Technical University of Denmark

Google Loon Project

B.Sc.Thesis, June 10th 2016

Kári Thrastarson, s131896

Supervisor:

Jan Madsen

Professor in the Department Mathematics and Computer Science DTU

Contents

List of Figures

Abstract

This project explores Google's endeavours to launch balloons to the stratosphere to supply rural areas with an Internet connection. A model is constructed to simulate the movement of the balloons and a framework is created for further development of the control algorithm that dictates whether a balloon should move to a new wind layer or not. Several algorithms are created and compared. The result is a JAVA tool that can be further built upon.

1 Introduction

Nowadays the Internet is no longer only a source of entertainment and leisurely activities, but an essential part of everyday operations. There should be no need to list all of the important roles this technology plays in people's everyday routine, but this list is actually getting longer every day. In light of these facts it is astounding to read the annual report from the *Internet Society* which states that in 2013 60% (Brown, 2015) of the global population was still not able to connect to the Internet.

The race has begun and many technology giants are now pursuing to hook the rest of the world up with an Internet connection for a reasonable price. But how can this be achieved? The companies each have their way of implementing this but they are all headed in the same general direction; up. They have all turned the usual way of communicating with an Internet provider upside down. As a person moved around he would connect with different radio towers, based on his current location. With the balloons this example is turned on its head. As the person stands still, he communicates with different balloons as they hover above and change location.

Whether the provider of the signal is a balloon, drone or some other device, the principal is the same. The technology has been strapped to a mobile object which moves around in the stratosphere and provides rural and remote ares with a 4G Internet connection. The aim of this project is to simulate the movement of balloons in the stratosphere and answer the question whether this is a feasible idea or not. Is the stratosphere a viable habitat for these Internet emitting balloons? The model will generalize the behaviour and simplify the universe to some extent. Reasoning for all assumptions will be provided as they are made. Rather than honing in on a perfect solution, several control algorithms are developed for the balloons during this project and they aim to explore different aspects of the problem in question. Each algorithm is explained and tested, and thoughts given on its pitfalls and how they could be improved.

The data used is not actual weather data but randomly (and realistically) generated mock data. To further enhance the accuracy of the model actual stratospheric weather data could be used as an input.

2 The loon project

The Project Loon is carried out by X (formerly Google X), a semi-secret department under the Alphabet umbrella. The project was started unofficially in 2011 and, as with all projects, was small in scope. The first prototypes were constructed with cheap material bought on the Internet and the launches included letting a balloon loose and following it with an antenna, fast car and reckless driving. The project was so secretive that none of the equipment was labeled with Google's name, but rather a sticker that read

”If found, return to Paul”, and a promise of a reward. With this promise Google hoped to retrieve all crashed balloons during these trial months.

In June 2013 the team announced that it had successfully provided some 50 families with Internet connection through balloons. This announcement served as a proof of concept and the development continued. The lifespan of each balloon was extended with better latex and manufacturing process. One notable improvement was ”fluffier” socks for the manufacturing crew. Stepping on the plastic in shoes or rough socks appeared to damage the quality of the plastic and therefor shorten the life expectancy of a balloon. The controlling of the balloons has also improved greatly with better analysis of weather data, but in addition to using NOAA’s data, the balloons now collect their own weather data.

The project has gone through several stages of improvement and the manufacturing process has been optimized so that a new balloon can be created and launched in a matter of hours. This shows that Google takes this project seriously and is on the brink of scaling significantly. They still have some obstacles to overcome before this project can be launched on a global scale, like air traffic regulations, landing pads and service centers for the balloons and such. But these obstacles pale in comparison to the ones already overcome.

3 The Model

3.1 Introduction

The loon project aims to launch thousands of balloons, each carrying heavy and expensive equipment, not to mention dangerous when falling freely. It would be foolish to attempt such a thing before making sure that the end goal is plausible, if not definitely possible. This can be done relatively cheaply using computer simulations, which is exactly the aim of this project.

The balloons are located in the stratosphere so they escape the winds and weather conditions that we experience from the troposphere. The balloons float with stratospheric currents that are more regular than the wind in the troposphere and more predictable. The layers are numerous and differ between altitude. This means that the steering of the balloons is done simply by going up or going down. The balloons can decrease or increase their altitude to catch a new wind layer and go in another direction.

Even though the world has been simplified significantly and many laws of physics overlooked, the simulation should give a good idea as to whether this idea is good or poor. Arguments are provided for all assumptions made during the modelling and their implications.

3.2 Description

The model focuses on the balloons and their movement around the stratosphere. The key components in the model are the following:

The balloons: Each balloon travels around the stratosphere and supplies Internet connection to the point on the grid it corresponds to.

A ”grid”: The earth in this model is represented by 2-dimensional arrays, or grids. There are several, equally sized, grids that contain information about the status of the system.

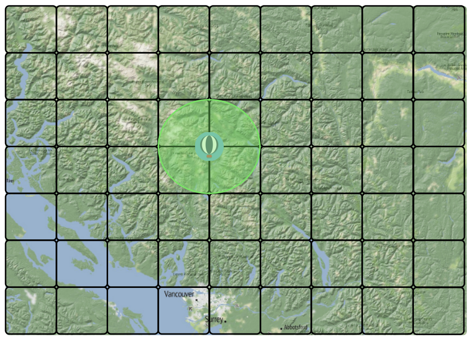

Figure 1: Grid representation.

One grid is of type Boolean and is computed based on the position of the balloons and it displays the coverage on the surface of the earth. A point in the grid with value ”True” represents a covered spot. This is calculated using the position of the balloons and the *RANGE* parameter in the model. The goal is to fill this grid with as many true values as possible. A similar grid object is used to keep track of the position of the balloons. This object is described better in the section dedicated to the implementation.

The stratosphere: The stratosphere is a collection of stratospheric wind layers which also are represented by a grid. The size of each wind layer equals the size of the earth grid.

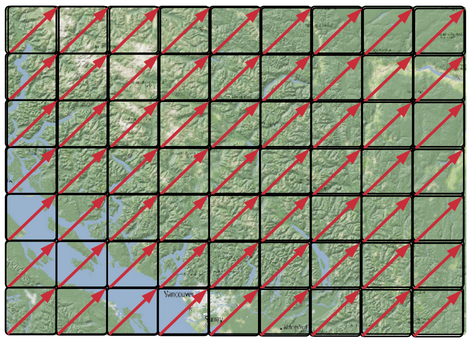

Figure 2: An example of a wind layer.

The layer object holds a collection of vectors which carry the wind strength in directions x and y at that exact point. The stratosphere is organized so each wind layer occupies a certain altitude. That way each balloon is affected by only one wind layer at a time, determined by its current altitude.

The model has some modifiable key parameters which are explained in chapter 3.4. The model creates and initializes the ecosystem, which consists of the wind layers and the grid representing the earth. The simulation then starts populating the model with balloons, one during each iteration. The model then moves each balloon according to its corresponding wind layer. The decision a balloon has to make is whether it should start moving upwards, downwards or stay put. This decision is the core of the model and will be developed in different ways and compared.

3.3 Data

The balloons will be located in the stratosphere, approximately 20 kilometers above the earth's surface (*Project Loon*, n.d.). The stratosphere is a collection of layers, each with a different jet stream. The movement of the jet streams in each layer is non-uniform and slowly changes as you move over the globe. The exact intersection between layers is not necessarily clear, but this model will present a simplified version of the stratosphere.

The stratosphere in the model is a collection of four layers, each with its own collection of wind vectors. The general direction of each is the same throughout the plane, but each vector has a deviation of about 25%, both in size and direction, to make the behaviour realistic and give the control algorithms more directions to choose from. Each of the four layers has its own general direction, pointed towards one corner of the grid. This is done to increase variety of wind directions and possibilities to choose from. The average speed of the jet streams in the stratosphere is about 45 meters per second (Roginsky, 1999), but the coverage of each balloon is a circle

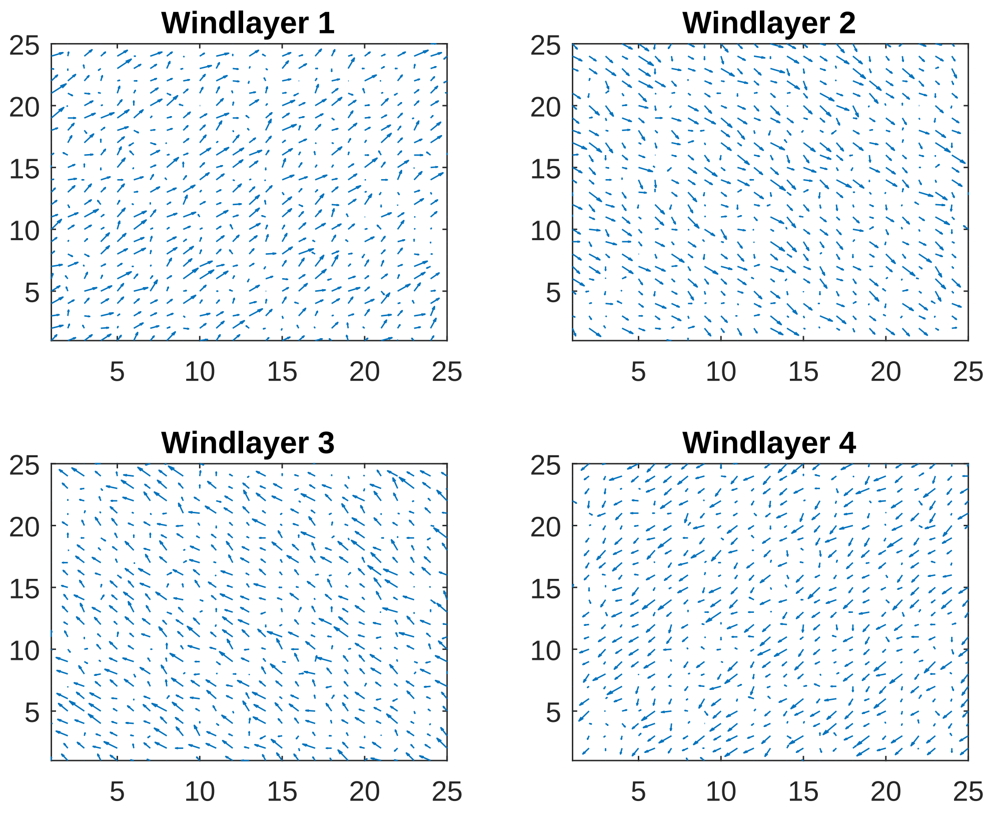

Figure 3: Wind Layers.

with diameter 80km. At this speed it takes a balloon approximately 15 minutes to escape its own circle of coverage.

