AALBORG UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
SOFTWARE SYSTEMS ENGINEERING

Route planning for an autonomous agent

Eglė Sasnauskaitė Joana I.L. Miguens
Marius Mikučionis Rasa Bonyadlou Stanislav Levchenko

Supervisor: Anders P. Ravn

December 19, 2001

Title:

Route planning for an autonomous agent

Subject:

Distributed systems

Project group:

SSE-101

Participants:

Eglė Sasnauskaitė
Joana I.L. Miguens
Marius Mikučionis
Rasa Bonyadlou
Stanislav Levchenko

Supervisor:

Anders P.Ravn

Abstract

Distributed systems are used in the agricultural business and among other issues an opportunity to use autonomous agents is analyzed. One of the problems for autonomous agents is route planning. Our aim is to analyze and describe several route planning algorithm and it's advantages and disadvantages. 4 route planning and one exclusion zone avoidance algorithms are implemented practically in the Simulator application. The route planning algorithm were simulated and tested in various situations that might appear in a farm field.

Time of writing:

The 3^{rd} September 2001 - the 19^{th} December 2001

Copies:

8

Pages:

57

Appendix:

Floppy disk

Preface

This project is submitted by the master group of Software Systems Engineering for the Distributed Systems course at the Department of Computer science at the Faculty of Engineering and Science, autumn 2001. The purpose of the project is to design and integrate a route planning algorithm into a distributed system, which is a simulator application of the agricultural autonomous agents.

Eglė Sasnauskaitė

Joana I.L. Miguens

Marius Mikučionis

Stanislav Levchenko

Rasa Bonyadlou

Contents

List of Figures

List of Tables

1 Introduction

Technology made a revolution in agriculture during the last 50 years. Big farms with a high degree of mechanization and automation by IT researched the methods and ways how to reduce the cost of farming for years. This issue is related with more efficient information management, manual work replacement with sensors and robotics, yields monitoring using IT, etc.

Manual work replacement with robotics and sensors or in other words precision technologies in the agricultural sector is a relatively new issue because it requires more comprehensive approach than the industry [1]. Integration of lightweight autonomous agents in a farm requires to pay attention to such problems as agent control, maintenance, job scheduling, route planning, etc.

In our project we will deal with the route planning for autonomous agents. Our goal is to find the best route for the agricultural autonomous agents among given points that would include necessary points in a field avoiding exclusion zones, considering the position of agents, costs and the environmental issues.

As an example let us take a tractor or any moving vehicle which can move without a human driver. It moves performing certain tasks. Different tasks that an autonomous agent could perform could be distinguished in three main groups. For every group of tasks there can be different strategy for the route planning:

Observation: gather the information about the soil contents (humidity, acidity), detect grain quality and types, check weed and crop density, detect the vermin. Here the agent should follow the planned route from one point to another to perform certain tasks.

Tillage: plaguing, loosing, harrowing, sowing, fertilizing, vermin sprinkling, harvesting. For those tasks the agent has additional implements attached and the route should cover the entire field. In this case optimality criteria could be route crossing absence.

Other agent support: deliver seeds, implements, fuel, take a broken agent from the field for maintenance. For this kind of tasks the agent does a job by visiting only one or a few certain points in the field.

In our project we concentrate only on the first group of tasks - *field observation by an autonomous agent*.

An agent must visit certain points of the field to get information which would be suitable to represent a general situation in the field. For that purpose points to be visited usually form a grid structure. Agent must avoid exclusion zones in the field, pay attention to plant damages and still follow the shortest route. So the route planning is not trivial and very important issue to get certain results from field observation with minimal cost.

The structure of our report includes five main chapters. The first section introduces the system definition and gives analysis for our problem and application domains. At the end of the section we define requirements for our project. Next section describes algorithms which are used for solving general route planning problems. We'll discus advantages and disadvantages of the algorithms and some of them will be used for further implementation in our project. We describe an input for the system and how we get the output - the final route.

We expect that the reader is acquainted with the computer science terms or has software engineering background.

2 Analysis

The analysis section is structured according to the object oriented analysis and design paradigm described in [3]. In the problem domain section 2.1 we analyze the agricultural objects described in [1] and [2] projects and identify the role of route planning for the autonomous agents. We draw our main goal during this project in the system definition section 2.2. The application domain section 2.3 identifies and describes the details of the objects used in our project. Finally the requirements section 2.4 summarizes the sub-goals and assumptions to be accomplished to reach our main goal.

2.1 Problem domain

Route planning for autonomous agent is related with a wide concept of efficient farm management [2]. From the farm management model point of view, the field observation is a *job* to be performed by an agent in a certain field. An autonomous agent consists of a self controlling moving *platform* and several *implements* attached to it (see figure 1). The agent can perform several jobs simultaneously with several independent implements, so the main *job* associated with the *platform* actually consists of smaller sub-jobs.

Each *job* is associated with a field which shows where the actual work should be performed. An average farm takes care of many fields that is about 600ha area in total. Usually each *field* is divided into sub-*fields* (figure 2) that can be divided further into smaller "sub-sub-*fields*" and so on. Besides that some places in a field can be unreachable as the surface of the field is not a perfect plane and a variety of obstacles (rocks, lakes, trees, bushes, windmills, buildings and others) can be met in the field.

The unreachable field places are called *exclusion zones* in the agricultural terminology. Most of the exclusion zones are known to the station and marked in the map, but there may appear new obstacles forming exclusion zones: fallen

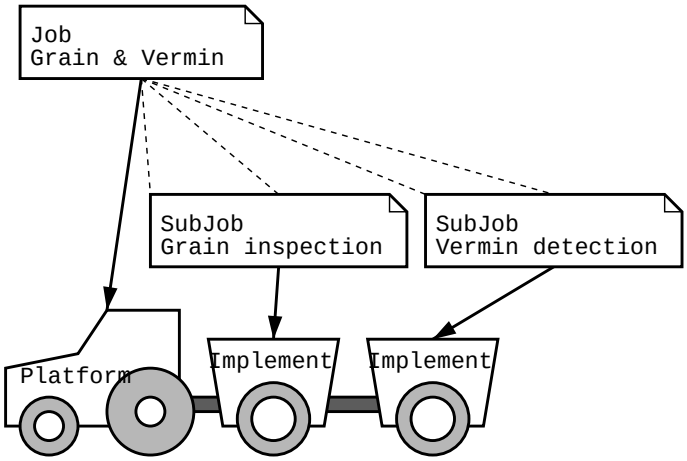


Figure 1: Agent and job relationship.

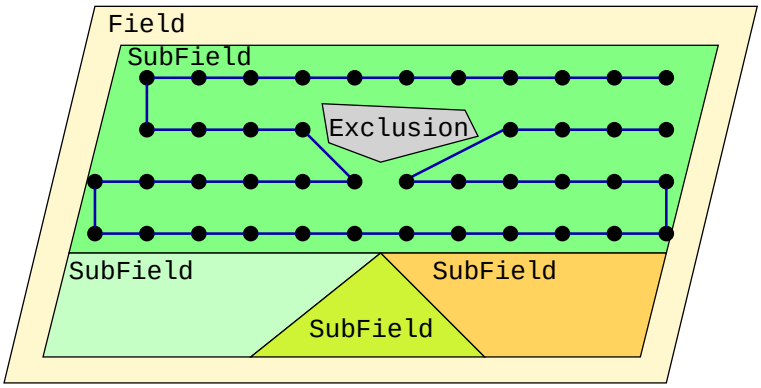


Figure 2: Field structure and points to be visited.

tree, big pools after the rain, a human or a cow walking in the field. The moving or unknown obstacles must be detected by the agent in run-time and the map of exclusion zones must be updated. In our project we will consider only those exclusion zones which are known to the system.

The specific observation job is associated with a field and therefore must tell the exact places of the field to be observed. These places are represented by the *points*, which usually can be arranged into a grid structure (figure 2). The platform assigned to do the job must visit those points by following the route.

A *route* is an ordered list of points to be visited in the field. The optimal route avoids exclusion zones and saves costs in terms of minimal fuel consumption, shortest time and minimal plant damages. The plant locations are described by

the crop structure, which is also arranged into a grid.

The class diagram in figure 3 summarizes the relationships between entities in the farm.

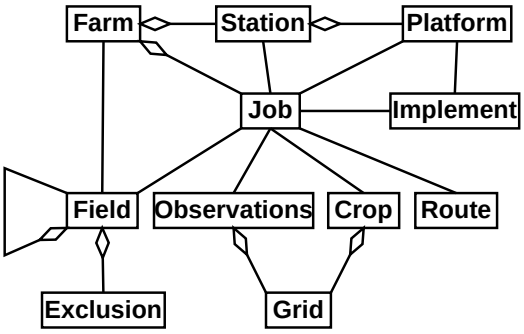


Figure 3: Class diagram of farm entities.

All components (farm office, station, platforms) must be connected into a distributed system, which handles the field map, job data, the agent positions and still reacts to environment changes in real-time [2]. The proposed system architecture is in Figure 4.

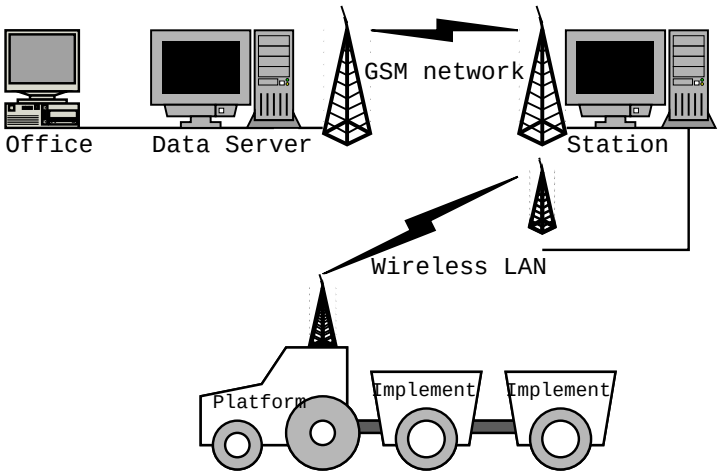


Figure 4: Farm components.

There is an office, where the farm manager works with business data (specifies what job and where should be done) on a data server. The system receives job structures describing the task, the field structure, the crop structure and the grid of points to be visited. The station computes the optimal route and passes it to

the selected platform to do the work in the field. Note that platform is always connected to the station via a wireless local area network, thus it can request a new route if an unexpected exclusion zone occurs in real time.

2.2 System definition

Our goal is to design a route planning algorithm, which resolves the optimal route for the autonomous agent to do field observation tasks. The designed algorithm is to be integrated into the simulator application on the Java platform.

2.3 Application domain

In this part we will give a more detailed presentation of formalized concepts in the class diagram. Figure 5 shows the packages and the main classes in the simulator application software.

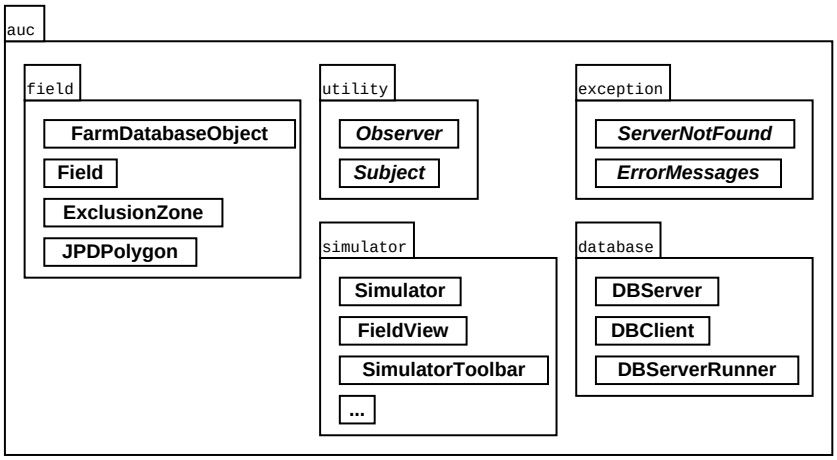


Figure 5: Classes and packages in the Simulator application.

