Extended Essay in Computer Science

Enhancement of the FAR Multi-agent Pathfinding Algorithm

Zbyněk Stara

Candidate Number: 000889-045

September 25, 2012

Word count: 3975

Abstract:

The FAR multi-agent pathfinding algorithm is one of the most current developments in the field of pathfinding. This study investigates the FAR algorithm and raises the question, “In what ways can the FAR algorithm be enhanced to provide better solution to the multi-agent pathfinding problem?”

Multi-agent pathfinding is in the focus of active research in Computer Science as it has no universal solution in terms of both the accuracy of the solution and speed of calculation. The problem is, in its most general sense, given as follows: In a defined solution space, navigate a set of agents from their starts to their respective goals, without them hitting each other or a defined set of obstacles. For that end, the FAR algorithm consists of three fundamental phases: First, the field is *flow-annotated*, second, the agents find their paths through the field, and third, the agents’ paths are coordinated, so that no pathfinding conflicts occur. This study identifies inefficiencies in all three stages of the original FAR method, and provides solutions to these, enhancing both the efficiency and simplicity of the algorithm, as well as the faithfulness to the original design.

Generalization of the flow annotation rule of reverse accessibility allowed for the development of a more concise code for the algorithm’s first stage, and eliminated the need for diagonal movement support. Furthermore, the change from Pythagorean to Manhattan heuristics has allowed the agents to follow as straight paths as possible and thus comply more with the FAR requirements set out in the original research paper. Finally, the inclusion of proxy paths instead of arbitrary step-aside commands, and their universal applicability, has allowed big portions of task-specific routines to be taken out from the algorithm in favor of a more complete general solution.

Word count: 291

Table of Contents:

1. Introduction:

The FAR multi-agent pathfinding algorithm is one of the most current developments in the field of pathfinding. This study investigates and implements the FAR algorithm and raises the question, "In what ways can the FAR algorithm be enhanced to provide better solution to the multi-agent pathfinding problem?" It strives to continue the line of questioning of the current approaches to pathfinding to provide a more suitable solution to the increasingly complex challenges of this discipline of Computer Science.

1.1. Pathfinding:

Pathfinding is one of the most challenging problems in Computer Science, and one that is extremely relevant to current real-life applications, robot motion planning the most prominent amongst them.

The problem is, in its most general sense, given as follows: Given a start node and a goal node in a defined solution space, find the shortest path connecting the two nodes. It is most commonly assumed that the solution space is known in its entirety to the pathfinding agents at the start of the simulation, and that the *passability* of a given node is a binary value – a node can only be passable or not.

1.2. Single-Agent Pathfinding:

Initially, the term *pathfinding* has been understood to be synonymous with the more specific challenge of *single-agent pathfinding*, the task of navigating *one* agent from assigned start to its goal.

The first such solution was the Dijkstra's algorithm, exploring the nodes of a field according to the distance from the start node until it reaches the goal. However, this solution was still very inefficient, mainly because the node exploration was *unfocused* –, thereby exploring too many unnecessary nodes.

A* algorithm, coming in the 1960s, solved this problem by using *heuristics* – the estimated distance to the goal – to concentrate the searching efforts of the algorithm along the ideal, straight route to the goal (Hart, Nils & Raphael)

1.3. Multi-Agent Pathfinding:

However, single-agent pathfinding algorithms are far from suitable to offer answers to a much more complex problem of *multi-agent pathfinding*, the challenge of navigating *multiple* agents through a single solution space.

As a way to solve this problem, a three-dimensional approach has been devised, enhancing the two dimensions of the solution space with a third, temporal dimension, as presented in the Cooperative Pathfinding – CA* – algorithm (Silver). Reservations are made of specific nodes at specific times, allowing them to be visited multiple times per simulation by different agents, but not by two of them at the same time.

Various extensions have been developed to enhance its performance, including the subject matter of this study, the Flow Annotation Replanning algorithm (Wang & Boten).

The issue of multi-agent pathfinding is in the focus of active research in Computer Science as it has no universal solution in terms of both the accuracy of the solution and speed of calculation. Different algorithms are proposed, usually trading off the optimality of the solution for the speed of the calculation.

2. FAR Algorithm:

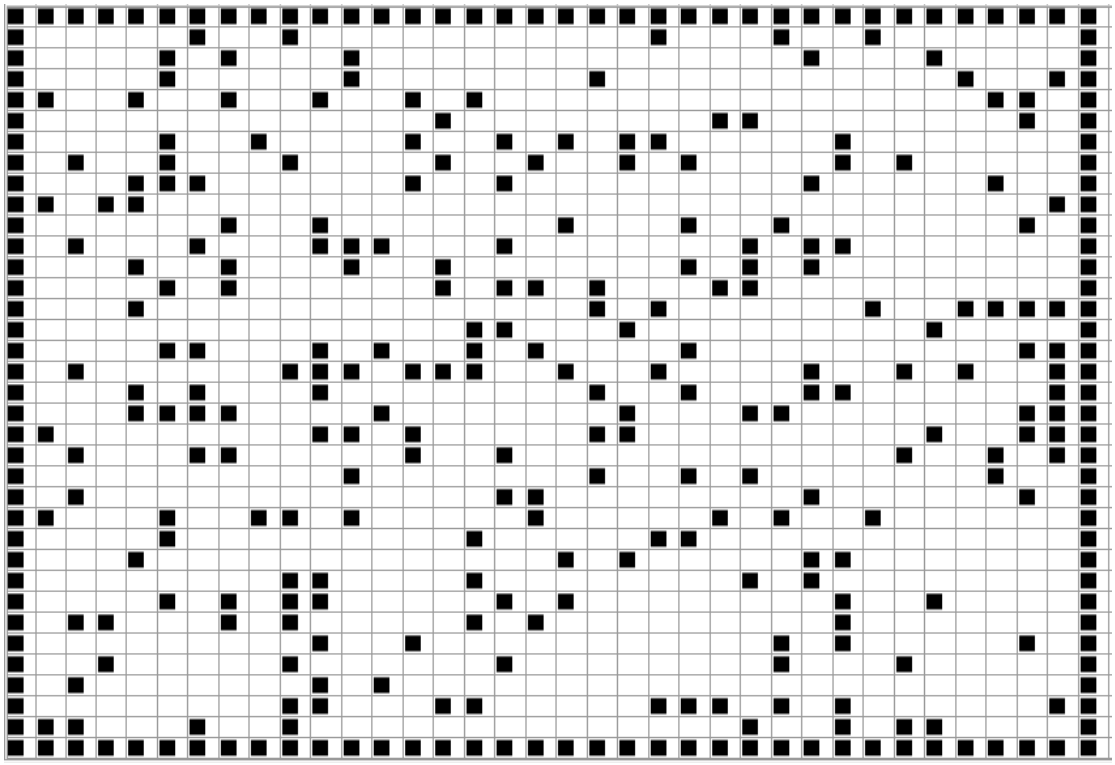
The Flow Annotation Replanning algorithm is an extension of the Cooperative Pathfinding multi-agent pathfinding algorithm. In turn, the CA*, as its acronym suggests, is itself an extension to the A* single-agent pathfinding solution. Therefore, the two multi-agent pathfinding algorithms are putting the principles of single-agent pathfinding into new contexts, and providing solutions to the challenges of multi-agent pathfinding.

2.1. Definition of Terms:

Pathfinding is the task of navigating a particular *agent* through the *solution space* avoiding *obstacles* and other agents along a *path* identified to be the shortest. It is the task of a pathfinding algorithm to identify such path. One attempt to do so for all of the agents is termed a *simulation*.

A *simulation* consists of several *time steps*, which add a temporal dimension to the simulation. There may be an arbitrary limit on the number of steps for which to reserve, this is called the *failure criterion*; if an agent has not visited its goal node by that time step, it is considered a failure (Wang & Boten).

The *solution space* is the set of all *nodes* of the simulation. It is important to note that, while, in general, solution space could take on the form of any set of nodes with defined *edges* – that is to say, connections to other nodes –, for the purposes of this research it has been implemented as a grid map, which is arguably the standard adopted by most researchers (Wang & Boten, Silver). This means that the solution space is represented as a two-dimensional array of square nodes, and lends space for defining more focused terms, for clarity. Therefore, the solution space represented as a grid map is termed a *field* and a node of a field is called an *element*.



Screenshot 1 **Field of 36x36 elements. Black squares show blocked elements (obstacles and boundaries).**

Element is the core of the grid map concept. It can either be *traversable*, able to be accessed or moved through, or *impassable*, as is the case with obstacles. If it is traversable, it has a definite set of *neighbor elements*. *Neighbors* are traversable elements surrounding the element, excluding those closed off by diagonal walls of obstacles. However, not all of the remaining edges have to be traversable in the direction from the element, or traversable at all, resulting in a smaller pool of options for movement out of a given element. Edges can be both *bidirectional* – possible to be traversed from both ends equally – or *unidirectional* – only traversable in one way .

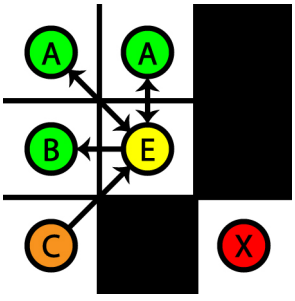


Figure 1 **A, B, and C are neighbor elements of E**
X is not a neighbor element as the edge between E and X is not traversable
The edges between A and E are bidirectional
Edges between B/C and E are unidirectional

An *agent* is one pathfinding unit; it starts its *path* at a given start element and ends it at a given goal element. The agent makes *reservations* of elements along its path at different steps of the simulation; this is to signal that that element is requested and, if no reservation *conflict* occurs, will be *occupied* by that agent at the following time step. If the agent successfully reserved an element distinct from the current element, it is said to have *moved*; otherwise, the agent has *waited*.

2.2. The FAR Method:

The FAR algorithm consists of three fundamental phases: First, the field is *flow-annotated*, second, the agents find their paths through the field, and third, the agents' paths are coordinated, so that no pathfinding conflicts occur. This section describes the original FAR algorithm, in the form it was presented in (Wang & Boten).

The proposed enhancements to the algorithm found by this study are presented in the section 3 of this paper, and they answer the research question "In what ways can the FAR algorithm be enhanced to provide better solution to the multi-agent pathfinding problem?"

2.2.1. Flow Annotation:

The flow annotation phase is perhaps the most fundamental to FAR and, certainly its biggest contribution to the multi-agent pathfinding approach. It transforms the way edges are marked in the field. In the standard A* – and consequently, the CA* – all edges in the field are bidirectional, which gives a lot of freedom to the pathfinding algorithm but is not very good for the coordination stage that comes afterwards.

Consider the following example (Figure 2): There is a significant cluster of obstacles in the center of the field and it is almost certain that some agents would need to find their path around that cluster. And without any constraints as to where they can plan to go, they are going to plan the shortest path, that is, the path right around the obstacles. But if all the agents do so, there is going to be a very high number of head-on collisions around the cluster which would be hard to coordinate – and a lot of idle space elsewhere.