The wind layer takes this into account so the average size of each wind vector reflects this relationship between the range and the speed. Figure 3 shows an example of a generated mock data set.

3.4 Simulation and Parameters

The model is the structure and relationship between all objects. It is modifiable in some ways to be able to measure and compare performance of different algorithms. The parameters that should be adjusted before simulation are the following:

WORLD_SIZE: The integer dimension of the grid that represents the earth. This number greatly affects the run time of the model as well as accuracy.

VERTICAL_SPEED: The rate of the balloon in directions up and down. This number will be added/subtracted from the balloon's altitude each step, until its vertical motion is stopped.

NUMBER_OF_STEPS: How many steps the simulation should run. This number should be as big as possible to properly model the behaviour of the system.

NUMBER_OF_CURRENTS: The number of wind layers, or wind currents, in the system. The more layers there is to choose from the more accurate direction the balloon should be able to choose.

MIN/MAX_ALTITUDE: Numbers used to divide the layers. The total distance between MIN and MAX is divided equally between all the layers in the model. This is used to determine whether a balloon has reached a new wind layer. The units are abstract but should be taken into consideration when choosing vertical speed. The vertical speed corresponds to the units on this MIN/MAX scale.

RANGE: The grid of balloons correspond to a certain surface on earth. This parameter dictates the size of the area that one balloon can impact. That is the radius of the circular area that one balloon can service.

LIFETIME: The maximum lifetime of a balloon is fixed. This is the number of steps each balloon can participate in. This number should be adjusted in accordance with the size of the grid, and the number of steps, since it would be difficult to get any results in a model where the balloons are removed just as they're about to get into position.

COMMUNICATION_RADIUS: The communication radius of a balloon is a parameter applied in the more strict algorithms. This variable controls how far the balloons can communicate between each other to share information about location and wind conditions.

When all objects have been created and configured, the simulation is run. The simulation moves all balloons according to the established stratosphere.

A simulation starts with no balloons in the system and repeats the following steps:

1. Create a balloon? Is the world saturated with balloons? The param-

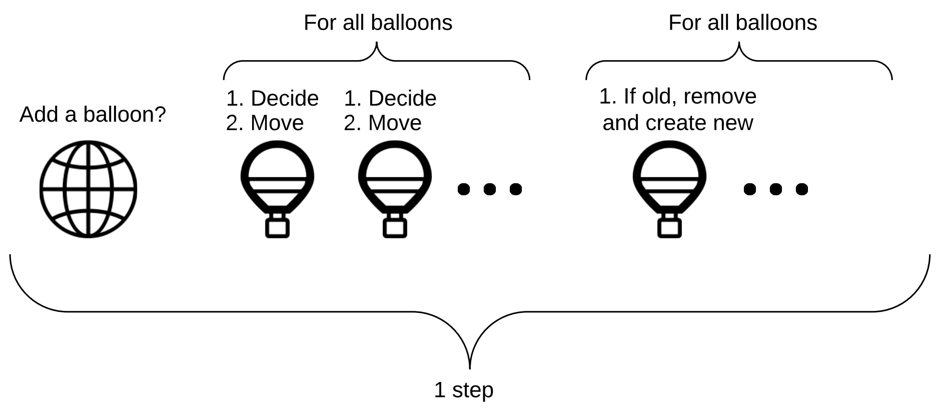

Figure 4: 1 step in the simulation.

eter *NUMBER_OF_BALLOONS* is used to determine whether a new balloon should be spawned or not.

2. Apply decisions: All balloons decide, using a control algorithm, whether they should start moving up or down or stay in the current altitude.

3. Apply currents: When the decision has been made, all the balloons are moved according to the wind layer they are currently located in. Furthermore, they are moved up or down according to the model parameter for vertical speed.

4. Update statistics: The statistics of the simulation are gathered on the fly so after each iteration the data is re-evaluated and stored for further analysis.

5. Apply age: Once all balloons have been moved, and their age in-

cremented by 1, their age is examined. The ones that have reached $LIFETIME$ are removed from the system, and new balloons created at point (0,0).

The movement and the age checking could of course be done in the same for-loop but the two procedures were separated for implementation reasons. Since the for-loop was iterating through the list of balloons, it proved unstable to remove and add to that list during that same iteration.

3.4.1 Number of balloons

The model has a parameter called NUMBER_OF_BALLOONS which dictates the maximum number of balloons allowed in the system. A fairly logical method was used to determine a reasonable value for the this parameter.

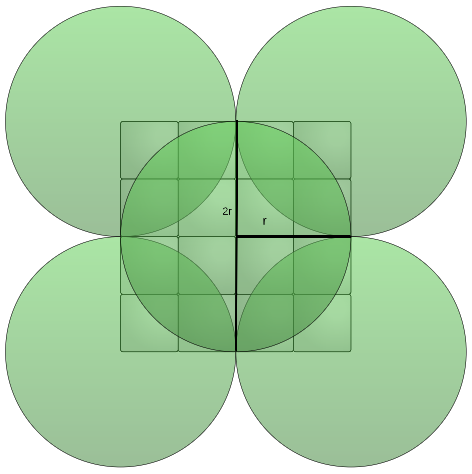

Figure 5: Flower pattern.

When arranged in a flower pattern (see Figure 5) each tile of dimension $(2r)$x$(2r)$ is covered with the help of 5 circles with radius r. One whole, and 4 quarts of circles that also cover the neighbouring tiles. This means that the model needs two circles for each $(2r)$x$(2r)$ tile in the grid. So the number of balloons is calculated as follows:

$$n = 2 \cdot \frac{w^2}{(2r)^2}$$
$$n = \frac{1}{2} \cdot \frac{w^2}{r^2}$$

where

$$\begin{aligned} w &= \text{WORLD\_SIZE} \\ r &= \text{RANGE} \\ n &= \text{NUMBER\_OF\_BALLOONS} \end{aligned} \tag{1}$$

The flower pattern is a safe method to cover a square area with circles, albeit not the most efficient one. Overlaps are unavoidable and since the equivalent of two circles are allocated to one tile, the ratio between the coverage is

$$\begin{aligned}&\frac{\text{Surface of circles}}{\text{Surface of square}}\\ =&\frac{2\cdot r^2\cdot\pi}{(2r)^2}\\ =&\frac{\pi}{2}\end{aligned}$$

i.e roughly 57% redundancy.

3.5 Assumptions

The assumption is made that the grid is fine grained and corresponds to a surface on earth. Each point in the grid has a Boolean and

$grid[x][y] == true$ for $x, y \in [0;\text{WORLD\_SIZE}]$

would represent a surface with lossless internet connection on every point. Each balloon occupies exactly one point in the grid but supplies coverage to the neighbouring area. The size of that area is controlled by the parameter RANGE. This is better displayed in Figure 6.

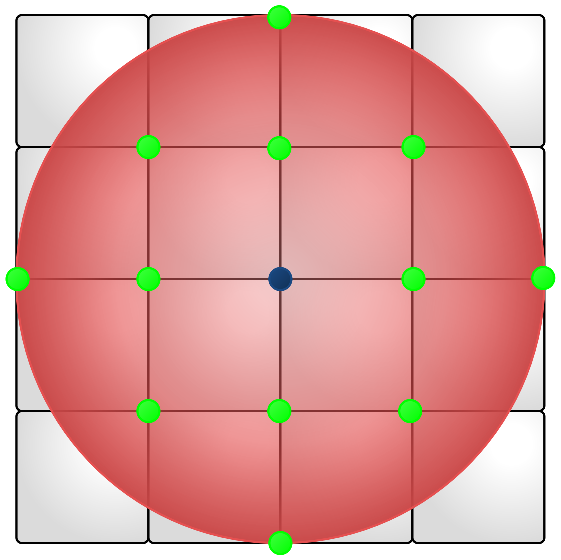

Figure 6: Coverage in a 5x5 grid with a balloon at (2,2) and RANGE=2.

The movement of a balloon in a grid is similar to the old Nintendo games.