All packages are placed into main *auc* package (Figure 5):

simulator contains classes related to the simulation start and graphical user interface (main window frame, menus, control and viewing panels).

utility holds auxiliary classes for a close interaction between *field*, *database* and *simulator* classes.

exception package contains various exceptions in the Simulator application.

field package contains classes that represent sets of objects in the farm.

database package classes are responsible for the data transfer between the farm office database and the station. The goal of these classes is to produce *field* package objects at the station which correspond to the data in the database server at the farm office.

General concepts of our problem domain were presented earlier. Figure 6 represents structures of a field more detailed from an application point of view.

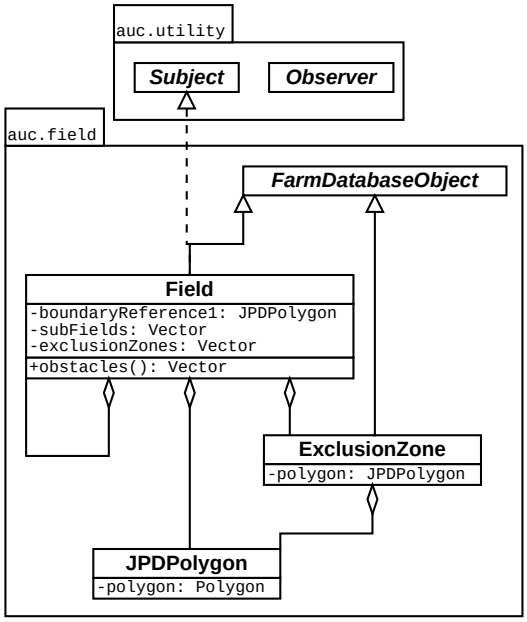


Figure 6: Field package class diagram.

A *field* holds a recursive structure since a field can have several sub-fields which are also fields. From a geometrical point of view a field has a boundary which is a polygon. The boundary polygon is represented as a sequence of ordered points connected by edges where the last one is connected with the first one. Note that none of the normal field boundary edges can be crossed.

The *field* structure also contains references to *exclusion zones*. A boundary of an exclusion zone is also marked by a polygon, but it requires different handling because these areas should not be crossed by a platform.

Further more we need to describe additional classes to deal with coordinates of a field boundary, observation points and finally route objects (Figure 7).

We need a *job* data structure for the route planning algorithm. The *job* should have references to the *field*, *crop* and *observations* data structures. The reference to the *route* data is needed for the platform to follow the route.

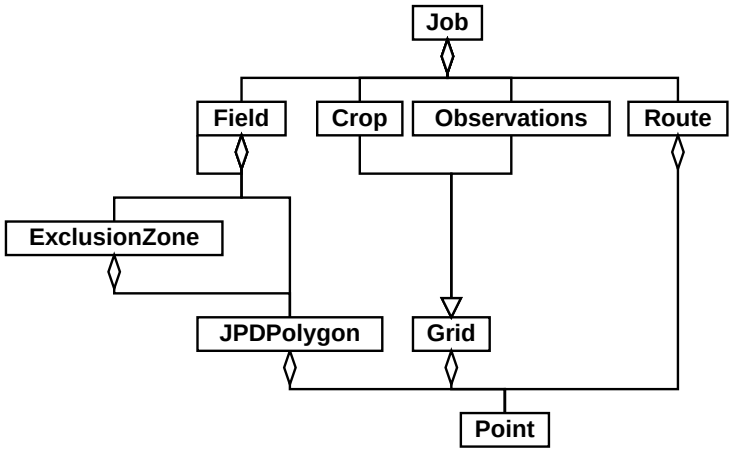


Figure 7: Other data structures for the route algorithm.

The *crop* and *observations* structures have the similar properties which are summarized in the *grid* structure, which has a list of *point*s.

A *route* is presented as a polyline, which consists of points of a field connected in a certain order. The *route* data is planned according to the specifications of a certain job in the station machine. It covers the grid of observation points which have to be visited by platform.

In the real world physical properties of the platform (such as dimensions) should be considered while planning a route (this involves the speed of the platform, the radius of turns, the width of the path and so on). In our route planning we will narrow down the problem and consider an agent as a moving dot without dimensions.

There can be some cases when the path to the given observation point can not be found:

1. Obstacles block the way from one point to another, e.g. a fence, too narrow entrance of a gate, etc.

2. The route is inefficient because of the high cost when platform tries to go around the obstacle or does wide turns, e.g. if the dimensions of the platform are so large then it has to make a wide U-turn of 180° when moving from one row to another.

Single points as fuel filling and maintenance points could be taken into account too, but in our project we will not consider this option.

2.4 Requirements

We have analysed on what we should consider in our project and distinguished requirements:

1. The route should include all points selected unless they are unreachable.
2. The route should avoid exclusion zones.
3. The route should prefer to follow direction of plant rows in the field to minimize plant damage.
4. The route should be optimal regarding:
   (a) The sequence of points implying the minimum cost of the route.
   (b) Travel cost should include distance and plant damage.
5. The algorithm must be efficient, that is polynomial complexity in terms of the number of the observation points.
6. The algorithm must be able to deal with convex and concave field boundaries.

In order to achieve the route planning goals we assume the minimal requirements for the algorithm input components:

- Job:
  1. Contains all needed initial data for the route planning: field, crop and observation point structures.
- Field:
  1. Has recursive definition as it holds sub-fields which are fields itself.
  2. Field and exclusions have boundaries which are polygons.
  3. Coordinates for the field geometry are given in Cartesian coordinates system measured in metres.
- Crop and observation point structures:
  1. Defined by a two-dimensional grid.
  2. Are able to generate points having grid dimensions and field boundaries.
  3. Are able to construct the grid from parameters in the job data file.

4. Cover given entire field.
5. Does not contain points that are outside the field boundary or within exclusion zone boundary.
6. Has a possibility to address/locate each point by grid coordinates.

3 Design

Our goal is to implement the algorithm, which would find the shortest possible path regarding the cost of the route or would inform if there is no possible path to the some given points.

The design section is split into several logical parts: at first we specify the input for the route planning (job, grid and field structures), then we introduce route segment construction (exclusion zone avoidance) as a sub-algorithm for the route planning, and finally we present several general and grid-oriented algorithms for the route planning.

3.1 Job structure

The job structure is meant to contain sufficient information for the initial route planning, so the objects of this class contain references to the observation points, crop structure and a field (Figure 7). The job has many more attributes in the FieldStar [2] database (e.g. agent machine and implements assigned to do the task, the planned schedule and so on) but we concentrate on the data which is required for one platform to do the observation task.

3.2 Grid of points

The points to be visited by autonomous intelligent agent are stored in the database as a parameterized grid.The grid structure also can be used to mark the plant positions. The grid has three parameters: starting point, horizontal and vertical vectors (Figure 8).

The grid is generated by adding and subtracting $\vec{h}$ and $\vec{v}$ from the starting point S:

$$P(n, m) = S + n\vec{h} + m\vec{v}, \tag{1}$$

where $n, m \in \mathcal{Z}$ (are integer numbers).

We have developed a way to generate a grid to cover the field and order the points into a two-dimensional array. All the points generated are verified to be

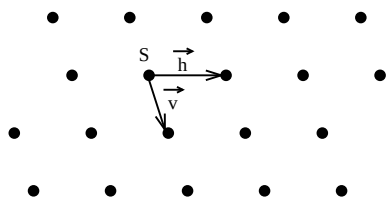


Figure 8: Illustration of grid parameters.

valid, i.e. the point should be inside the field and not belong to any exclusion zone. Non-valid points are filtered out as described in the *Point filtering* section.

3.2.1 Point array generator

This algorithm uses the transformation from given Cartesian coordinate system to the grid coordinate system. The grid coordinate system is also Cartesian but can be shifted from real world coordinates and the vector base need not to be orthogonal. The idea is based on linear mapping between two linear spaces (algebras) [6]. So the only assumption is that the angle between two base vectors $\vec{h}$ and $\vec{v}$ should not be equal to $0°$ nor to $180°$ and both vectors can not be zero length (to have a linear mapping we need to have an independent[1] base vectors [6]).

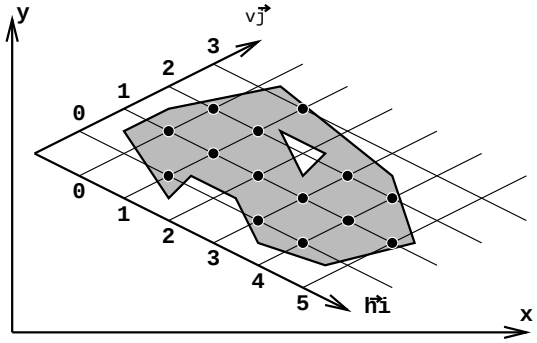


Figure 9: The grid point structure in the two dimensional array.

The Figure 9 illustrates the array of points that covers the field structure. To minimize the memory used the point enumeration begins from row 0 and column 0 and ends with last row and column, i.e. the first (number 0) and the last row and/or column contain at least one point.

[1] Vectors $\vec{v}$ and $\vec{h}$ are independent if and only if $a \cdot \vec{v} + b \cdot \vec{h} \neq 0$ for $\forall a, b \in \mathcal{R}$

To find the coordinates of the first and the last rows and columns we need to transform the real world polygon coordinates into the grid coordinates. To enumerate the points in the grid we need to transform the grid coordinates back to the real world coordinates. So we need a complete bidirectional mapping from and to the grid coordinate system.

Figure 10 introduces the grid coordinate system and tells how to calculate the grid coordinates for any vector $\vec{p}$:

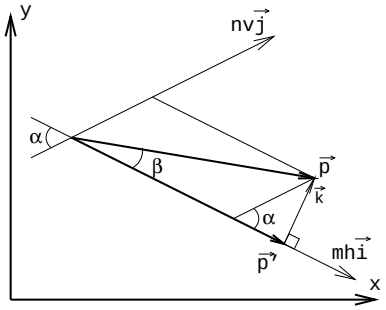


Figure 10: The $\vec{p}$ transformation from the real fieled to the grid coordinate system.

- The direction of grid coordinate system axis is described by 1-length vectors $\vec{i}$ and $\vec{j}$:

$$\vec{i} = \frac{\vec{h}}{|\vec{h}|},\ \vec{j} = \frac{\vec{v}}{|\vec{v}|}. \tag{2}$$

- The coordinates on $\vec{i}$ and $\vec{j}$ axes are described by nh and mv numbers: $h = |\vec{h}|, v = |\vec{v}|, m, n \in \mathcal{R}$. m and n are the real numbers denoting the grid coordinates of the given point. Note that $\vec{h} = h\vec{i}$ and $\vec{v} = v\vec{j}$, and the point $nh\vec{i} + mv\vec{j}$ belongs to the grid if and only if $n, m \in \mathcal{Z}$.

- The real world coordinates can be calculated just adding the grid coordinate components:

$$\vec{p} = mh\vec{i} + nv\vec{j} = m\vec{h} + n\vec{v}. \tag{3}$$

- The projection of $\vec{p}$ to axis $\vec{j}$ is:

$$\vec{p'} = (\vec{i} \cdot \vec{p})\vec{i} = (p\cos\beta)\vec{i}. \tag{4}$$

- The vector $\vec{k}$ is perpendicular to $\vec{p'}$ then $k = nv\sin\alpha$ and $\vec{k} = \vec{p} - \vec{p'}$, so:

$$n = \frac{|\vec{p} - \vec{p'}|}{v\sin\alpha} = \frac{|\vec{p} - (p\cos\beta)\vec{i}|}{v\sin\alpha}. \tag{5}$$

- From the Cartesian coordinate system property (3) we have:

$$m = \frac{|\vec{p} - nv\vec{j}|}{|h\vec{i}|} = \frac{|\vec{p} - n\vec{v}|}{|\vec{h}|}. \tag{6}$$

3.2.2 Point filtering

Each point of the grid generated is checked if it belongs to the field or not. Due to rather complex field structure, point filtering is made in the following way:

1. Check if the point belongs to the field boundary polygon.
2. Check if the point does not belong to any exclusion zone (the boundary is a polygon too).

The following algorithm (see the Figure 11) describes how to verify whether the point belongs to the polygon [4]:

1. Emanate a horizontal ray from a given point towards positive infinity of X-axis.
2. Count how many edges of the polygon are crossed by the ray.
3. If the count of crossed edges is odd then the point is inside polygon, otherwise it is outside.

