Syddansk Universitet
IMADA

27-11-2020
DS801

INTRODUKTION TIL OBJEKTORIENTERET PROGRAMMERING:

Projekt 2/3 (Gruppe 25)

Koordinator: Simon Lorentzen

Stefan Randa

Sarah Pedersen

Indhold

1. Indledning **2**

2. Kravspecifikation **2**

3. Implementering **4**

3.1. Node 4

3.2. Edge 4

3.3. Ant 5

3.4. Simulator 6

4. Test **10**

4.1. Modultest 11

4.2. Fejl 13

5. Konklusion **14**

Referencer **14**

Bilag I (Node.java) **15**

Bilag II (Edge.java) **15**

Bilag III (Ant.java) **16**

Bilag IV (Simulator.java) **18**

Bilag V (Test.java) **21**

Bilag VI (Metodeoversigt for Simulator) **24**

1. Indledning

Denne rapport dokumenterer den anden fase i eksamensprojektet til kurset *Introduktion til objektorienteret programmering* (DS801). Kravene til projektet er specificeret i det kommende afsnit. Derefter beskrives nogle af de designbeslutninger, som gruppen har taget, og derpå uddybes implementeringen af relevante metoder. Til sidst foreligger et afsnit om testscenarier (herunder modultest med dedikeret kode), en konklusion og relevante bilag.[1]

2. Kravspecifikation

Gruppen skal i denne fase af projektet udarbejde fire klasser (*Node*, *Edge*, *Ant* og *Simulator*), hvortil følgende krav er specificeret. For klassen *Node*:

- En standardkonstruktør uden argumenter, som opretter en ny knude uden sukker.
- En konstruktør med ét argument, som opretter en ny knude med en given mængde sukker.
- En metode, **int sugar()**, som returnerer mængden af sukker i denne knude.
- En metode, **void decreaseSugar()**, som reducerer mængden af sukker i denne knude med én enhed.
- En metode, **void setSugar(int sugar)**, som nulstiller mængden af sukker i denne knude.

For klassen *Edge*:

- En konstruktør, som har to **Node**s som argumenter, og som opretter en ny kant mellem dem uden feromoner.
- To metoder, **Node source()** og **Node target()**, som returnerer referencerne til hhv. kilden og målet af denne kant.
- En metode, **int pheromones()**, som returnerer niveauet af feromoner i denne kant.
- En metode, **int decreasePheromones()**, som reducerer mængden af feromoner i denne kant med én enhed.
- En metode, **raisePheromones()**, som øger niveauet af feromoner i denne kant med en given mængde.

[1] OBS: Formuleringerne i dette afsnit er delvist kopieret fra vores første rapport.

For klassen *Ant*:

- En konstruktør, som har en **Colony** som argument, og som opretter en ny myre, der tilhører en given koloni.
- En metode, **void move(Node location)**, som transporterer denne myre til en specificeret lokation.
- En metode, **Node current()**, som returnerer denne myres aktuelle lokation.
- En metode, **Node previous()**, som returnerer denne myres forrige lokation.
- En metode, **Colony home()**, som returnerer denne myres hjemkoloni.
- To metoder, **boolean isAtHome()** og **boolean wasAtHome()**, som indikerer, hvorvidt denne myres forrige lokation stemmer overens med dens hjemkoloni.
- En metode, **boolean carrying()**, som indikerer, hvorvidt denne myre i øjeblikket bærer sukker.
- En metode, **void pickUpSugar()**, som modellerer denne myres opsamling af en fast mængde sukker fra sin aktuelle lokation.
- En metode, **void dropSugar()**, som modellerer denne myres afgivelse af sin sukkerbelastning i den koloni, den tilhører.

For klassen *Simulator*:

- En konstruktør, som opretter en ny simulation, og som har en **Graph**, en array af **Ant**s samt mængden af sukker og feromoner, en myre bærer og afgiver, som argumenter.
- En metode, **void tick()**, som opdaterer simulationen med én tidsenhed.

Samt en rapport, *report.pdf*, som inkluderer:

- En beskrivelse af og eksempler på den implementerede kode.
- Relevante designbeslutninger.
- Scenarier, metoder og resultater fra test af koden (med dedikeret kode).
- Et bilag, som inkluderer kildekoden.

Disse fem filer skal afleveres i én samlet ZIP-fil uden undermapper.[2]

[2] OBS: Formuleringerne i dette afsnit er delvist kopieret fra vores første rapport.

3. Implementering

Dette afsnit indeholder en forklaring af klassernes interne betydning samt en uddybende beskrivelse af den implementerede kode og/eller andre relevante designbeslutninger. Bemærk, at denne uddybning primært vedrører *Simulator*, eftersom de tre resterende klasserne er overvejende selvforklarende.

3.1. Node

Denne klasse[3] repræsenterer de lokationer på grafen, hvorigennem myrerne bevæger sig. Instanser af *Node* indeholder enten tomme eller tilfældige mængder af sukker. Hertil implementeres bl.a. en standardkonstruktør, **public Node()**, der etablerer knudeinstanser uden sukker, og en et-argument-konstruktør, **public Node(int sugar)**, der etablerer knudeinstanser med prædefinerede mængder af sukker.

Klassens resterende metoder:

- **public int sugar()** En getter, som returnerer det samlede antal sukkerenheder i en knudeinstans.
- **public void decreaseSugar()** Trækker én sukkerenhed fra det samlede antal i denne knudeinstans.
- **public void setSugar(int sugar)** Sætter mængden af sukker i en knudeinstans til en prædefineret mængde.

3.2. Edge

Denne klasse repræsenterer forbindelserne mellem myrernes mulige lokationer. *Edge* rummer med andre ord stierne mellem tilsluttede knudeinstanser, som inkluderer en kilde- og en målknude (defineret af en to-argument-konstruktør, **public Edge(final Node source, final Node target**) samt det afgivne feromonniveau fra de forbipasserende myrer. Myrerne kan bevæge sig i alle retninger.

Klassens resterende metoder:

- **public Node source()** En getter, der henter den første knudeinstans, som kantinstansen er forbundet til.

[3] Bemærk, at *Colony*, som gruppen skal implementere i tredje og sidste fase af projektet, nedarver sine attributter og metoder fra *Node*.

- **public Node target()** En getter, der henter den anden knudeinstans, som kantinstansen er forbundet til.
- **public int pheromones()** En getter, der returnerer mængden af feromoner, som er til stede i kantinstansen.
- **public void raisePheromones(int amount)** Øger feromonniveauet på kantinstansen.
- **public void decreasePheromones()** Trækker én fra feromonniveauet i kantinstansen.

