimulationParameters = GetCustomSimulation();

break;

}

## //END CHANGE

New GetCustomSimulation method:

//CHANGE

static List<int> GetCustomSimulation()

{

List<int> SimulationParameters = new List<int>();

SimulationParameters.Add(GetInputInRange("Enter number of nests (1-5): ", 1, 5));

SimulationParameters.Add(GetInputInRange("Enter number of rows (5-20): ", 5, 20));

SimulationParameters.Add(GetInputInRange("Enter number of columns (5-20): ", 5, 20));

SimulationParameters.Add(GetInputInRange("Enter starting food in each nest (100-1000): ", 100, 1000));

SimulationParameters.Add(GetInputInRange("Enter starting number of food cells (1-10): ", 1, 10));

SimulationParameters.Add(GetInputInRange("Enter starting ants in each nest (5-20): ", 5, 20));

SimulationParameters.Add(GetInputInRange("Enter new pheromone strength (500-2000): ", 500, 2000));

SimulationParameters.Add(GetInputInRange("Enter pheromone decay per stage (10-200): ", 10, 200));

return SimulationParameters;

}

Format and range checks and error messaging, together with opportunity to re-input if required:

static int GetInputInRange(string Prompt, int Min, int Max)

{

int Input;

do

{

try

{

Console.Write(Prompt);

Input = Convert.ToInt32(Console.ReadLine());

if (Input < Min || Input > Max)

{

Console.WriteLine($"Input must be between {Min} and {Max}");

}

}

catch (FormatException)

{

Console.WriteLine("Not a valid integer");

Input = -1;

}

} while (Input < 0 || Input > Max);

return Input;

}

//END CHANGE

## Testing

● Show the program displaying a suitable error message that the value is not in range. **[1 mark]**

A computer screen with white text

AI-generated content may be incorrect.

● Show the program displaying the custom simulation details. *(As per the original code,   
it is possible that some of the food sources could be in the same cell as each other.)* **[1 mark]**

A black screen with white text

AI-generated content may be incorrect.

# Task 13 (4 marks)

## Coding

 Input the starting amount of food units (no error handling is required). **[1 mark]**

 Pass starting amount of food units as additional parameter into the simulation and store appropriately in the Simulation class. **[1 mark]**

 Correctly use the new parameter when setting up food cells. [1 mark]

### Teacher Notes

N/A

### Example Solution

Modification to the Main method to input how many food units start off in food cells and pass this in as an additional simulation parameter:

//CHANGE

Console.WriteLine("Enter in the starting amount of food units in food cells: ");

int StartingAmountOfFoodInFoodCells = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter simulation number: ");

string SimNo = Console.ReadLine();

switch (SimNo)

{

case "1":

SimulationParameters = new List<int> { 1, 5, 5, 500, 3, 5, 1000, 50, StartingAmountOfFoodInFoodCells };

break;

case "2":

SimulationParameters = new List<int> { 1, 5, 5, 500, 3, 5, 1000, 100, StartingAmountOfFoodInFoodCells };

break;

case "3":

SimulationParameters = new List<int> { 1, 10, 10, 500, 3, 9, 1000, 25, StartingAmountOfFoodInFoodCells };

break;

case "4":

SimulationParameters = new List<int> { 2, 10, 10, 500, 3, 6, 1000, 25, StartingAmountOfFoodInFoodCells };

break;

}

Simulation ThisSimulation = new Simulation(SimulationParameters);

string Choice;

## //END CHANGE

Storing the new parameter data in the Simulation class:

//CHANGE

protected int NumberOfRows, NumberOfColumns, StartingFoodInNest, StartingNumberOfFoodCells, StartingNumberOfNests, StartingAmountOfFoodInFoodCells;

//END CHANGE

protected int StartingAntsInNest, NewPheromoneStrength, PheromoneDecay;

public Simulation(List<int> SimulationParameters)

{

StartingNumberOfNests = SimulationParameters[0];

NumberOfRows = SimulationParameters[1];

NumberOfColumns = SimulationParameters[2];

StartingFoodInNest = SimulationParameters[3];

StartingNumberOfFoodCells = SimulationParameters[4];

StartingAntsInNest = SimulationParameters[5];

NewPheromoneStrength = SimulationParameters[6];

PheromoneDecay = SimulationParameters[7];

//CHANGE

StartingAmountOfFoodInFoodCells = SimulationParameters[8];

//END CHANGE

Assigning the new parameter data when setting up a food cell:

for (int Count = 1; Count <= StartingNumberOfFoodCells; Count++)

{

bool Allowed;

do

{

Allowed = true;

Row = RGen.Next(1, NumberOfRows + 1);

Column = RGen.Next(1, NumberOfColumns + 1);

foreach (Nest N in Nests)

{

if (N.GetRow() == Row && N.GetColumn() == Column)

{

Allowed = false;

}

}

} while (!Allowed);

//CHANGE

AddFoodToCell(Row, Column, StartingAmountOfFoodInFoodCells);