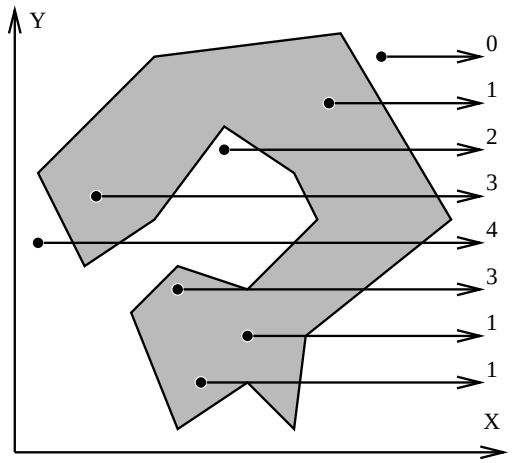


Figure 11: Inside polygon point checking.

The ray intersection with the edge of the polygon is evaluated by the following predicate:

$$Intersects(P_1P_2, P) = \Big(\big((y_1 \leq y) \wedge (y < y_2)\big) \vee \big((y_2 \leq y) \wedge (y < y_1)\big)\Big) \wedge \big(x < (x_2 - x_1) \cdot (y - y_1)/(y_2 - y_1) + x_1\big). \quad (7)$$

where $P_1(x_1, y_1)$ is one of the polygon vertices, $P_2(x_2, y_2)$ is next to P_1 polygon vertex and $P(x, y)$ is the point to be tested.

At first the predicate $Intersects(P_1, P_2, P)$ checks whether the given point P coordinate y is between y_1 and y_2 coordinates of the edge vertices P_1 and P_2 (otherwise the ray has no chance to cross the given edge).

Secondly the predicate calculates the crossing coordinate $x_c = (x_2 - x_1) \cdot (y - y_1)/(y_2 - y_1) + x_1$ and it is true only if the X coordinate of the given point is less than x_c, which means that given point is to the left of the polygon edge and the ray will cross the segment.

Note that:

1. The ray may cross the polygon edge at the vertex point. In this case one of the segment endpoints is omitted to belong to the segment (strict less sign $<$ when comparing the y coordinates). This ensures that the edge endpoints are counted properly (the last three rays in the Figure 11):
   (a) two times, if the ray touches the angle which "looks" downwards;
   (b) once, if the ray crosses the angle which "looks" neither downwards nor upwards;
   (c) not counted at all, if the ray crosses the angle which "looks" upwards.
2. y_2 may be equal to y_1, in this case we will get positive or negative infinity when calculating x_c, but still we can compare it with the x and Java handles this without any additional checking.

3.3 Field as a structure of polygons

As already mentioned in the application domain section 2.3, a *field* contains references to its subfields and exclusion zones. The database stores the recursive structure of fields by the additional attribute *PartOf* which contains the reference to the "parent" field.

However the exclusion zone does not have a recursive structure, it has single reference to the last sub-field in field structure it belongs to. So none of the "parent" fields knows about its exclusion zones unless it searches for exclusion zones in all of its subfields. To fix this we add handles to the *field* methods which

control the exclusion zone setting: in the early construction of sub-field, when sub-field receives the references to the exclusion zones, it passes the references to the parent-field, which registers the exclusion zones and passes them to its parents recursively. In this way the system will contain just one copy of each exclusion zone, but every field in the hierarchy will be notified about the exclusion zones they contain.

The exclusion zones are not the only obstacles for the autonomous agents. The field may be concave, thus the platform may go out of the field by crossing a concavity.

To simplify the route planning algorithm we propose to extend the field polygon to a minimal convex polygon and add the concavities to the *list of obstacles* in the field. Note that we can not add the field concavities to the list of its exclusion zones since the parent field may "think" of them as an exclusion zones while they are not. In this way the route planning algorithms should concentrate only on the list of obstacles and not care about the concavities nor the exclusion zones. In future this list may be updated by the list of moving obstacles (exclusion zones) in the field.

A detailed solution to the concavities problem is described in *Polygon* design section 3.4.

3.4 Polygon

Our Polygon2D class extends the standard java.awt.Polygon, overrides the integer methods with double and defines additional methods for use in the *Field* and the *ExclusionZone*. The point containment method algorithm is described in the Points Filter section 3.2.2. Let's describe two more methods:

Concavities. The polygon may be concave or convex. So, to avoid concavities we will think over them as virtual exclusion zones listed in the list of obstacles in the field. To get the concavities we start with a vertex orientation checking for convex polygons [6]. It is based on the cross product between vectors of adjacent edges. If the crossproduct is positive then the cross product vector rises above the pane (z axis up out of the plane) and if negative then the cross product is into the plane:

$$crossProduct = (x_i - x_{i-1}) \cdot (y_{i+1} - y_i) - (y_i - y_{i-1}) \cdot (x_{i+1} - x_i). \quad (8)$$

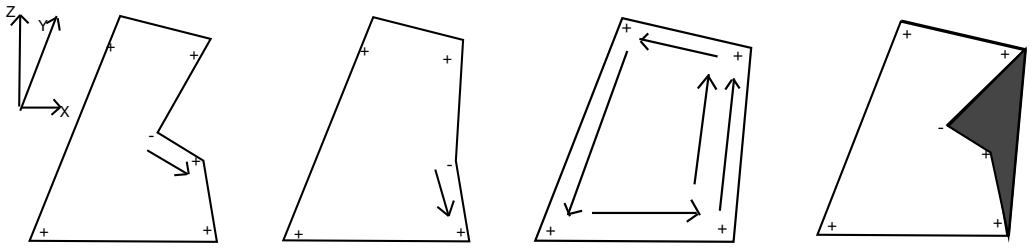
where x_i and y_i are the coordinates of corresponding polygon vertices.

If the cross product is positive we have a clockwise listed polygon (see Figure 12(a)). To determine the vertex ordering we calculate the polygon

area by this equation:

$$Area = \frac{1}{2}\sum_{i=0}^{N-1}(x_i \cdot y_{i+1} - x_{i+1} \cdot y_i). \quad (9)$$

So, we go one way around checking the vertex cross product sign. If cross product has different sign from the entire polygon area sign, we have a concavity. This vertex is removed and we continue from one vertex back (Figure 12(b)). At the end we are able to get the concavities by checking the vertices which are in the last polygon and not in the first (Figure 12(d)). We check each vertex just once, unless it belongs to the concavity (then



(a) Given polygon and cross product signs.

(b) Go back after the vertex removal.

(c) Convex polygon.

(d) Concavities.

Figure 12: Convex polygon algorithm.

we check it twice), so our algorithm belongs to a complexity class $O(n)$ in terms of the polygon vertices number n.

Intersection. Another method is to detect the intersection point of two given segments, which is used to detect whether the given segment crosses the polygon and furthermore whether the polygon contains another polygon. Let's start calculating the following formulas assuming one segment (called a) from A to B and another from C to D (called b). See Figure 13.

The line equations are:

$$Segment_a = B + u_a(B - A) \quad and \quad Segment_b = C + u_b(D - C) \quad (10)$$

Then we equalize the two equations in X and Y:

$$X_A + u_a(X_B - X_A) = X_C + u_b(X_D - X_C) \quad (11)$$

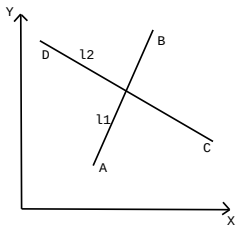


Figure 13: Intersection point.

$$Y_A + u_a(Y_B - Y_A) = Y_C + u_b(Y_D - Y_C) \tag{12}$$

to get u_a and u_b coefficients:

$$u_a = \frac{(X_C - X_D) \cdot (Y_A - Y_D) - (Y_C - Y_D) \cdot (X_A - X_D)}{(Y_C - Y_D) \cdot (X_B - X_A) - (X_C - X_D) \cdot (Y_B - Y_A)} \tag{13}$$

$$u_b = \frac{(X_B - X_A) \cdot (Y_A - Y_D) - (Y_B - Y_A) \cdot (X_A - X_D)}{(Y_C - Y_D) \cdot (X_B - X_A) - (X_C - X_D) \cdot (Y_B - Y_A)} \tag{14}$$

Note that these equations have the same denominator: if the denominator is zero, the segments are parallel. And if the denominator and numerator are both zero we have coincident lines. Since we are interested in segments intersection besides lines, it is necessary to test if u_a and u_b lie between 0 and 1: the segments have a single intersection point if and only if they lie in this interval. The intersection point has the coordinates:

$$X = X_A + u_a(X_B - X_A) \quad and \quad Y = Y_A + u_b(Y_B - Y_A) \tag{15}$$

3.5 Exclusion zone avoidance

There are several path-planning strategies considering exclusion zones. One of them is when the agent periodically updates knowledge about the surrounding environment and re-plans the path. Another strategy is used when the agent knows the exclusion zones and the goal location. In the second case only known static exclusion zones can be considered in advance and that is the case of our project.

3.5.1 Exclusion zone crossing elimination

Calculation of the optimal route segment avoiding exclusion zones can be done in the following way:

1. Calculate the entire route and check which lines include exclusion zone crossing.

2. Modify the route between the certain points correspondingly considering that the route will become longer.

3.5.2 Simplest algorithm for circuiting the exclusion zone

The simplest algorithm for finding the route segment to avoid the exclusion zone could be the following:

1. Make line from *a* to *b*.

2. Separate polygon's (an obstacle) nodes into those which lies on the different sides of the line.

3. Connect the points consequently on every side excluding con-cavities.

For exclusion of con-cavities we agree that if the shape of an obstacle is concave as in Figure 14 P1P2P3P4P5P6 then we'll find the convex shape around it P1P2P3P4P6, that is excluding point P5. The same strategy can be applied for Figure 15 excluding the point P4.

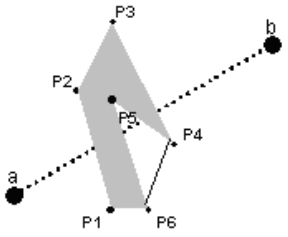


Figure 14: Concave-shaped obstacle is considered as a convex-shaped figure.

There can be two possible approaches to go around the obstacle, but both of them are based on the same idea: find the shortest way around an obstacle either from one or another side making a sidetrack in the route. In Figure 15 we separate two possible routes aP2b and aP1P5P4P3b while the second one needs to be optimized.

Optimization can be done using other more advanced algorithms.

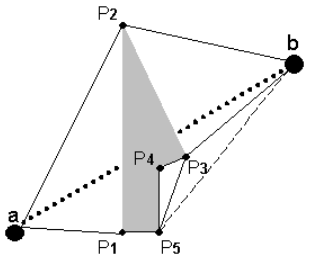


Figure 15: Exclusion zone crossing

3.5.3 Dropping of perpendiculars algorithm

We can calculate distance that is expressed by the length of perpendicular lines from the primary route line to each point of polygon. If the next distance is shorter than the previous one then the point is omitted and next distance is calculated.

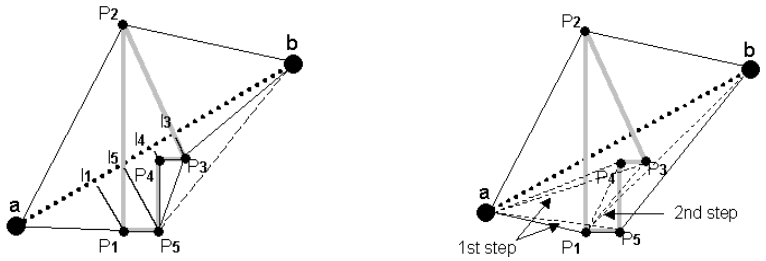
We will follow the previous example. Let us denote distances from the line *ab* to the corner points of the polygon as $l1, l5, l4$ and $l3$ – correspondingly with the point numbers. If we start from point $P5$ and compare $l5$ and $l4$ we will see that $l5 > l4$, though we could skip the point $P4$ and go to a next point $P3$. Next comparing : is $l5 < l3$. The answer is yes and we skip further to the point b. Thus, the shortest way from $P5$ is directly to b (Figure 16(a)).

We assume that the shortest way around the obstacle must be as close to the primary base line as possible. Thus we don't need to compare distances from point to point. But we will do it in another further proposed algorithm.