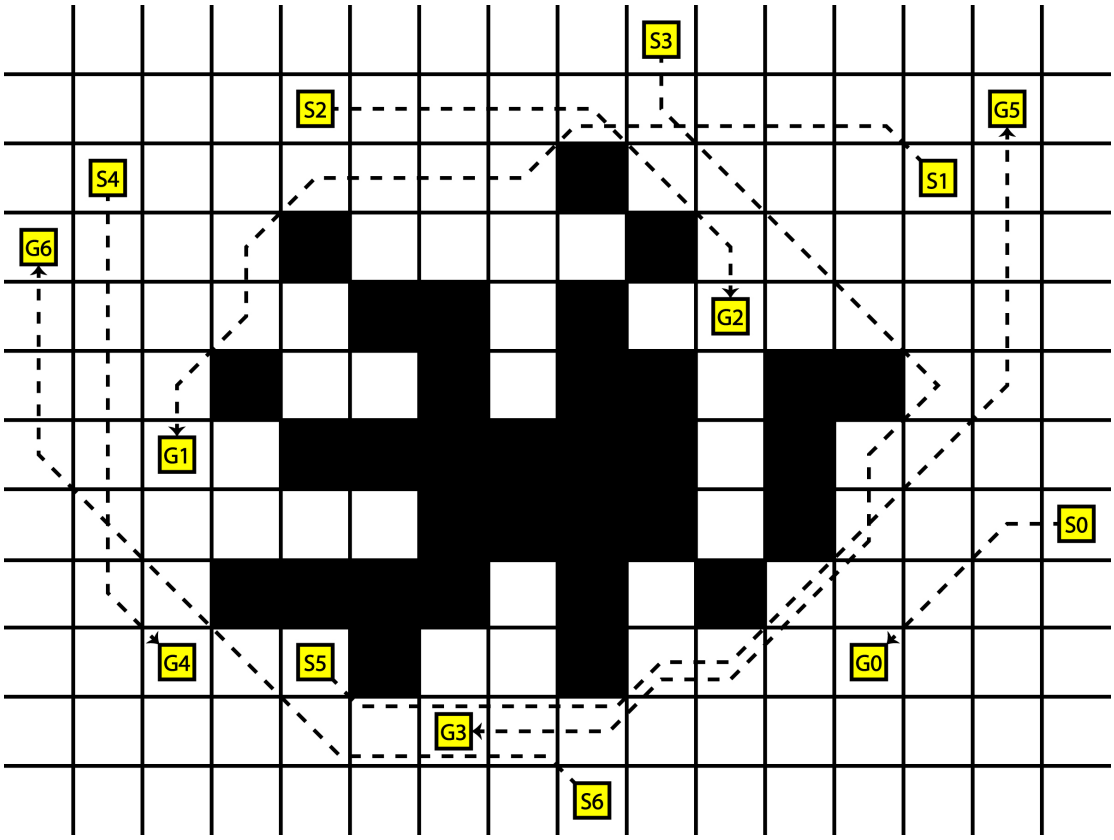


Figure 2 Cluster of obstacles in the center. S squares are starts; G squares are goals.

The original FAR algorithm strives to consider such problems beforehand, in its flow annotation phase. In a method "inspired by two-way roads" (Wang & Boten 1), it only marks edges as traversable in one direction in a particular column or row of the field. Moreover, the neighboring rows and columns have always the opposite direction, producing a structure of "lanes" (Wang & Boten 1), as on two-way roads.

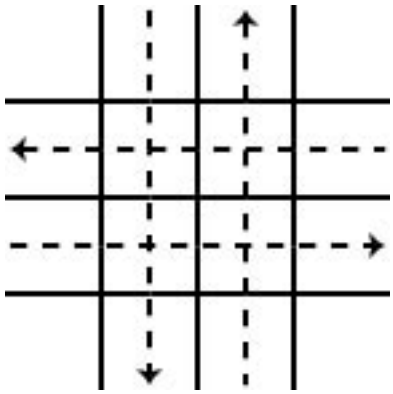


Figure 3 Structure of alternating lanes

There are additional rules that are applied after the initial markup, which are to guarantee that any element is still accessible in both directions. Not only it must be

possible for the elements to have at least one outgoing and one incoming flow; in order to ensure that connectivity is maintained in the whole field, there needs to be a simple *reverse* path accessible for any two orthogonally neighboring elements. In other words, when an agent travels from element A to element B in an orthogonal direction, it needs to have a simple path available to get back to A from B should that be needed. There are two possible reverse paths in general, which are shown on the following diagram:

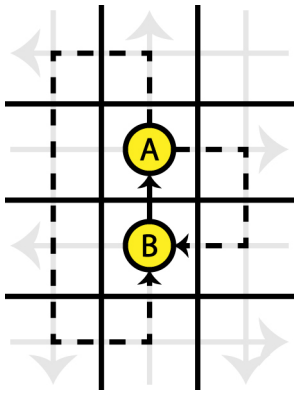


Figure 4 To ensure reverse connectivity between A and B, at least one of the dashed paths must be free of obstacles

If both of the two possible reverse paths, shown in Figure 4, are blocked by obstacles, reverse connectivity is ensured by inserting a bidirectional edge between the two elements in question. If this is ensured in the whole field, it is guaranteed that short proxy paths for any eventuality can be found.

Therefore, if these minimal requirements are met, all of the field is usefully flow-annotated, and the algorithm may progress to another phase, which is concerned with agents' pathfinding. For comparison, we can see the obstacle cluster example in the following diagram:

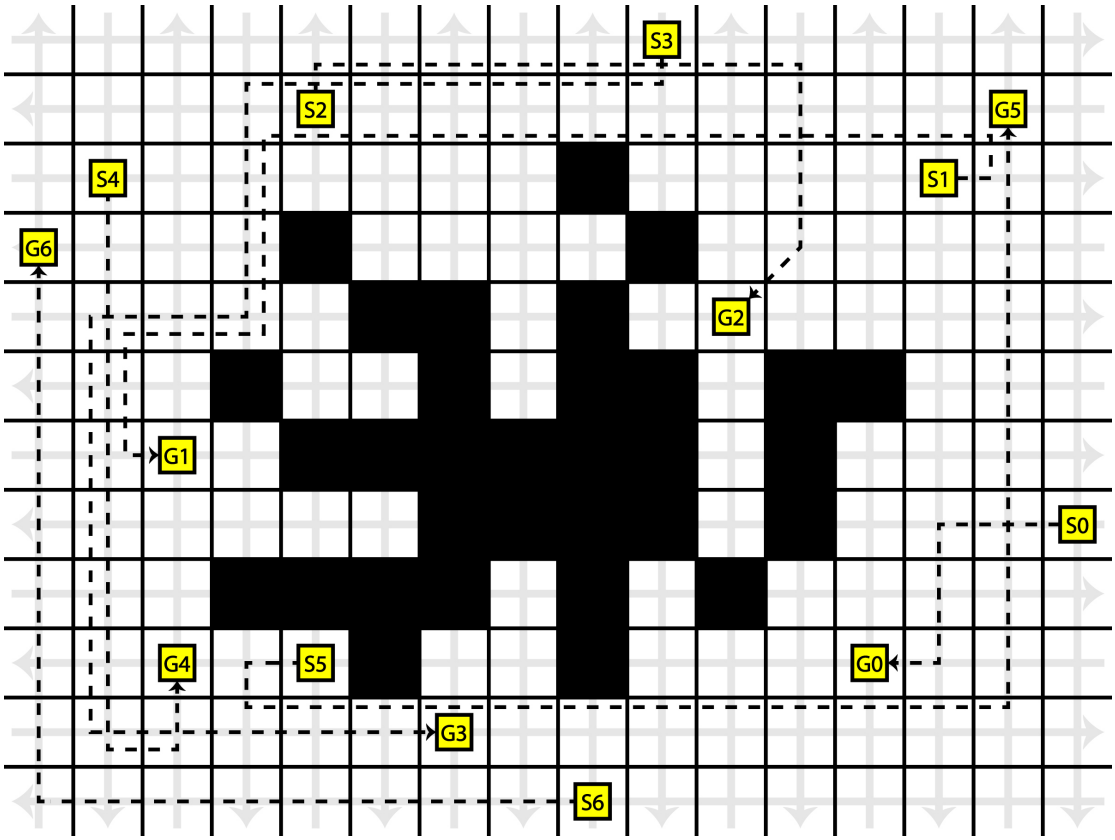


Figure 5 The obstacle cluster example after FAR annotation. Note the absence of any head-on crashes

2.2.2. Pathfinding:

The pathfinding algorithm used in original FAR is a standard A*, but with one important addition of a tie-breaking procedure favoring paths that continue in the current direction of motion (Wang & Boten 4).

The A* algorithm is described in detail in (Hart, Nils & Raphael). It works as follows: The algorithm keeps two sets of elements – one, the *open set*, stores elements which were previously found to be accessible, but were not examined yet; while the *closed set* stores elements which were already examined. The open set initially contains only the start element; the closed set starts empty.

The algorithm uses *f-score* to determine how promising does the element look in terms of offering the best path. The *f-score* is calculated as the sum of the cost of moving to the element along the shortest known path (*g-score*) and the heuristic estimate of the cost of travel to the goal (*h-score*).

The original A* makes use of the *Pythagorean heuristics*: the expected distance to goal is the length of a hypotenuse of a triangle with sides of the x- and y-

distance between current element and goal. This function is explained on the following diagram:

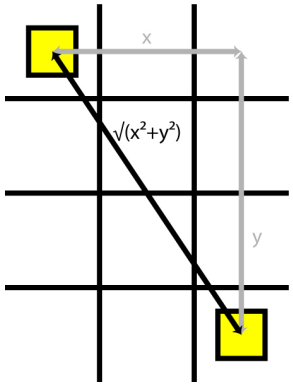


Figure 6 Pythagorean heuristics

The algorithm itself operates in a loop – it takes the lowest f-score element from the open set, and retrieves all of its accessible neighbors. For all neighbors not in the closed set, the g-score is updated with a value calculated from going to current neighbor through the currently investigated element – if the new g-score is lower than what it used to be. If the element was not in the open set yet, its h-score is also calculated, and the agent is added to the open set according to its f-score.

The loop reiterates until the goal element is chosen from the open set as the investigated element. This means that any further investigation of elements is unnecessary since there is simply no element that could possibly have a lower f-score. (This is true if and only if a consistent heuristics function is used – and the Pythagorean heuristics is consistent.) At the very end, the path is reconstructed. The algorithm accesses the goal element, and then retrieves the sequence of predecessor elements, adding each to the front of a *path* list. The path is then returned as the outcome of the algorithm.

2.2.3. Coordination:

The original FAR solution included path coordination in the following way: Every unit reserves a certain number of elements ahead of his path – this value is referred to as the *reservation depth* of the simulation. This means that the agent's coordination algorithm accesses the element and the agent is saved into a *reservation slot*, which corresponds to how many steps ahead in time the agent is reserving.

For example, if the reservation depth of the simulation is 3 – "a good value that was empirically found" (Wang & Boten 6) in the experiments on the original FAR –, there have to be 3 reservation slots for each element. Each agent then posts reservations to the reservation slots corresponding to which element of its path it is reserving.

The agent can post a reservation to an element's reservation slot only if it has successfully reserved a reservation slot on the preceding element of its path on the preceding time step. If a reservation conflict occurs at an element, with two agents trying to reserve the same node, the flows alternate, much like the "traffic lights at road intersections" (Wang & Boten 4).

If a unit reaches its goal destination before the failure criterion step, it is deemed a success. However, it still needs to take part in the coordination process. It waits on its destination element unless it blocks the path of another agent. In that case, in the original FAR, the waiting agent is required to "take a step away from the target" (Wang & Boten 5) and allow the passage of the blocked agent. It then calls the A* algorithm to find a way back to its goal, still coordinating with other units.

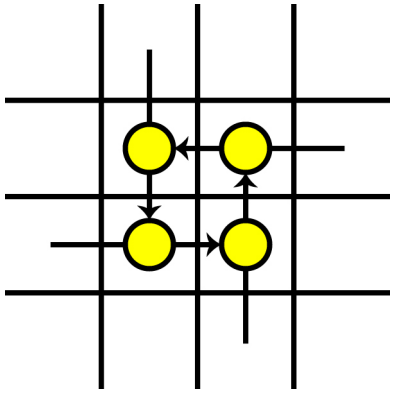


Figure 7 Deadlock

The original FAR discusses also another situation, the so-called *deadlock* (four units waiting for each other in a cycle).