//END CHANGE

}

## Testing

A screenshot of a computer program

AI-generated content may be incorrect.● Show the program displaying the custom simulation details. *(As per the original code, it is possible that some of the food sources could be in the same cell as   
each other.)* **[1 mark]**

# Task 14 (13 marks)

## Coding

 Create new SoldierAnt class which inherits Ant. [1 mark]

 Mechanism for SoldierAnt to have states and move between them together with any required modifications to the base class to allow the soldier ant to   
operate correctly. [1 mark]

 Create new Predator class which inherits Entity. [1 mark]

 Modification to the AdvanceStage method in the Simulation class so that the soldier ant waits around the nest (“patrolling”) until a predator is detected in its   
neighbouring cells. [1 mark]

 Mechanism to move towards the predator if detected to engage with it. [1 mark]

 Modify the UpdateAntPheromoneInCell method in the Simulation class to add five times the amount of pheromone to a cell containing a soldier ant which is engaging with a predator. [1 mark]

 Modify the SetupANestAt to ensure the same number of soldier ants as there are worker ants. [1 mark]

 Modify the Main static method to add Option 6 to the menu to add a predator to a cell. [1 mark]

 Create new method in the Simulation class which adds new predators to the simulation (through option 6 in the main menu). The modification should allow the user to add multiple predators by repeatedly selecting the new option 6. [1 mark]

 Correct modification of the GetDetails method to show the number of soldier ants in a cell when the simulation details are displayed. **[1 mark]**

 Creation of any helper methods to allow the GetDetails method to operate correctly. [1 mark]

### Teacher Notes

*Although a soldier ant does not have the functionality to return to the nest, it still needs a small food capacity in order to operate simply within AQA’s Version 2 of the pre-release code.*

### Example Solution

Create new SoldierAnt Class which inherits Ant:

//CHANGE

class SoldierAnt : Ant

{

private string CurrentState = "";

public SoldierAnt(int StartRow, int StartColumn, int NestInRow, int NestInColumn)

: base(StartRow, StartColumn, NestInRow, NestInColumn)

{

TypeOfAnt = "soldier";

FoodCapacity = 5;

}

public override void ChooseCellToMoveTo(List<int> ListOfNeighbours, int IndexOfNeighbourWithStrongestPheromone)

{

if (IndexOfNeighbourWithStrongestPheromone == -1)

{

int IndexToUse = ChooseRandomNeighbour(ListOfNeighbours);

ChangeCell(IndexToUse, ref Row, ref Column);

}

else

{

int IndexToUse = ListOfNeighbours.IndexOf(IndexOfNeighbourWithStrongestPheromone);

ChangeCell(IndexToUse, ref Row, ref Column);

}

}

public override string GetDetails()

{

return CurrentState;

}

public override void UpdateState(string NewState)

{

CurrentState = NewState;

}

}

//END CHANGE

Update required in the Ant base class to allow the soldier ant to operate:

//CHANGE

public virtual void UpdateState(string NewState)

{

}

//END CHANGE

}

Creation of Predator class which inherits Entity and data structure to store it:

//CHANGE

class Predator : Entity

{

public Predator(int StartRow, int StartColumn) : base(StartRow, StartColumn)

{

}

public override string GetDetails()

{

return $"{base.GetDetails()} Predator present";

}

}

//END CHANGE

class Simulation

{

protected List<Cell> Grid = new List<Cell>();

protected List<Ant> Ants = new List<Ant>();

protected List<Pheromone> Pheromones = new List<Pheromone>();

protected List<Nest> Nests = new List<Nest>();

//CHANGE

protected List<Predator> Predators = new List<Predator>();

## //CHANGE

Modification to the AdvanceStage method in the Simulation class:

foreach (Ant A in Ants)

{

A.AdvanceStage(Nests, Ants, Pheromones);

Cell CurrentCell = Grid[GetIndex(A.GetRow(), A.GetColumn())];

if (A.GetFoodCarried() > 0 && A.IsAtOwnNest())

{

AddFoodToNest(A.GetFoodCarried(), A.GetRow(), A.GetColumn());

A.UpdateFoodCarried(-A.GetFoodCarried());

}

//CHANGE

else if (CurrentCell.GetAmountOfFood() > 0 && A.GetFoodCarried() == 0 && A.GetFoodCapacity() > 0 && A.GetTypeOfAnt() != "soldier")

{

//END CHANGE

int FoodObtained;

do

{

FoodObtained = RGen.Next(1, A.GetFoodCapacity() + 1);

} while (FoodObtained > CurrentCell.GetAmountOfFood() || (A.GetFoodCarried() + FoodObtained) > A.GetFoodCapacity());

CurrentCell.UpdateFoodInCell(-FoodObtained);

A.UpdateFoodCarried(FoodObtained);

}

else

{

if (A.GetFoodCarried() > 0)

{

UpdateAntsPheromoneInCell(A);

}

//CHANGE

if (A.GetTypeOfAnt() == "soldier" && A.GetDetails() == "patrolling")