3.3. Ant

Denne klasse repræsenterer de individuelle myrer, som hver især kender sin hjemkoloni og nuværende såvel som tidligere lokation på grafen. Instanser af *Ant* registrerer således myrernes positionelle variabler ved at pege på deres hjemkoloni (defineret af en et-argument-konstruktør, **public Ant(Colony home)**). Myrerne kan bære en fast mængde af sukker.

Klassens resterende metoder:

- **public void move(Node location)** Denne metode fortæller myreinstanser, hvor de skal bevæge sig til.
- **public Node current()** En getter, der returnere den knudeinstans, som myreinstansen befinder sig i.
- **public Node previous()** En getter, der henter den knudeinstans, som en myreinstansen befandt sig i under det forrige tick.
- **public Colony home()** En getter, der henter den koloniinstans, som myreinstansen blev oprettet i.
- **public boolean isAtHome()** Tjekker om myreinstansen befinder sig i den koloninstans, som den blev oprettet i.
- **public boolean wasAtHome()** Tjekker om myreinstansen befandt sig i den koloniinstans, som den blev oprettet i, under forrige tick.
- **public boolean carrying()** Tjekker om myreinstansen bærer på sukker.
- **public void pickUpSugar()** Får myreinstansen til at samle sukker op fra en knudeinstans med sukker i.
- **public void dropSugar()** Får myreinstans til at smide sukker fra sig.

3.4. Simulator

Denne klasse opretter og eksekverer en simulation. *Simulator* er en klient af klasserne *RandomUtils*, *Graph* og *Ant*, og har følgende attributter: **Graph graph**, **Ant ants**, **int sugarCapacity** og **int droppedPheromones**.[4]

For at tilgå informationer fra *Edge*, *Colony* og *Node*, skal der sættes attributter for **graph** og **ant**, så de førnævnte klasser kan tilgås gennem metoderne for **graph** og **ants**. *Simulator* opdaterer alle informationer om myrer, idet de påvirkes af mængden af sukker i kolonien såvel som feromoner. I procesdiagrammet herunder gives et high level overblik over hvilke handlinger, *Simulator* er ansvarlig for, og i hvilken rækkefølge, denne proces bliver kaldt.

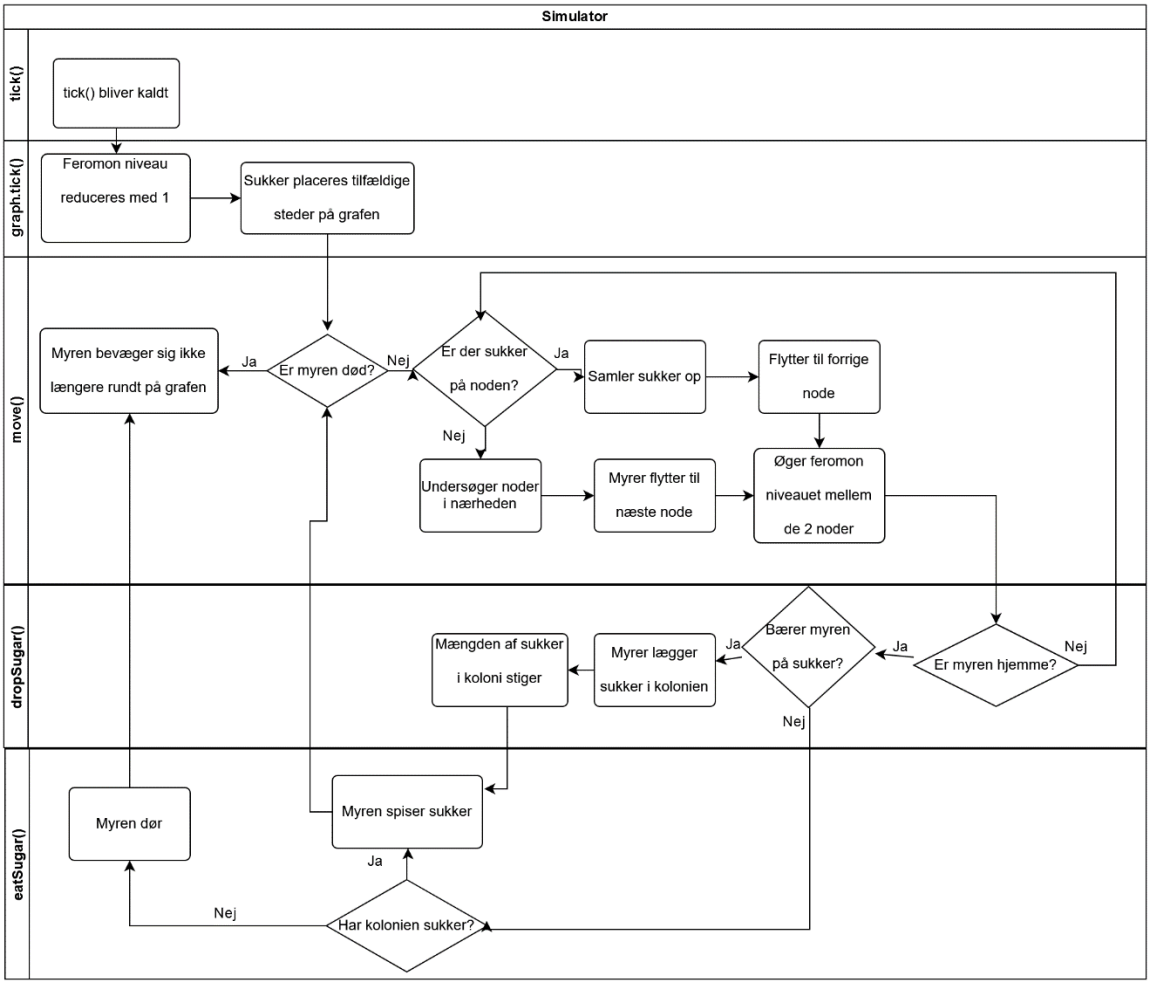

Fig 1. *Procesdiagram*

[4] For et overblik over og en beskrivelse af samtlige metoder for *Simulator*, se Bilag VI.

Processen starter, når **tick()** bliver kaldt, hvorefter simulationen opdateres med argumenter fra *Graph*. Hver handling bliver udført så mange gange, som **tick()** bliver kaldt og så længe der er levende myrer i simulationen. Simulationen er påvirket af argumenter fra brugeren. Dette ses bl.a. ved myrernes levetid: hvis der er mange myrer, lidt sukker og myrerne kan bære meget lidt sukker, så dør myrerne hurtigt. Hvis der derimod er få stærke myrer, som kan bære meget sukker, så lever myrerne længere. Hvis feromonniveauet er højt, forbliver myrerne tættere på kolonien, men hvis det er lavt, bevæger de sig længere væk fra kolonien. Størrelsen på grafen og antallet af kolonier påvirker, hvor hurtigt sukkeret indsamles, og dermed også hvor ofte, der bliver lagt nyt sukker på knuderne.