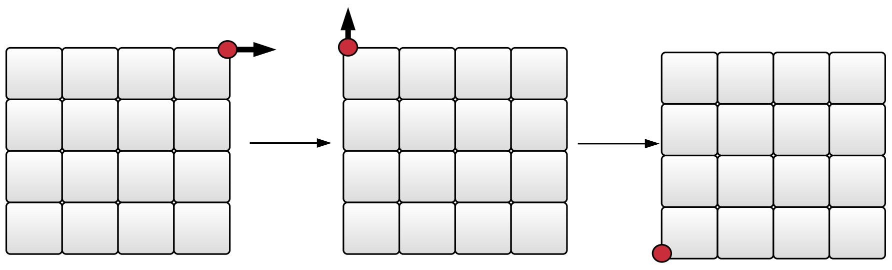

Figure 7: Border Behaviour.

When a balloon exits the grid it appears on the opposite site. For the same reason as New Zealand is typically both on the far right and left side of a map, this is done to resemble the behaviour of a spherical object when flattened out.

In the model the balloons are a point in a grid, without a mass or volume. Also, the balloons do not move to a different spot in a continuous motion but are teleported between places. Removed from point A, and spawned at a point B. Multiple balloons are allowed to occupy the same space without any conflicts or implications. This is a simplification of the real life problem, but does not have to be included in the model as the odds of a crash are trivial when the size of the space the balloons occupy are taken into consideration.

The maximum lifetime of a balloon is a fixed number in this model, which is an unlikely event in the real life problem. The balloons can however still not hover forever so this model parameter was necessary. The balloon is removed from the grid at the point where it was located when it reached is maximum age. This is convenient when modelling, but maybe not in the real life problem, since the balloon could be hovering over the sea, or a city, when it reaches its limit. The assumption is made that the balloon can be safely landed at every point in the grid.

All new balloons are launched from the same place, at point (0,0) in the grid. More launching places could easily be implemented by modifying the createBalloon() function or by using the override function createBalloon(int x, int y) which lets the user choose a specific location to which the new instance of the Balloon class should be created.

The weather data and the vertical movement of the balloons has been simplified greatly. The mock data has been created really conveniently to give the balloons as many directions to choose from as possible. This is not always the case in the stratosphere. Furthermore, the division between different wind layers is not as clear as portrayed in this model. The data is however realistic enough to give weight to the model and display a lifelike behaviour of the balloons to some extent. The balloons move between two points in every step of the simulation, whether it is with one layer or another, and that behaviour is sufficient at this stage.

The assumption is made that the coverage on earth does not depend on the altitude of a balloon. This is a simplification since the signal strength from a balloon differs from altitude to altitude. This is overlooked in this model but can easily be implemented in an updated version, since this is only a matter of calculations and usage of units and measurements that already are available in this model.

3.6 Measurements and output

The collection of data in the model is twofold. First there is static data which is collected before and after the simulation. Those measurements include the parameters of the model as well as:

droppedConnections: The number of times a balloon leaves a spot and no other balloons takes its place. That would mean that the connection would drop in that area until another balloon would fly over.

accumulatedCoverage: The accumulated coverage of the simulation divided by the number of steps. This gives a concrete value for the quality of the control algorithm.

runtime: The time it took to run the simulation.

These three results, as well as the values for the parameters, are appended to a file called *simulation_results.txt* which is a log file for the model.

The current coverage at step t is output to a text file which is named after the algorithm used in that simulation, for example *simulation_coverage_alg1.txt*. This file is overwritten every simulation.

4 Design

4.1 The Model

This model was designed following the *General Responsibility Assignment Software Patterns* (GRASP). The GRASP design principles provide guidelines for object-oriented software design and are well suited for a project such as this. The main focus of the principles, like the name states, is assigning responsibilities to the appropriate objects in the model. The basic roles in GRASP can be divided into *knowing* and *doing*. The design of the loon model is simple so the division was straight forward:

BALLOON: Knows: A balloon object knows its own location, altitude and in which wind layer it is currently located.

Does: A balloon object can move with the wind. It gets the wind vectors from the wind layer in which it is located.

WIND LAYER: Knows: A wind layer object knows its own identification number, as well as the wind vector at any given place in the grid.

Does: A wind layer does nothing but provide access to its data.

WORLD: Knows: The world know about the status of the simulation and how many steps have been taken. It collects statistical data about the grid in which the balloons hover.

Does: The world has access to all balloons and wind layers and is the controller of the simulation. The world initiates all decisions and movements of every balloon.

The *Balloon* and *WindLayer* are expert classes while *World* is a controller class in this model.

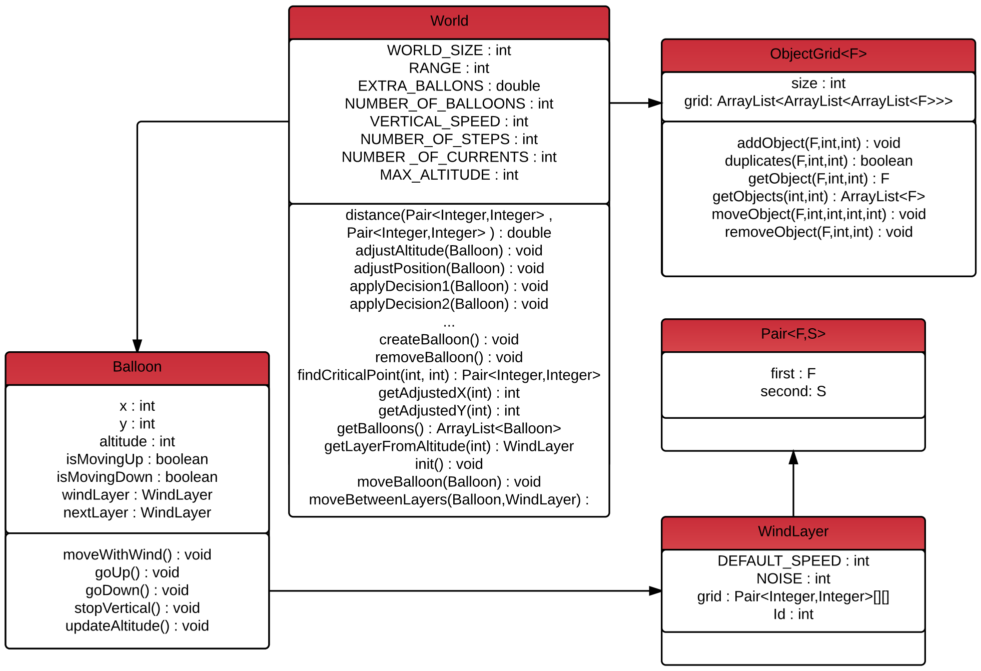
World
WORLD_SIZE : int
RANGE : int
EXTRA_BALLONS : double
NUMBER_OF_BALLOONS : int
VERTICAL_SPEED : int
NUMBER_OF_STEPS : int
NUMBER _OF_CURRENTS : int
MAX_ALTITUDE : int
distance(Pair<Integer,Integer> , Pair<Integer,Integer>) : double
adjustAltitude(Balloon) : void
adjustPosition(Balloon) : void
applyDecision1(Balloon) : void
applyDecision2(Balloon) : void
...
createBalloon() : void
removeBalloon() : void
findCriticalPoint(int, int) : Pair<Integer,Integer>
getAdjustedX(int) : int
getAdjustedY(int) : int
getBalloons() : ArrayList<Balloon>
getLayerFromAltitude(int) : WindLayer
init() : void
moveBalloon(Balloon) : void
moveBetweenLayers(Balloon,WindLayer) :
ObjectGrid<F>
size : int
grid: ArrayList<ArrayList<ArrayList<F>>>
addObject(F,int,int) : void
duplicates(F,int,int) : boolean
getObject(F,int,int) : F
getObjects(int,int) : ArrayList<F>
moveObject(F,int,int,int,int) : void
removeObject(F,int,int) : void
Pair<F,S>
first : F
second: S
Balloon
x : int
y : int
altitude : int
isMovingUp : boolean
isMovingDown : boolean
windLayer : WindLayer
nextLayer : WindLayer
moveWithWind() : void
goUp() : void
goDown() : void
stopVertical() : void
updateAltitude() : void
WindLayer
DEFAULT_SPEED : int
NOISE : int
grid : Pair<Integer,Integer>[][]
Id : int

Figure 8: Class Diagram.