{

List<int> NeighbourIndices = GetIndicesOfNeighbours(A.GetRow(), A.GetColumn());

foreach (int Index in NeighbourIndices)

{

if (Index != -1)

{

Cell TempCell = Grid[Index];

foreach (Predator PotentialPredator in Predators)

{

if (PotentialPredator.InSameLocation(TempCell))

{

A.ChooseCellToMoveTo(NeighbourIndices, Index);

A.UpdateState("engaging"); //engage with predator

Console.WriteLine("Soldier Ant engaging with predator");

}

}

}

}

}

else if (A.GetTypeOfAnt() == "soldier" && A.GetDetails() == "engaging" && CurrentCell.InSameLocation(A))

{

UpdateAntsPheromoneInCell(A);

Console.WriteLine("Soldier Ant is fighting with predator");

}

else if (A.GetTypeOfAnt() == "soldier" && A.IsAtOwnNest())

{

A.ChooseCellToMoveTo(GetIndicesOfNeighbours(A.GetRow(), A.GetColumn()),

GetIndexOfNeighbourWithStrongestPheromone(A.GetRow(), A.GetColumn()));

A.UpdateState("patrolling");//Set the soldier's action to patrolling when it leaves the nest

}

//END CHANGE

else

{

A.ChooseCellToMoveTo(GetIndicesOfNeighbours(A.GetRow(), A.GetColumn()),

GetIndexOfNeighbourWithStrongestPheromone(A.GetRow(), A.GetColumn()));

}

}

}

Modification to the GetDetails method to show where soldier ants and predators are:

//CHANGE

int NumberOfSoldierAnts = GetNumberOfSoldierAntsInCell(TempCell);

if (NumberOfSoldierAnts > 0)

{

Details += $"| Soldier Ants: {NumberOfSoldierAnts} | ";

}

int NumberOfPredators = GetNumberOfPredatorsInCell(TempCell);

if (NumberOfPredators > 0)

{

Details += $"| Predators: {NumberOfPredators} | ";

}

//END CHANGE

Modification to the helper methods for GetDetails to count the number of ants, soldier ants and predators:

//CHANGE

public int GetNumberOfAntsInCell(Cell C)

{

int Count = 0;

foreach (Ant A in Ants)

{

if (A.InSameLocation(C) && (A.GetTypeOfAnt() == "worker" || A.GetTypeOfAnt() == "queen"))

{

Count++;

}

}

return Count;

}

public int GetNumberOfSoldierAntsInCell(Cell C)

{

int Count = 0;

foreach (Ant A in Ants)

{

if (A.InSameLocation(C) && A.GetTypeOfAnt() == "soldier")

{

Count++;

}

}

return Count;

}

public int GetNumberOfPredatorsInCell(Cell C)

{

int Count = 0;

foreach (Predator P in Predators)

{

if (P.InSameLocation(C))

{

Count++;

}

}

return Count;

}

//END CHANGE

Modification to the UpdateAntsPheromoneInCell method in the Simulation class to be able to lay down stronger pheromones when a soldier ant is engaging with a predator:

public void UpdateAntsPheromoneInCell(Ant A)

{

//CHANGE

foreach (Pheromone P in Pheromones)

{

if (P.InSameLocation(A) && P.GetBelongsTo() == A.GetID())

{

if (A.GetTypeOfAnt() == "soldier" && A.GetDetails() == "engaging")

{

NewPheromoneStrength \*= 5; //soldier ants leave stronger pheromones when engaging predators

}

P.UpdateStrength(NewPheromoneStrength);

return;

}

}

if (A.GetTypeOfAnt() == "soldier" && A.GetDetails() == "engaging")

{

NewPheromoneStrength \*= 5; //soldier ants leave stronger pheromones when engaging predators

}

else

{

Pheromones.Add(new Pheromone(A.GetRow(), A.GetColumn(), A.GetID(), NewPheromoneStrength, PheromoneDecay));

}

}

Modification to the main method to include a new option 6 menu and associated methods required to place a predator in the simulation “world”:

//CHANGE

public void AddPredatorToCell(int Row, int Column)

{

Predators.Add(new Predator(Row, Column));

}

//END CHANGE

//CHANGE

case "6":

int PredatorRow = 0, PredatorColumn = 0;

GetCellReference(ref PredatorRow, ref PredatorColumn);

ThisSimulation.AddPredatorToCell(PredatorRow, PredatorColumn);

Console.WriteLine($"Predator added to cell {PredatorRow}, {PredatorColumn}{Environment.NewLine}");

break;

//END CHANGE

}

} while (Choice != "9");

Console.ReadLine();

}

static void DisplayMenu()

{

Console.WriteLine();

Console.WriteLine("1. Display overall details");

Console.WriteLine("2. Display area details");

Console.WriteLine("3. Inspect cell");

Console.WriteLine("4. Advance one stage");

Console.WriteLine("5. Advance X stages");

//CHANGE

Console.WriteLine("6. Put Predator into cell");