For at gøre koden letlæselig, har gruppen valgt at bruge for loops, idet længden af iterationer er kendt og statisk. Derudover har gruppen valgt at bruge flere return statements, eftersom det derved er nemmere at identificere, hvad metoden returnerer. Nedenfor ses et client/provider-inspireret metodetræ. Det illustrerer opbygningen af metoden **tick()**, som kaldes i *RunSimulation* for at få programmet til at simulere. De grå kasser betyder, at metoden IKKE er udviklet af gruppen.

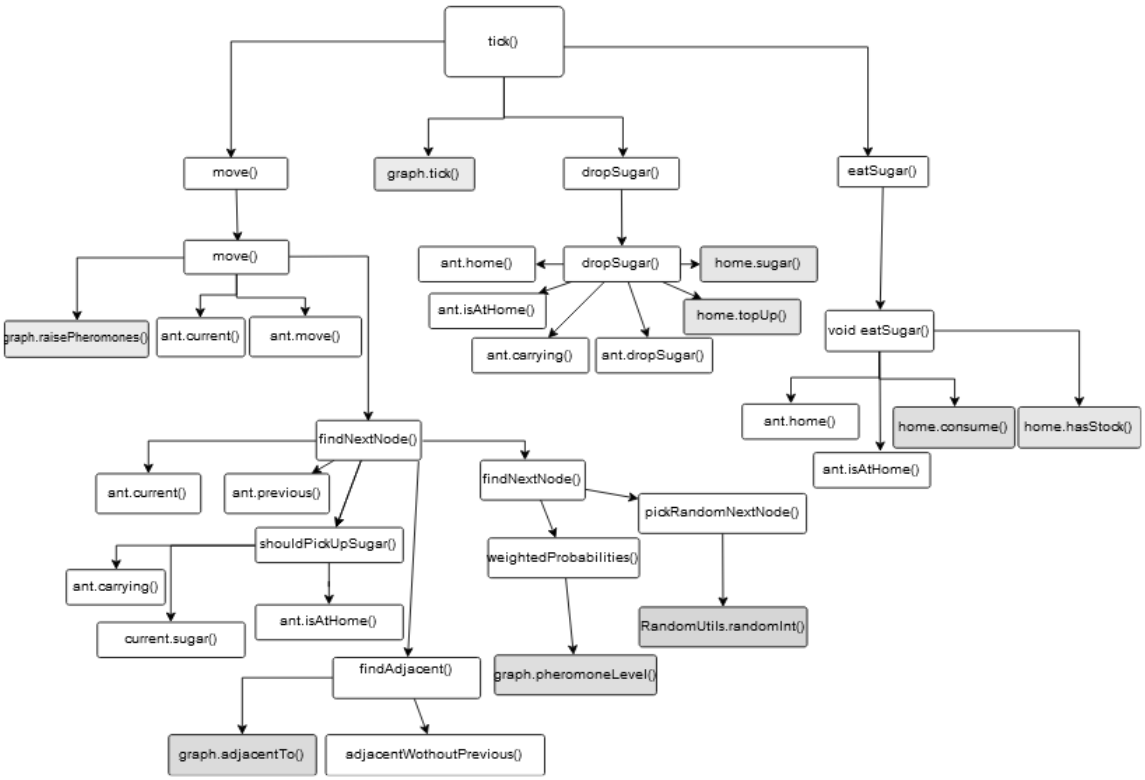

Fig 2. *Metodetræ*

Som metoder kaldes ned gennem træet, sendes informationer op til hovedmetoden **tick()**, der endelig opdaterer simulationen. Som det ses i træet, er der en række metoder, der fungerer som knudepunkter for information og eksekvering, herunder: **move()**, **dropSugar()**, **eatSugar()** og **findNextNode()**. For at reducere kompleksiteten og antallet af eksekveringer i hver metode, er princippet vedr. overloading taget i brug ved knudepunktsmetoderne. Bemærk, at klassen indeholder tre offentlige metoder. Metoderne er offentlige for at muliggøre kravet om test via dedikeret kode (jf. projektbeskrivelsen). Mere herom i testafsnittet. I tabellen nedenfor ses et overblik over de overloadede knudepunktsmetoders navne, tilgængelighed og parametrene.

Overloadede metoder								
Navn	**dropSugar()**		**move()**		**eatSugar()**		**findNextNode()**	
Tilgængelighed	Offentlig	Privat	Offentlig	Privat	Offentlig	Privat	Privat	Privat
Parameter	**Ant ant**		**Ant ant**		**int i, Ant[] ants**		**Ant ant**	**Node [] adjecent, Node current**

dropSugar(): Som det ses i tabellen, optræder metoden både som offentlig og privat. Den offentlige version skal bruge et objekt af typen myrer som argument, hvorimod den private ikke modtager argumenter. Den private itererer over listen af myrer, og kalder den offentlige version af metoden på hver enkel myre. Den offentlige version laver objektet hjem, og tjekker følgende: er myren i live, er den hjemme i kolonien og bærer den sukker. Hvis dette er sandt, øges sukkerniveauet i kolonien.

```
private void dropSugar() {
       for (Ant ant : ants) {
           dropSugar(ant);
       }
}
```

```
public void dropSugar(Ant ant) {
        if (ant != null) {
            Colony home = ant.home();
            if (ant.isAtHome() && ant.carrying()) {
                   ant.dropSugar();
                   home.topUp(home.sugar() + sugarCapacity);
            }
        }
    }
```

move(): Denne metode er også tilgængelig i en offentlig og privat version. Den offentlige skal bruge et objekt af typen myrer som argument, hvor den private ikke modtager argumenter. Den private itererer over listen af myrer: hvis en myre er i live, kaldes den offentlige version af metoden med myren som argument. Den offentlige version sætter variablerne for den nuværende og den næste knude. Derefter øges feromonniveauet mellem de to knuder, og myren flyttes til næste knude.

```
private void move() {
	for (Ant ant : ants) {
	    if (ant != null) {
		move(ant);
	    }
	}
}
```

```
    public void move(Ant ant) {
	    Node current = ant.current();
	    Node nextNode = findNextNode(ant);
	    graph.raisePheromones(current, nextNode,
droppedPheromones);
	    ant.move(nextNode);
    }
```