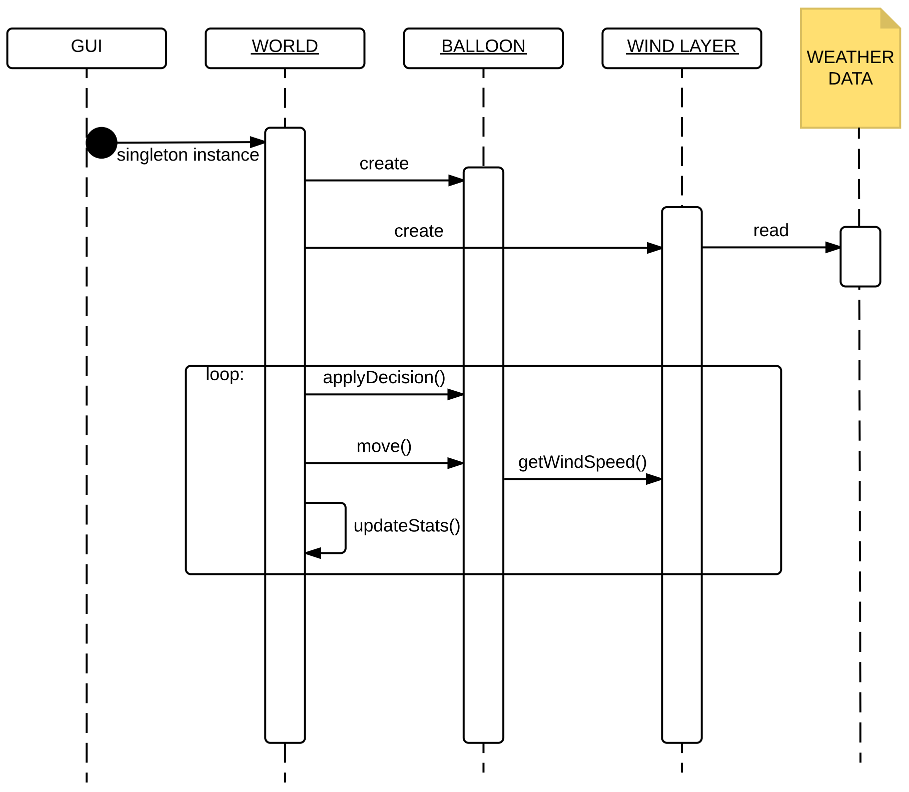

Figure 9: Simplified Sequence Diagram.

4.2 GUI

Two user interfaces were developed for the model. The first one, named *TextGUI*, a pure textual output displaying all the statistical data for the simulation as well as generating text files with the sequence of coverage data during the run. This was used comprehensively to analyze and compare the performance of different algorithms.

The second one, named *MainWindow*, provided graphical representation of the balloons at each step. The design was very rudimentary and was developed purely for debugging purposes at early stages of the project. Seeing the initial movements of the heard of balloons gives a clear image of the behaviour of the model and whether it is working properly or not. The simulation window gives the user an opportunity to take a single step, or up to 200 steps at at time, using the slider object.

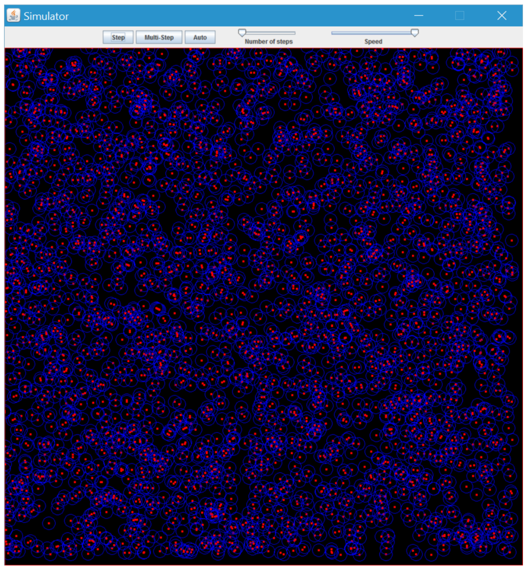

Figure 10: Main Window - red dots are balloons, and the blue halo is the coverage.

The speed of the simulation can be controlled with another slider object. This second interface is not bound to the model parameter NUMBER_OF_STEPS so it will run until stopped. This is useful to see how an algorithm works during long simulations. In this project a visual representation is often the fastest measure of quality of an algorithm.

5 Implementation

The programming language of choice for this project was JAVA. That was well suitable for an object oriented problem such as this. Eclipse was used as an IDE not only because of familiarity, but also due to all the handy refactoring tools that come built in to the environment. This is helpful when the code starts to look complicated, to be easily able to extract lines of code and store it within a private helper function. The toolkits SWING and AWT were used for the development of the GUI.

The project was stored on GitHub[1] and is accessible to everyone. The process was iterative where a running version of the model was ready early, and then built upon for the duration of the period.

5.1 Balloon

The *Balloon* class is an expert for the balloon. The following attributes are used to describe the object:

int x : X coordinates in the plane.

[1]www.github.com/karithrastarson

int y : Y coordinates in the plane.

int altitude : The altitude is used to determine wind layer.

int age : The number of steps this balloon object has taken.

boolean isMovingUp : Used to determine whether this balloon is currently moving up.

boolean isMovingDown : Used to determine whether this balloon is currently moving down.

WindLayer windLayer : The wind layer in which this balloon is currently located.

WindLayer nextLayer : Used to determine where this balloon is headed.

The balloon's main function and responsibility to the model is to move with the wind. This is done with the public function *moveWithWind()*. The balloon retrieves a wind vector from its current wind layer and adds the x and y components of that vector to its own current location in the grid. The new coordinates are then adjusted by the model, according to the border behaviour as described earlier. The adjustment is made as follows:

```
if new x $\geq$ WORLD_SIZE then
    set x = x-WORLD_SIZE
end
if new x ¡ 0 then
    set x = x+WORLD_SIZE
end
repeat for Y.
```

The rest of the functions in the *Balloon* class are classic getters and setters, as well as a *toString()* and *equals()* functions.

5.2 WindLayer

The *WindLayer* class is an expert of weather data. Its main attribute is a grid of size equal to the *WORLD_SIZE* global variable. The default constructor of the class takes three variables: an ID, the size of the world in which it will be and the default size of wind vectors. From that information the constructor takes care of generating mock data for the wind layer. One additional attribute is computed from the default size of wind vectors, but that is the *NOISE* variable. This is used in the generation of mock data, to determine how much each vector should be able to deviate from another. This variable is set to 25% of the default wind vector.

Algorithm .1 displays how the mock data is generated. This model uses 4 layers and the constructor uses the ID to determine the general direction. The default speed variable is set to one of the four corners of the plane, based on the ID. The size of this vector will variate from zero to *DEFAULT_SPEED*. Then a random number is generated on the scale from zero to *NOISE*. Then a random boolean variable determines whether that random number should be added or subtracted from the x direction of the wind vector. Same routine is performed for the y direction.

Algorithm .1 Pseudocode for generating mock data

```
for i from 0 to WORLD_SIZE do
    for j from 0 to WORLD_SIZE do
        ds = random number from 0 to DEFAULT_SPEED

        if ld=0 then
            x=ds and y=ds
        end
        if ld=1 then
            x=ds and y=-ds
        end
        if ld=2 then
            x=-ds and y=ds
        end
        if ld=3 then
            -ds and y=-ds
        end
        div = random number from 0 to NOISE
        signX = random boolean
        if signX then
            x + div
        end
        else
            x - div
        end
        div = random number from 0 to NOISE
        signY = random boolean
        if signY then
            y + div
        end
        else
            y - div
        end
        wind[i][j] = wind vector(x,y)
    end
end
```

5.3 Pair

This class was created when a short surf on the Internet showed that JAVA was not equipped with a convenient generic container class that would store pairs, or tuples. It was easier to implement one from scratch. This class is really simple: it carries two variables of generic types T and R. Then the class is equipped with getters and setters.

This class was created to contain the wind vectors. Each wind layer has a grid of Pairs so a Pair at coordinates x and y in the grid tells the wind speed and direction at that particular point.

5.4 ObjectGrid

No Java library had a suitable container for this project. The object that this project demanded was a two dimensional grid of a fixed size that could store zero or more objects in each cell. A generic class was implemented, but it was then utilized with *Balloon* objects in this particular project. This makes it easier to access a particular balloon object based on its coordinates.

The main container of the class looks complex but is in fact really simple. The class has the following attributes:

```
private ArrayList<ArrayList<ArrayList<F>>> grid;
```

```
    private int size;
```

The outer two array lists are initialized in the constructor, and a grid of empty array lists created. This grid has the dimensions size x size. Each cell in the grid holds an array list of a generic type that can be populated and modified easily. To find the number of balloons occupying a certain spot, the size of the array list at the corresponding point in the grid is computed and returned.

5.5 World

The *World* class is the controller of the model and contains all the balloons, wind layers and the control algorithms. The class contains a lot of attributes who's sole purpose is debugging and printing to to create graphs. Those attributes will be left out of this documentation.

The *World* class has the attributes listed in section 3.4. The class has two main containers that make up the model:

```
private ArrayList<WindLayer> stratosphere;
private ArrayList<Balloon> balloons;
```

In addition, the class has two containers that support the decision making and the statistical overview of the simulation:

```
private ObjectGrid<Balloon> balloons_grid;
```

```java
private boolean[][] coverage;
```

The ArrayList called *stratosphere* holds all wind layers of the model in an order. The ArrayList called *balloons* is a collection of all balloons created by the model. The boolean grid called *coverage* is updated in every step of the simulation with boolean values; true for covered and false for not covered. The container of type *ObjectGrid* called *balloons_grid* keeps track of the position of all balloons in the system.