//END CHANGE

Console.WriteLine("9. Quit");

Add as many soldiers as worker ants to the simulation:

public void SetUpANestAt(int Row, int Column)

{

Nests.Add(new Nest(Row, Column, StartingFoodInNest));

Ants.Add(new QueenAnt(Row, Column, Row, Column));

for (int Worker = 2; Worker <= StartingAntsInNest; Worker++)

{

Ants.Add(new WorkerAnt(Row, Column, Row, Column));

//CHANGE

Ants.Add(new Soldier(Row, Column, Row, Column));

//END CHANGE

}

}

## A computer screen with white text AI-generated content may be incorrect.Testing

● Show the program displaying simulation details showing soldier ants are patrolling in the cells   
surrounding the nest. (Any combination of cells: 1:3, 1:4, 1:5, 2:3, 2:5, 3:3, 3:4, 3:5.) **[1 mark]** ▶

● Show the program displaying that a soldier ant is / soldier ants are engaging with the predator, and  
program displaying that a soldier ant is / soldier ants are fighting with the predator. [1 mark] ▼

A screenshot of a computer screen

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.

# Task 15 (6 marks)

## Coding

 Introduction of new option 6 into main menu. [1 mark]

 New accessor method in the Ant class to expose the Stages attribute. [1 mark]

 Calculating how many ants are carrying food and how much food is in each nest. [1 mark]

 Calculating the average age of the ants currently in the simulation. [1 mark]

 Presenting the summary data to the user in a simple way. [1 mark]

### Teacher Notes

*The pre-release code uses the technique of returning a string to print out to the console in the main static method. This example solution prints directly from the GetColonySummary method. Both are acceptable solutions.*

### Example Solution

New option 6 on the main menu:

//CHANGE

case "6":

ThisSimulation.GetColonySummary();

break;

}

} while (Choice != "9");

Console.ReadLine();

}

static void DisplayMenu()

{

Console.WriteLine();

Console.WriteLine("1. Display overall details");

Console.WriteLine("2. Display area details");

Console.WriteLine("3. Inspect cell");

Console.WriteLine("4. Advance one stage");

Console.WriteLine("5. Advance X stages");

//CHANGE

Console.WriteLine("6: Display Colony Stats");

## //END CHANGE

Exposing the Stages attribute from within the Ant class:

//CHANGE

public int GetStages()

{

return Stages;

}

//END CHANGE

New GetColonySummary method in the Simulation class which performs required calculations and presents the data to the user:

//CHANGE

public void GetColonySummary()

{

int TotalAnts = Ants.Count;

int AntsCarryingFood = 0;

int TotalFoodInNests = 0;

int TotalStagesAlive = 0;

// Count ants carrying food and calculate total stages alive

foreach (Ant A in Ants)

{

if (A.GetFoodCarried() > 0)

{

AntsCarryingFood++;

}

TotalStagesAlive += A.GetStages();

}

// Calculate total food in all nests

foreach (Nest N in Nests)

{

TotalFoodInNests += N.GetFoodLevel();

}

A screenshot of a computer

AI-generated content may be incorrect. // Calculate average age of ants

double AverageAge = 0;

if (TotalAnts > 0)

{

AverageAge = TotalStagesAlive / TotalAnts;

}

// Display the summary

Console.WriteLine("Colony Summary:");

Console.WriteLine($"Total Ants: {TotalAnts}");

Console.WriteLine($"Ants Carrying Food: {AntsCarryingFood}");

Console.WriteLine($"Total Food in Nests: {TotalFoodInNests}");

Console.WriteLine($"Average Age of Ants: {AverageAge:F2} stages");

}

//END CHANGE

}

## Testing

● Show the program displaying the colony summary data. **[1 mark]** ▶

# Task 16 (5 marks)

## Coding

 Introduction of new option 6 into main menu. [1 mark]

 New helper method to get the total strength of the pheromones in cell. [1 mark]

 New DisplayPheromoneHeatMap method in the Simulation class which correctly uses the NumberOfColumns and NumberOfRows attributes. [1 mark]

 Correctly displaying the total pheromone strength for each cell in a way which represents a grid/table. [1 mark]

### Teacher Notes

*The example solution uses padding to ensure the table is evenly spaced and laid out. This is not necessary. Candidates should be assessed on their ability to collect and display the required information, not on its appearance. A rough representation of the simulation “world” in a grid / tabular form is acceptable.*

### Example Solution

New option 6 on the main menu:

//CHANGE

case "6":

Console.WriteLine("Heat Map for the simulation");

Console.WriteLine(ThisSimulation.DisplayPheromoneHeatMap());

break;

//END CHANGE

}

} while (Choice != "9");

Console.ReadLine();

}

static void DisplayMenu()

{

Console.WriteLine();

Console.WriteLine("1. Display overall details");

Console.WriteLine("2. Display area details");

Console.WriteLine("3. Inspect cell");

Console.WriteLine("4. Advance one stage");

Console.WriteLine("5. Advance X stages");