eatSugar(): Denne metode er atter tilgængelig i en offentlig og privat version. Den offentlige skal bruge et indeksnummer og en myreliste som argumenter, hvorimod den private ikke modtager argumenter. Den private løber gennem myrelisten, og giver den offentlige argumenterne til parametrene. Den offentlige tjekker, om hver enkelte myrer er i live, er hjemme, og om hjemmet har sukker på lager; hvis ikke, dør myren, idet dens indeks bliver sat til null. Ellers spiser myren.

```
private void eatSugar() {
	for (int i = 0; i < ants.length; i = i + 1) {
	    eatSugar(i, ants);
	}
}
```

```
public void eatSugar(int i, Ant[] ants) {
	    Ant ant = ants[i];
	    if (ant != null) {
		Colony home = ant.home();
		if (ant.isAtHome()) {
			if (!home.hasStock()) {
			    ants[i] = null;
			} else {
			    home.consume();
			}
		}
	    }
    }
```

findNextNode(): Denne metode findes i to private versioner. Den ene tager et myreobjekt som argument, mens den anden har brug for en nodeliste og den nuværende knude, som myreren står på. **findNextNode(Ant ant)** laver to knudeinstanser, tjekker om myreren skal samle sukker op, og hvis den skal det, returneres der til forrige knude. Ellers kigger den på listen over andre knuder, som returnerer resultatet for **findNextNode(Node [] adjecent, Node current)**. **findNextNode(Node [] adjecent, Node current)** tjekker nodelistens længde. Hvis der kun er ét element i listen, returnerer metoden dette. Hvis der er flere elementer, returnerer metoden en tilfældig knude baseret på en vægtet sandsynlighed, samlet antal feromoner og de nærmeste naboknuder.

```
private Node findNextNode(Ant ant) {
       Node current = ant.current();
       Node previous = ant.previous();

       if (shouldPickUpSugar(current, ant)) {
           current.decreaseSugar();
           ant.pickUpSugar();
           return previous;
       }
       Node[] adjacent = findAdjacent(current, previous);
       return findNextNode(adjacent, current);
   }
```

```
    private Node findNextNode(Node[] adjacent, Node current) {
           if (adjacent.length == 1) {
               return adjacent[0];
           }
           int[] nodeProbabilities = new int[adjacent.length];
           int totalPheromones =
weightedProbabilities(adjacent, current, nodeProbabilities);
           Node nextNode = pickRandomNextNode(totalPheromones,
nodeProbabilities, adjacent);
           return nextNode;
       }
```

4. Test

Projektgruppen har løbende udført test ved at printe værdier for kritiske data, såsom:

- Niveauet af feromoner ved start.
- Myrerne dør, når de sulter.
- Myrerne spiser sukker i kolonien.
- Feromonniveauet stiger mellem to knuder, når myren flytter sig.
- Myrerne samler sukker op.
- Myrerne bevæger sig som forventet.
- Den vægtede sandsynlighedsliste virker.

Dette er løst ved simple print-udsagn, som er bibeholdt i kildekoden. Nedenfor ses to eksempler på sådan et udsagn. Hvis noget fejler, kan man aktivere print-udsagnet ved at tage det ud af kommentarfeltet (dvs. ved at slette enten // eller /* */).

```
public void eatSugar(int i, Ant[] ants) {
    Ant ant = ants[i];
    if (ant != null) {
        Colony home = ant.home();
        if (ant.isAtHome()) {
            if (!home.hasStock()) {
                ants[i] = null;
                // System.out.println(String.format(Ant: %s died from starvation, ant));
            } else {
                home.consume();
                /*
                 * System.out.println(String.format(Ant: %s ate sugar %d left, ant,
                 * home.sugar()));
                 */
            }
        }
    }
}
```

4.1. Modultest

Da programmet så ud til at køre som forventet, udviklede gruppen modultest for at sikre, at programmet levede op til projektbeskrivelsens krav.[5] I tabellen nedenfor gives et overblik over testklassens metoder, formål og resultat:

Metode	Formål	Resultat
Main(String[] args)	Kalder de andre testmetoder i klassen.	N/A
testPheromonesRaiseOnEdge()	Testen får et givet feromonniveau mellem to knuder, en myrer flyttes mellem knuderne og testen tjekker, om feromonniveauet er forskelligt fra det forrige.	Testen kører uden fejl, hvilket betyder, at programmet opfører sig som forventet. Det nye feromonniveau er forskelligt fra the gamle.
testAdjacentDoesNotContainCurrent()	Testen tjekker om adjacentTo-listen ikke indeholder den nuværende knude.	Testen kører uden fejl, hvilket betyder, at programmet opfører sig som forventet.
testAntMoved();	Testen tjekker om en myre flytter sig fra en knude til den anden ved at se om de to hashkoder er ens.	Testen kører uden fejl, hvilket betyder, at programmet opfører sig som forventet.
testAntAte()	Testen tjekker om en myre spiser, når den er i kolonien og om der er sukker.	Testen kører uden fejl, hvilket betyder, at programmet opfører sig som forventet.
testAntStarving()	Testen tjekker om en myres indeks bliver sat til null, hvis den sulter.	Testen kører uden fejl, hvilket betyder, at programmet opfører sig som forventet.
testAntPickUpSugar()	Testen tjekker om en myre samler sukker op, når alle kriterier er opfyldt: der er sukker på knuden, myren ikke bærer på sukker og myren ikke hjemme.	Testen kører uden fejl, hvilket betyder, at programmet opfører sig som forventet.
testAntDropSugar()	Testen tjekker om en myrer afgiver sukker, hvis følgende kriterier er opfyldt: myre bærer på sukker. Den tjekker også, om sukkerindholdet stiger i kolonien.	Testen kører uden fejl, hvilket betyder, at programmet opfører sig som forventet.

Alle metoder er også testet i negative scenarier for at sikre, at testen melder fejl.

[5] Se Bilag VI for modultestens kildekode.