Algorithm .2 Update Coverage

```
Clear the coverage grid.
forall Balloon b in balloons do
    center = Pair(b.getX(), b.getY())
    for int i = b.getX() - RANGE; i < b.getX() + RANGE; i++ do
        for int j = b.getY() - RANGE; j < b.getY() + RANGE; j++ do
            point = Pair(i,j)

            coverage[i][j] = inCircle(center, point)
        end
    end
end
```

The *World* class has a private function called *updateCoverage()* which is in charge of updating the boolean grid. This is described with Algorithm .2. This function iterates through all the balloon in the system and inspects a rectangular area around the center of the balloon. This corresponds to the rectangle displayed in Figure 6. Each point in this area is passed through a function called inCircle, which applies the mathematical function for a circle to determine whether the point is within the coverage range or not. Either this points is within range or not. The sum of of all connected points is calculated and compared against the sum of the previous step. The difference in covered cells is interpreted as a dropped connection. If the connected cells er fewer than they were in the round before, that means that somebody has lost his Internet connection.

$$(x-a)^2 + (y-b)^2 = r^2.$$

where

$x =$ x-coordinate

$y =$ y-coordinate

$a =$ x-coordinate of the center point

$b =$ y-coordinate of the center point

$r =$ radius (Range)

The constructor is short and only initializes the containers mentioned above without populating them. Then a more elaborate initialization function (called *init(String str)*) can be called with an input string, indicating which algorithm to use for the simulation. The *init* function creates four wind layers and adds them to the stratosphere. Furthermore it outputs the wind layers to text files so they can be plotted and analyzed. The outcome is displayed in figure 3.

One by one the balloons are created and placed at point (0,0) in the grid. In each step of the simulation only one (or zero if the system is saturated) balloon is added while the rest of the balloons are moved. This will prevent the cell (0,0) from clogging on the first step. The simluation is a repetition of the private function *step()* and repeated as many times as the parameter of the model says. Each step is an execution of the following:

```
if No. of balloons in system < NUMBER_OF_BALLOONS then
    createBalloon()
end
forall balloons in system do
    applyDecision() moveBalloon()
end
forall balloons in system do
    if Age of balloon $\geq$ LIFETIME then
        removeBalloon()
    end
end
updateStatistics()
```

The *applyDecision()* function is the actual control algorithm to which another section of this paper is dedicated. The *moveBalloon()* function is the thoughtful procedure of moving the balloon in the model and updating the appropriate containers. First the altitude is adjusted according to the vertical speed if the balloon is marked as moving up or down. Then a check is

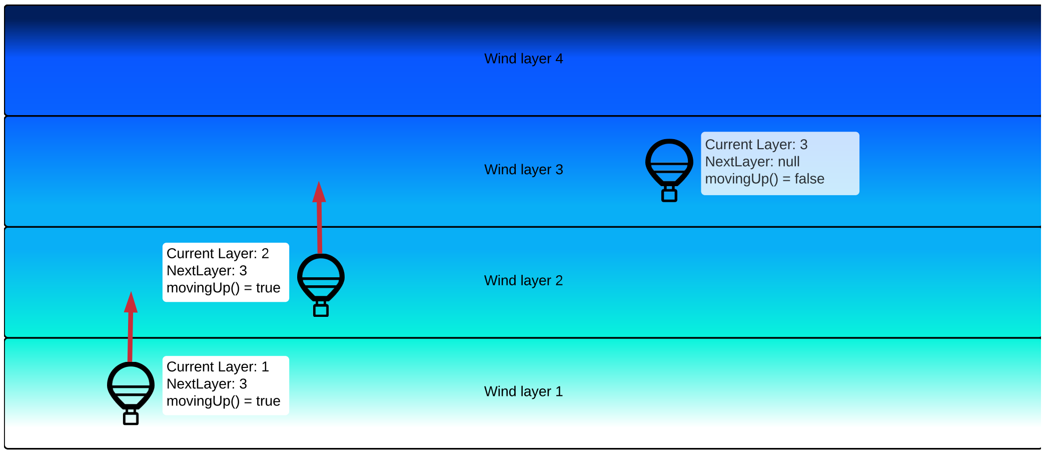

Figure 11: The wind layer attributes of a balloon while it moves upwards

performed to see if the change in altitude affected the wind layer in which the balloon is. This is done with a function called *getLayerFromAltitude* in which each wind layer is assigned a certain space between the global variables that indicate the maximum and minimum altitude. In a simulation, for example, where MIN_ALTITUDE = 0 and MAX_ALTITUDE=400, the function would return wind layer 1 for altitudes from 0-99, wind layer 2 for altitudes 100-199 etc.

The *windLayer* attribute of the balloon is then updated and compared to the *nextLayer* attribute to see if the vertical movement of the balloon should be stopped or not. If the balloon's current layer equals its next layer variable, then the vertical movement is stopped and the next layer variable set to *null*.

When the balloon has been placed in the correct wind layer it can be moved around in the grid of balloons with the corresponding wind vector. The previous location as well as the new location of the balloons are updated in the grid to represent the actual movement.

6 Control Algorithms

6.1 Simulations

The following values were used for the simulations unless otherwise stated:

World Size	500
Lifetime	200 or LONG
Number of Balloons	313 (computed)
Range	20
Communication Radius	20
Vertical Speed	10
Number of Steps	2000
Number of Currents	4
Max Altitude	400
Min Altitude	0

Table 1: Simulation Parameters

6.2 Algorithm 1

The first algorithm is straightforward and rudimentary. Like the description of the model stated the simulation first checks if it can add balloons to the system. After doing so each balloon in the system uses the following logic to determine whether it should move or not:

Algorithm .3 Control Algorithm 1

```
if Multiple balloons same spot then
    if At bottom layer then
        Go up
    end
    else if At top layer then
        Go Down
    end
    else
        Choose direction at random
    end
end
else
    Stay in current layer
end
```

This algorithm has an obvious downside since it contains an element of random, which is usually not helpful in a *control* algorithm. Nevertheless, it spreads the balloons thoroughly around the grid relatively fast. The control over the grid is however limited and this would not be a suitable solution if the end goal is to provide stable coverage.

This algorithm might however be utilized as a part of a larger more complex algorithm. At the beginning stages of a balloon's life cycle it could prove helpful to get the balloon out of that first wind layer and join the huddle.

6.3 Algorithm 2

This algorithm is a step towards a more complex control algorithm. Some minor logic replaces the random generator in the previous solution.

Algorithm .4 Control Algorithm 2

```
if Multiple balloons same spot then
    Calculate projections
    if $optionUp < optionDown \; optionUp < currentSpace$ then
        Go up
    end
    else if $optionDown < currentSpace$ then
        Go Down
    end
end
else
    Stay in current layer
end
```

The balloon fetches the wind vectors from neighbouring wind layers and uses them to weigh its options. It computes the number of balloons in three different points:

1. The point to which the balloon's current wind layer will move it.
2. The point to which the the wind layer above the balloon's current wind layer would move it.
3. The point to which the the wind layer below the balloon's current wind layer would move it.

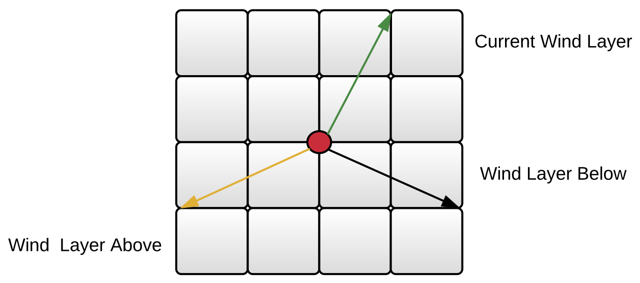

Figure 12: Calculate projections

The algorithm then determines which option has the cell with the lowest number of balloons and chooses that direction.

A1 does a much better job than A2 like Figure 13 shows clearly. A1 has the advantage that it is free, and the spreading of balloons happens quickly and spontaneously. A2 is more boring in its movement and only moves to a new layer if the numbers add up.

It is interesting to see that the system reaches a certain state where A2's graph resembles a dead man's heart monitor. The flat line indicates that all balloons have reached a point in space where they deem it not feasible to move. The projection points that they compute indicate that they are located in the optimal spot. The graphs clearly show that they are wrong in that decision.

In the second simulation, every 200 steps a balloon is removed, and recreated at coordinates (0,0). Like mentioned earlier, A1 does an excel-

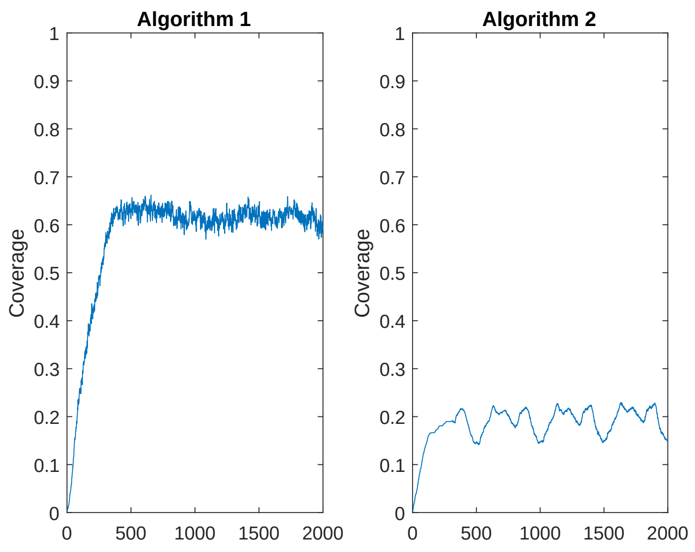

Figure 13: Algorithm 1 vs. Algorithm 2 in 1000 steps with an infinite lifetime of each balloon