//CHANGE

Console.WriteLine("6. Display Pheromone Heat Map");

## //END CHANGE

Helper method to calculate the total strength of pheromone in a cell:

//CHANGE

public int GetTotalPheromoneStrengthInCell(Cell C)

{

int TotalStrength = 0;

foreach (Pheromone P in Pheromones)

{

if (P.InSameLocation(C))

{

TotalStrength += P.GetStrength();

}

}

return TotalStrength;

}

//END CHANGE

New DisplayPheromoneHeatMap method which collects together the required information and displays it:

//CHANGE

public string DisplayPheromoneHeatMap()

{

int cellWidth = 10;

string details = "";

string rowSeparator = "";

string headerRow = "";

for (int col = 1; col <= NumberOfColumns; col++)

{

headerRow += $"| {col} ".PadRight(cellWidth - 2) + " ";

}

headerRow += "|\n";

for (int col = 1; col <= NumberOfColumns; col++)

{

rowSeparator += "+" + new string('-', cellWidth);

}

rowSeparator += "+\n";

details += " " + headerRow;

for (int row = 1; row <= NumberOfRows; row++)

{

details += " " + rowSeparator;

string rowLine = "";

for (int col = 1; col <= NumberOfColumns; col++)

{

Cell cell = Grid[GetIndex(row, col)];

if (col == 1)

{

rowLine += $"{row}".PadRight(3) + "|" + GetTotalPheromoneStrengthInCell(cell).ToString().PadRight(cellWidth);

}

else

{

rowLine += $"|" + GetTotalPheromoneStrengthInCell(cell).ToString().PadRight(cellWidth);

}

}

rowLine += "|\n";

details += rowLine;

}

details += " " + rowSeparator;

return details;

}

//END CHANGE

## A screenshot of a computer AI-generated content may be incorrect.Testing

 Show the program displaying the heat map. **[1 mark]** ▶

# Task 17 (7 marks)

## Coding

 Introduction of new option 6 into main menu. [1 mark]

 Modification of the Ant class to mark that it has been infected, and for how long, together with suitable access methods to expose these attributes. [1 mark]

 Increment the infection period at the correct time. [1 mark]

 Modify the AdvanceStage method in the Nest class to cull infected ants at 10 stages. [1 mark]

 Modify the AdvanceStage method in the Simulation class so that infected ants no longer move. [1 mark]

 Create new method InfectAntsInCell which infects any ants in that cell which are not already infected. Inform the user how many ants have been infected. [1 mark]

### Teacher Notes

N/A

### Example Solution

New option 6 in main menu:

//CHANGE

case "6":

int RowToInfect = 0, ColumnToInfect = 0;

GetCellReference(ref RowToInfect, ref ColumnToInfect);

Console.WriteLine(ThisSimulation.InfectAntsInCell(RowToInfect, ColumnToInfect));

break;

//END CHANGE

}

} while (Choice != "9");

Console.ReadLine();

}

static void DisplayMenu()

{

Console.WriteLine();

Console.WriteLine("1. Display overall details");

Console.WriteLine("2. Display area details");

Console.WriteLine("3. Inspect cell");

Console.WriteLine("4. Advance one stage");

Console.WriteLine("5. Advance X stages");

//CHANGE

Console.WriteLine("6. Infect Ants in a cell");

//END CHANGE

## Console.WriteLine("9. Quit");

Modification to the Ant class to identify it as infected, and for how long, and increment the infection period appropriately:

class Ant : Entity

{

protected int NestRow, NestColumn, AmountOfFoodCarried, Stages, FoodCapacity;

protected string TypeOfAnt;

protected static int NextAntID = 1;

//CHANGE

protected bool Diseased = false;

protected int StagesSinceInfection = 0;

//END CHANGE

public Ant(int StartRow, int StartColumn, int NestInRow, int NestInColumn)

: base(StartRow, StartColumn)

{

NestRow = NestInRow;

NestColumn = NestInColumn;

ID = NextAntID;

NextAntID++;

Stages = 0;

AmountOfFoodCarried = 0;

FoodCapacity = 0;

TypeOfAnt = "";

}

//CHANGE

public void Infect()

{

Diseased = true;

StagesSinceInfection = 0;

}

public bool IsDiseased()

{

return Diseased;

}

public int GetStagesSinceInfection()

{

return StagesSinceInfection;

}

//END CHANGE

…

public override void AdvanceStage(List<Nest> Nests, List<Ant> Ants, List<Pheromone> Pheromones)

{

//CHANGE

Stages++;

if (IsDiseased())

{

StagesSinceInfection++;

}

//END CHANGE

}

Modify the AdvanceStage method in the Nest class to remove ants which have been diseased for 10 or more stages:

//CHANGE

int NumberOfAntsDiedThroughDisease = 0;

for (int i = 0; i < Ants.Count; i++)

{

if (Ants[i].GetTypeOfAnt() == "worker" && Ants[i].IsDiseased() && Ants[i].GetStagesSinceInfection() >= 10)

{

Ants.RemoveAt(i);

i--;