3.5.4 Measuring distances algorithm

Following that algorithm we check if the line crosses polygon. Lets follow the example starting from point a. Algorithm starts checking from the furthest destination point b and if it is not possible to reach it directly (without crossing the obstacle) then closer points are checked. Points $b, P3, P4, P5$ are not acceptable directly either. Hens the next step is point $P2$. After that we can move to the point $P5$ the same way. Thus the shortest way is $a - P1 - P5 - b$ (Figure 16(b)).

It is obvious that the number of iterations will be less if we start from point b. In that case we will have to check a less number of lines. We guess that as soon as the number of obstacle polygon point is less than 1000 it requires not too much time for calculation, thus optimization of that algorithm is not vitally required.



(a) Example of the Dropping perpendiculars algorithm.

(b) Example of the Measuring distances algorithm.

Figure 16: Exclusion zone avoidance algorithms.

3.5.5 Furthest visible point algorithm

It begins with separating points lying on both sides of the primary route line. The next step is to find such way to the next point that the angle between it and the primary line (for the first step it would be the primary route) would be maximal. In case if there is only one point aside - we just follow the route $a - P2 - b$. The other side of the primary route line is more complicated (Figure 15.

We start to measure angles between the primary route line and all of the points. The biggest angle is $b - a - P1$, thus we move to $P1$. From $P1$ we do the same again, but now we find and compare angles between the rest of the points and line $a - P1$. (That means that at the each step we make found new "primary route line" for the next angle calculation.) The biggest angle is $a - P1 - P5$ and we move to the point $P5$. From $P5$ we find our destination point b directly. The result is the shortest way as $a - P1 - P5 - b$ (Figure 17). Algorithm implies method of "elastic line stretching" from point a to b over both sid of a obstacle.

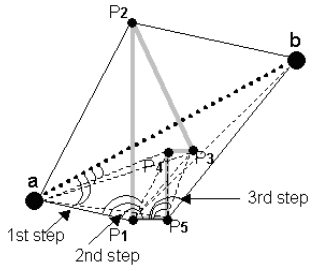


Figure 17: Example of the furthest visible point algorithm.

The advantage of that algorithm is absence of checking belongings of lines

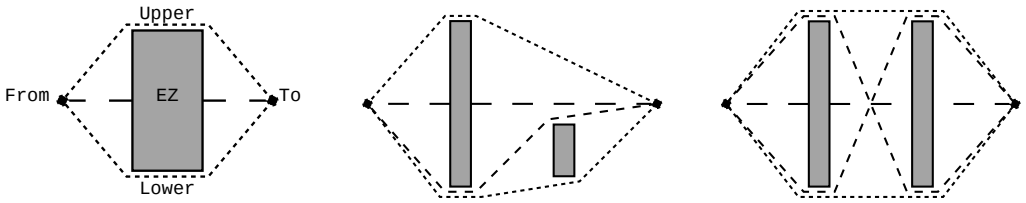
or points to polygon on iteration steps. Finding angles requires less calculation resources and as a result reduces processing time.

In the case of unknown exclusion or obstacle or break down of the agent:

1. Agent reacts to unknown obstacle by stopping.
2. Agent sends message about the problem to control center.
3. Control center re-plans the route

3.5.6 Multiple exclusion zone avoidance

It is not enough to have just one exclusion zone avoidance, since a route segment may cross several exclusion zones in a field. Figure 18 illustrates different ways to avoid two exclusion zones.



(a) Two ways to patch a segment from $From$ to To points.

(b) Three ways to avoid two exclusion zones.

(c) Four ways to avoid two exclusion zones.

Figure 18: Exclusion zones avoidance cases.

In general case there are no more than 2^n ways to avoid n exclusion zones which are intersected by the segment. We developed a recursive procedure to fix the segment by adding new points to avoid the exclusion zones it intersects:

1. Given the segment S: find the exclusion zone EZ crossed by S.
2. Find two ways how to avoid EZ and form two segments $Upper$ and $Lower$ to replace S.
3. Assume that $Upper$ is not a perfect segment and may cross another exclusion zone: invoke this procedure to patch the $Upper$ segment.
4. Assume that $Lower$ is not a perfect segment and may cross another exclusion zone: invoke this procedure to patch the $Lower$ segment.

5. Compare the length of the patched $Upper$ and $Lower$ segments: return the shorter one as the patched segment S.

This procedure works fine for the most of the cases, but fails if one of the given segment ends is completely blocked by exclusion zones. In the blocked point case the algorithm loops, so we limited the recursive calls to the number of all exclusion zones in the field. If the procedure does not find a suitable segment replacement which does not intersect any exclusion zone it returns $null$ which means that the upper procedure calling should choose the alternative segment.

This algorithm is not suitable for general route planning (as you may think of) since it is exponential in terms of exclusion zones crossed, but it is ideal to avoid several (up to 7 at a time) exclusion zones.

3.6 General route planning

The route planning problem is similar to the known *traveling salesman problem* (TSP),which has a goal to find the shortest tour and visit N different cities as we have a goal to find the optimal route among N different points. In the route planning we do not consider the end point of the route to be important while in TSP it is required that the salesman should end up in the same city he started the tour. The TSP is assumed to have an exponential complexity and in terms of complexity the TSP belongs to the class of NP (non-deterministic polynomial) tasks. Mathematicians suppose that it is still hardly possible that a better algorithm will ever be found. The explanation of that problem is described in the section 3.6.1. Although it was known that Brute Force is impossible for big number of cities (more than 12-15), some heuristic and generic methods can give acceptable but not the best solution.

3.6.1 Brute-force algorithm

Brute-force algorithms generate all possible answers to the problem and checks whether it satisfies the conditions of the problem.

The route planning brute force algorithm generates all possible routes to visit all points selected in the field. Figure 19 illustrates all possible routes for five points. The route in the tree is the path from root node to any leaf node, and the length of the route is the sum of distances when traveling from one point-node to another.

Testing every possible route for an N city tour would require evaluating $N!$ routes. A 30-point route would give us $2.65 \cdot 10^{32}$ cases. Assuming modern computer speed calculation of approximately 1 billion (10^9) cases per second, would take over 8,400,000,000,000,000 years. Obviously, this is not suitable solution.

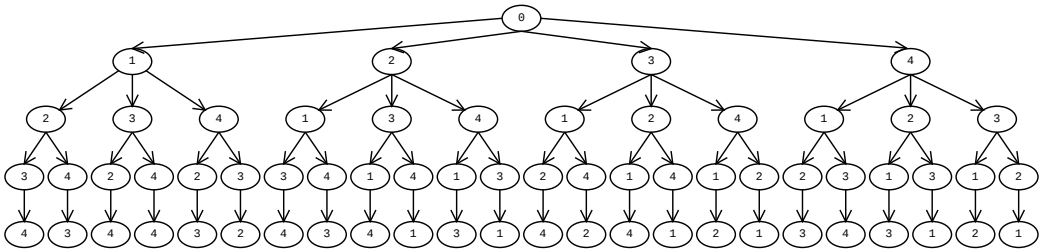


Figure 19: Tree traversed by brute-force algorithm for 5 points.

In next section we will try to find better solution for that problem applicable for the project goals.

3.6.2 Heuristic insertion algorithm

The algorithm originally was found to solve TSP [5] and we adopted it for the route planning. The idea is based on a point insertion to the initial route forming a new route which is used as the initial until all points are inserted:

1. Compute the travel cost (segments) matrix M $n \times n$ from any point to any other point in the field:

$$M[i][j] = Segment(P_i, P_j) \tag{16}$$

   Each segment should avoid exclusion zones and have a cost evaluated by sub-algorithm described in section 3.5.6.

2. Form a list L of all given observation points.

3. Choose the segment from the first to the second point in L as the initial route R. Remove the first and the second points from L.

4. For each point P in L:

   (a) For each position I in R:

      i. Calculate the insertion cost of P into R in position I:

$$IC(P, I) = M[I][P] + M[P][I + 1] - M[I][I + 1] \tag{17}$$

      ii. Remember the insertion position and the insertion cost if $IC(P, I)$ is less than any other insertion cost calculated before.

(b) Remember the point P and the best insertion position I if the point insertion cost is less than any other point insertion cost calculated before.

5. Insert the best insertion point P into the route R and remove P from L.
6. If L is not empty, go to step 4.
7. Return R.

It is obvious from the algorithm description that the algorithm requires n^3 iterations to calculate a route, where n is the number of points. The route optimality depends very much on the initial two points selected. We added a procedure which creates several different permutations of observation points and chooses the best route produced by the algorithm (the complexity increases to n^4).

3.6.3 Heuristic genetic algorithm

Genetic Algorithms (GAs) were invented by John Holland and they are a part of evolutionary computing, which is a rapidly growing sphere of artificial intelligence [8]. GAs are optimization methods which don't necessary find the best solution but they can find satisfactory solution in considerable time.

Genetical algorithms are used:

- The search space is large, complex or poorly understood. In our case we have many points, complex structures of fields.
- Domain knowledge is scarce or expert knowledge is difficult to encode to narrow the search space. In our case we need to find the route from the begining to the end without partial routes given.
- No mathematical analysis is available.
- Traditional search methods fail.

Genetic algorithms use such definitions as a member or gene, population, generation, fitness, parents and children/offsprings. In our case a *g*ene of a member is a point. *A* population consists of several members, i.e. routes. A measure of *f*itness for each member is the cost (length) of the route. Throughout the search for the optimal solution - the shortest route, a survival of the fittest procedure is used, which means that a shorter route is chosen over longer route [7]. Primary routes are called *p*arents and newly generated routes *c*hildren which form a new generation. Children evolve from the population of parents using selection, mutation and/or recombination mechanisms.

Basic steps:

1. Randomly generate an initial population - a group of random routes ($P_1P_2...P_n$, where 1, 2, 3,...n is the index of the point).
2. Compute and save the length for each individual in the current population.
3. Create new generation of children.
   (a) Select 1 or more shorter routes from the parent population.
   (b) Combine 2 better members using recombination mechanism or the best member using mutation mechanism.
4. Repeat step 2 until satisfying solution is obtained

It should be considered that after combinations of parents there shouldn't be repetition or missing points in the offsprings. Point adjacencies should be preserved from the parents to the children. Ordering problem is solved with permutation encoding.

Recombination mechanism . Single point crossover is performed when one crossover point is selected. The permutation is copied from the first parent till the selected point. Then the second parent is scanned from the beginning and if the number is not yet in the offspring then it is added.

$(P_1P_2P_3P_4P_5P_6P_7P_8P_9)+(P_4P_5P_3P_6P_8P_9P_7P_2P_1) \rightarrow (P_1P_2P_3P_4P_5P_6P_8P_9P_7)$

Mutation mechanism . Mutation is performed on a single parent using order changing - two numbers are selected and exchanged.

$(P_1P_2P_3P_4P_5P_6P_7P_8P_9) \rightarrow (P_1P_8P_3P_4P_5P_6P_7P_2P_9)$

3.7 Grid-oriented route planning

In this section we will discuss various route planning algorithms that are designed specially for the grid structured visit-points.

3.7.1 Criteria of optimality

We assume that we have a set of visiting points as a grid. The set is non-complete - we consider only points outlined by a field border -visiting points filled by black (see example in Figure 20). Thus presence of obstacles on the field means just ignoring of certain points, been covered by obstacles - grey oval inside the field. We are not taking into consideration points outside the field border.

Having a grid layout might simplify our task. We will assume that the grid vertical and horisontal lines are perpendicular, any travel by the grid from one

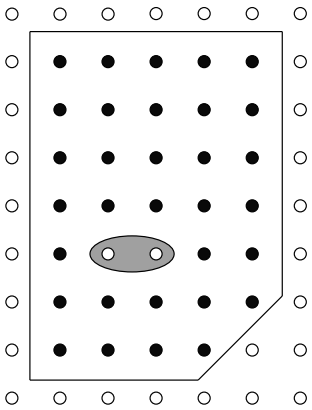


Figure 20: An example of a field.

node to the next will have the same length and only diagonal travel will be x1.42 times longer, or more precise: multiplied by $\sqrt{2}$. The agent can make only three possible moves between nearest nodes: vertical, horizontal or diagonal.