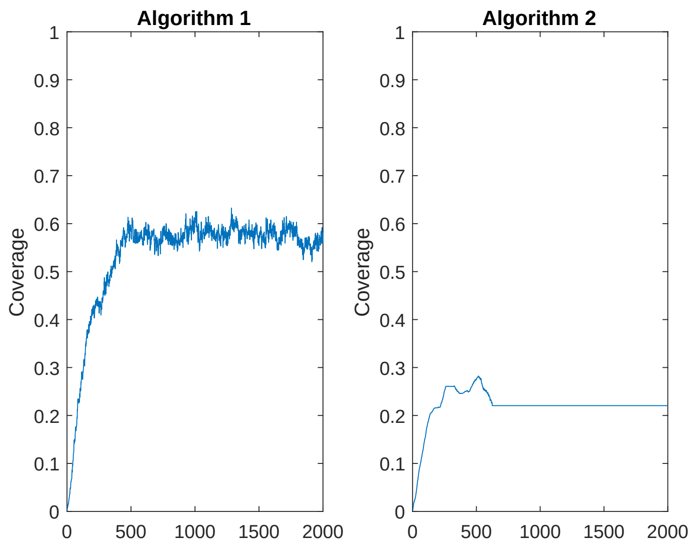

Figure 14: Algorithm 1 vs. Algorithm 2 in 1000 steps with lifetime 200

lent job of spreading the balloons quickly and thoroughly so this lifetime heavily favours that algorithm. A2 is never able to spread the balloons well enough before it has to start all over again.

6.4 Algorithm 3

The third algorithm is a modification of Algorithm 2 and introduces a new feature to the balloon. A2 based its decision on the number of balloons in the neighbouring cells, and which wind layer would blow it to the most vacant cell.

A3 communicates with the balloons within its communication radius and makes a decision based on the data it receives. This is done through a private function. The function creates a list of all balloons within the communication radius. Then each of the balloons in that list counts it own neighbours and returns the value. The point on which the balloon with the fewest number of neighbours is located is deemed as a critical point.

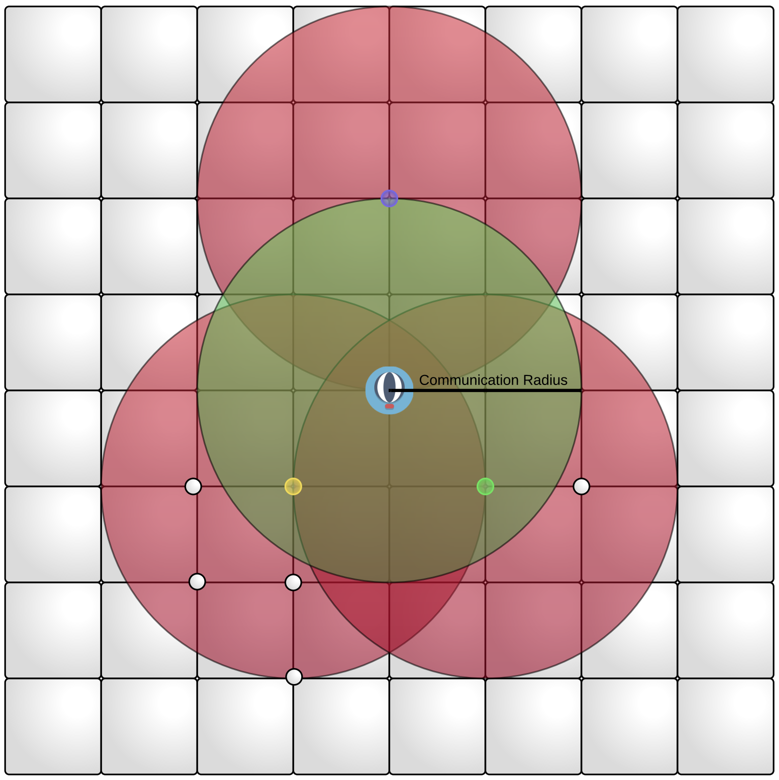

Figure 15: Critical point computed. The purple point will be deemed critical

Figure 15 shows an example of calculations of a critical point. The main balloon has three neighbours represented with different colours. Each of the three neighbours has its own number of neighbours. The yellow, green and purple balloons each have 6, 3 and 1 neighbours respectively. This means that the coordinates of the purple balloon are returned as a critical point.

A3 is also the first algorithm to explore more than just the wind layer above and below, but the entire stratosphere. To be able to navigate towards that layer, a *nextLayer* attribute has been introduced to the balloon object. When a balloon is moved between layers, this variable is used to determine if the vertical movement should be stopped or not, i.e if the balloon has reached is destined layer or not.

Algorithm .5 Control Algorithm 3

```
if More than 1 balloon at current space then
    Find point c (critical point)

    Calculate projectedPair (pp) for current layer

    smallestPP = distance(pp, c)
    forall WindLayer wl : stratosphere do
        Calculate projectedPair (pp)
        Calculate distance(pp,c)
        if pp<smallestPP then
            wl is chosen as NextLayer
        end
    end
    NextLayer set as attribute of balloon
    if NextLayer is above CurrentLayer then
        goUp()
    end
    else
        goDown()
    end
end
else
    Stay in current layer
end
```

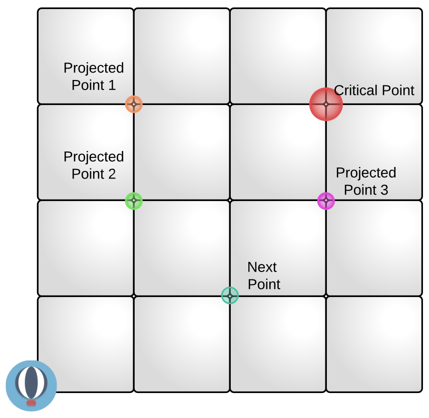

Figure 16: The distance between projected points and the critical point is measured.

When a critical point has been determined, a projection point is created for all wind layers in the stratosphere. A projection point is a hypothetical point that shows where the balloon would go if it were located in another wind layer at this very moment. This is represented in figure 16. The distance d between each projection point (x_p, y_p) and the critical point (x_c, y_c) is calculated with the distance formula:

$$d = \sqrt{(x_c - x_p)^2 + (y_c - y_p)^2}$$

6.5 Algorithm 3s

Algorithm 3 was the first algorithm to offer a more long term vision, where a balloon headed not only for the adjacent wind layer, but maybe rather the one after that. The transition does not happen instantaneously since a balloon has a fixed vertical speed. This means that a balloon can be interrupted in its transition, and the destination layer changed before the balloon reaches its target. Algorithm 3s is a minor twist on A3 as it only adds one criteria. No decision is made by balloons that are already in motion:

Algorithm .6 Control Algorithm 3s

```
forall Ballon b : balloons do
    if b is not moving vertically then
        applyDecision3(b)
    end
end
```

Figures 17 and 18 show the performance of the algorithms, first with an infinite lifetime of all balloons. The first thing you notice is that A3s obviously outperforms A3 in first simulation, where the lifetime is infinite. A little slower to begin with, but achieves higher coverage in the end. The statistics worth noting for this simulation are:

Statistic	**Algorithm 3**	**Algorithm 3s**
Runtime:	113s	14,4s
Coverage over simulation:	49,1%	53,5 %
Dropped Connections:	1.489.340	1.566.077

Table 2: Algorithm 3 vs. 3s with infinite lifetime

A3 has however a slight upper hand when it comes to dropped connections. That is understandable since every balloon makes a calculated

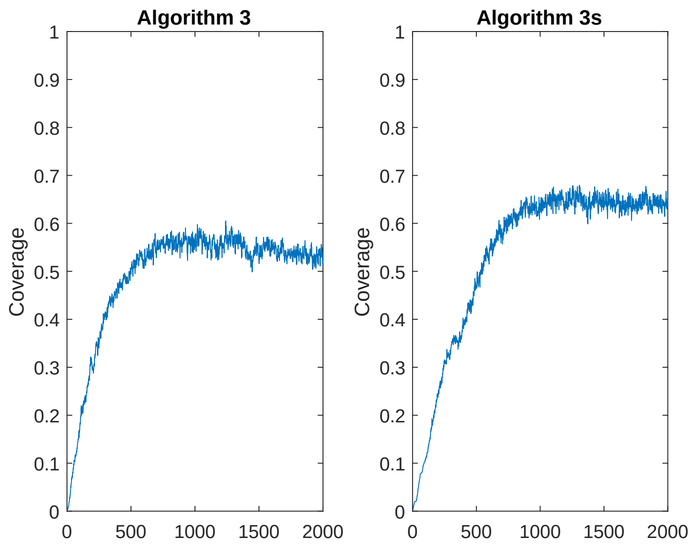

Figure 17: Algorithm 3 vs. Algorithm 3s in 1000 steps with an infinite lifetime of each balloon

decision in every step while a balloon in A3s disregards all decisions while moving vertically.