NumberOfAntsDiedThroughDisease++;

}

}

if (NumberOfAntsDiedThroughDisease > 0)

{

Console.WriteLine($"{NumberOfAntsDiedThroughDisease} have died through disease");

}

//END CHANGE

Modify the AdvanceStage method in the Simulation class so that infected ants can no longer move:

else

{

if (A.GetFoodCarried() > 0)

{

UpdateAntsPheromoneInCell(A);

}

//CHANGE

if (!A.IsDiseased())

{

A.ChooseCellToMoveTo(GetIndicesOfNeighbours(A.GetRow(), A.GetColumn()),

GetIndexOfNeighbourWithStrongestPheromone(A.GetRow(), A.GetColumn()));

}

}

Create new method InfectAntsInCell which infects all ants in a cell not already infected and advises the user how many ants have been infected:

//CHANGE

public string InfectAntsInCell(int Row, int Column)

{

int TotalNumberOfAntsInfectedInCell = 0;

foreach (Ant A in Ants)

{

if (A.GetRow() == Row && A.GetColumn() == Column && A.GetTypeOfAnt() == "worker" && !A.IsDiseased())

{

A.Infect();

TotalNumberOfAntsInfectedInCell++;

}

}

return $"Total number of ants infected in cell {Row}, {Column}: {TotalNumberOfAntsInfectedInCell}{Environment.NewLine}";

}

A screenshot of a computer program

AI-generated content may be incorrect. //END CHANGE

## Testing

● Show the program displaying how many ants were originally infected and how   
many ants have been removed. **[1 mark]** ▶

# Task 18 (8 marks)

## Coding

 Introduction of new option 6 into main menu. [1 mark]

 Create new DangerPheromone class which inherits Pheromone. **[1 mark]**

 Modification of the Nest class to include a DangerPheromone attribute including appropriate accessor and mutator method for the attribute. [1 mark]

 Modify the AdvanceStage method in the Simulation class to trigger a DangerPheromone when the food levels in a nest drop below 50 units. **[1 mark]**

 Correct modification of the GetDetails, GetAreaDetails and GetCellDetails methods to indicate when a nest’s food levels are dangerously low. **[1 mark]**

 Helper methods required to allow GetDetails, GetAreaDetails and GetCellDetails to operate. **[1 mark]**

 Create new method GetDangerPheromoneLocations method in the Simulation class which collects together the location details of the nests in the simulation “world” which are dangerously low on food. [1 mark]

### Teacher Notes

*This example solution uses a composition relationship between the Nest class and a DangerPheromone object to demonstrate a different approach. This could be done using a “DangerPheromone” list but with only one element in it, following the same structure as the “Ants” or “Pheromones” lists that the original solution uses. Both are acceptable. Setting a nest as “no longer in danger” is difficult to demonstrate at runtime due to the number of variables that influence it happening.*

### Example Solution

New option 6 in main menu:

//CHANGE

case "6":

Console.WriteLine(ThisSimulation.GetDangerPheromoneLocations());

break;

//END CHANGE

}

} while (Choice != "9");

Console.ReadLine();

}

static void DisplayMenu()

{

Console.WriteLine();

Console.WriteLine("1. Display overall details");

Console.WriteLine("2. Display area details");

Console.WriteLine("3. Inspect cell");

Console.WriteLine("4. Advance one stage");

Console.WriteLine("5. Advance X stages");

//CHANGE

Console.WriteLine("6. Display Nests with dangerously low food levels");

//END CHANGE

## Console.WriteLine("9. Quit");

Creation of new DangerPheromone class which inherits Pheromone:

//CHANGE

class DangerPheromone : Pheromone

{

public DangerPheromone(int Row, int Column, int BelongsToAnt, int InitialStrength, int Decay)

: base(Row, Column, BelongsToAnt, InitialStrength, Decay)

{

BelongsTo = BelongsToAnt;

Strength = InitialStrength;

PheromoneDecay = Decay;

}

}

//END CHANGE

Modification to the Nest class to include DangerPheromone:

class Nest : Entity

{

protected int FoodLevel, NumberOfQueens;

protected static int NextNestID = 1;

//CHANGE

private DangerPheromone D;

//END CHANGE

public Nest(int StartRow, int StartColumn, int StartFood) : base(StartRow, StartColumn)

{

FoodLevel = StartFood;

NumberOfQueens = 1;

ID = NextNestID;

NextNestID++;

}

public void ChangeFood(int Change)

{

FoodLevel += Change;

if (FoodLevel < 0)

{

FoodLevel = 0;

}

}

//CHANGE

public void SetInDanger(bool status)

{

D = new DangerPheromone(GetRow(), GetColumn(), 0, 100, 10);

}

public bool InDanger()

{

if (D != null)

{

return true;

}

else

{

return false;

}

}

//END CHANGE

Triggering a DangerPheromone in a nest if the food level drops too low and resetting if the levels rise back up again:

foreach (Nest N in Nests)