Efter testen havde sikret, at programmet simulerede, som det skulle, blev klasserne *Simulator*, *Ant*, *Egde* og *Node* samlet med de udleveret klasser (samt den udviklede *RunSimulation* fra første fase af projektet). Derpå blev følgende input givet:

- Sandsynlighed for sukker: **0,5**
- Gennemsnitlig sukkermængde: **5**
- Sukker, myrer kan bære: **2**
- Feromoner tabt: **2**
- Ticks: **10**
- Kolonier: **5**
- Myrer i koloni 1: **10**
- Myrer i koloni 2: **20**
- Myrer i koloni 3: **30**
- Myrer i koloni 4: **40**
- Myrer i koloni 5: **50**
- Gridstørrelse: **5*5**
- Simulationsform: **Visuel**

Nedenfor ses den eksekverede visuelle simulation (fra den udleverede *RunSimulation*-klasse):

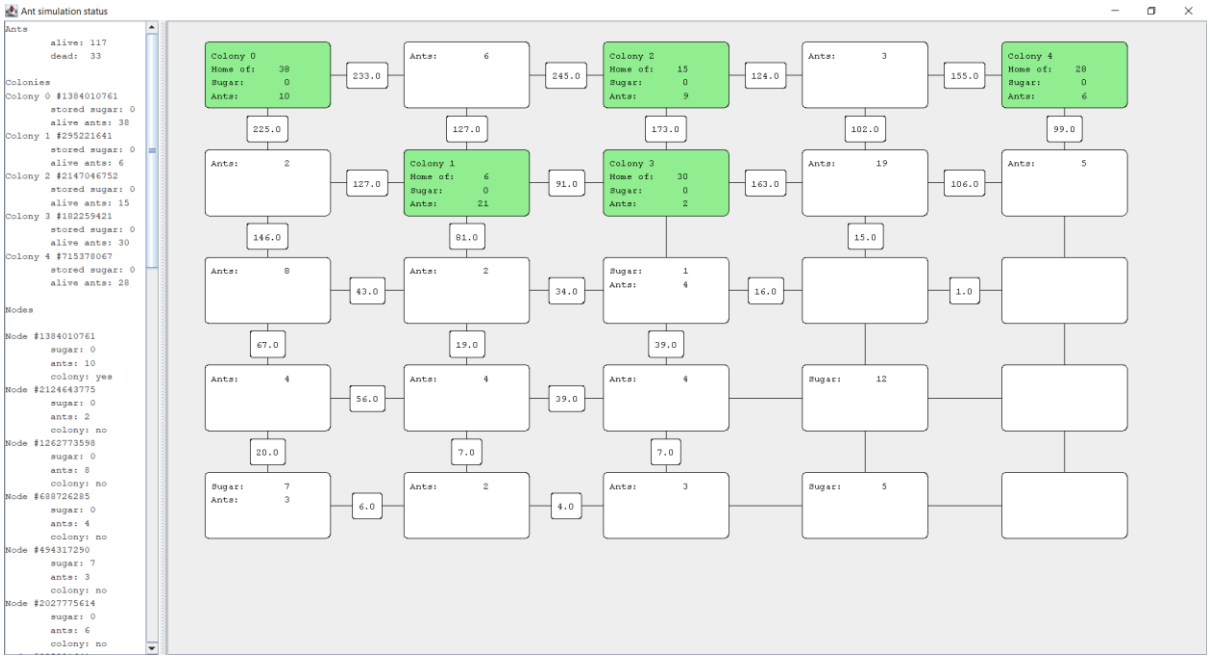

Fig 3. *Eksekveret visuel simulation*

Efter bekræftelse af, at de udviklede klasser virkede med begge *RunSimulation*-klasser, blev følgende scenarier testet.

	Visuel simulation	Læs fra fil	Tekstuel simulation
Udleveret *RunSimulation*	**Virker**	**Virker**	**Virker**
Udviklet *RunSimulation*	**Virker**	**Virker**	**Virker**

Testscenarierne bekræfter, at *Simulator* og de andre klasser kører fejlfrit, når de integreres med den udleverede såvel som udviklede *RunSimulation*.

4.2. Fejl

Under den endelige test af alle klasser, blev der fundet en "Exception in thread main java.lang.NullPointerException"-fejl ved tekstopdateringssporet. Gruppen arbejdede sig baglæns for at identificere, hvor fejlen opstod. Det undrede gruppen, at den opstod sporadisk i systemet. Nedenfor ses fejlmeddelelsen.

```
Exception in thread "main" java.lang.NullPointerException
        at Simulator.shouldPickUpSugar(Simulator.java:111)
        at Simulator.findNextNode(Simulator.java:99)
        at Simulator.move(Simulator.java:78)
        at Simulator.move(Simulator.java:39)
        at Simulator.tick(Simulator.java:25)
        at RunSimulation.main(RunSimulation.java:202)
```

Først blev en simulationen printet, hvorefter gruppen tjekkede, hvorfra myrer samlede sukker op. Det viste sig, at myrerne ind imellem samlede sukker op på ikke-eksisterende knuder. Dette ledte til metoden **adjacentWithoutPrevious()**, der afhænger af argumentet **adjacent**, som stammer fra **graph.adjacentTo()**. Det viste sig, at **graph.adjacentTo()** returnerede den samme knude flere gange i listen. Nedenfor ses systemudskriftet.

```
Colony@4554617c
Node@677327b6 -
Node@14ae5a5 - C
Colony@1540e19d
Colony@4554617c
```

Gruppen antog, at **adjacentTo()** kun returnerede hver knude én gang i listen, men det var ikke tilfældet. Da gruppen frasorterede den forrige knude (ved at reducere længden af listen med én) var den nye liste ikke fyldt op, hvis den forrige knude optrådte flere gange. Dette gav mulighed for, at en myre valgte den knude, der ikke var sat, dvs. null. Gruppen lavede en løsning i **adjacentTo()**, som sikrede, at de knuder, som blev lagt ind i listen, ikke var ækvivalente med den forrige knude.

Baseret på projektbeskrivelsens kravsspecifikationer og forudgående test, vurderer gruppen, at de udviklede klasser efterlever udviklingskontrakten. Simulationen opfører sig som forventet iht. de angivne argumenter fra brugeren. Hvis simulationen skulle gøres mere virkelighedsnær, burde det være muligt for myren også at dø uden for kolonien og ikke kun i selve kolonien.

5. Konklusion

Kravspecifikationen til koden såvel som rapporten er opfyldt. Sammen med de udleverede klassefiler, eksekveres de udviklede klasser for fase to (*Node*, *Edge*, *Ant* og *Simulator*) problemfrit i kommandolinjegrænsefladen, hvor alle relevante argumenter og informationer bliver indsamlet og fremvist. Design og implementering er forklaret, og programmets funktionalitet er testet gennem flere scenarier med dedikeret kode. Derfor kan det konkluderes, at gruppen efterlever kravspecifikationerne.[6]

Referencer

Foruden den udleverede klassedokumentation, har gruppen konsulteret lærebogen og Oracles officielle kodekonventioner efter behov:

- Oracle (1999). *Code Conventions for the Java™ Programming Language*. Lokaliseret den 16/10/2020: https://www.oracle.com/java/technologies/javase/codeconventions-contents.html.
- Savitch, W. (2019). *An Introduction to Problem Solving & Programming*. Harlow, UK: Pearson Education Limited.

[6] OBS: Formuleringerne i dette afsnit er delvist kopieret fra vores første rapport.