Consequentially our criteria for optimality will be minimal number of diagonal travels. Eventhough we assumed that an agent is a dot on the map, we will try to optimize the root in a way that it would have few of turns and majority of straight travels, which should lie along the rows of a crop. Crop damages can be avoid because AA should be used to observe fiels soon after sowing when crops are low and could recover easily afterwards.

Thus the optimal route among n points where l is distance between points will be:

$$S_{optimal} = n \cdot l$$

Real route to travel among the same number of points with k number of diagonal travels will be:

$$S_{real} = l \cdot (n - k) + k \cdot l \cdot \sqrt{2}$$

and optimality criteria is the devision of optimal rout to a real:

$$C_{opt} = \frac{S_{optimal}}{S_{real}} = \frac{n \cdot l}{l \cdot (n - k) + k \cdot l \cdot \sqrt{2}} = \frac{n}{n + k \cdot (\sqrt{2} - 1)}$$

Let us calculate the optimality criteria for 3 fields. Each field contains 10, 100, 1000 points respectively and each of fields contain 10 diagonal travels out of total points number (for the first field all travels are diagonal). And for that example

we say that distance between points is 1.

$$C_{opt}(10) = 0.707107$$

$$C_{opt}(100) = 0.96022$$

$$C_{opt}(1000) = 0.995875$$

The difference between $C_{opt}(10)$ and $C_{opt}(100)$ is only 0.035649 or 3.5%. That an algorithm results quite acceptable when about 10% of travels are diagonal.

Set up a possible algorithm non optimality value (e.g. 5% or 0.95 of optimal) we can use that criteria to check acceptability of a new solution (rout). That allows us to beginx applying the most simplest algorithms.

3.7.2 Grid-oriented heuristic algorithm

The idea of algorithm is to split the field into cells. We consider that cells could contain 1, 2, 3 or 4 points. Number of possible travels over the cell is 1, 2, 6 and 8 correspondingly (Figure 21).

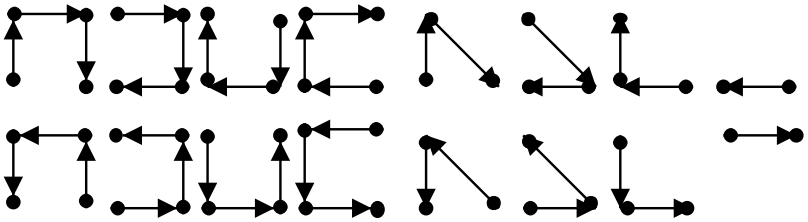


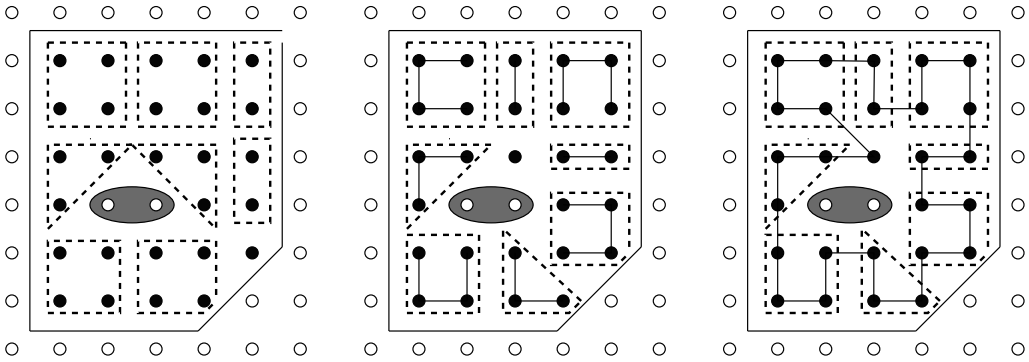
Figure 21: Possible travels over the 4, 3, and 2 -point cells.

Having that we can reduce the task to find the best combination of the following two tasks:

- Splitting into cells
- Travel over cells (or cell rotate position)
- Connection between cells

The lust item of that trio (connection between cells) could also contain diagonal connection. Actually that is almost the same as a Puzzle game.

The logic of splitting could be the following: we will try to split the field in the way that there would be no two- or one-point cell next to the field board. That is to prevent blocking of the way near a board or in a corner of the field. Considering



(a) An example of one possible splitting of field into cells.

(b) New splitting of field into cells with added connections inside cells.

(c) Connections among cells.

Figure 22: Field splitting into cells and connecting into a continuous route.

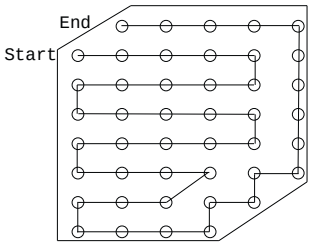


Figure 23: Line moving algorithm illustration.

that kind of field splitting (Figure 22(a)) the field is modified as on Figure 22(b). We can add connection inside cells as well.

Now we can make connections between cells in Figure 22(c).

As it can be seen in Figure 22(c) the route is very close to optimal because it is only one diagonal connection between cells.

3.7.3 Line moving algorithm

Let we have a field quite the same as we had in a previous example.

Step 1. At the start of the algorithm the shell-principle is used. We outline the whole field by the route including all shell points. Of cause we try to avoid diagonal travels. After that the field looks as on Figure 23.

Step 2. On each field there direction of a preferable movement which goes along

the crop raws. Lets assume in the current example that crops are planted in horizontal rows. The algorithm makes right-left travels starting from the longest-straight side. Travels are stretched to the left to the until obstacle or root covered point is reached.

After that we can check does that solution suites or not by using the above mentioned criteria of optimality.

4 Implementation

The Simulator *Javadoc*[2] documentation for central classes is found in the appendix floppy disc. We would like to discuss some main ideas in this implementation section, since the Javadoc documentation is split in many HTML files and is written for other developers.

We have added two separate packages to the Simulator application and some classes to the already existing $auc.field$ package (Figure 24):

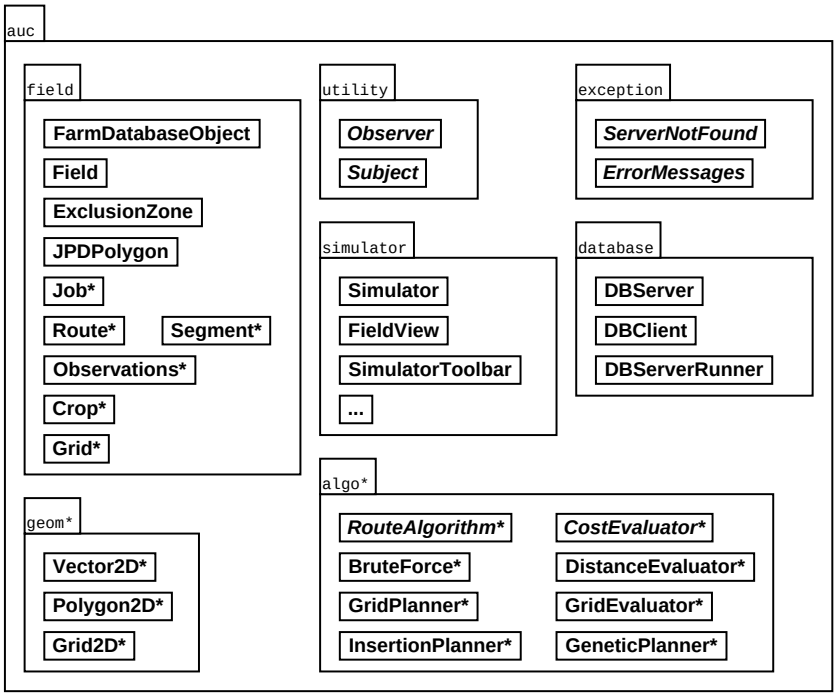


Figure 24: The main package: auc.geom.

geom has mathematical abstractions to simulate the geometrical properties of objects in the *field* package. Unfortunately the JDK[3] does not contain the geometry structures we need (the JDK Polygon class deals with integers and does not have sufficient operations).

algo contains algorithm related classes, which neither fit in the *field* nor in *geometry* packages, such as route planning algorithms and travel cost evaluation algorithms.

[2] A utility program by Sun corporation, which creates HTML pages representing the java class interface from specially formated comments in the class source.

[3] The Sun Java Standard Development Kit 1.3.1.

You may find the summary of the Simulator packages in Figure 24. The added packages and classes are marked with a star (∗).

4.1 The field package

The field package contains the data objects from the database. The summary is presented in Figure 25.

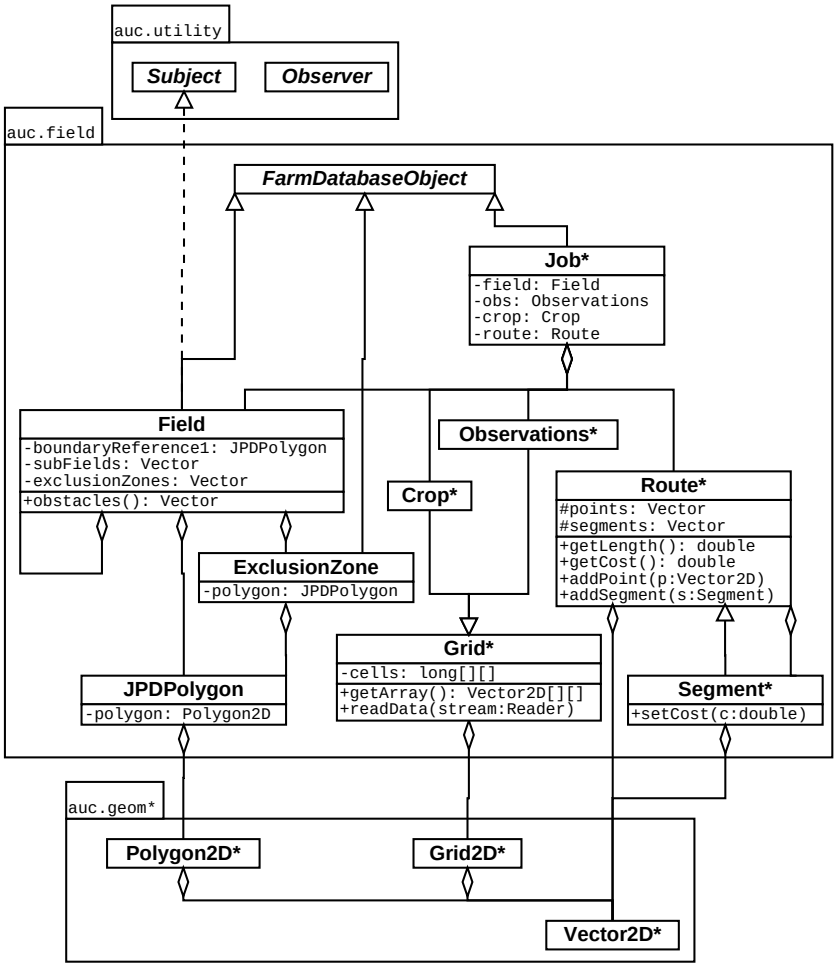


Figure 25: Field objects package: auc.field.

A *job* structure contains a lot of data associated with jobs in the farm (the planned schedule of tasks, the actual state of tasks and so on), but we are interested in data associated with observation tasks, i.e. the *field* structure reference, the data about observation points and we could also get the information about the current

plant structure in the given field. The observation points are given in the *grid* structure which is described in the grid file associated with a job.

The *grid* holds a set of points which are arranged in a certain order. In a field observation job the grid structure is used for two purposes:

- A grid holds geometrical properties of a crop (e.g. wheat, oats, ...) in a certain field: crop density which is the distance between parallel rows and direction of rows.
- A grid holds observation points which have to be visited according to a job. While forming this grid we should consider the properties of the crop grid unless there is a case of exclusion zone avoidance.

4.2 The geometry package

Most of the calculations lies in the *auc.geom* package (Figure 26). We have implemented the *Vector2D* class to represent a mathematical vector[4], which has all needed operations for vector algebra and is very handy to represent a point.