{

N.AdvanceStage(Nests, Ants, Pheromones);

//CHANGE

if (N.GetFoodLevel() < 50 && !N.InDanger())

{

N.SetInDanger(true);

}

else

{

N.SetInDanger(false);

}

//CHANGE

}

Modification to the GetDetails, GetAreaDetails and GetCellDetails methods *(typically this will be the same code in both methods, therefore only one is shown here)*:

//CHANGE

if (IsDangerPheromoneInCell(CurrentCell))

{

Details += "| Nest has dangerously low food levels | ";

}

//END CHANGE

Helper method for the GetDetails, GetAreaDetails and GetCellDetails methods:

//CHANGE

public bool IsDangerPheromoneInCell(Cell C)

{

foreach (Nest N in Nests)

{

if (N.InDanger() && N.InSameLocation(C))

{

return true;

}

}

return false;

}

A screenshot of a computer program

AI-generated content may be incorrect.Create new GetDangerPheromoneLocations method:

public string GetDangerPheromoneLocations()

{

String Details = "";

foreach (Nest N in Nests)

{

if (N.InDanger())

{

Details += $"Nest at ({N.GetRow()}, {N.GetColumn()}) has dangerously low food levels.{Environment.NewLine}";

}

}

return Details;

}

//END CHANGE

## Testing

 Show the program displaying the location and food levels of nests which are dangerously low   
on food and details of one of the nests in danger. **[1 mark]** ▶

# Task 19 (6 marks)

## Coding

 Introduction of new option 6 into main menu. [1 mark]

 Modification of the Cell class to flag when it has cinnamon in it and suitable access and mutator methods to expose this flag. [1 mark]

 New RemovePheromonesFromCell method which identifies a cell as containing cinnamon and removes all pheromones from it. [1 mark]

 Mechanism to prevent an ant from updating the pheromones in a cell if the cell contains cinnamon. [1 mark]

 Mechanism to prevent a worker ant from entering a cell containing cinnamon when it is searching randomly for food. [1 mark]

### Teacher Notes

*Assessing a solution working with an ant searching for food is difficult because ants move randomly; therefore, the assessment is mostly through the code.*

### Example Solution

New option 6 in main menu:

//CHANGE

case "6":

int DropRow = 0, DropColumn = 0;

GetCellReference(ref DropRow, ref DropColumn);

ThisSimulation.RemovePheromonesFromCell(DropRow, DropColumn);

Console.WriteLine($"Cinnamon dropped on cell {DropRow}, {DropColumn} to remove pheromones.{Environment.NewLine}");

break;

//END CHANGE

}

} while (Choice != "9");

Console.ReadLine();

}

static void DisplayMenu()

{

Console.WriteLine();

Console.WriteLine("1. Display overall details");

Console.WriteLine("2. Display area details");

Console.WriteLine("3. Inspect cell");

Console.WriteLine("4. Advance one stage");

Console.WriteLine("5. Advance X stages");

//CHANGE

Console.WriteLine("6: Drop Cinnamon onto a cell to remove pheromones");

## //END CHANGE

Modification to the Cell class to identify it as containing cinnamon:

class Cell : Entity

{

protected int AmountOfFood;

//CHANGE

private bool HasCinnamon = false;

//END CHANGE

public Cell(int StartRow, int StartColumn) : base(StartRow, StartColumn)

{

AmountOfFood = 0;

}

…

//CHANGE

public void PlaceCinnamon()

{

AmountOfFood = 0;

HasCinnamon = true;

}

public bool ContainsCinnamon()

{

return HasCinnamon;

}

//END CHANGE

Create new RemovePheromonesFromCell method:

//CHANGE

public void RemovePheromonesFromCell(int Row, int Column)

{

Cell TargetCell = Grid[GetIndex(Row, Column)];

TargetCell.PlaceCinnamon();

List<Pheromone> PheromonesToRemove = new List<Pheromone>();

foreach (Pheromone P in Pheromones)

{

if (P.InSameLocation(TargetCell))

{

PheromonesToRemove.Add(P);

}

}

foreach (Pheromone P in PheromonesToRemove)

{

Pheromones.Remove(P);

}

}

//END CHANGE

Preventing an ant from updating the pheromones in a cell:

public void UpdateAntsPheromoneInCell(Ant A)

{

//CHANGE

if (Grid[GetIndex(A.GetRow(), A.GetColumn())].ContainsCinnamon())

{

return;

}

//END CHANGE

foreach (Pheromone P in Pheromones)

{

if (P.InSameLocation(A) && P.GetBelongsTo() == A.GetID())

{

P.UpdateStrength(NewPheromoneStrength);

return;

}

}

Pheromones.Add(new Pheromone(A.GetRow(), A.GetColumn(), A.GetID(), NewPheromoneStrength, PheromoneDecay));

}

Preventing an ant which is exploring from entering a neighbouring cell which contains cinnamon:

foreach (int ColumnDirection in new int[] { -1, 0, 1 })