Bilag I (Node.java)

```java
/**
 * Instances of class Node are traversable sugar deposits. They contain either
 * none or a random amount of sugar, as defined by the user.
 */
public class Node {

    private int sugar = 0;

    /**
     * Default constructor creates a node with no sugar.
     */
    public Node() {
    }

    /**
     * Starts the instance with a set amount of sugar.
     */
    public Node(int sugar) {
        this.sugar = sugar;
    }

    /**
     * Gets the amount of sugar inside the instance.
     */
    public int sugar() {
        return sugar;
    }

    /**
     * Decreases amount of sugar inside the instance by one.
     */
    public void decreaseSugar() {
        if (sugar > 0) {
            sugar = sugar - 1;
        }
    }

    /**
     * Sets the amount of sugar inside the instance to a specified amount.
     */
    public void setSugar(int sugar) {
        this.sugar = sugar;
    }
}
```

Bilag II (Edge.java)

```java
/**
 * Instances of class Edge act as pathways between connected Node instances.
 * Each instance holds a source and a target Node, as well as the amount of
 * pheromones dropped by ants passing through the instance.
 */
public class Edge {

    /*
     * Node source and Node target are declared as final, since they do not change
     * once set.
     */
    private final Node source;
    private final Node target;
    private int pheromones = 0;

    /**
     * Constructs an instance of Edge, taking two Nodes as inputs.
     */
    public Edge(final Node source, final Node target) {
        this.source = source;
        this.target = target;
    }
```

```
    /**
     * Gets the source Node from instance.
     */
    public Node source() {
        return this.source;
    }

    /**
     * Gets the target Node from instance.
     */
    public Node target() {
        return this.target;
    }

    /**
     * Gets the amount of pheromones inside the instance.
     */
    public int pheromones() {
        return pheromones;
    }

    /**
     * Increases pheromones by a specified amount.
     */
    public void raisePheromones(int amount) {
        pheromones += amount;
    }

    /**
     * Decreases amount of pheromones by one.
     */
    public void decreasePheromones() {
        // Checks if there are pheromones before decreasing.
        if (pheromones > 0) {
            pheromones = pheromones - 1;
        }
    }
}
```

Bilag III (Ant.java)

```
/**
 * Instances of class Ant records the Ants' position on the grid by storing
 * their home Colony, as well as its previous Node instances and the one it is
 * currently in. Ants can also carry sugar.
 */
public class Ant {

    private Colony home;
    private Node current;
    private Node previous;
    private boolean carrying;

    /**
     * Ants start out with all their positional variables pointing to their home
     * Colony.
     */
    public Ant(Colony home) {
        this.home = home;
        this.current = home;
        this.previous = home;
    }

    /**
     * Instances can move between Nodes by traversing the Edge instance between
     * them.
     */
    public void move(Node location) {
        this.previous = current;
        this.current = location;
    }
```

```
    /**
     * Gets the Ant's current Node instance.
     */
    public Node current() {
        return current;
    }

    /**
     * Gets the Ant's previous Node instance.
     */
    public Node previous() {
        return previous;
    }

    /**
     * Gets the Colony instance that the Ant calls home.
     */
    public Colony home() {
        return home;
    }

    /**
     * Checks if the Ant is currently in its home Colony.
     */
    public boolean isAtHome() {
        return current == home;
    }

    /**
     * Checks if the Ant's previous location was its home Colony.
     */
    public boolean wasAtHome() {
        return previous == home;
    }

    /**
     * Returns whether or not the Ant is currently carrying sugar.
     */
    public boolean carrying() {
        return carrying;
    }

    /**
     * Sets boolean carrying to true.
     */
    public void pickUpSugar() {
        this.carrying = true;
    }

    /**
     * Sets boolean carrying to false.
     */
    public void dropSugar() {
        this.carrying = false;
    }
}
```