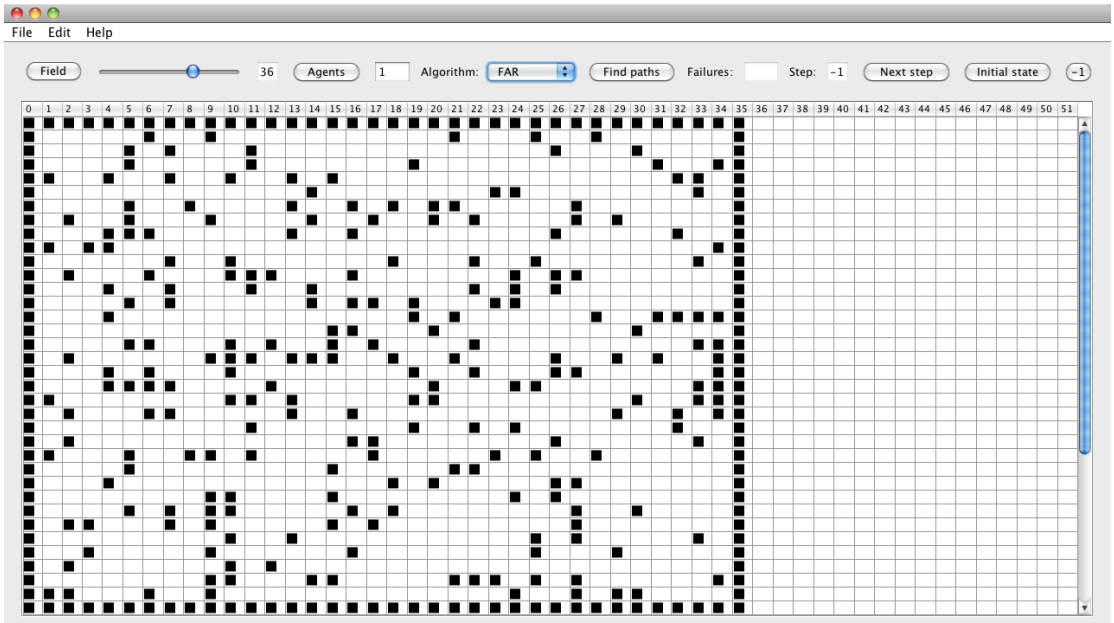
Deadlocks have to checked for every time an agent finds its path is blocked by another agent. The problem is solved by making the *critical agent* of the cycle to move aside, allowing the other deadlocked units to pass.

3. Enhancements:

The question is, then, how can the FAR algorithm be enhanced to provide better solution to the multi-agent pathfinding problem? This section provides the answers found by practical implementation of the algorithm and successive investigation of the areas identified by this study to cause problems and inefficiencies.

3.1. This Study's Implementation:

The algorithm was investigated using a visualizing program, which was specifically developed by the author of this paper. (Complete code is included as an appendix to this research paper.) The program proved useful in visualizing the paths of the agents at a given time step and modeling the influence of algorithm changes to the output.



Screenshot 2 **Screenshot of the program's graphical user interface. Maximum field length supported is 50x50 elements**

3.2. Identification of Inefficiencies:

This algorithm was thoroughly investigated during its implementation for the visualizing program. The program had to reflect, as close as possible, the FAR descriptions in the original research paper. This was done to ensure that the algorithm was as effective as possible, while still being, essentially, FAR algorithm.

In what ways can the FAR algorithm be enhanced to provide better solution to the multi-agent pathfinding problem? This study identified areas for improvement in all three stages of the original FAR method, enhancing both the efficiency and simplicity of the algorithm.

3.2.1. Flow Annotation:

No problems were identified with the basic flow annotation system. The original FAR algorithm's general rule of separating the field traffic into specific lanes – discussed in section 2.2.1 of this paper – does exactly what it is supposed to, reducing, in effect, the number of options (and therefore the memory requirements of taking them all into account) of the pathfinding as well as the coordination stage of the simulation.

The original FAR's problems with flow annotation were found by this study to stem from the way the algorithm deals with the inevitable problem of running into obstacles while flow-annotating the field.

While the original FAR algorithm does present the requirement that any two orthogonally neighboring agents must be easily accessible in both directions, the rule is presented just as an overarching principle, used to justify the introduction of a number of additional rules dealing with special cases. These rules are specifically aimed at the elimination of so-called *sinks* and *sources*, resolving the *twisted tunnel* problem, and restoring connectivity in *narrow tunnels*.

A *sink* is an element from which there is no exit route; a *source* is its opposite, an element that cannot be entered. A *twisted tunnel* is a narrow diagonal corridor two elements in width, while a *narrow tunnel* is an orthogonal corridor one element wide. All four situations are outlined in Figure 8, along with their proposed solutions from the original FAR algorithm (Wang & Boten 4). (Note the use of diagonal edges in eliminating the sinks and sources and their absence in the twisted tunnel setting.)

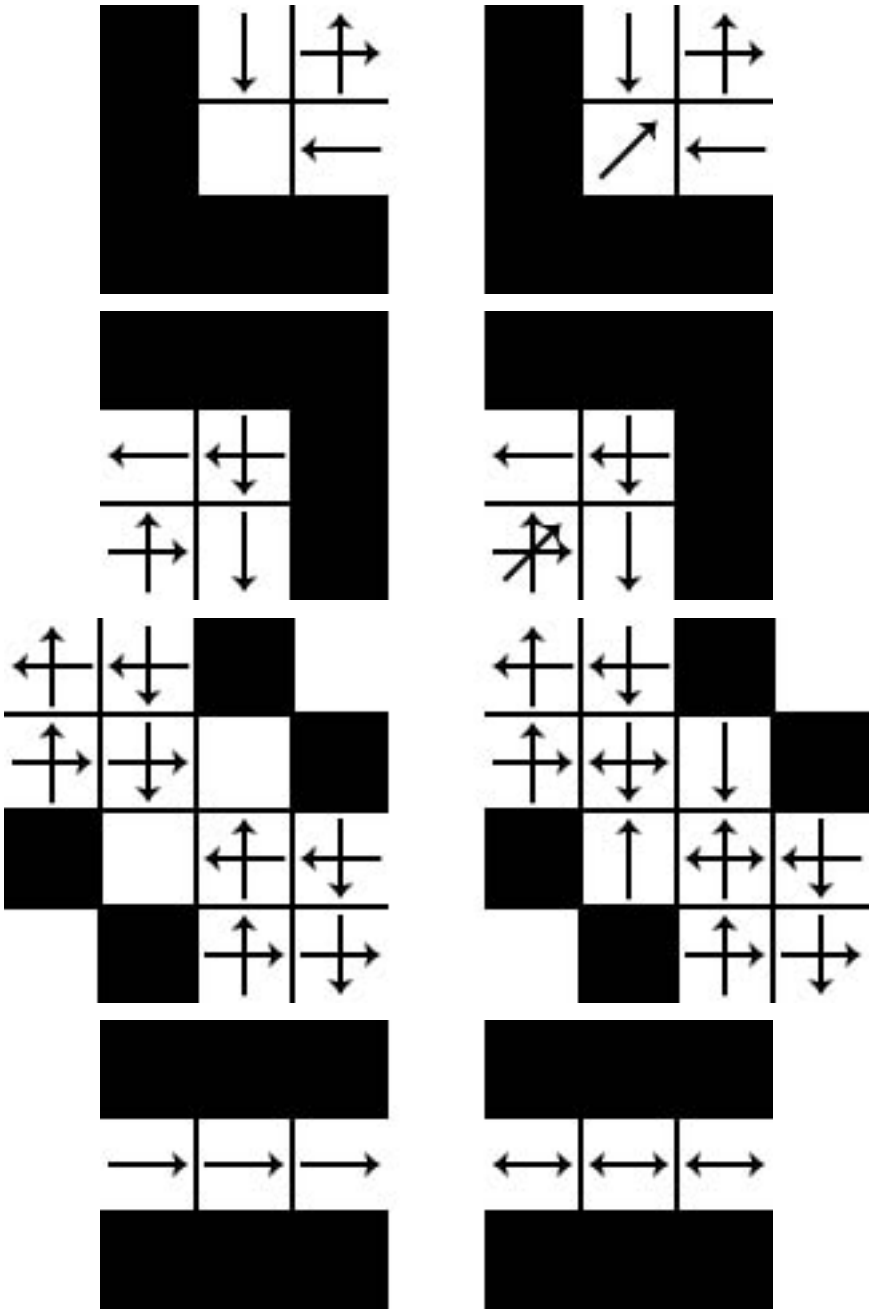


Figure 8 Situations addressed by FAR accessibility rules (top to bottom) with solutions (right)
sink,
source,
twisted tunnel,
narrow tunnel.

This study found, however, that using diagonal edges to resolve the sinks and sources is not systematic, standing out as the only use of diagonals in otherwise entirely orthogonal-based pathfinding algorithm. That would not be a problem if the inclusion of diagonal moves in the coordination stage did not necessitate extensive checks to ensure moves are *legal*, that is, there is not a diagonal row of obstacles or

other agents in the way – all of that in both space and time. Therefore, it was concluded that it is much more favorable to just change the orthogonal edges to be bidirectional

Moreover, looking at many special cases instead of the general rule is not a very good strategy in itself – the code required to handle all of possibilities can easily grow very complex and hard to maintain. This study determined that it is much easier to just have the algorithm checking the local reverse connectivity for every pair of elements connected by an edge, in the worst case making that edge bidirectional.

```
// Zbyněk Stara
if (flowNorthward && !northBlocked) {
    boolean nnBl = field.getNorthElement(field.getNorthElement(currentElement)).isBlocked();
    if (flowEastward) {
        boolean nneBl = field.getNorthEastElement(field.getNorthElement(currentElement)).isBlocked();
        if (!((!wBl && !nwBl) || (!nnBl && !nneBl && !neBl && !eBl && !seBl && !sBl))) {
            field.getNorthElement(currentElement).getFARExtension().setExtraVerticalAccessTo(currentElement);
        }
    } else if (flowWestward) {
        boolean nnwBl = field.getNorthWestElement(field.getNorthElement(currentElement)).isBlocked();
        if (!((!eBl && !neBl) || (!nnBl && !nnwBl && !nwBl && ! wBl && !swBl && !sBl))) {
            field.getNorthElement(currentElement).getFARExtension().setExtraVerticalAccessTo(currentElement);
        }
    }
} else if (flowSouthward && !southBlocked) {
    boolean ssBl = field.getSouthElement(field.getSouthElement(currentElement)).isBlocked();
    if (flowEastward) {
        boolean sseBl = field.getSouthEastElement(field.getSouthElement(currentElement)).isBlocked();
        if (!((!wBl && !swBl) || (!ssBl && !sseBl && !seBl && !eBl && !neBl && !nBl))) {
            field.getSouthElement(currentElement).getFARExtension().setExtraVerticalAccessTo(currentElement);
        }
    } else if (flowWestward) {
        boolean sswBl = field.getSouthWestElement(field.getSouthElement(currentElement)).isBlocked();
        if (!((!eBl && !seBl) || (!ssBl && !sswBl && !swBl && !wBl && !nwBl && !nBl))) {
            field.getSouthElement(currentElement).getFARExtension().setExtraVerticalAccessTo(currentElement);
        }
    }
}
```

Code 1 **Two of the flow annotation condition blocks showing reverse connectivity checks for different flow arrangements of the investigated pair of elements (the very first check describes situation in Figure 4); the codes (like wBl) are shorthand checks for blocked elements (here at west element)**

The approach used to implement the reverse connectivity checks can be seen in Code 1. Unfortunately, there is no better way to check if a reverse path is obstacle-free than actually checking every single element of the path (ssBl, sseBl, seBl…). However, it is still better to have only one set of these elaborate checks than four.

It was found that this study's approach achieves the same results as the original algorithm – the field is usefully annotated –, but thanks to the usage of the underlying principle as the rule, the code has been much simplified, and the need for diagonal edges was removed.

3.2.2. Manhattan Heuristics:

The second enhancement this study has identified relates to the pathfinding section. Because it was concluded that it is most favorable when there are no diagonal moves allowed in the field, the so-called *Manhattan heuristics* can be used instead of the Pythagorean heuristics of standard FAR. The difference is that Manhattan heuristics adds up the x- and y-distances to the goal; there is no hypotenuse involved in the calculation. Therefore, the Manhattan heuristics only considers possible orthogonal movements. Comparison of the two, both in code and diagram, can be seen below.

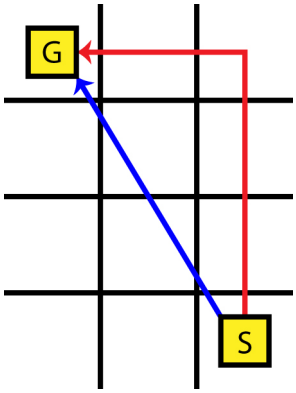


Figure 9 Shortest path from S to G according to Pythagorean (blue = $\sqrt{13}$ = 3.6) and Manhattan (red = 5) heuristics

```
//Zbyněk Stara
public double distanceBetween(Element element1, Element element2) {
    return (Math.sqrt(((element2.X_ID-element1.X_ID)*(element2.X_ID-element1.X_ID))
            + ((element2.Y_ID-element1.Y_ID)*(element2.Y_ID-element1.Y_ID))));
}

public double manhattanDistanceBetween(Element element1, Element element2) {
    return (Math.abs(element2.X_ID-element1.X_ID) + Math.abs(element2.Y_ID-element1.Y_ID));
}
```

Code 2 Implementation of the two heuristic functions – Pythagorean at the top, Manhattan at the bottom

Moreover, it was shown in this investigation that switching away from the Pythagorean heuristics also has some far-reaching positive implications on the closeness with which the algorithm reflects the requirements set up for it in the original research paper – the FAR researchers have included a condition stating that pathfinding ties in the f-value are broken in favor of straight paths. The study has found that the Pythagorean heuristics without diagonal movement produces suboptimal results, as the agents tend to end up zigzagging around the best diagonal path. This went against the intention of the researchers who tried to keep paths “as

straight as possible, since fewer turns reduce the chance for side-on collisions" (Wang & Boten 4).