{

int NeighbourRow = Row + RowDirection, NeighbourColumn = Column + ColumnDirection;

if ((RowDirection != 0 || ColumnDirection != 0) && NeighbourRow >= 1 && NeighbourRow <= NumberOfRows &&

NeighbourColumn >= 1 && NeighbourColumn <= NumberOfColumns)

{

//CHANGE

if (Grid[GetIndex(NeighbourRow, NeighbourColumn)].ContainsCinnamon())

{

ListOfNeighbours.Add(-1);

continue;

}

//END CHANGE

ListOfNeighbours.Add(GetIndex(NeighbourRow, NeighbourColumn));

}

else

{

ListOfNeighbours.Add(-1);

}

## Testing

● Show the program displaying all the pheromones removed from the selected cell. **[1 mark]**

A screenshot of a computer program

AI-generated content may be incorrect.

# Task 20 (5 marks)

## Coding

 Add a suitable data structure to the WorkerAnt class which enables an ant to store its movement history. **[1 mark]**

 Store the movement history of the worker ant from one stage to another. **[1 mark]**

 Override the normal worker ant movement behaviour to backtrack through the movement history completely back to the nest after the ant is six stages “old”. [1 mark]

 Reset the history if the worker ant finds food. [1 mark]

### Teacher Notes

*The LIFO structure of a stack lends itself very effectively to this question; however, it could be done with a normal list or array if interacted with appropriately. Equally this solution uses an attribute “ForagingStages” to keep track of the number of stages. This could also be done by checking the length of the data store being used for backtracking.*

### Example Solution

Modification of the WorkerAnt class to introduce a new data structure to store the movement history:

class WorkerAnt : Ant

{

//CHANGE

private Stack<Tuple<int, int>> MovementHistory;

private int ForagingStages;

private bool BackTracking = false;

public WorkerAnt(int StartRow, int StartColumn, int NestInRow, int NestInColumn)

: base(StartRow, StartColumn, NestInRow, NestInColumn)

{

TypeOfAnt = "worker";

FoodCapacity = 30;

MovementHistory = new Stack<Tuple<int, int>>();

ForagingStages = 0;

}

//END CHANGE

Modification of the ChooseCellToMoveTo method in the WorkerAnt class to push movement history onto a suitable data structure:

public override void ChooseCellToMoveTo(List<int> ListOfNeighbours, int IndexOfNeighbourWithStrongestPheromone)

{

if (AmountOfFoodCarried > 0)

{

//CHANGE

ForagingStages = 0; //Covers the scenario of ant finds food by luck while foraging

MovementHistory.Clear();

if (Row > NestRow)

{

Row--;

}

else if (Row < NestRow)

{

Row++;

}

if (Column > NestColumn)

{

Column--;

}

else if (Column < NestColumn)

{

Column++;

}

}

//CHANGE

else if (ForagingStages >= 7)

{

BacktrackToPreviousCell();

}

else if (IndexOfNeighbourWithStrongestPheromone == -1)

{

int IndexToUse = ChooseRandomNeighbour(ListOfNeighbours);

ChangeCell(IndexToUse, ref Row, ref Column);

//Console.WriteLine($"ForagingStages for this Ant: {ForagingStages}");

MovementHistory.Push(Tuple.Create(Row, Column)); //The ant is foraging whilst it can't smell a pheromone trail.

ForagingStages++;

}

else

{

int IndexToUse = ListOfNeighbours.IndexOf(IndexOfNeighbourWithStrongestPheromone);

ChangeCell(IndexToUse, ref Row, ref Column);

ForagingStages = 0; //Once it finds a pheromone trail, the ant is no longer foraging.

MovementHistory.Clear();

}

//END CHANGE

}

Creation of BacktrackToPreviousCell method which is called when ant reaches seven stages of movement without finding food:

//CHANGE

else if (ForagingStages >= 7)

{

BacktrackToPreviousCell();

}

//END CHANGE

else if (IndexOfNeighbourWithStrongestPheromone == -1)

{

int IndexToUse = ChooseRandomNeighbour(ListOfNeighbours);

ChangeCell(IndexToUse, ref Row, ref Column);

}

else

{

int IndexToUse = ListOfNeighbours.IndexOf(IndexOfNeighbourWithStrongestPheromone);

ChangeCell(IndexToUse, ref Row, ref Column);

}

}

//CHANGE

public void BacktrackToPreviousCell()

{

BackTracking = true;

if (MovementHistory.Count > 0)

{

Console.Write($"Ant {ID} backtracked from {Row}, {Column}");

Tuple<int, int> PreviousPosition = MovementHistory.Pop();

Row = PreviousPosition.Item1;

Column = PreviousPosition.Item2;

Console.WriteLine($" to {Row}, {Column}");

if (Row == NestRow && Column == NestColumn)

{

BackTracking = false;

ForagingStages = 0;

}

}

}

//END CHANGE

## Testing

 Show the program displaying backtracking route(s) of ants which have not successfully found food. **[1 mark]**

A screenshot of a computer program

AI-generated content may be incorrect.