The *Polygon2D* class uses *Vector2D* to store its vertices and almost all methods in this class manipulates the vectorial information in vertices:

addVertex adds a new vertex to the polygon.

getVertexIterator returns a vertex iterator to access each vertex one-by-one. It is used in vector transformation before painting on the screen and for determining the boundaries of the *grid*.

getVertexCount is used to get the count of vertices, e.g. when the size of the vertices storage is needed to know in advance.

convex returns the minimal convex polygon of this polygon. This method creates a copy of the polygon and takes away the concavities as described in section 3.4.

concavities returns an array of all concavities met in this polygon (section 3.4).

contains decides whether the given point is inside the polygon (section 3.2.2).

intersects decides whether the given segment intersects the polygon.

intersection decides the intersection point of two given segments.

[4]Note that Vector2D is a completely different utility than java.util.Vector in JDK.

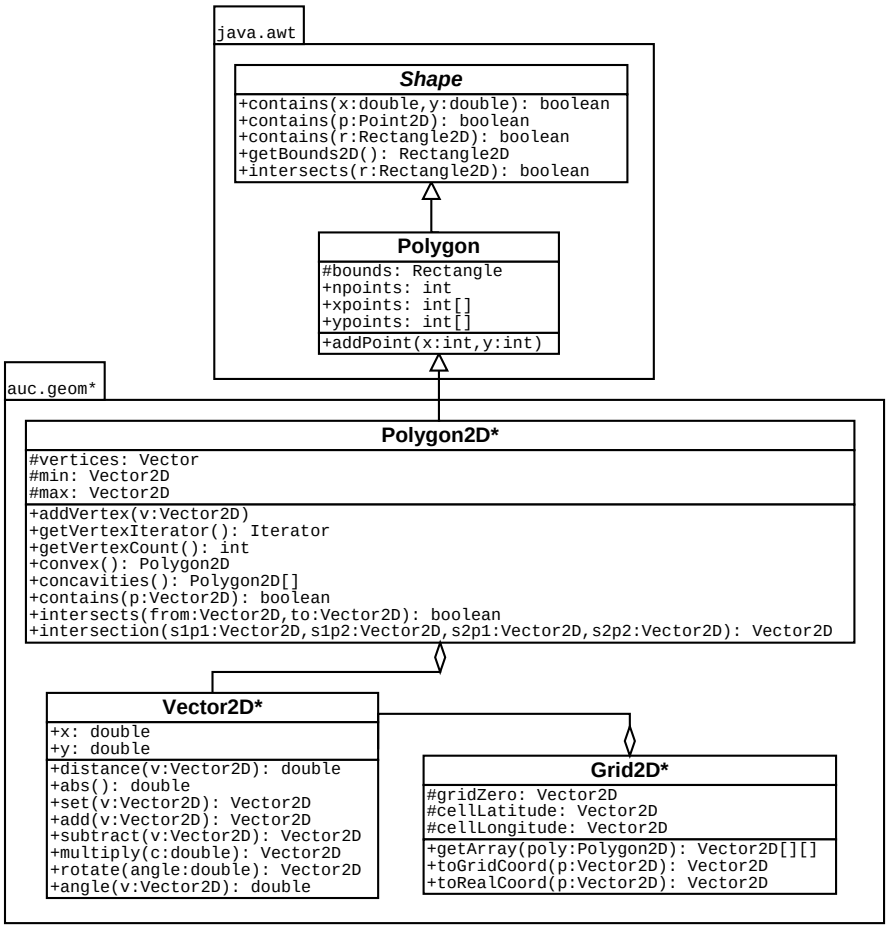


Figure 26: Geometry objects package: auc.geom.

The *Grid2D* class represents abstract infinite mathematical grid (*toGridCoord* and *toRealCoord* coordinates transformation methods) and can calculate all the points which are limited by the given polygon (*getArray* method). However the *Grid2d* parameters (*gridZero*, *cellLatitude* and *cellLogitude*) a stored as *Vector2D* objects too.

4.3 The algorithms package

The *auc.algo* package contains all route planning related algorithms. The package organization is based on the route algorithm and cost evaluation sub-algorithm idea.

The sub-algorithm (we call it the *CostEvaluator*) calculates the travel cost from one point to another and returns the evaluated segment of the route which

may contain additional points to avoid the exclusion zones.

All route planning algorithms implement the *RouteAlgorithm* interface (figure 27). The *RouteAlgorithm* interface requires two methods to be implemented:

planRoute calculates the route from the given *Job* data.

setCostEvaluator gives an opportunity to select other than the default cost evaluator for the algorithm (any of algorithms can have its own cost evaluator).

So far we have implemented four different algorithms:

BruteForce is described in section 3.6.1. It is the only one that uses the *DistanceEvaluator* which gives the cost equal to the length of the route and does not care about the exclusion zone intersection. The latter algorithms use the *GridEvaluator*, which takes into account the grid structure of the visit points and avoids the exclusion zones if any.

InsertionsPlanner is described in section 3.6.2.

GridPlanner is described in section 3.7.3.

GeneticPlanner is described in section 3.6.3.

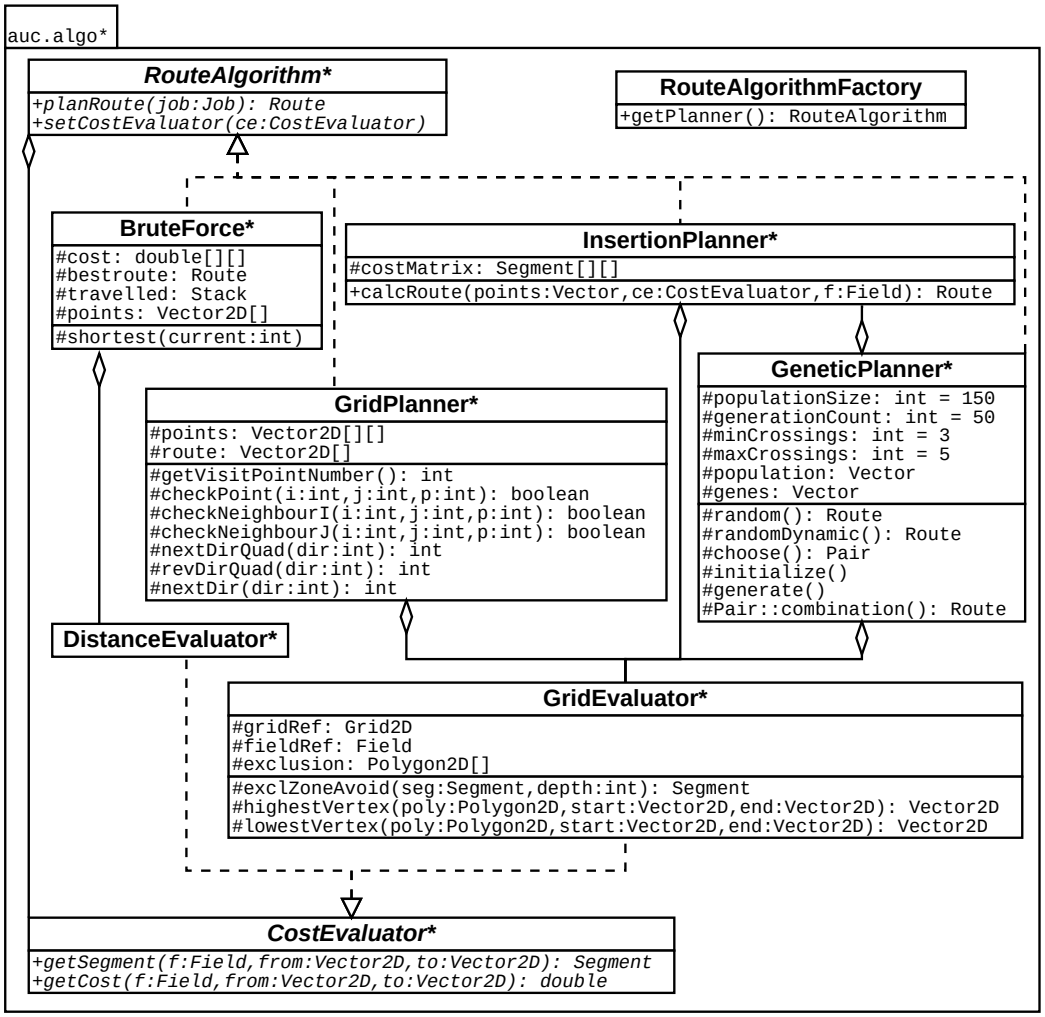


Figure 27: Algorithms objects package: auc.algo.

5 Testing

The testing section consists of test specifications and test reports. We will test the grid generator, the grid filtering function and the route planning algorithms. At the end of testing we summarize the faults found.

5.1 Test specifications

There are several software testing strategies [9]:

1. *Unit testing* includes an indication of the components that will undergo unit

tests or the criteria to be used to select components for unit test. Usually white box testing techniques are used in this strategy..

2. *Integration testing* includes a discussion of the order of integration by software function. Dual problems of verification and program construction. There are used black box and some white box testing techniques.

3. *Validation testing* testing of validation criteria (established during requirements analysis). Black box testing techniques are used.

4. *System Testing* considers integration of software with other system components.

There are several testing techniques but the main, mentioned above are [9]:

1. White box testing - testing control structures of a procedural design (ensure that all independent paths are exercised at least once, all logical decisions are exercised for both true and false paths, all loops are executed at their boundaries and within operational bounds, all internal data structures are exercised to ensure validity).

2. Black box testing focuses on functional requirements and input/output relations. It compares expected and actual result but don't know control flow structure. Black box testing attempts to find incorrect or missing functions, interface errors, errors in data structures or external database, access performance errors, initialisation and termination errors).

We will use both techniques in our project.

For test specification we will also define what data (tools) are used for testing, what is the target of a test and what is the final product of the test and what are requirements for every different testing case. Results are recorded into the testing report.

5.1.1 Test specification for the grid generator

The goal is to test the grid generator, which transforms coordinates of a vector from coordinates of Cartesian plane to the coordinate system of the grid and vice versa.

The scope of the test is to find out whether a transformation of the vector is performed correctly. Vector's coordinates might be negative; depending in which quadrant of the Cartesian plane the vector is located. The behavior of the grid generator is tested considering vectors with coordinates from different quadrants.

Testing strategy: the unit testing is used. We have two classes Grid2D and Vector2D for testing the grid generator, which is integrated into the simulator.

Grid 2D deals with the plane, which is, arranged to the grid coordinates which are measured by units (a cell length and width). Vector 2D holds Cartesian coordinates which are measured in meters.

Testing data:

- Simulator version 2001119;
- Observation point grid generator module version 2;
- Sample grid;
- Sample vector.

Test target: coordinate transformation function.

Test product: vector coordinates after transformation from Cartesian plane to the grid coordinate system and back.

Test instructions: procedure for testing cases.

Table 1: Test instructions testing the grid generator

Actions	**Expected result**
1. Preparation 1.1. Create a sample grid in the plane with orthogonal axes (parallel to Cartesian axes). 1.2. Create a sample vector in Cartesian plane.	Simulator application prepared. Sample grid object is created. A vector object in Descartes coordinates is created (v1).
2. Execution 2.1. Convert the coordinates of the vector to the coordinates of the grid coordinate system. 2.2. Convert to the vector into Cartesian coordinates 2.3. Compare coordinates of the first sample vector to the coordinates of the last sample vector.	– New sample vector in grid coordinates (v2) is created. New sample vector in Cartesian coordinates is created (v3). Coordinates of the vector before transformation and after two transformations should be the same.
3. Record results of the case	record.

Test cases:

1. A sample grid is orthogonal and coordinates are proportional to the field coordinates. A sample vector coordinates must be from four different quadrants of the grid coordinate system:
   (a) from the quadrant I;
   (b) from the quadrant II;
   (c) from the quadrant III;
   (d) from the quadrant IV.
2. Axes of a sample grid are non-orthogonal and shifted from the axes of the field plane. A sample vector coordinates must be from four different quadrants of the grid coordinate system:
   (a) from the quadrant I;
   (b) from the quadrant II;
   (c) from the quadrant III;
   (d) from the quadrant IV.

Test results of the test cases are recorded into the report summary below 5.2.1

5.1.2 Test specification for grid filtering

Testing strategy:

- Unit testing. We check how the grid filtering function works, which is in the Grid class.
- Validation testing. We check whether filtering function filters the grid points according to the requirements for the grid (section 2.4).

Test data:

- sample field structure;
- sample grid of the crop structure.