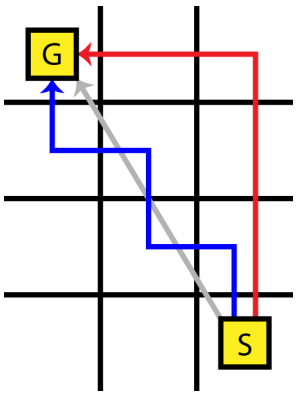


Figure 10 **When using Pythagorean heuristics in a field without diagonal moves, the path (blue) tends to zigzag around the ideal line (grey)**
Manhattan heuristics does not suffer from such a problem if the same-direction preference is implemented (red)

With the Manhattan heuristics, then, the imaginary diagonal path is not disproportionately weighted, and so the agent can move along a straight line, turn once, and then go along a straight line again towards the goal.

Not only does this study's switch to Manhattan heuristics in the FAR pathfinding stage makes sense since diagonal paths are no longer possible, it also improves the overall performance of the algorithm.

3.2.3. Proxy Paths:

In the path coordination section, this study introduces a concept novel to the original FAR algorithm with the aim to generalize the process of waiting agent clearing the way to moving agents that needs to pass through the waiter's position. Though this process is already included in the original FAR algorithm, it is only implemented for specific cases – namely, moving through the position of agent that has found its goal, and resolving a deadlock.

However, as the solution in the original FAR is not generalized, there are unaccounted-for situations, in which it would be more efficient to have a waiting agent giving way to a moving agent. To resolve it, a generalized system of *proxy paths* has incorporated in this study.

When a waiting agent is displaced from its position by a moving agent, the displaced agent has to call A* to find its way back to its original position. The path returned by this secondary A* search is the *proxy path*. It subsequently becomes an integral part of the agent's path, as if it has been there from the beginning.

```
// Zbyněk Stara
public boolean isProxyAvailable(int reservationSlot, FARAgent agent) {
    List accessibleElements = thisElement.getFARExtension().getAccessibleElements();
    boolean proxyFound = false;
    for (int i = 0; i < accessibleElements.size(); i++) {
        Element currentAccessible = (Element) accessibleElements.getNodeData(i);
        if ((reservationSlot - 1) >= 0) {
            if (currentAccessible.getFARExtension().getReservation(reservationSlot) == null) {
                agent.assignProxyPath(step, reservationSlot, currentAccessible);
                proxyFound = true;
                break;
            } else {
                if (currentAccessible.getFARExtension().getReservation(reservationSlot).getAgent() == null) {
                    agent.assignProxyPath(step, reservationSlot, currentAccessible);
                    proxyFound = true;
                    break;
                } else {
                    if (currentAccessible.getFARExtension().isTopPriority(reservationDepth, reservationSlot,
                            currentAccessible.getFARExtension().getCurrentAgent(), step, thisElement)) {
                        agent.assignProxyPath(step, reservationSlot, currentAccessible);
                        proxyFound = true;
                        break;
                    }
                }
            }
        } else {
            thisElement.getFARExtension().getCurrentAgent().assignProxyPath(step, reservationSlot, currentAccessible);
            proxyFound = true;
            break;
        }
    } if (proxyFound) {
        return true;
    } else {
        return false;
    }
}
```

Code 3 **The algorithm checking the availability of a suitable proxy location for agent on a given element**

This study introduced several changes to the proxy-path mechanism, in which its approach differs from the original FAR's one. Unlike the original FAR solution, the displaced agent must make sure that its move does not disrupt the paths of other agents – demonstrated by the requirement on the proxy-seeking agent to have top-priority movement rights onto the proxy (the rights are rotated periodically). Furthermore, then the inclusion of the proxy path into the agent's path permits better subsequent cooperation of the agent, even allowing – if it is necessary – another proxy path to be accessed while on a proxy path.

However, the biggest advantage of this study's solution is that it allows for an automatic solution of the *deadlock* problem (also discussed in section 2.3.3) without any additional checking measures needed. That allowed a huge part of the deadlock-specific code of the original FAR to be removed. Again, therefore, a general solution proved to be more efficient than several specific ones.

3.3. Areas for Further Investigation:

Besides enhancing the original FAR algorithm, this study identified several areas for further investigation and enhancement of the algorithm, which were not solved by this research paper, and are, therefore, open for further investigation.

One of those is the problem of *head-on collisions*, agents passing through each other as they progress to the position the other one occupied on the previous time step. It was discussed neither in the original FAR nor in this research; however, this problem cannot be ignored completely – although bidirectional edges are rarer in FAR than in standard CA*, they still do exist.

Another area for further investigation is the possibility for dynamic setting of the reservation depth of the simulation. According to the initial setup of the field and number of agents placed, then, the algorithm would adjust the reservation depth's value. For example, a field with narrow tunnels and many agents would require higher value for reservation depth than an empty open field. Currently, the reservation depth is set externally as a parameter to the algorithm.

3.4. Conclusions:

This study identified several enhancements of the FAR multi-agent pathfinding algorithm. They improve and optimize the algorithm's performance in all three stages of finding the solution – flow annotation, pathfinding, and coordination.

The generalization of the flow annotation rule of reverse accessibility led to the development of a more concise code for the algorithm's first stage, and eliminated the need for diagonal movement support.

Furthermore, the change from Pythagorean to Manhattan heuristics has allowed the agents to follow as straight paths as possible and thus comply more with the FAR requirements set out in the original research paper.

Finally, the inclusion of proxy paths in this study instead of arbitrary step-aside commands, and their universal applicability, have allowed big portions of task-specific routines to be taken out from the algorithm in favor of a more complete general solution.

4. Bibliography:

4.1. References:

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* Menlo Park, California: Artificial Intelligence Group of the Applied Physics Laboratory, Stanford Research Institute, 1968. N. pag. *IEEE Xplore Digital Library.* Web. 6 May 2012. <http://ieeexplore.ieee.org>. (A*)

Silver, David. *Cooperative Pathfinding.* UCL Department of Computer Science. University College London, 4 Sep. 2005. Web. 6 May 2012. <http://www.cs.ucl.ac.uk/>. (CA*)

Wang, Ko-Hsin Cindy, and Adi Boten. *Fast and Memory-Efficient Multi-Agent Pathfinding.* Association for the Advancement of Artificial Intelligence. Association for the Advancement of Artificial Intelligence, 2008. Web. 6 May 2012. <http://www.aaai.org>. (FAR)

4.2. Additional Literature Consulted:

Dijkstra, E. W. A Note on Two Problems in Connexion with Graphs. Amsterdam, The Netherlands: Springer-Verlag, 1959. *Springer link.* Web. 25 Sept. 2012. <http://link.springer.com>

Jansen, Renee, and Nathan Sturtevart, *A New Approach to Cooperative Pathfinding.* Edmonton, Alberta: International Foundation for Autonomous Agents and Multiagent Systems, 2008. *ACM Digital Library.* Web. 25 Sept. 2012 <http://dl.acm.org>

Orkin, Jeff. *Three States and a Plan: The A.I. of F.E.A.R.* Cambridge, Massachusetts: Game Developers Conference, 2006. *CiteseerX.* Web. 25. Sept. 2012. <http://www.jorkin.com>

Silver, David. *Cooperative Pathfinding*. Department of Computer Science, University of Alberta. 22 June 2007. Web. 6 May 2012. <http://cs.ualberta.ca>

Stentz, Anthony. *The Focused D* Algorithm for Real-Time Replanning.* Pittsburgh, Pennsylvania: International Joint Conference on Artificial Intelligence, 1995. Carnegie Mellon University. Web. 25 Sept. 2012. <http://cs.cmu.edu>

- - -. *Optimal and Efficient Path Planning for Unknown and Dynamic Environments.* Pittsburgh, Pennsylvania: Is Carnegie Mellon, 1993. *CiteseerX.* Web. 6 May 2012. <http://citeseerx.ist.psu.edu>

Sturtevart, Nathan, and Michael Buro. *Improving Collaborative Pathfinding.* Edmonton, Alberta: Department of Computer Science, University of Alberta, 2006. *Association for the Advancement of Artificial Intelligence.* Web. 25 Sept. 2012. <http://www.aaai.org>