Bilag IV (Simulator.java)

```java
/**
 * The Simulator class handles the operations performed during ticks. From the
 * navigation and movement of Ant instances to the gathering and consumption of
 * sugar. From the instantiation of object, to the secretion and storage of
 * pheromones. It is all tracked in the Simulator class.
 */
public class Simulator {

    private Graph graph;
    private Ant[] ants;
    private int sugarCapacity;
    private int droppedPheromones;

    /**
     * Constructor creates a new simulation with the related parameters.
     */
    public Simulator(Graph graph, Ant[] ants, int sugar, int pheromones) {
        this.graph = graph;
        this.ants = ants;
        this.sugarCapacity = sugar;
        this.droppedPheromones = pheromones;
    }

    /**
     * The order of the commands for the tick method (which runs this simulation for
     * one unit of time) must be:
     *
     * 1. move(), 2. dropSugar() and 3. eatSugar().
     *
     * If eatSugar() is applied after move(), the ants die when they return to the
     * colony. If eatSugar() is applied before move(), all the ants die immediately.
     */
    public void tick() {
        graph.tick();
        move();
        dropSugar();
        eatSugar();
    }

    /**
     * Uses for-each loop to iterate through instances of Ant in ants array. Calling
     * the dropSugar method on each Ant in the ants array.
     */
    private void dropSugar() {
        for (Ant ant : ants) {
            dropSugar(ant);
        }
    }

    /**
     * Loops through Ant instances in ants array check if Ant is alive, and then
     * moves it.
     */
    private void move() {
        for (Ant ant : ants) {
            if (ant != null) {
                move(ant);
            }
        }
    }

    /**
     * Loops through ants and calls eatSugar at each index at ants.
     */
    private void eatSugar() {
        for (int i = 0; i < ants.length; i = i + 1) {
            eatSugar(i, ants);
        }
    }
```

```
/**
 * Checks if Ant instance is alive. Also checks if Ant is at home and if it's
 * carrying sugar. If so, ant.dropSugar method is called, whereby the sugar
 * level in the colony increases.
 */
public void dropSugar(Ant ant) {
    if (ant != null) {
        Colony home = ant.home();
        if (ant.isAtHome() && ant.carrying()) {
            ant.dropSugar();
            home.topUp(home.sugar() + sugarCapacity);
        }
    }
}

/**
 * Checks if Ant is alive. Sets ant's home Colony. If home has no sugar, the Ant
 * dies or is set to null. If the home colony has sugar, one unit of sugar is
 * subtracted from Colony.
 */
public void eatSugar(int i, Ant[] ants) {
    Ant ant = ants[i];
    if (ant != null) {
        Colony home = ant.home();
        if (ant.isAtHome()) {
            if (!home.hasStock()) {
                ants[i] = null;
                // System.out.println(String.format(Ant: %s died from starvation, ant));
            } else {
                home.consume();
                /*
                 * System.out.println(String.format(Ant: %s ate sugar %d left, ant,
                 * home.sugar()));
                 */
            }
        }
    }
}

/**
 * Records the location of Ant instance as well as a new target Node. Once
 * found, the pheromone level of the Edge (which the the Ant traverse) is
 * increased, and the Ant is moved to the new Node.
 */
public void move(Ant ant) {
    Node current = ant.current();
    Node nextNode = findNextNode(ant);

    /*
     * int pheromoneLevel = graph.pheromoneLevel(current, nextNode)
     * System.out.println(String.
     * format(Ant: %s, Raising pheromoneLevel from: %d to: %d, between nodes: %s <-> %s
     * , ant, pheromoneLevel, pheromoneLevel + droppedPheromones, current, nextNode)
     * );
     */
    graph.raisePheromones(current, nextNode, droppedPheromones);
    ant.move(nextNode);
    /*
     * System.out.println(String.format(Ant %s, new pheromoneLevel: %d, ant,
     * graph.pheromoneLevel(current, nextNode)));
     */
}

/**
 * This method returns a new Node for the ant to move to. It checks whether the
 * current node has sugar. If so, that sugar is picked up (if the Ant is not
 * already carrying). One unit of sugar is also subtracted from the node, and
 * the Ant moves to its previous location. If there is no sugar in the current
 * node, an array of adjacent nodes and the current node is used to find the
 * next new node.
 */
private Node findNextNode(Ant ant) {
    Node current = ant.current();
    Node previous = ant.previous();
```

```
    if (shouldPickUpSugar(current, ant)) {
        // System.out.println(String.format(Ant: %s, is picking up sugar, ant));
        current.decreaseSugar();
        ant.pickUpSugar();
        return previous;
    }
    Node[] adjacent = findAdjacent(current, previous);
    return findNextNode(adjacent, current);
}

/**
 * This method returns the decision of whether or not to pick up sugar. It
 * checks if Ant isn't carrying sugar, if the sugar is placed on the node and if
 * the ant is not in the colony. If these statements are fulfilled, the ant will
 * pick up sugar.
 */
private boolean shouldPickUpSugar(Node current, Ant ant) {
    /*
     * System.out.println(String.format(Ant: %s, is picking up sugar at: %s, ant,
     * current));
     */
    boolean decision = !ant.carrying() && current.sugar() > 0 && !ant.isAtHome();
    return decision;
}

/**
 * This method returns an array of nodes. Called by ants when trying to find a
 * new Node to travel to. If the ant only have one node to travel to, this node
 * will be returned in the array. If the ant has multiple options an array
 * without the previous node is returned.
 */
private Node[] findAdjacent(Node current, Node previous) {
    Node[] adjacent = graph.adjacentTo(current);
    if (current == previous || previous == null || adjacent.length == 1) {
        return adjacent;
    }
    return adjacentWithoutPrevious(adjacent, previous);
}

/**
 * This method returns an array of nodes without the previous node. It counts
 * all nodes in adjacent. If the node is not the previous node, it is added to
 * the new array of nodes.
 */
private Node[] adjacentWithoutPrevious(Node[] adjacent, Node previous) {
    /*
     * Find how many nodes are not a previous node. Hack to avoid faulty
     * graph.adjacentTo giving multiple of the same entry.
     */
    int newLength = 0;
    for (Node node : adjacent) {
        if (node != previous) {
            newLength = newLength + 1;
        }
    }
    Node[] newAdjacent = new Node[newLength];
    int inputIndex = 0;
    for (Node node : adjacent) {
        if (node != previous) {
            newAdjacent[inputIndex] = node;
            inputIndex = inputIndex + 1;
        }
    }
    return newAdjacent;
}
```

```
    /**
     * This method returns node to move to. Determines which node an instance of Ant
     * should go to next on its way to its intended location. Checks for dead-ends,
     * in which case the Ant is instructed to return to its previous Node. If more
     * than one Edge is connected to the current Node, the decision (about which
     * Edge instance to traverse) is calculated using the weightedProbability method
     * – and by comparing the amount of Pheromones in each connected Edge.
     */
    private Node findNextNode(Node[] adjacent, Node current) {
        if (adjacent.length == 1) {
            /*
             * System.out.println(String.format(Only way Moving %s -> %s, current,
             * adjacent[0]));
             */
            return adjacent[0];
        }
        int[] nodeProbabilities = new int[adjacent.length];
        int totalPheromones = weightedProbabilities(adjacent, current, nodeProbabilities);
        Node nextNode = pickRandomNextNode(totalPheromones, nodeProbabilities, adjacent);
        /*
         * System.out.println(String.format(Random Moving %s -> %s, current,
         * nextNode));
         */
        return nextNode;
    }

    /**
     * This method returns int of total pheromones. It loops through connected
     * Edges, pheromoneLevel and sums the pheromones.
     */
    private int weightedProbabilities(Node[] adjacent, Node current, int[] nodeProbabilities) {
        int totalPheromones = 0;
        for (int i = 0; i < nodeProbabilities.length; i = i + 1) {
            int pheromones = graph.pheromoneLevel(current, adjacent[i]);
            totalPheromones += pheromones + 1;
            nodeProbabilities[i] = totalPheromones;
        }
        return totalPheromones;
    }

    /**
     * This method returns a node, which is picked randomly based on the weighed
     * probability. If none of the other nodes is picked in the loop, the last
     * element in the array is chosen.
     */
    private Node pickRandomNextNode(int totalPheromones, int[] nodeProbabilities, Node[] adjacent) {
        int randomNumber = RandomUtils.randomInt(totalPheromones);
        for (int i = 0; i < nodeProbabilities.length - 1; i = i + 1) {
            if (randomNumber < nodeProbabilities[i]) {
                return adjacent[i];
            }
        }
        return adjacent[adjacent.length - 1];
    }
}
```