Statistic	**Algorithm 3**	**Algorithm 3s**
Runtime:	73,1s	15,4s
Coverage over simulation:	39,0%	38,5 %
Dropped Connections:	1.168.119	1.194.726

Table 3: Algorithm 3 vs. 3s with lifetime 200

When the lifetime is 200 the performance of the algorithms are similar. During the first replacing phase, after the first 200 steps, A3s takes more of a bump. Like Table 3 shows the most noticeable difference in the this simulation is the run time but that makes sense since A3s only interacts with a portion of the balloons each iteration, whereas A3 generates a decision

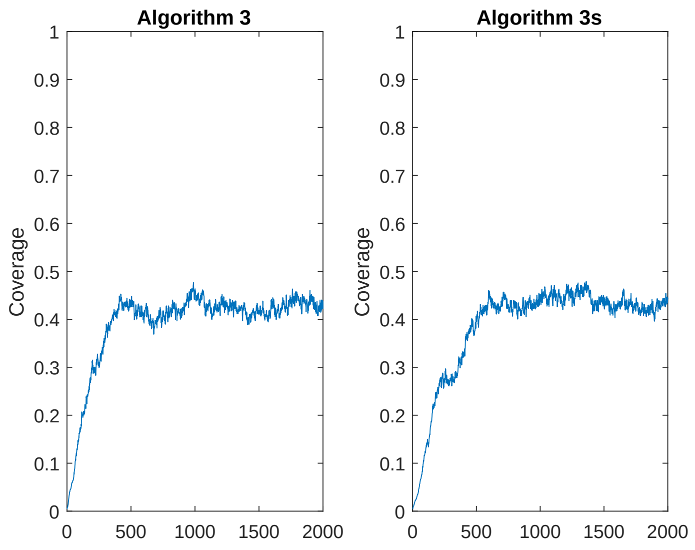

Figure 18: Algorithm 3 vs. Algorithm 3s in 1000 steps with lifetime 200

for all balloons every step.

6.6 Algorithm 4

How does a balloon know the wind direction and speed in its neighbouring wind layers? How does a balloon that has drifted several hundred kilometers from the others determine which direction to take? The most realistic way of determining what the weather conditions are like in another layer, is by communicating with a balloon currently located in said layer. This is what Algorithm 4 introduces.

This approach is more realistic and accurate as it can provide the balloons with live weather data. The model is however still constructed with static wind data which is initialized when the class is created and never updated. But the access, and the way in which balloons acquire data for their decision is changed as of Algorithm 4. This method does however have an obvious pitfall. Since all balloons are launched in the same wind layer, no one will ever know the condition in other wind layers and therefore stay put. This problem is circumvented with a quick fix: Algorithm 1. Since that algorithm did such an excellent job of spreading the balloons on the initial stages it is utilized here in A4.

Algorithm .7 Control Algorithm 4

```
forall Balloon b : balloons do
    if b.age < 50 then
        applyDecision1(b)
    end
    else
        ArrayList n = find all neighbours

        ArrayList w = extract wind layers from n

        Find critical point

        Choose wind layer (from w) that directs towards it
    end
end
```

When the age of a balloon has reached 50 (number chosen randomly) the logic is the same as A3, but with more limited resources. A3 had access to all layers in the system while A4 has to depend on the position of its neighbours. This method clearly favours herding since staying close

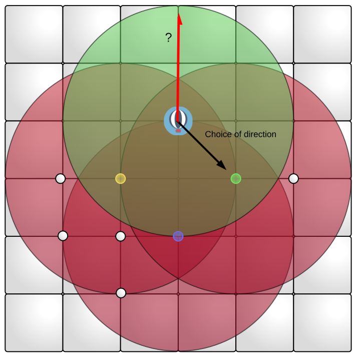

Figure 19: The problem is that the algorithm moves to the least occupied area, but not the empty area

to the pack gives a balloon more data to work from. But the logic however always tries to steer the balloon away from the most crowded areas.

The flaw with the logic of A3 and A4 is displayed in Figure 19. As the balloon does not have access to every point on the grid, it makes its decision based on the neighbouring balloons. This means that the balloon will always navigate towards another balloon. This is a flaw when it comes to spreading, but an advantage as the balloons always need to be in contact with each other.

6.7 Algorithm 4s

Like the ”s” suggests, Algorithm 4s is minor tweak of Algorithm 4. Like displayed in Figure 20 this algorithm focuses on moving a balloon away from all of its neighbours. This is done by adding the direction vectors from the

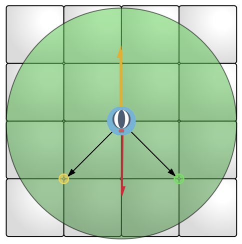

Figure 20: Algorithm 4s - The red vector is the sum of the black vectors and the yellow is its opposite.

balloon to each of the neighbours together, and computing the vector that points to the opposite direction. The movement to that particular direction is however still dependent on only the wind layers found in the neighbourhood. The balloon can still not choose perfectly its direction, but rather chooses between from a limited selection of wind layers.

Algorithm .8 Control Algorithm 4s

```
forall Balloon b : balloons do
    if b.age < 50 then
        applyDecision1(b)
    end
    else
        ArrayList n = find all neighbours

        ArrayList w = all wind layers from n and b

        Compute vector r (Red vector on figure 20)

        Create vector y (Yellow vector on figure 20 )

        Choose layer from w with direction closest to y.
    end
end
```

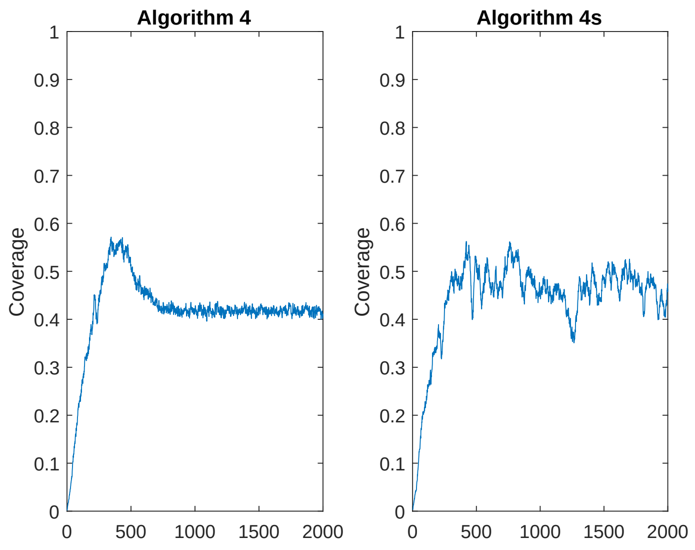

Figure 21: Algorithm 4 vs. Algorithm 4s in 3000 steps with an infinite lifetime of each balloon

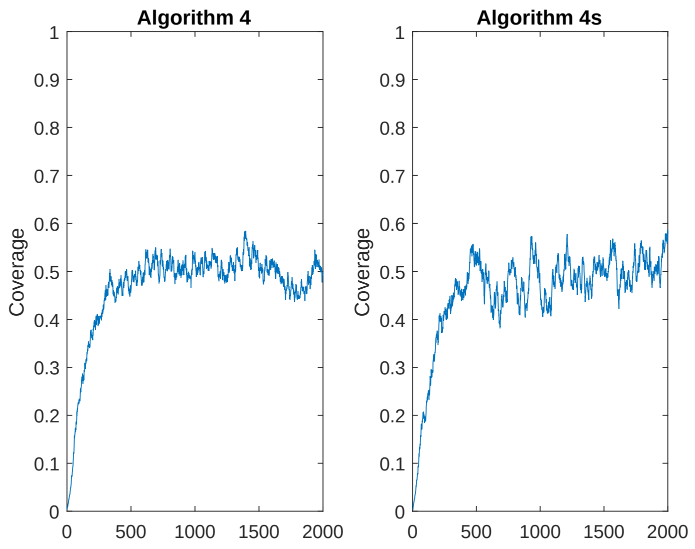

Figure 22: Algorithm 4 vs. Algorithm 4s in 2000 steps with lifetime 200

Statistic	Algorithm 4	Algorithm 4s
Runtime:	35,5s	25,7s
Coverage over simulation:	41,6%	44,2%
Dropped Connections:	1.077.058	1.292.397

Table 4: Algorithm 4 vs. 4s with long lifetime

Statistic	Algorithm 4	Algorithm 4s
Runtime:	29,8s	21,8s
Coverage over simulation:	46,2%	45,9%
Dropped Connections:	1.259.908	1.385.315

Table 5: Algorithm 4 vs. 4s with a lifetime of 200 steps

Both algorithms are relatively stable but A4 appears to have the upper hand. It is interesting to see the curve in A4, displayed in Figure 21, but that is most likely the effect of the first 50 steps, in which A1 was applied.

A4 does a better job maintaining coverage while replacing old balloons. That is not a surprise since the balloons in A4 have the tendency to stay close to at least one other balloon, while A4s tries to keep every one apart. This means that replacing a balloon is less of a shock for A4 than it is for A4s.

Tables 4 and 5 show the difference between the two, but in the realistic scenario, i.e the one with a fixed lifetime, A4 has both higher coverage and lower number of dropped connections.