5. Appendix:

5.1. The Algorithm Visualizing Program:

5.1.1. MainGUI.java:

```
package guipackage;

import adtpackage.*;
import generalpackage.*;
import apackage.*;
import farpackage.*;

/**
 *
 * @author Zbyněk Stara
 */
public class MainGUI extends javax.swing.JFrame {
    // IMPORTANT: the setValueAt methods use ordering of (…, y, x); the Elements have
       (x, y)

    private final int MAX_FIELD = 52; // 52 = defalut: 50x50 field, 2500 elements =
       500 obstacles + 1000 starts + 1000 goals
                                // another suggested value would be 37: 35x35 field,
       1225 elements = 245 obstacles + 980 free spaces
                                // or 36: 34x34 field, which fits nicely on the screen

    private final int INIT_FIELD = 36;

    private final String BOUNDARY = "▐";
    private final String OBSTACLE = "▐";
    private final String BLANK = "";
    private final String START = "S";
    private final String GOAL = "G";
    private final String PATH = "o";
    private final String AGENT = "•";

    private Field field;

    private APathfinder aPathfinder;
    private FARPathfinder farPathfinder;

    private boolean isFieldSet = false;
    private boolean areAgentsSet = false;
    private boolean arePathsSet = false;

    int stepCounter;

    int algorithmToUse = 0; // 0 = A*, 1 = D*, 2 = CA*, 3 = WHCA*, 4 = FAR

    private Pathfinder choosePathfinder() {
        switch (algorithmToUse) {
            case 0:
                return aPathfinder;
            case 1:
                // return dPathfinder.getDAgentSet();
            case 2:
                // return caPathfinder.getCAAgentSet();
            case 3:
                // return whcaPathfinder.getWHCAAgentSet();
            case 4:
                return farPathfinder;
            default: // will not happen
                return null;
        }
    }

    private void drawFieldTable(int size) {
        for (int i = 0; i < MAX_FIELD; i++) {
            for (int j = 0; j < MAX_FIELD; j++) {
                fieldTable.setValueAt("", i, j);
            }
```

```
    }
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (field.getElement(i, j).isBoundary()) {
                fieldTable.setValueAt(BOUNDARY, j, i);
            } else if (field.getElement(i, j).isObstacle()) {
                fieldTable.setValueAt(OBSTACLE, j, i);
            } else {
                fieldTable.setValueAt(BLANK, j, i);
                /*String debug = "";
                List accessibleList = field.getElement(i,
   j).getFARExtension().getAccessibleElements();
                int accessibleListSize = accessibleList.size();
                for (int k = 0; k < accessibleListSize; k++) {
                    Element currentAccessible = (Element)
   accessibleList.removeFirst();
                    if (field.getNorthElement(field.getElement(i, j)) ==
   currentAccessible) debug = debug + "↑";
                    else if(field.getNorthEastElement(field.getElement(i, j)) ==
   currentAccessible) debug = debug + "↗";
                    else if(field.getEastElement(field.getElement(i, j)) ==
   currentAccessible) debug = debug + "→";
                    else if(field.getSouthEastElement(field.getElement(i, j)) ==
   currentAccessible) debug = debug + "↘";
                    else if(field.getSouthElement(field.getElement(i, j)) ==
   currentAccessible) debug = debug + "↓";
                    else if(field.getSouthWestElement(field.getElement(i, j)) ==
   currentAccessible) debug = debug + "↙";
                    else if(field.getWestElement(field.getElement(i, j)) ==
   currentAccessible) debug = debug + "←";
                    else if(field.getNorthWestElement(field.getElement(i, j)) ==
   currentAccessible) debug = debug + "↖";
                }
                fieldTable.setValueAt(debug, j, i);*/
            }
        }
    }
}

private void drawFieldTableStarts(TreeSet startElementSet) {
    for (int i = 0; i < startElementSet.size(); i++) {
        Element currentStartElement = (Element) startElementSet.getNodeData(i);
        fieldTable.setValueAt(START + i, currentStartElement.Y_ID,
   currentStartElement.X_ID);
    }
}

private void drawFieldTableGoals(TreeSet goalElementSet) {
    for (int i = 0; i < goalElementSet.size(); i++) {
        Element currentStartElement = (Element) goalElementSet.getNodeData(i);
        fieldTable.setValueAt(GOAL + i, currentStartElement.Y_ID,
   currentStartElement.X_ID);
    }
}

private void drawFieldTableAgentsAtStep(TreeSet agentElementSet) {
    for (int i = 0; i < agentElementSet.size(); i++) {
        Element currentAgentElement = (Element) agentElementSet.getNodeData(i);
        fieldTable.setValueAt(AGENT + i, currentAgentElement.Y_ID,
   currentAgentElement.X_ID);
    }
}

private void drawFieldTablePathsUntilStep(TreeSet pathElementSet) {
    while (!pathElementSet.isEmpty()) {
        Element currentPathElement = (Element) pathElementSet.removeMin();
        fieldTable.setValueAt(PATH, currentPathElement.Y_ID,
   currentPathElement.X_ID);
    }
}

private void myInitComponents() {
    sizeSlider.setMaximum(MAX_FIELD);
    sizeSlider.setMinimum(4);
    sizeSlider.setValue(INIT_FIELD);
```

```
    fieldSizeTF.setEditable(false);
    numberAgentsTF.setText(1 + "");
    numberPathFailuresTF.setEditable(false);
    currentStepTF.setText(-1 + "");
    currentStepTF.setEditable(false);
}

public MainGUI() {
    initComponents();
    myInitComponents();
}

// Generated code (290 lines)

private void exitMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    System.exit(0);
}
private void sizeSliderStateChanged(javax.swing.event.ChangeEvent evt) {
    fieldSizeTF.setText(sizeSlider.getValue() + "");
}
private void refreshFieldButtonMouseReleased(java.awt.event.MouseEvent evt) {
    isFieldSet = true;
    areAgentsSet = false;
    arePathsSet = false;
    stepCounter = -1;
    field = new Field(sizeSlider.getValue(), sizeSlider.getValue());
    field.generateObstacles();
    field.assignFreeGroups();
    field.fillClosedGroups();
    //field.printFreeGroupNumberList();
    drawFieldTable(field.X_DIMENSION);
    currentStepTF.setText(-1 + "");
}
private void findPathsButtonMouseReleased(java.awt.event.MouseEvent evt) {
    if (isFieldSet && areAgentsSet) {
        arePathsSet = true;
        drawFieldTable(field.X_DIMENSION); // assuming field is a square
        stepCounter = -1;
        currentStepTF.setText(stepCounter + "");
        switch (algorithmToUse) {
            case 0:
                aPathfinder = new APathfinder(field);
                break;
            case 1:
                // return dPathfinder.getDAgentSet();
                break;
            case 2:
                // return caPathfinder.getCAAgentSet();
                break;
            case 3:
                // return whcaPathfinder.getWHCAAgentSet();
                break;
            case 4:
                farPathfinder = new FARPathfinder(field);
                break;
        }

    drawFieldTablePathsUntilStep(choosePathfinder().getAgentPathsUntilStep(chooseP
    athfinder().getFailureCriterion()));
        drawFieldTableStarts(choosePathfinder().getAgentStartElements());
        drawFieldTableGoals(choosePathfinder().getAgentGoalElements());
    }
}
private void nextStepButtonMouseReleased(java.awt.event.MouseEvent evt) {
    if (isFieldSet && areAgentsSet && arePathsSet) {
        stepCounter++;
        drawFieldTable(field.X_DIMENSION);
        currentStepTF.setText(stepCounter + "");

    drawFieldTablePathsUntilStep(choosePathfinder().getAgentPathsUntilStep(stepCou
    nter));
        drawFieldTableStarts(choosePathfinder().getAgentStartElements());
        drawFieldTableGoals(choosePathfinder().getAgentGoalElements());

    drawFieldTableAgentsAtStep(choosePathfinder().getAgentsAtStep(stepCounter));
    }
```

```
    }
    private void initialStateButtonMouseReleased(java.awt.event.MouseEvent evt) {
        if (isFieldSet && areAgentsSet && arePathsSet) {
            drawFieldTable(field.X_DIMENSION);
            stepCounter = 0;
            currentStepTF.setText(stepCounter + "");

        drawFieldTablePathsUntilStep(choosePathfinder().getAgentPathsUntilStep(stepCou
        nter));
            drawFieldTableStarts(choosePathfinder().getAgentStartElements());
            drawFieldTableGoals(choosePathfinder().getAgentGoalElements());

        drawFieldTableAgentsAtStep(choosePathfinder().getAgentsAtStep(stepCounter));
        }
    }
    private void resetPathButtonMouseReleased(java.awt.event.MouseEvent evt) {
        if (isFieldSet && areAgentsSet && arePathsSet) {
            drawFieldTable(field.X_DIMENSION); // assuming field is a square
            stepCounter = -1;
            currentStepTF.setText(stepCounter + "");

        drawFieldTablePathsUntilStep(choosePathfinder().getAgentPathsUntilStep(chooseP
        athfinder().getFailureCriterion()));
            drawFieldTableStarts(choosePathfinder().getAgentStartElements());
            drawFieldTableGoals(choosePathfinder().getAgentGoalElements());
        }
    }
    private void algorithmComboBoxItemStateChanged(java.awt.event.ItemEvent evt) {
        algorithmToUse = algorithmComboBox.getSelectedIndex();
        isFieldSet = true;
        arePathsSet = false;
        stepCounter = -1;
        drawFieldTable(field.X_DIMENSION);
        drawFieldTableStarts(field.getAgentStartElements());
        drawFieldTableGoals(field.getAgentGoalElements());
        currentStepTF.setText(-1 + "");
    }
    private void refreshAgentsButtonMouseReleased(java.awt.event.MouseEvent evt) {
        if (isFieldSet) {
            areAgentsSet = true;
            stepCounter = -1;
            currentStepTF.setText(stepCounter + "");
            field.assignAgents(Integer.parseInt(numberAgentsTF.getText()));
            drawFieldTable(field.X_DIMENSION);
            drawFieldTableStarts(field.getAgentStartElements());
            drawFieldTableGoals(field.getAgentGoalElements());
        }
    }
    /**
    * @param args the command line arguments
    */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new MainGUI().setVisible(true);
            }
        });
    }

    // Variables declaration (32 lines)

}
```