Bilag V (Test.java)

```
public class Test {

    public static void main(String[] args) {
        Test test = new Test();
        test.testPheromonesRaiseOnEdge();
        test.testAdjacentDoesNotContainCurrent();
        test.testAntMoved();
        test.testAntAte();
        test.testAntStarving();
        test.testAntPickUpSugar();
        test.testAntDropSugar();

    }
```

```
    public void testPheromonesRaiseOnEdge() {
        // Setup test data
        Colony colony = new Colony();
        Ant ant = new Ant(colony);
        Graph graph = new Graph(3, 3, new Colony[] {colony}, 0.5, 5);

        Node[] randomNodes = graph.adjacentTo(colony);
        Node firstNode = randomNodes[0];

        int currentPheromones = graph.pheromoneLevel(colony, firstNode);
        // Act
        graph.raisePheromones(colony, firstNode, 10);
        int newPheromones = graph.pheromoneLevel(colony, firstNode);

        assert newPheromones != currentPheromones;
    }

    public void testAdjacentDoesNotContainCurrent() {
        // Setup test data
        Colony colony = new Colony();
        Ant ant = new Ant(colony);
        Graph graph = new Graph(3, 3, new Colony[] {colony}, 0.5, 5);

        Node current = ant.current();
        Node[] randomNodes = graph.adjacentTo(current);
        for (Node node : randomNodes) {
            assert node != current;
        }
    }

    public void testAntMoved() {
        // Setup test data
        Colony colony = new Colony();
        Ant ant = new Ant(colony);
        Graph graph = new Graph(3, 3, new Colony[] {colony}, 0.5, 5);
        Simulator simulator = new Simulator(graph, new Ant[] {ant}, 1, 1);

        // Act
        Node currentLocation = ant.current();
        simulator.move(ant);
        Node newLocation = ant.current();

        // Assert
        checkNotEqual(currentLocation, newLocation);
        checkNotEqual(graph.pheromoneLevel(currentLocation, newLocation), 0);
    }

    private void checkNotEqual(Object a, Object b) {
        if (a.equals(b)) {
            System.err.println(String.format(Object a: %s, was unexpectedly equal to Object b: %s, a,
b));
        }
    }

    public void testAntAte() {
        // Setup test data
        Colony colony = new Colony();
        Ant ant = new Ant(colony);
        Graph graph = new Graph(3, 3, new Colony[] {colony}, 0.5, 5);
        Ant[] ants = new Ant[] {ant};
        Simulator simulator = new Simulator(graph, ants, 1, 1);
        colony.topUp(1);

        // Act
        simulator.eatSugar(0, ants);

        // Assert
        if (colony.hasStock()) {
            System.err.println( Error: Ant did not eat at colony as expected!);
        }
    }

    public void testAntStarving() {
```

```
        // Setup test data
        Colony colony = new Colony();
        Ant ant = new Ant(colony);
        Graph graph = new Graph(3, 3, new Colony[] {colony}, 0.5, 5);
        Ant[] ants = new Ant[] {ant};
        Simulator simulator = new Simulator(graph, ants, 1, 1);

        // Act
        simulator.eatSugar(0, ants);

        // Assert
        if (ants[0] != null) {
            System.err.println( Error: Ant did not die at colony as expected!);

        }
    }

    public void testAntPickUpSugar() {
        // Setup test data
        Colony colony = new Colony();
        Ant ant = new Ant(colony);
        Graph graph = new Graph(3, 3, new Colony[] {colony}, 0.5, 5);
        Ant[] ants = new Ant[] {ant};
        Simulator simulator = new Simulator(graph, ants, 1, 1);
        Node[] nodes = graph.adjacentTo(colony);
        Node node = nodes[0];
        node.setSugar(1);
        ant.move(node);

        // Act
        simulator.move(ant);

        // Assert
        if (node.sugar() != 0) {
            System.err.println( Error: Ant did not pick up sugar as expected!);
        }
        if (ant.carrying() == false) {
            System.err.println( Error: Ant did not pick up sugar as expected!);
        }
        if (ant.isAtHome() == false) {
            System.err.println( Error: Ant did not return to previous node as expected!);
        }
    }

    public void testAntDropSugar() {
        // Setup test data
        Colony colony = new Colony();
        Ant ant = new Ant(colony);
        Graph graph = new Graph(3, 3, new Colony[] {colony}, 0.5, 5);
        Ant[] ants = new Ant[] {ant};
        Simulator simulator = new Simulator(graph, ants, 1, 1);
        ant.pickUpSugar();

        // Act
        simulator.dropSugar(ant);

        // Assert
        if (ant.carrying() == true) {
            System.err.println( Error: Ant did not drop sugar in colony as expected!);
        }
        if (colony.hasStock() == false) {
            System.err.println( Error: Ant did not drop sugar in colony as expected!);
        }
    }
}
```

Bilag VI (Metodeoversigt for Simulator)

- **public Simulator(Graph graph, Ant[] ants, int sugar, int pheromones)** En konstruktør, der etablerer en instans af Simulator-klassen, med en prædefineret Graph-instans, Ant-array, sukkermængde og feromonniveau.
- **public void tick()** Eksekverer løbende (og flere gange i sekundet) et prædefineret antal ticks.
- **private void dropSugar()** Bruger et for each loop til at gennemgå instanserne i ants-arrayet. dropSugar-metoden kaldes for hver instans i arrayet.
- **private void move()** Gennemgår ants-arrayet for at rykke Ant-instanser mod deres mål, alt imens der tjekkes for instanser, som er null, hvilket indikerer, at en Ant-instans er død.
- **private void eatSugar()** Gennemgår ants-arrayet, og fjerne en sukkerenheder fra Ant-instansernes home Colony. Hvis der ikke er sukker i en Ant-instans' home Colony, bliver den instans' plads i ants-arrayet sat til null.
- **public void dropSugar(Ant ant)** Tjekker om en Ant-instans er i sin home Colony, og om den bærer på en sukkerenhed. Hvis disse kræv er opfyldt, øges Colony-instansens sukkerniveau med én.
- **private Node findNextNode(Ant ant)** Tjekker hvilken Node-instans skal være en Ant-instans' nye rejsemål.
- **private boolean shouldPickUpSugar(Node current, Ant ant)** Kaldes når en Ant-instans kommer i kontakt med sukkerenheder. Beslutningen om Ant-instansen skal samle sukkeret op træffes ved at finde ud af, om Ant-instansen allerede bærer på sukker eller ej.
- **private Node[] findAdjacent(Node current, Node previous)** Finder all Node-instanser, som er forbundet med den, som en Ant-instans står i.
- **private Node[] adjacentWithoutPrevious(Node[] adjacent, Node previous)** Finder all Node-instanser, som er forbundet med den, som en Ant-instans står i, uden at returnere den Node-instans, som den kom fra.
- **private Node findNextNode(Node[] adjacent, Node current)** Finder den næste Node-instans, og gør det til en Ant-instans' næste rejsemål.
- **private int weightedProbabilities(Node[] adjacent, Node current, int[] nodeProbabilities)** Gennemgår de Edge-instanser, der er forbundne, således instansen med det højeste feromonniveau bliver valgt som Ant-instansens rejsemål.
- **private Node pickRandomNextNode(int totalPheromones, int[] nodeProbabilities, Node[] adjacent)** Hvis to eller flere Edge-instanser har samme feromonniveau, bliver Ant-instansens rejsemål tilfældigt udvalgt.