Test target: Grid point filtering function in the Grid class.

Test product: Field fully covered with the grid. No grid outside the field or inside the obstacles.

Test instructions: the test is based on the measurements and visual testing.

Table 2: Test instructions for testing grid filtering

Actions	**Expected result**
1. Preparation	–
1.1. Configure the field structure	The coordinates of the sample fields, obstacles, crop grid structure are set.
1.2. Start JAVA applet	Applet opened. A field, subfields, obstacles, crop-grid is drawn.
2. Execution	–
2.1. Put the grid for a certain job	The grid for observation has been set at once or point by point.
2.1. Check visually all field reachable places	Reachable places should be covered by a crop or observation point grid.
2.2. Check visually all exclusion zones	Exclusion zones should not contain any crop nor observation point.

Test cases:

- A field has different configuration;
- A grid has different coordinates.

Results of the test cases are put in the report summary in the section 5.2.2

5.1.3 Test specification for route planning algorithms

The goal is to test the perfomance of the route planning algorithms.

The scope of the test is to find out how the algorithms plan the route in the simple fields and in more complex fields. Complexity of the fields will be described step by step.

Testing strategy:

- Unit testing includes classes of different algorithms and checks for their performance.
- Validation testing tests what is the relation between a certain input (complex structure of the field) and the output (the length of a route, computation time, visual checking of how the algorithms deal with obstacles and concavities of fields).

Testing data:

- sample field structure;
- sample grid structure.

Testing resources:

- Hardware: AMD Athlon(tm) 1.333GHz, 512MB RAM.
- Software: MS Windows 2000 professional SP2, MS Access 2000, Sun JAVA Development Kit 1.3.1_01, Simulator application build 20011119.

Test target: one of the route planning algorithms:

- Brute-force;
- Heuristic Insertion algorithm;
- Grid-planner algorithm;
- Heuristic Genetic algorithm;
- exclusion zone avoidance algorithm.

Test product: route with certain requirements which depend on the complexity of the field:

- minimal length of the route;
- time for producing the route;
- shortest way around the obstacle;
- validate crossing of the obstacle (Y/N);
- validate crossing of concavity (Y/N);
- validate going outside the field (Y/N).

Test instructions: the test is based on the visual testing.

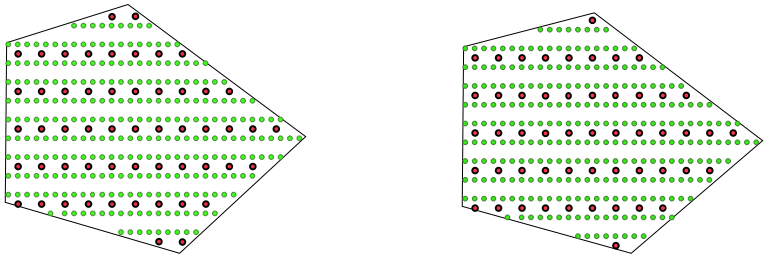
Test cases: For testing a new case change the structure of the field and repeat from step 2.1. After testing one algorithm take another algorithm and repeat from step 2.1 for all testing cases. Testing cases with pictures are listed below.

1. Simple field without exclusions:
   (a) with a rectangular grid (Figure 28(a)).
   (b) when the rectangular grid has a row with a single point (Figure 28(b)).

Table 3: Test instructions for an algorithm testing

Actions	**Expected result**
1. Preparation	–
1.1. Load data in the Field Star database	Data for field, obstacle structure and job is loaded.
1.2. Start Simulator	Simulator interface window is opened.
2. Execution	–
2.1. Select a field structure	Observation job selected and a certain field from the test cases is drawn.
2.1. Run a certain algorithm for route planning	Observation points are connected by a route in the Simulator window.
2.2. Check the data which evaluates the performance of the algorithm:	checked.
2.2.1. The length of the route	record (m).
2.2.2. The time that algorithm took to make the route	record time (sec).
2.2.3. Validate if the distance avoiding an exclusion zone between two points is the shortest	validate.
2.2.4. Validate crossing of an exclusion zone (Y/N)	validate.
2.2.5 Validate crossing of concavity (Y/N)	validate.
2.2.6. Validate going outside the field (Y/N)	validate.
3. Record the results of the test case	recorded.

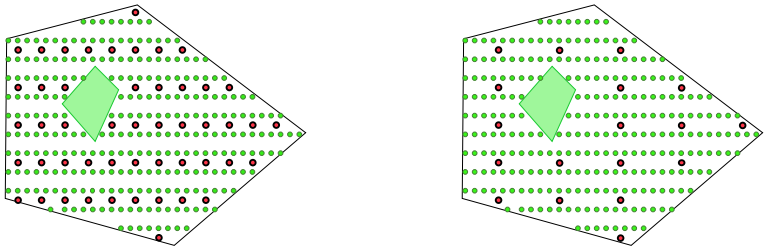
2. The field with one exclusion zone (Figure 29(a)).
3. The field with an exclusion zone and different structure of the rectangular grid (Figure 29(b)).
4. Complex shaped field:
   (a) Concave shape with one concavity (Figure 30(a)).
   (b) Concavity with multiconcave shape (Figure 30(b)).
   (c) Very narrow shape of the concavity (Figure 30(c)).
5. Complex shape of exclusion zone:
   (a) Concave shape of the exclusion zone (Figure 31(a)).
   (b) Several exclusion zones in the field (Figure 31(b)).



(a) A simple field without exclusion zones.

(b) Grid with a single point in a row.

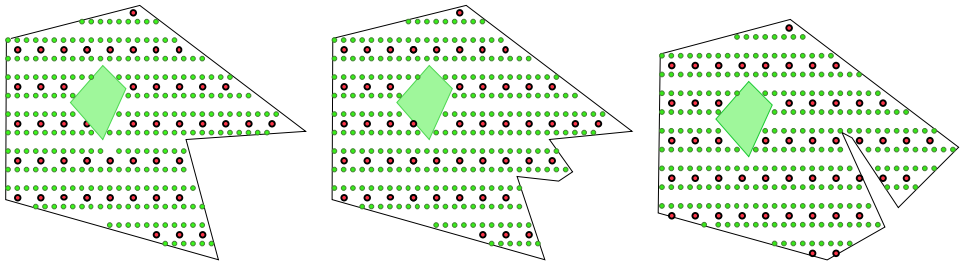
Figure 28: Cases of the simple fields.



(a) A field with one exclusion zone.

(b) A field with one exclusion zone and different structure of rectangular grid.

Figure 29: Cases of the simple fields.

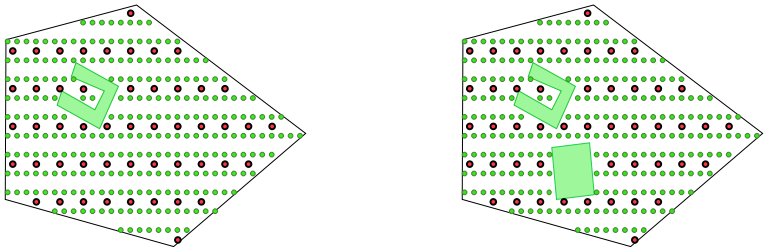


(a) Concave shape with one concavity.

(b) Multiconcavity of the field.

(c) A field with a narrow concavity.

Figure 30: Cases of the complex-shaped fields.

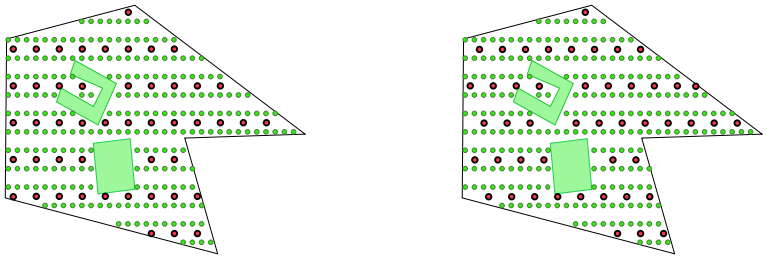


(a) Concave shape of the exclusion zone.

(b) A field with several exclusion zones.

Figure 31: Cases of complex-shaped exclusion zones.

6. Complex shape of the field and of exclusion zones (Figure 32(a))
7. Complex grid structure (Figure 32(b)).



(a) A complex field.

(b) A field with non-rectangular observation grid.

Figure 32: Cases of a complex field and grid.

Pictures contain crop-grid structure (thicker rows of dots) which has not been implemented in the simulator yet, but it was considered to be in the future versions of the simulator.

Results of the test cases are put in the report (section 5.2.3).

5.2 Test reports

5.2.1 Test report for the grid generator

Table 4: A sumarry test report for grid generator

Case ID	Sample vectors	Expected vector	Result
Grid coordinates(0, 0; 10, 0; 0, 10)			
QI	v1(15.0, 5.0) v2(1.5, 0.5) v3(15.0, 5.0)	v3(15.0, 5.0)	pass
QII	v1(-15.0, 5.0) v2(-1.5, 0.5) v3(-15.0, 5.0)	v3(-15.0, 5.0)	pass
QIII	v1(15.0, -5.0) v2(1.5, -0.5) v3(15.0, -5.0)	v3(15.0, -5.0)	pass
QIV	v1(-15.0, -5.0) v2(-1.5, -0.5) v3(-15.0, -5.0)	v3(-15.0, -5.0)	pass
Grid coordinates(0, 0; 10, 5; 5, 5)			
QI	v1(15.0, 5.0) v2(2.0, -1.0) v3(15.0, 5.0)	v3(15.0, 5.0)	pass
QII	v1(-15.0, 5.0) v2(-4.0, 5.0) v3(-15.0, 5.0)	v3(-15.0, 5.0)	pass
QIII	v1(15.0, -5.0) v2(4.0, -5.0) v3(15.0, -5.0)	v3(15.0, -5.0)	pass
QIV	v1(-15.0, -5.0) v2(-2.0, 1.0) v3(-15.0, -5.0)	v3(-15.0, -5.0)	pass

The test results were rounded with a precision of 10^{-14}. Results with that deviation are acceptible in both computer computation and the farming technologies. We conclude that the grid generator makes transformations from the Cartesian coordinates to the grid coordinates and back properly.

5.2.2 Test report for the grid filtering

Figure 33 contains a snapshot from the test window and test results are in the table below.

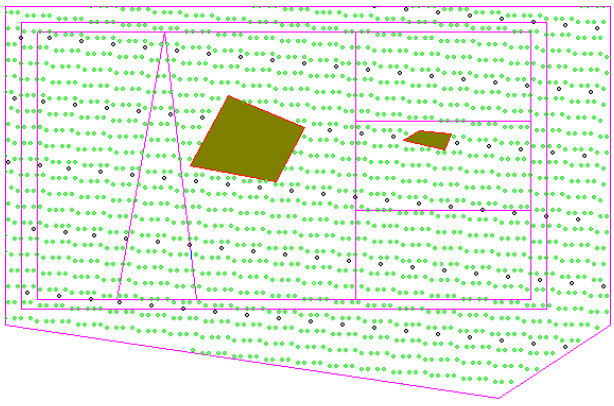


Figure 33: Testing the grid filtering function.

Table 5: A sumarry test report for grid filtering

Actions	**Result**
1. Preparation	–
1.1. Configure the field structure	passed
1.2. Start JAVA applet	passed
2. Execution	–
2.1. Put the grid for a certain job	passed
2.1. Check visually all field reachable places	passed
2.2. Check visually all exclusion zones	passed

We can conclude that the job-grid filtering function works properly.

5.2.3 Test report for the route planning algorithms

Testing Grid Planner Algorithm

Table 6: A test report for Grid-planner Algorithm

Test case	**Length (m)**	**Time (sec)**	**Cross EZ (Y/N)**	**Cross concav (Y/N)**	**Outside (Y/N)**	**Result**
Case 1a)	781.06	0.05	-	-	No	pass
Case 1b)	...	no response	-	-	...	fail
Case 2	790.26	0.05	No	-	No	pass
Case 3	...	no response	-	-	...	fail
Case 4a)	675.78	0.05	No	No	No	pass
Case 4b)	614.21	0.04	No	No	No	pass
Case 4c)	792.25	0.04	No	No	No	pass
Case 5a)	798.46	0.03	Yes	-	No	fail
Case 5b)	801.35	0.08	Yes	-	No	fail
Case 6	691.34	0.03	Yes	No	No	fail
Case 7	...	no response	...	...	...	fail

The Grid-planner algorithm gives very good results considering the time and the route length in the cases where it gives results. We found out that it didn't work in the case where there is a single point left in a row. We treat this problem as a failure in the source code.