5.1.2. Field.java:

```
package generalpackage;
import adtpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class Field {
    // TERMINOLOGY OF ELEMENTS:
        // BLOCKED ELEMENTS
```

```
        // OBSTACLE
        // BOUNDARY
    // TRAVERSABLE ELEMENTS - no obstacle or boundary
        // OCCUPIED - blocked by agent
        // ASSIGNED - (creation of new agents) blocked by another agent's start or
    goal
        // RESERVED - an agent has reserved this element because it intends to
    move there
        // FREE - no agent there
        // NEIGHBOR - traversable elements around a given element
        // ACCESSIBLE - elements that can be accessed from a given element
public final int X_DIMENSION; // includes boundaries
public final int Y_DIMENSION;

private Element [][] elementArray;
private int freeGroupNumber = -999;
private List freeGroupList = new List();
private HashSet traversableSet;
private Agent [] agentArray;
public Field() {
    X_DIMENSION = 52;
    Y_DIMENSION = 52;
    freeGroupNumber = 0;
    elementArray = new Element[X_DIMENSION][Y_DIMENSION];
    traversableSet = new HashSet((X_DIMENSION - 2) * (Y_DIMENSION - 2));
    generateBoundaries();
}
public Field(int x_dimension, int y_dimension) {
    X_DIMENSION = x_dimension;
    Y_DIMENSION = y_dimension;
    freeGroupNumber = 0;
    elementArray = new Element[X_DIMENSION][Y_DIMENSION];
    traversableSet = new HashSet((X_DIMENSION - 2) * (Y_DIMENSION - 2));
    generateBoundaries();
}
private void generateBoundaries() { // STEP ONE
    for (int i = 0; i < X_DIMENSION; i++) {
        if ((i == 0) || (i == (X_DIMENSION - 1))) {
            for (int j = 0; j < Y_DIMENSION; j++) {
                elementArray[i][j] = new Element(i, j, true, false); // the top
    and bottom boundaries
            }
        } else {
            for (int j = 0; j < Y_DIMENSION; j++) {
                if ((j == 0) || (j == (Y_DIMENSION - 1))) {
                    elementArray[i][j] = new Element(i, j, true, false); // the
    left and right boundaries
                } else {
                    elementArray[i][j] = new Element(i, j, false, false);
                    traversableSet.add(getElement(i, j), getElement(i,
    j).getKey(Y_DIMENSION));
                }
            }
        }
    }
}
public void generateObstacles() { // STEP TWO - improved version with guaranteed
    20% of "raw" obstacles (before freeGroup adjustment) = CA*
    int obstacleX = -999;
    int obstacleY = -999;
    double area = (X_DIMENSION - 2) * (Y_DIMENSION - 2);
    int obstaclesLeft = (int) Math.round(area / 5.0);
    System.out.println("Area is " + area + " and 20% of that is " + (area / 5.0) +
    " (" + obstaclesLeft + ") obstacles");
    while (obstaclesLeft > 0) {
        do {
            obstacleX = (int) ((Math.random() * (X_DIMENSION - 2)) + 1);
            obstacleY = (int) ((Math.random() * (Y_DIMENSION - 2)) + 1);
        } while (!traversableSet.contains(getElement(obstacleX,
    obstacleY).getKey(Y_DIMENSION)));
        traversableSet.remove(getElement(obstacleX,
    obstacleY).getKey(Y_DIMENSION));
        getElement(obstacleX, obstacleY).setIsObstacle(true);
        obstaclesLeft -= 1;
    }
}
```

```
public void assignFreeGroups() { // STEP THREE
    for (int i = 1; i < (Y_DIMENSION - 1); i++) {
        for (int j = 1; j < (X_DIMENSION - 1); j++) {
            Element currentElement = getElement(j, i);
            System.out.print(currentElement.toString() + ": ");
            if (!currentElement.isObstacle()) { // if this element is not an
    obstacle
                if (getWestElement(currentElement).isBoundary() ||
    getWestElement(currentElement).isObstacle()) { // if west element is blocked
                    if (getNorthElement(currentElement).isBoundary() ||
    getNorthElement(currentElement).isObstacle()) { // if north element is also
    blocked
                        // assign this element to a new freeGroup
                        currentElement.setFreeGroup(freeGroupNumber);
                        FreeGroup newFreeGroup = new FreeGroup(freeGroupNumber,
    currentElement, this);
                        freeGroupList.insertAtRear(newFreeGroup);
                        freeGroupNumber++;
                    } else { // if west element is blocked but north element is
    not
                        // assign this element to north element's freeGroup
                        int northFreeGroup =
    getNorthElement(currentElement).getFreeGroup();
                        currentElement.setFreeGroup(northFreeGroup);
                        ((FreeGroup)
    freeGroupList.getNodeData(northFreeGroup)).addElement(currentElement);
                    }
                } else { // if west element is not blocked
                    // assign freeGroup from west element
                    int westFreeGroup =
    getWestElement(currentElement).getFreeGroup();
                    currentElement.setFreeGroup(westFreeGroup);
                    ((FreeGroup)
    freeGroupList.getNodeData(westFreeGroup)).addElement(currentElement);
                    if (!(getNorthElement(currentElement).isBoundary()) &&
    !(getNorthElement(currentElement).isObstacle()) &&
    !(getNorthElement(currentElement).getFreeGroup() ==
    currentElement.getFreeGroup())) {
                        // add elements of north element's freeGroup to this
    element's freeGroup
                        if (getNorthElement(currentElement).getFreeGroup() >
    currentElement.getFreeGroup()) { // traceBack for north's freeGroup
                            freeGroupTraceBack(getNorthElement(currentElement),
    currentElement.getFreeGroup());
                        } else { // traceBack for west's freeGroup
                            freeGroupTraceBack(currentElement,
    getNorthElement(currentElement).getFreeGroup());
                        }
                    } else {
                    }
                }
            } else { // if this element is an obstacle
                // skip this element
            }
        }
    }
}
private void freeGroupTraceBack(Element element, int newFreeGroup) { // Recursive
    helper function
    int oldFreeGroup = element.getFreeGroup();
    element.setFreeGroup(newFreeGroup);
    ((FreeGroup) freeGroupList.getNodeData(oldFreeGroup)).removeElement(element);
    ((FreeGroup) freeGroupList.getNodeData(newFreeGroup)).addElement(element);

    List elementNeighborList = getAxisNeighborElements(element);
    while (!elementNeighborList.isEmpty()) {
        Element currentNeighbor = (Element) elementNeighborList.removeLast();
        int neighborFreeGroup = currentNeighbor.getFreeGroup();
        if ((neighborFreeGroup != newFreeGroup) && (neighborFreeGroup != -999) &&
    (!currentNeighbor.isBoundary()) && (!currentNeighbor.isObstacle())) {
            freeGroupTraceBack(currentNeighbor, newFreeGroup);
        }
    }
}
public void fillClosedGroups() { // STEP FOUR
    FreeGroup mainFreeGroup = (FreeGroup) freeGroupList.getNodeData(0);
```

```
    for (int i = 1; i < freeGroupList.size(); i++) {
        FreeGroup currentFreeGroup = (FreeGroup) freeGroupList.getNodeData(i);
        if (currentFreeGroup.getNumberElements() >
    mainFreeGroup.getNumberElements()) {
            mainFreeGroup = currentFreeGroup;
        }
        if (mainFreeGroup.getNumberElements() > (((X_DIMENSION - 2)*(Y_DIMENSION -
    2))/2)) {
            break;
        }
    }
    TreeSet changeElementSet = new TreeSet();
    for (int i = 0; i < freeGroupList.size(); i++) {
        FreeGroup currentFreeGroup = (FreeGroup) freeGroupList.getNodeData(i);
        if (currentFreeGroup.getFreeGroupNumber() ==
    mainFreeGroup.getFreeGroupNumber()) {
            continue; // skip this because it's the main free group
        } else {
            while (currentFreeGroup.getNumberElements() > 0) {
                Element minElement = currentFreeGroup.removeMinElement();
                if (minElement != null) {
                    changeElementSet.add(minElement,
    minElement.getKey(Y_DIMENSION));
                }
            }
        }
    }
    System.out.println("Changing " + changeElementSet.size() + " elements to
    obstacles.");
    while (!(changeElementSet.isEmpty())) {
        Element minElement = (Element) changeElementSet.removeMin();
        minElement.setIsObstacle(true);
        traversableSet.remove(minElement.getKey(Y_DIMENSION));
        minElement.setFreeGroup(-1); // OPTIONAL
    }
}
public void assignAgents(int numAgents) {
    agentArray = new Agent[numAgents];
    HashSet freeSet = new HashSet((X_DIMENSION - 2) * (Y_DIMENSION - 2));
    for (int i = 1; i < (X_DIMENSION - 1); i++) {
        for (int j = 1; j < (Y_DIMENSION - 1); j++) {
            if (traversableSet.contains(getElement(i, j).getKey(Y_DIMENSION))) {
                freeSet.add(getElement(i, j), getElement(i,
    j).getKey(Y_DIMENSION));
            }
        }
    }
    for (int i = 0; i < agentArray.length; i++) {
        try {
            agentArray[i] = new Agent(this, freeSet, i);
        } catch (Exception ex) {
            System.out.println("Error in newRandomAAgent. Breaking assignment
    loop…");
            break;
        }
    }
}
public void printFreeGroupNumberList() {
    for (int i = 0; i < freeGroupList.size(); i++) {
        System.out.println("freeGroupNumberList at " + i + ": freeGroupNumber " +
    ((FreeGroup) freeGroupList.getNodeData(i)).getFreeGroupNumber() + ",
    numberElements " + ((FreeGroup)
    freeGroupList.getNodeData(i)).getNumberElements());
    }
}

public Element [][] getElementArray() {
    return elementArray;
}
public HashSet getTraversableSet() {
    return traversableSet;
}
public Agent [] getAgentArray() {
    return agentArray;
}
public TreeSet getAgentStartElements() {
```

```
    TreeSet elementSet = new TreeSet();
    for (int i = 0; i < agentArray.length; i++) {
        Agent currentAgent = agentArray[i];
        elementSet.add(currentAgent.getStart(), currentAgent.getAgentID());
    }
    return elementSet;
}
public TreeSet getAgentGoalElements() {
    TreeSet elementSet = new TreeSet();
    for (int i = 0; i < agentArray.length; i++) {
        Agent currentAgent = agentArray[i];
        elementSet.add(currentAgent.getGoal(), currentAgent.getAgentID());
    }
    return elementSet;
}
public Element getElement(int x_position, int y_position) {
    return elementArray[x_position][y_position];
}
public List getNeighborElements(Element element) {
    List neighborList = new List();
    neighborList.insertAtFront(getNorthElement(element));
    neighborList.insertAtFront(getNorthEastElement(element));
    neighborList.insertAtFront(getEastElement(element));
    neighborList.insertAtFront(getSouthEastElement(element));
    neighborList.insertAtFront(getSouthElement(element));
    neighborList.insertAtFront(getSouthWestElement(element));
    neighborList.insertAtFront(getWestElement(element));
    neighborList.insertAtFront(getNorthWestElement(element));
    return neighborList;
}
public List getAxisNeighborElements(Element element) {
    List neighborList = new List();
    neighborList.insertAtFront(getNorthElement(element));
    neighborList.insertAtFront(getEastElement(element));
    neighborList.insertAtFront(getSouthElement(element));
    neighborList.insertAtFront(getWestElement(element));
    return neighborList;
}
public double distanceBetween(Element element1, Element element2) {
    return (Math.sqrt(((element2.X_ID-element1.X_ID)*(element2.X_ID-
    element1.X_ID)) + ((element2.Y_ID-element1.Y_ID)*(element2.Y_ID-
    element1.Y_ID))));
}
public double manhattanDistanceBetween(Element element1, Element element2) {
    return (Math.abs(element2.X_ID-element1.X_ID) + Math.abs(element2.Y_ID-
    element1.Y_ID));
}
public boolean isLegalMove(Element element, Element question) {
    if ((question == getNorthEastElement(element)) &&
    getNorthElement(element).isObstacle() && getEastElement(element).isObstacle())
    return false;
    else if ((question == getSouthEastElement(element)) &&
    getEastElement(element).isObstacle() && getSouthElement(element).isObstacle())
    return false;
    else if ((question == getSouthWestElement(element)) &&
    getSouthElement(element).isObstacle() && getWestElement(element).isObstacle())
    return false;
    else if ((question == getNorthWestElement(element)) &&
    getWestElement(element).isObstacle() && getNorthElement(element).isObstacle())
    return false;
    else return true;
}
public Element getNorthElement(Element element) {
    return elementArray[element.X_ID][element.Y_ID - 1];
}
public Element getNorthEastElement(Element element) {
    if ((element.X_ID + 1) < X_DIMENSION) return elementArray[element.X_ID +
    1][element.Y_ID - 1];
    else return null;
}
public Element getEastElement(Element element) {
    if ((element.X_ID + 1) < X_DIMENSION) return elementArray[element.X_ID +
    1][element.Y_ID];
    else return null;
}
public Element getSouthEastElement(Element element) {
```

```
        if (((element.X_ID + 1) < X_DIMENSION) && ((element.Y_ID + 1) < Y_DIMENSION))
        return elementArray[element.X_ID + 1][element.Y_ID + 1];
        else return null;
    }
    public Element getSouthElement(Element element) {
        if ((element.Y_ID + 1) < Y_DIMENSION) return
        elementArray[element.X_ID][element.Y_ID + 1];
        else return null;
    }
    public Element getSouthWestElement(Element element) {
        if ((element.Y_ID + 1) < Y_DIMENSION) return elementArray[element.X_ID -
        1][element.Y_ID + 1];
        else return null;
    }
    public Element getWestElement(Element element) {
        return elementArray[element.X_ID - 1][element.Y_ID];
    }
    public Element getNorthWestElement(Element element) {
        return elementArray[element.X_ID - 1][element.Y_ID - 1];
    }
    @Override public String toString() {
        return ("Dimensions: " + X_DIMENSION + ", " + Y_DIMENSION + "; FreeGroups
        assigned: " + freeGroupList.size());
    }
}
```

5.1.3 Element.java:

```
package generalpackage;

import apackage.*;
import farpackage.*;

/**
 *
 * @author Zbyněk Stara
 */
public class Element implements Printable {
    // IMPORTANT: the setValueAt methods use ordering of (…, y, x); the Elements have
(x, y)

    private class ElementExtension {
        private AExtension aExtension;
        private FARExtension farExtension;
        // other extensions will be here

        public ElementExtension() {
            aExtension = new AExtension();
        }

        // A* algorithm
        public AExtension getAExtension() {
            return aExtension;
        }
        public void setAExtension(AExtension aExtension) {
            this.aExtension = aExtension;
        }

        // FAR algorithm
        public FARExtension getFARExtension() {
            return farExtension;
        }
        public void setFARExtension(FARExtension farExtension) {
            this.farExtension = farExtension;
        }
    }

    public final int X_ID;
    public final int Y_ID;

    private boolean isBoundary = false;
    private boolean isObstacle = false;

    // for use by field hole-filling algorithm
```

```
private int freeGroup = -999;

// for use by search algorithms
private ElementExtension extension = new ElementExtension();

// for drawing the field table
private boolean isStart = false;
private int startForAgent = -999; // ID of agent that starts here

private boolean isGoal = false;
private int goalForAgent = -999;

public Element(int x_id, int y_id) {
    this.X_ID = x_id;
    this.Y_ID = y_id;
}
public Element(int x_id, int y_id, boolean isBoundary, boolean isObstacle) {
    this.X_ID = x_id;
    this.Y_ID = y_id;

    this.isBoundary = isBoundary;
    this.isObstacle = isObstacle;
}

public void setIsObstacle(boolean isObstacle) {
    this.isObstacle = isObstacle;
}

public void setFreeGroup(int freeGroup) {
    this.freeGroup = freeGroup;
}

public void setAExtension(AExtension aExtension) {
    extension.setAExtension(aExtension);
}
public void setFARExtension(FARExtension farExtension) {
    extension.setFARExtension(farExtension);
}

public void setStartForAgent(int startForAgent) {
    this.startForAgent = startForAgent;
}
public void setGoalForAgent(int goalForAgent) {
    this.goalForAgent = goalForAgent;
}

public void setIsStart(boolean isStart) {
    this.isStart = isStart;
}
public void setIsGoal(boolean isGoal) {
    this.isGoal = isGoal;
}

public int getFreeGroup() {
    return freeGroup;
}

public AExtension getAExtension() {
    return extension.getAExtension();
}
public FARExtension getFARExtension() {
    return extension.getFARExtension();
}

public int getStartForAgent() {
    return startForAgent;
}
public int getGoalForAgent() {
    return goalForAgent;
}

public boolean isBoundary() {
    return isBoundary;
}
public boolean isObstacle() {
    return isObstacle;
```

```
    }
    public boolean isBlocked() {
        return (isBoundary || isObstacle);
    }

    public boolean isStart() {
        return isStart;
    }
    public boolean isGoal() {
        return isGoal;
    }

    public boolean isEqual(Element element) {
        return ((X_ID == element.X_ID) && (Y_ID == element.Y_ID));
    }

    public int getKey(int fieldDimension) {
        return ((X_ID - 1) + ((Y_ID - 1) * (fieldDimension - 2)));
    }

    @Override public String print() {
        return toString();
    }

    @Override public String toString() {
        return ("[" + X_ID + ", " + Y_ID + "]");
    }
}
```