A4 showed some potential but what happens when the number of balloons is increased? Figure 23 shows the results when the original number

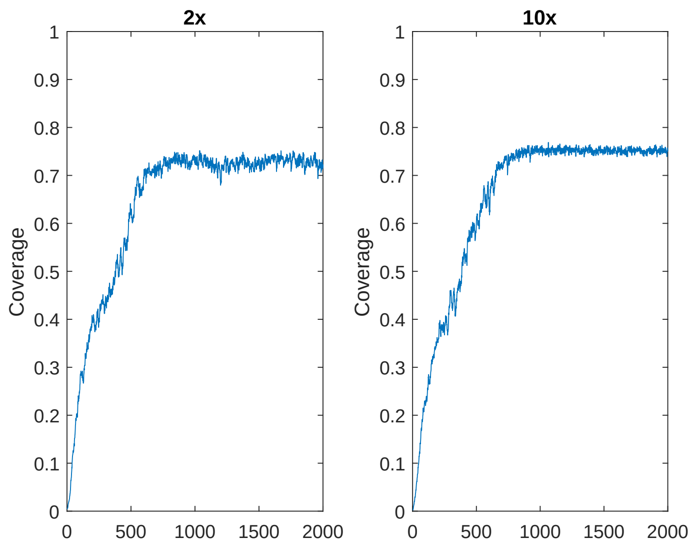

Figure 23: Algorithm 4 - Number of balloons multiplied to reach higher coverage.

of balloon is multiplied two times, and then ten times. The difference is not as much as one would hope and it seems as if the algorithm reaches its full potential with little over 70% coverage. The only noticeable difference is the stability.

7 Discussion

This project is like Pandora's box. The idea is so simple and conceivable and creating a model for it sounds like a piece of cake. Circles floating around some area according to some predefined wind speed. But the considerations and details are endless. A long discussion could be had about each detail of the model but that is material for a whole other project. To keep things short and precise some key discussions have been chosen that are interesting to give a second thought to, and should maybe have been implemented differently. For clarity a sub chapter has been created for each of the discussion topics.

7.1 Decentralized vs. Centralized

This model was designed and implemented with no concern over decentralized vs. centralized algorithms. It was designed in a way that made sense with regard to JAVA and the GRASP pattern. In retrospect it would have made sense to give more weight to the role of the balloon in the system. As it is now, it serves only an abstract role indicating its position in the grid. All other communications are channeled through the *World* class, but that indicates a centralized solution. This was sidestepped by putting some restraints on the flow of communication between the balloon and the world class, but the information was still there; it was just not allowed to pass on to the balloon. This was done to imitate the decentralized solution.

If the question of centralized vs. decentralized would have come earlier in the process the model would have taken a different shape. More accurate attributes in the balloon class could have given more weight to it, to better resemble the communication between balloons and hence, the decision to move or not.

7.2 Number of balloons

The selection of the number of balloons was on a relatively low level compared to the references available. Some extensive studies have been done on the issue of covering a square area with circles, and the approach offered in this project is really naive. It expects a lot of overlapping which in the end would suggest more balloons than actually needed. That is an expensive assumption to make. But of course the problem description is just abstract, and the real life problem will not deal with a fixed sized, square grid. It is way more complicated than that.

7.3 Time management

In a project such as this it is easy to get hung up on some minor details. The ambition for a perfect control algorithm was way too much and too much time spent on debugging some tiny aspects, instead of focusing on the bigger picture. It is easy to feel invested in this and constantly try and improve some control algorithm instead of documenting its performance and moving on. With wind data as complex as this and this number of balloons, this problem becomes very difficult to debug and analyze by hand. Running this step by step is not helpful if you don't have some automatically generated statistics because keeping up with the movement and position of all balloons by hand is incredibly tedious, if even possible.

7.4 GUI

The only good use for a GUI, in my opinion, in this project is to see the location of each balloon. The user interface could be extended endlessly and the user experience could be engineered to perfection, but that is besides the point. What is useful is the placement of the balloons at each step, and the statistics of the program. I feel that two weeks went were wasted as I studied the depths of GUI programming in JAVA in order to squeeze a certain element out of the model. The functionality was completely unnecessary.

7.5 Live weather data

The weather data at this height is difficult to attain and maintain. One of Google's ambition in this project is to make the balloons gather data while deployed. This would give them Google better data to work with, as well as the research community of course. The thought came to add this functionality to the program. It is really simple in practice. The balloon measures the speed and direction of its surroundings and reports it. But adding this to the program, as another mock data set, did not seem important at this stage. But a simple version of this could be implemented in a similar way as the distortion of the wind vectors. A balloon could be given some ”noise” it could add or subtract from its current spot in the wind layer. Example: Balloon b at point (x,y) where the wind vector is (33,40). If the ”noise” were 10, two random numbers between -10 and 10 would be generated and added to 33 and 40. This position in this current wind layer would be updated with these numbers, and that would affect all balloons in that spot immediately.

7.6 Algorithms and measurements

The difficult thing about implementing the control algorithms is the number of steps in a simulation. A logic might sound good for the initial steps, or under certain circumstances, but when executed over and over again thousands of times, it is difficult to foresee the pattern. Some implementations would look good for the first thousand steps, but then all balloons would huddle together and stay there and coverage would drop. In those cases some state was eventually reached where the system would stay unchanged, or locked, in a really inconvenient state. To circumvent a problem such as this, some detection mechanism should be introduced to the model, where the system could be shuffled if a lock was detected.

More accurate measurements should be implemented at the next stages of this project. The current graphs show the coverage in the system at any given time but that does not tell the whole story. It would be interesting to install random sampling into the model, which would measure the coverage at a given point for the duration of the simulation. The coverage in the system can be stable, while the coverage in each individual point is extremely unstable. This could be implemented by introducing yet another grid, counting the times each point is connected. This could be represented graphically in a heat map.

8 Conclusion

The question posed in the introduction is whether Google's endevours, placing its balloons in the stratosphere, are feasible or not. To answer that question a model was constructed which simulates the balloons and their behaviour in the stratosphere. Their behaviour depends greatly upon the data they are provided with but this model included only its own mock data. With that in mind, the results are promising.

The algorithms developed during this project showed that stable coverage could be maintained in the system even while replacing old balloons. Furthermore, the model showed that even algorithms with limited access to data were still able to provide stable coverage with balloon to balloon communication. Even though the best algorithm only reached roughly 70% coverage, it still concludes that the idea is not far fetched.

References

Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1988). Network Flows. , 1–231.

Aldrich, J., & Garrod, C. (2013). Principles of Software Construction : Objects , Design and Concurrency Java Collections.

and Lo, N. (2015). *How Google[x]'s Project Loon Tests Its Giant Internet Balloons - #NatAndLo Ep 9.* Retrieved 2016-01-02, from `https://www.youtube.com/watch?v=kQDQ3Ps_-b4`

Brown, K. C. o. I. S. (2015). *INTERNET SOCIETY GLOBAL INTERNET REPORT 2015* (Tech. Rep.). Internet Society.

Cabello, S. (2007). Approximation algorithms for spreading points. *Journal of Algorithms*. doi: 10.1016/j.jalgor.2004.06.009

Černý, T. (2010). GRASP Patterns. *X36Ass*.

Gelsbo, P. (2015). Modelling Google Loon.

Google. (n.d.). *Facts and Figures.* Retrieved 2016-05-08, from `https://sites.google.com/a/pressatgoogle.com/project-loon/facts-and-figures`

Google. (2013). *Introducing Project Loon.* Retrieved 2016-02-01, from `https://www.youtube.com/watch?v=m96tYpEk1Ao`

Google. (2015). *Project Loon - Scaling Up.* Retrieved 2016-02-01, from `https://www.youtube.com/watch?v=HOndhtfIXSY`

Katikala, S. (2014). Google Project Loon. *Rivier Academic Journal*, *10*(2), 3–8.

Levy, S. (2013). The Untold Story of Google's Quest to Bring the Internet EverywhereBy Balloon. *Wired*. Retrieved from `http://www.wired.com/2013/08/googlex-project-loon/all/`

Loon, P. (2013a). *Ask Away: How can balloons provide stable coverage?* Retrieved 2016-02-01, from `https://www.youtube.com/watch?v=mjyLynnQuC4`

Loon, P. (2013b). *Ask Away: How do the balloons move up and down?* Retrieved 2016-01-02, from `https://www.youtube.com/watch?v=E4DOc6yQftc`

Loon, P. (2015). *Ask Away: How are Loon balloons recovered?* Retrieved 2016-02-01, from `https://www.youtube.com/watch?v=zpzwQajQxZ4`

Moon, S., Moon, I., & Yi, K. (2009). Design, tuning, and evaluation of a full-range adaptive cruise control system with collision avoidance. *Control Engineering Practice*. doi: 10.1016/j.conengprac.2008.09.006

Murtagh, D., & Chalmers, . (n.d.). Measurement of Stratospheric and Mesospheric Wind Fields Using SMILES and with an Outlook to the Future.

Project Loon. (n.d.). Retrieved 2016-01-02, from https://www.google.com/loon/

Roginsky, M. L. M. N. (1999). *What is the avg. (and max.) wind speed in the stratosphere?* Retrieved 2016-05-06, from http://www.madsci.org/posts/archives/1999-09/938402041.Es.r.html

Simonite, T. (n.d.). *Project Loon.* Retrieved 2016-04-17, from https://www.technologyreview.com/s/534986/project-loon/

Swinbank, R., & Ortland, D. A. (n.d.). COMPILATION OF WIND DATA FOR THE UARS REFERENCE ATMOSPHERE PROJECT.

The Verge. (2015). *Inside Google's wildly ambitious internet balloon project.* Retrieved 2016-01-02, from https://www.youtube.com/watch?v=OFGW2sZsUiQ

Witold Mertens, S. (2014). Google Loon Control Systems.