Testing Heuristic Insertion algorithm

Table 7: A test report for Heuristic Insertion algorithm

Test case	**Length (m)**	**Time (sec)**	**Cross EZ (Y/N)**	**Cross concav (Y/N)**	**Outside (Y/N)**	**Result**
Case 1a)	762.11	8.92	-	-	No	pass
Case 1b)	747.11	8.53	-	-	No	pass
Case 2	802.25	10.02	No	-	No	pass
Case 3	735.55	10.06	No	-	No	pass
Case 4a)	686.29	8.18	No	No	No	pass
Case 4b)	623.17	6.76	No	No	No	pass
Case 4c)	763.43	20.05	No	No	No	pass
Case 5a)	806.32	13.82	Yes	-	No	fail
Case 5b)	809.68	19.94	Yes	-	No	fail
Case 6	683.26	17.06	Yes	No	No	fail
Case 7	723.61	15.12	No	No	No	pass

Heuristic Insertion algorithm is our best solution if we consider that it passed majority of cases (Figure 7) . This algorithm computes a zig-zag route in non-orthogonal grid, which means that the route is not optimal. The resutls of Grid-Planner show that there is a shorter route in the same cases (2, 4a, 4b, 5a and 5b).

Testing Heuristic Genetic algorithm

In the beginning we set certain values for the Genetic algorithm parameters:

- The number of routes in the population - 120;
- A number of generations - 30.
- A number of children combined from a single pair of routes per generation - 20;
- Minimum and maximum number of crossings in the crossover mechanism - [4, 6].

Every pair of routes in the population makes a new generation of 20 children using crossover mechanism with 4, 5 or 6 crossings. Next population is formed by selecting routes from the parent and children generations. There are 30 generations formed during the testing procedure.

Table 8: A test report for Heuristic Genetic algorithm

Test case	**Length (m)**	**Time (sec)**	**Cross EZ (Y/N)**	**Cross concav (Y/N)**	**Outside (Y/N)**	**Result**
Case 1a)	2464.49	90.65	-	-	No	pass
Case 1b)	2194.61	89.76	-	-	-	pass
Case 2	2543.30	106.06	No	-	No	pass
Case 3	2361.90	98.84	No	-	No	pass
Case 4a)	2177.69	122.67	No	No	No	pass
Case 4b)	2382.20	149.68	No	No	No	pass
Case 4c)	2587.62	174.18	Yes	No	No	fail
Case 5a)	2609.34	144.84	Yes	-	No	fail
Case 5b)	2571.65	189.68	Yes	-	No	fail
Case 6	2192.68	186.85	Yes	No	No	fail
Case 7	2322.31	191.46	No	No	No	pass

The test results of the Heuristic Genetic algorithm shows (Figure 8) that the routes are several times longer than previous algorithms produced. Long routes are the result of many intersections. This algorithm also requires more time to evolve into a better solution. It depends a lot on the combination of crossover and selection parameters.

Brute Force algorithm

Test case	**Length (m)**	**Time (sec)**	**Cross EZ (Y/N)**	**Cross concav (Y/N)**	**Outside (Y/N)**	**Result**
Case 1	...	No response	...	...	...	...

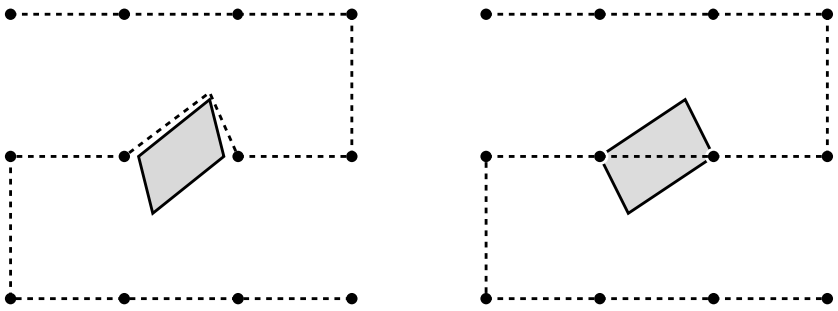
Table 9: A test report for Brute Force algorithm

Brute Force algorithm took far too much time to try even our simplest case. It is based on theoretical explanations but it is not efficient in the practical application.

5.3 Fault summary

Faults related to the exclusion zone avoidance:

1. The GridEvaluator finds a faulty way to avoid the exclusion zone when two neighbor grid points match the exclusion zone vertex points (Figure 34). The fault could be fixed by filtering out such points, but then the points on the field boundary will not be filtered out. In general the GridEvaluator should not use the exclusion zone vertex points to avoid it. The avoidance algorithm should choose a point which is a bit further from the exclusion zone, to make sure that the segment never contains points equal to any exclusion zone vertex.



(a) Proper exclusion zone avoidance.

(b) Faulty exclusion zone avoidance.

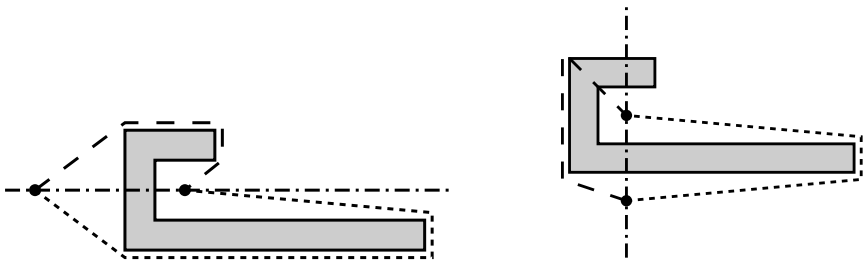
Figure 34: Concave exclusion zone avoidance.

2. The GridEvaluator finds a faulty way to avoid the concave exclusion zone: the route intersects with the exclusion zone (the algorithm chooses the thinner segment in Figure 35). The furthest visible point algorithm described in section 3.5.5 has a defect evaluating the exclusion zone vertex with a largest deviation angle from the direct segment.

3. The GridEvaluator finds a faulty way to avoid the concave concavity of the field (the algorithm chooses the thinner segment in Figure 36). It maybe be related to the previous fault, since the concavities are treated the same as exclusion zones.

The route planning algorithm related faults:

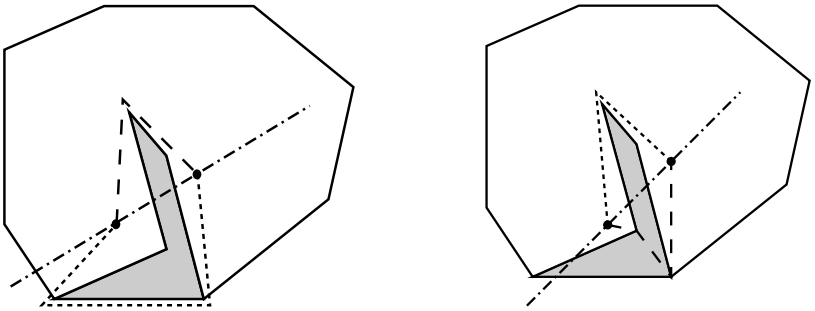
1. GridPlanner loops if some row or column of points in the grid contains just one point. The problem is that the algorithm does not foresee such situation and does not “know” where to go from that single point since the nearest neighbor is already in the route. The solution could be to visit the nearest neighbor point again.



(a) Proper concave exclusion zone avoidance.

(b) Faulty concave exclusion zone avoidance.

Figure 35: Concave exclusion zone avoidance.



(a) Proper concave concavity avoidance.

(b) Faulty concave concavity avoidance.

Figure 36: Concave field concavity avoidance.

2. GeneticPlanner computation takes a long time and the route still contains a lot of segment intersections. The optimal algorithm parameters should be investigated.

3. InsertionPlanner computed route contains many 5-point segment patterns with diagonal connections, which obviously are not optimal. The improvement could be to make a linear route patching, which detects the non-optimal segments with optimized ones (Figure 37). There are about 16 different 5-point patterns. After pattern replacement there can be a possibility to optimize S-shaped route patterns. Further investigation is needed.

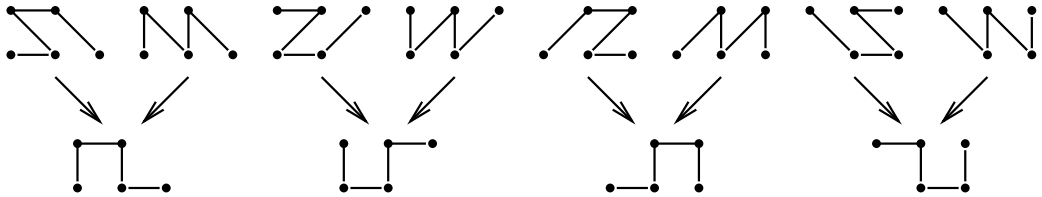


Figure 37: 5-point patterns in InsertionPlanner route.

6 Conclusion

We found out that only three route planning algorithms are feasible in practice. They are able to find the route among a large grid of points in considerable (polynomial) time, but their routes are not necessary optimal when complex exclusion zone configurations are given. Luckily there are not so many exclusion zones to avoid in a real field.

Unfortunately we have not solved the traveling salesman problem nor any other NP problem, but we have described six route planning algorithms and four exclusion zone avoidance algorithms. We have implemented four route planning algorithms and one exclusion zone avoidance algorithm. The route planning and the exclusion zone avoidance implementations are separated and can be changed if a better algorithm is found.

During testing we discovered several inadequacies which were described and solutions for correcting them were added.

There are several ideas that can be developed in future:

- The encountered faults in the exclusion zone avoidance algorithm and the route planning algorithms should be fixed in the future development.
- The route cost measure could include the cost of the plant damages since we did not have time to finish it.
- A route could include fuel filling and maintenance points. Those points could be stipulated by an operator or may be properties of a field or a group of fields. They could be also located outside the field.
- Adopt the route algorithm(s) for a multiple agent system.
- The future Simulator application may consider moving exclusion zones which could model a human or a cow walking in the field. For this purpose a radar or similar vision equipment should be simulated.

We had a good opportunity to see that robotics deals with numerous problems from the software systems point of view. Those problems are related with control over embedded systems, complex geometry structures and algorithm design and implementation. We encountered that precise solutions to the optimization problems in space usually have exponential complexity and are not practical to use.

During our project work we learned the main concepts of object oriented analysis and design, black and white box testing. Most of us learned object oriented programming principles with Java programming language. We gained much experience in writing a project report.

References

[1] Autonomous platform and information system for crop and weed monitoring http://www.cs.auc.dk/~api/.

[2] AGCO corporation project ”The Fieldstar Precision Farming system” http://www.Fieldstar.dk/.

[3] Lars Mathiasen, Andreas Munk-Madsen, Peter Axel Nielsen, Jan Stage. Object oriented analysis and design, 2000, Aalborg.

[4] Paul Burke. Geometry algorithms descriptions. http://astronomy.swin.edu.au/pbourke/geometry/.

[5] Robert Dakin. Traveling salesman problem heuristic solutions. http://www.pcug.org.au/~dakin/tspimp.htm.

[6] Rimvydas Krasauskas. Geometry, elements of linear algebra. http://www.mif.vu.lt/katedros/cs2/cagl/pdf/geo_12.pdf.

[7] Brian T. Luke. Overview of Evolutionary Programming Methods. http://members.aol.com/btluke/evprog.htm.

[8] Marek Obitko. Introduction to Genetic Algorithms. http://cs.felk.cvut.cz/~xobitko/ga/.

[9] Dr.Anthony Sobey. Software Testing Strategies. http://louisa.levels.unisa.edu.au/se1/testing-notes/test02.htm.

A Appendix - floppy disc

A floppy disc contains:

1. The Simulator application binary java classes.
2. The FieldStar sample database.
3. The source code of developed classes.
4. The Javadoc documentation of the Simulator classes.

Refer to README.TXT file to locate these items and INSTALL.TXT how to set up the Simulator application.