5.1.4 FARExtension.java:

```
package farpackage;

import generalpackage.*;
import adtpackage.*;

/**
 *
 * @author Zbyněk Stara
 */
public class FARExtension {
    public class ReservationSlot { // at a given reservation step
        private Reservation reservation = null;

        private List reservationPriorityQueue = null;

        public ReservationSlot() {

        }

        public ReservationSlot(List reservationPriorityQueue) {
            this.reservationPriorityQueue = reservationPriorityQueue;
        }

        private void initReservationPriorityQueue() {
            reservationPriorityQueue = new List();

            if (field.getEastElement(thisElement).getFARExtension() != null) {
                if
(field.getEastElement(thisElement).getFARExtension().horizontalAccessTo != null) {
                    if
(thisElement.isEqual(field.getEastElement(thisElement).getFARExtension().horizontalAcc
essTo)) reservationPriorityQueue.insertAtRear(field.getEastElement(thisElement));
                }
            } if (field.getWestElement(thisElement).getFARExtension() != null) {
                if
(field.getWestElement(thisElement).getFARExtension().horizontalAccessTo != null) {
                    if
(thisElement.isEqual(field.getWestElement(thisElement).getFARExtension().horizontalAcc
essTo)) reservationPriorityQueue.insertAtRear(field.getWestElement(thisElement));
                }
            }

            if (field.getNorthElement(thisElement).getFARExtension() != null) {
```

```
                if
(field.getNorthElement(thisElement).getFARExtension().verticalAccessTo != null) {
                    if
(thisElement.isEqual(field.getNorthElement(thisElement).getFARExtension().verticalAcce
ssTo)) reservationPriorityQueue.insertAtRear(field.getNorthElement(thisElement));
                }
            } if (field.getSouthElement(thisElement).getFARExtension() != null) {
                if
(field.getSouthElement(thisElement).getFARExtension().verticalAccessTo != null) {
                    if
(thisElement.isEqual(field.getSouthElement(thisElement).getFARExtension().verticalAcce
ssTo)) reservationPriorityQueue.insertAtRear(field.getSouthElement(thisElement));
                }
            }

            if (field.getEastElement(thisElement).getFARExtension() != null) {
                if
(field.getEastElement(thisElement).getFARExtension().extraHorizontalAccessTo != null)
{
                    if
(thisElement.isEqual(field.getEastElement(thisElement).getFARExtension().extraHorizont
alAccessTo)) reservationPriorityQueue.insertAtRear(field.getEastElement(thisElement));
                }
            } if (field.getWestElement(thisElement).getFARExtension() != null) {
                if
(field.getWestElement(thisElement).getFARExtension().extraHorizontalAccessTo != null)
{
                    if
(thisElement.isEqual(field.getWestElement(thisElement).getFARExtension().extraHorizont
alAccessTo)) reservationPriorityQueue.insertAtRear(field.getWestElement(thisElement));
                }
            }

            if (field.getNorthElement(thisElement).getFARExtension() != null) {
                if
(field.getNorthElement(thisElement).getFARExtension().extraVerticalAccessTo != null) {
                    if
(thisElement.isEqual(field.getNorthElement(thisElement).getFARExtension().extraVertica
lAccessTo)) {

reservationPriorityQueue.insertAtRear(field.getNorthElement(thisElement));
                    }
                }
            }
            if (field.getSouthElement(thisElement).getFARExtension() != null) {
                if
(field.getSouthElement(thisElement).getFARExtension().extraVerticalAccessTo != null) {
                    if
(thisElement.isEqual(field.getSouthElement(thisElement).getFARExtension().extraVertica
lAccessTo)) {

reservationPriorityQueue.insertAtRear(field.getSouthElement(thisElement));
                    }
                }
            }
        }

        public void rotateReservationPriorityQueue() {
            if (reservationPriorityQueue != null) {
                Element movedElement = (Element)
reservationPriorityQueue.removeFirst();
                reservationPriorityQueue.insertAtRear(movedElement);
            }
        }

        public List getReservationPriorityQueue() {
            return reservationPriorityQueue;
        }

        public boolean isTopPriority(int reservationDepth, int reservationSlot,
FARAgent agent, int step, Element element) {
            if (!reservation.isWaitReservation()) {
                if (reservationPriorityQueue != null) {
                    if (!reservationPriorityQueue.isEmpty()) {
                        for (int i = 0; i < reservationPriorityQueue.size(); i++) {
                            Element currentPriorityElement = (Element)
```

```
reservationPriorityQueue.getNodeData(i);
                            if (currentPriorityElement != null) {
                                if (element.isEqual(currentPriorityElement)) {
                                    return false;
                                } else if
(reservation.getCameFrom().isEqual(currentPriorityElement)) {
                                    return true;
                                } else {
                                    continue;
                                }
                            } else {
                                continue;
                            }
                        }
                        initReservationPriorityQueue();
                        return false;
                    } else {
                        initReservationPriorityQueue();
                        return true;
                    }
                } else {
                    initReservationPriorityQueue();
                    return true;
                }
            } else {
                if (isProxyAvailable(reservationSlot, agent)) return true;
                else return false;
            }
        }

        public boolean isProxyAvailable(int reservationSlot, FARAgent agent) {
            List accessibleElements =
thisElement.getFARExtension().getAccessibleElements();

            boolean proxyFound = false;

            for (int i = 0; i < accessibleElements.size(); i++) {
                Element currentAccessible = (Element)
accessibleElements.getNodeData(i);

                if ((reservationSlot - 1) >= 0) {
                    if
(currentAccessible.getFARExtension().getReservation(reservationSlot) == null) {
                        agent.assignProxyPath(step, reservationSlot,
currentAccessible);
                        proxyFound = true;
                        break;
                    } else {
                        if
(currentAccessible.getFARExtension().getReservation(reservationSlot).getAgent() ==
null) {
                            agent.assignProxyPath(step, reservationSlot,
currentAccessible);
                            proxyFound = true;
                            break;
                        } else {
                            if
(currentAccessible.getFARExtension().isTopPriority(reservationDepth, reservationSlot,
currentAccessible.getFARExtension().getCurrentAgent(), step, thisElement)) {
                                agent.assignProxyPath(step, reservationSlot,
currentAccessible);
                                proxyFound = true;
                                break;
                            }
                        }
                    }
                } else {

thisElement.getFARExtension().getCurrentAgent().assignProxyPath(step, reservationSlot,
currentAccessible);
                    proxyFound = true;
                    break;
                }
            }

            if (proxyFound) {
```

```
                return true;
            } else {
                return false;
            }
        }

        public Reservation getReservation() {
            return reservation;
        }

        public void setReservation(FARAgent agent, Element cameFrom, boolean
isWaitReservation, Element waitingFor, boolean isProxyReservation) {
            reservation = new Reservation(agent, cameFrom, isWaitReservation,
waitingFor, isProxyReservation);
        }
    }

    private int step = 0;

    private Field field;

    private Element thisElement = null;

    private Element verticalAccessTo = null; // priority = 3
    private Element horizontalAccessTo = null; // priority = 4
    private Element extraHorizontalAccessTo = null; // priority = 2
    private Element extraVerticalAccessTo = null; // priority = 1

    private ReservationSlot [] reservationSlotArray;
    private int reservationDepth;

    private FARAgent currentAgent;

    public FARExtension(Field field, Element thisElement, int reservationDepth) {
        this.field = field;

        this.thisElement = thisElement;

        this.reservationDepth = reservationDepth;

        reservationSlotArray = new ReservationSlot[reservationDepth];
        for (int i = 0; i < reservationDepth; i++) {
            reservationSlotArray[i] = new ReservationSlot();
        }
    }

    public void setVerticalAccessTo(Element element) {
        verticalAccessTo = element;
    }
    public void setHorizontalAccessTo(Element element) {
        horizontalAccessTo = element;
    }
    public void setExtraHorizontalAccessTo(Element element) {
        extraHorizontalAccessTo = element;
    }
    public void setExtraVerticalAccessTo(Element element) {
        extraVerticalAccessTo = element;
    }

    public List getAccessibleElements() {
        List returnList = new List();

        if (verticalAccessTo != null) returnList.insertAtFront(verticalAccessTo);
        if (horizontalAccessTo != null) returnList.insertAtFront(horizontalAccessTo);
        if (extraHorizontalAccessTo != null)
returnList.insertAtFront(extraHorizontalAccessTo);
        if (extraVerticalAccessTo != null)
returnList.insertAtFront(extraVerticalAccessTo);

        return returnList;
    }

    public boolean isTopPriority(int reservationDepth, int reservationSlot, FARAgent
agent, int step, Element element) {
        return (reservationSlotArray[reservationSlot].isTopPriority(reservationDepth,
reservationSlot, agent, step, element));
```

```
    }

    public Reservation getReservation(int reservationSlot) {
        return reservationSlotArray[reservationSlot].getReservation();
    }

    public void setReservation(int reservationSlot, FARAgent agent, Element cameFrom,
boolean isWaitReservation, Element waitingFor, boolean isProxyReservation) {
        reservationSlotArray[reservationSlot].setReservation(agent, cameFrom,
isWaitReservation, waitingFor, isProxyReservation);
    }

    public void setCurrentAgent(FARAgent agent) {
        currentAgent = agent;
    }

    public FARAgent getCurrentAgent() {
        return currentAgent;
    }

    public void initReservationPriorityQueue(int reservationSlot) {
        reservationSlotArray[reservationSlot].initReservationPriorityQueue();
    }

    public List getReservationPriorityQueue(int reservationSlot) {
        return reservationSlotArray[reservationSlot].getReservationPriorityQueue();
    }

    public void rotateReservationPriorityQueue(int reservationSlot) {
        reservationSlotArray[reservationSlot].rotateReservationPriorityQueue();
    }

    public void initReservationSlotArray() {
        reservationSlotArray = new ReservationSlot[reservationDepth];
        for (int i = 0; i < reservationSlotArray.length; i++) {
            reservationSlotArray[i] = new ReservationSlot();
            reservationSlotArray[i].initReservationPriorityQueue();
        }
    }

    public void resetReservationSlotArray(int step) { // new value of step from which
the reservation slots are counted
        List [] reservationPriorityQueueArray = new List[reservationSlotArray.length];
        for (int i = 0; i < reservationPriorityQueueArray.length; i++) {
            rotateReservationPriorityQueue(i);
            reservationPriorityQueueArray[i] = getReservationPriorityQueue(i);
        }

        reservationSlotArray = new ReservationSlot[reservationDepth];
        for (int i = 0; i < reservationSlotArray.length; i++) {
            reservationSlotArray[i] = new
ReservationSlot(reservationPriorityQueueArray[i]);
        }
    }
}
```

5.1.5 FARPathfinder.java:

```
package farpackage;

import generalpackage.*;
import adtpackage.*;

/**
 *
 * @author Zbyněk Stara
 */
public class FARPathfinder implements Pathfinder {
    private final int RESERVATION_DEPTH = 3; // FAR's k parameter - reservation depth
    private final int FAILURE_CRITERION = 100; // over how many steps is the
pathfinding considered a failure

    private final int LAST_STEP = (((FAILURE_CRITERION / RESERVATION_DEPTH) *
RESERVATION_DEPTH) + RESERVATION_DEPTH);
```

```
    private Field field;

    private FARAgent [] farAgentArray;

    private TreeSet [] farAgentGroupArray;

    TreeSet alreadyFlagged;

    private int step = 0; // simulation step

    private List agentReservationOrderAtStep = new List();

    private int numFailures = 0;

    public FARPathfinder(Field field) {
        this.field = field;

        for (int i = 1; i < (field.X_DIMENSION - 1); i++) {
           for (int j = 1; j < (field.Y_DIMENSION - 1); j++) {
               field.getElement(i, j).setFARExtension(new FARExtension(field,
field.getElement(i, j), RESERVATION_DEPTH));
           }
        }

        fieldFlowAnnotation();

        farAgentArray = new FARAgent[field.getAgentArray().length];

        for (int i = 0; i < field.getAgentArray().length; i++) {
            Agent agent = field.getAgentArray()[i];

            FARAgent newAAgent = new FARAgent(RESERVATION_DEPTH, LAST_STEP, field,
this, field.getElement(agent.getStartX(), agent.getStartY()),
field.getElement(agent.getGoalX(), agent.getGoalY()), i);

            farAgentArray[i] = newAAgent;
        }

        farAgentGroupArray = new TreeSet[RESERVATION_DEPTH];
        for (int i = 0; i < RESERVATION_DEPTH; i++) {
            farAgentGroupArray[i] = new TreeSet();
        }

        TreeSet ungrouppedSet = new TreeSet();
        for (int i = 0; i < farAgentArray.length; i++) {
            ungrouppedSet.add(farAgentArray[i], farAgentArray[i].getAgentID());
        }

        int currentGroupNum = 0;

        while (!ungrouppedSet.isEmpty()) {
            FARAgent addedAgent = null;
            do {
                int randomAgentNum = (int) (Math.random() * (farAgentArray.length));
                addedAgent = (FARAgent)
ungrouppedSet.remove(farAgentArray[randomAgentNum].getAgentID());
            } while (addedAgent == null);

            farAgentGroupArray[currentGroupNum].add(addedAgent,
addedAgent.getAgentID());

            addedAgent.setAgentGroup(currentGroupNum);

            currentGroupNum += 1;
            if (currentGroupNum == RESERVATION_DEPTH) {
                currentGroupNum = 0;
            }
        }

        TreeSet unreservedSet = new TreeSet();
        for (int i = 0; i < farAgentArray.length; i++) {
            unreservedSet.add(farAgentArray[i], farAgentArray[i].getAgentID());
        }

        TreeSet startElements = getAgentStartElements();
```

```
        for (int i = 0; i < startElements.size(); i++) {
            Element currentElement = (Element) startElements.getNodeData(i);


currentElement.getFARExtension().setCurrentAgent(farAgentArray[currentElement.getStart
ForAgent()]);
        }

        int agentGroupForStep = 0;

        for (step = 1; step < FAILURE_CRITERION; step++) { // step 0 is the start goal
for all of them - step 1 is the first where reservation is needed

            agentReservationOrderAtStep = new List();

            alreadyFlagged = new TreeSet();

            TreeSet unassignedSet = new TreeSet();
            for (int i = 0; i < farAgentGroupArray[agentGroupForStep].size(); i++) {
                unassignedSet.add((FARAgent)
farAgentGroupArray[agentGroupForStep].getNodeData(i), ((FARAgent)
farAgentGroupArray[agentGroupForStep].getNodeData(i)).getAgentID());
            }

            while (!unassignedSet.isEmpty()) {
                FARAgent chosenAgent = null;
                do {
                    int randomAgentNum = (int) (Math.random() *
(farAgentGroupArray[agentGroupForStep].size()));
                    chosenAgent = (FARAgent) unassignedSet.remove(((FARAgent)
farAgentGroupArray[agentGroupForStep].getNodeData(randomAgentNum)).getAgentID());
                } while (chosenAgent == null);

                agentReservationOrderAtStep.insertAtRear(chosenAgent);

                unreservedSet.remove(chosenAgent.getAgentID());
            }

            if (!unreservedSet.isEmpty()) {
                for (int j = 0; j < unreservedSet.size(); j++) {
                    FARAgent currentUnreservedAgent = (FARAgent)
unreservedSet.getNodeData(j);

                    currentUnreservedAgent.preplanningWaitReservation(step);
                }
            }

            for (int i = 0; i < agentReservationOrderAtStep.size(); i++) {
                FARAgent currentAgent = (FARAgent)
agentReservationOrderAtStep.getNodeData(i);

                System.out.println("\tcurrent agent for reservation: " +
currentAgent.getAgentID());
                currentAgent.reservePath(RESERVATION_DEPTH, step);
            }

            agentGroupForStep += 1;
            if (agentGroupForStep == RESERVATION_DEPTH) {
                agentGroupForStep = 0;
            }

            for (int i = 0; i < farAgentArray.length; i++) { // for the current step,
move everyone to their reserved positions
                farAgentArray[i].moveToReservation(RESERVATION_DEPTH, step);
            }

            for (int c = 0; c < field.getTraversableSet().size(); c++) { // move all
reservations back one reservation slot
                Element currentElement = (Element)
field.getTraversableSet().getNodeData(c);

                for (int i = 0; i < RESERVATION_DEPTH; i++) {
                    if ((i + 1) == RESERVATION_DEPTH) {
                        currentElement.getFARExtension().setReservation(i, null, null,
false, null, false);
                    } else {
```

```
                        Reservation movedReservation =
currentElement.getFARExtension().getReservation(i + 1);
                        if (movedReservation != null)
currentElement.getFARExtension().setReservation(i, movedReservation.getAgent(),
movedReservation.getCameFrom(), movedReservation.isWaitReservation(),
movedReservation.getWaitingFor(), movedReservation.isProxyReservation());
                        else currentElement.getFARExtension().setReservation(i, null,
null, false, null, false);
                    }
                }
            }
            System.out.println("Done");
        }

        System.out.println("After coordination stage");
        for (int i = 0; i < farAgentArray.length; i++) {
            FARAgent currentAgent = farAgentArray[i];

            if (!currentAgent.isComplete()) numFailures += 1;
        }
    }

    public void flagAgent(FARAgent agent, int step, boolean isEasy) {
        if (isEasy) {
            Element [] reservedElementsArray = agent.getReservedElementsArray();
            for (int i = 0; i < reservedElementsArray.length; i++) {
                if (reservedElementsArray[i] != null) {

reservedElementsArray[i].getFARExtension().resetReservationSlotArray(i);
                    reservedElementsArray[i] = null;
                }
            }

            agentReservationOrderAtStep.insertAtRear(agent);
        } else {
            Element[] reservedElementsArray = agent.getReservedElementsArray();
            for (int i = 0; i < reservedElementsArray.length; i++) {
                if (reservedElementsArray[i] != null) {

reservedElementsArray[i].getFARExtension().resetReservationSlotArray(i);
                    reservedElementsArray[i] = null;
                }
            }

            agentReservationOrderAtStep.insertAtRear(agent);
        }
    }

    public TreeSet getAlreadyFlagged() {
        return alreadyFlagged;
    }

    public void addAlreadyFlagged(FARAgent flagged) {
        alreadyFlagged.add(flagged, flagged.getAgentID());
    }

    public int getFailureCriterion() {
        return FAILURE_CRITERION;
    }

    public TreeSet getAgentsAtStep(int drawStep) {
        TreeSet elementSet = new TreeSet();

        for (int i = 0; i < farAgentArray.length; i++) {
            FARAgent currentAgent = farAgentArray[i];
            if (drawStep < currentAgent.getAgentPath().size()) {
                elementSet.add((Element)
currentAgent.getAgentPath().getNodeData(drawStep), currentAgent.getAgentID());
            } else {
                elementSet.add(currentAgent.GOAL, currentAgent.getAgentID());
            }
        }

        return elementSet;
    }
```

```
    public TreeSet getAgentPathsUntilStep(int drawStep) {
        TreeSet elementSet = new TreeSet();

        for (int i = 0; i < farAgentArray.length; i++) {
            FARAgent currentAgent = farAgentArray[i];
            for (int j = 1; j < drawStep; j++) {
                if (j < (currentAgent.getAgentPath().size() - 1)) {
                    Element currentElement = (Element)
currentAgent.getAgentPath().getNodeData(j);
                    elementSet.add(currentElement,
currentElement.getKey(field.Y_DIMENSION));
                }
            }
        }

        return elementSet;
    }

    public TreeSet getAgentStartElements() {
        TreeSet elementSet = new TreeSet();

        for (int i = 0; i < farAgentArray.length; i++) {
            FARAgent currentAgent = farAgentArray[i];
            elementSet.add(currentAgent.START, currentAgent.getAgentID());
        }

        return elementSet;
    }

    public TreeSet getAgentGoalElements() {
        TreeSet elementSet = new TreeSet();

        for (int i = 0; i < farAgentArray.length; i++) {
            FARAgent currentAgent = farAgentArray[i];
            elementSet.add(currentAgent.GOAL, currentAgent.getAgentID());
        }

        return elementSet;
    }

    public int getNumFailures() {
        return numFailures;
    }

    private void fieldFlowAnnotation() {
        for (int i = 1; i < (field.X_DIMENSION - 1); i++) {
            for (int j = 1; j < (field.Y_DIMENSION - 1); j++) {
                Element currentElement = field.getElement(i, j);

                boolean flowNorthward = false;
                boolean flowSouthward = false;
                boolean flowWestward = false;
                boolean flowEastward = false;

                boolean nBl = field.getNorthElement(currentElement).isBlocked();
                boolean neBl = field.getNorthEastElement(currentElement).isBlocked();
                boolean eBl = field.getEastElement(currentElement).isBlocked();
                boolean seBl = field.getSouthEastElement(currentElement).isBlocked();
                boolean sBl = field.getSouthElement(currentElement).isBlocked();
                boolean swBl = field.getSouthWestElement(currentElement).isBlocked();
                boolean wBl = field.getWestElement(currentElement).isBlocked();
                boolean nwBl = field.getNorthWestElement(currentElement).isBlocked();

                boolean northBlocked =
field.getNorthElement(currentElement).isBlocked();
                boolean eastBlocked =
field.getEastElement(currentElement).isBlocked();
                boolean southBlocked =
field.getSouthElement(currentElement).isBlocked();
                boolean westBlocked =
field.getWestElement(currentElement).isBlocked();

                if (!currentElement.isObstacle()) {
                    if ((i % 2) == 1) {
                        flowNorthward = true;
                    } else {
```

```
                        flowSouthward = true;
                    }

                    if ((j % 2) == 1) {
                        flowEastward = true;
                    } else {
                        flowWestward = true;
                    }

                    if (flowEastward && !eastBlocked) {

currentElement.getFARExtension().setHorizontalAccessTo(field.getEastElement(currentEle
ment));
                    } else if (flowWestward && !westBlocked) {

currentElement.getFARExtension().setHorizontalAccessTo(field.getWestElement(currentEle
ment));
                    }

                    if (flowNorthward && !northBlocked) {

currentElement.getFARExtension().setVerticalAccessTo(field.getNorthElement(currentElem
ent));
                    } else if (flowSouthward && !southBlocked) {

currentElement.getFARExtension().setVerticalAccessTo(field.getSouthElement(currentElem
ent));
                    }

                    if (flowNorthward && !northBlocked) {
                        boolean nnBl =
field.getNorthElement(field.getNorthElement(currentElement)).isBlocked();
                        if (flowEastward) {
                            boolean nneBl =
field.getNorthEastElement(field.getNorthElement(currentElement)).isBlocked();
                            if (!((!wBl && !nwBl) || (!nnBl && !nneBl && !neBl && !eBl
&& !seBl && !sBl))) {

field.getNorthElement(currentElement).getFARExtension().setExtraVerticalAccessTo(curre
ntElement);
                            }
                        } else if (flowWestward) {
                            boolean nnwBl =
field.getNorthWestElement(field.getNorthElement(currentElement)).isBlocked();
                            if (!((!eBl && !neBl) || (!nnBl && !nnwBl && !nwBl && !
wBl && !swBl && !sBl))) {

field.getNorthElement(currentElement).getFARExtension().setExtraVerticalAccessTo(curre
ntElement);
                            }
                        }
                    } else if (flowSouthward && !southBlocked) {
                        boolean ssBl =
field.getSouthElement(field.getSouthElement(currentElement)).isBlocked();
                        if (flowEastward) {
                            boolean sseBl =
field.getSouthEastElement(field.getSouthElement(currentElement)).isBlocked();
                            if (!((!wBl && !swBl) || (!ssBl && !sseBl && !seBl && !eBl
&& !neBl && !nBl))) {

field.getSouthElement(currentElement).getFARExtension().setExtraVerticalAccessTo(curre
ntElement);
                            }
                        } else if (flowWestward) {
                            boolean sswBl =
field.getSouthWestElement(field.getSouthElement(currentElement)).isBlocked();
                            if (!((!eBl && !seBl) || (!ssBl && !sswBl && !swBl && !wBl
&& !nwBl && !nBl))) {

field.getSouthElement(currentElement).getFARExtension().setExtraVerticalAccessTo(curre
ntElement);
                            }
                        }
                    }

                    if (flowEastward && !eastBlocked) {
```

```
                        boolean eeBl =
field.getEastElement(field.getEastElement(currentElement)).isBlocked();
                        if (flowNorthward) {
                            boolean eneBl =
field.getNorthEastElement(field.getEastElement(currentElement)).isBlocked();
                            if (!((!sBl && !seBl) || (!eeBl && !eneBl && !neBl && !nBl
&& !nwBl && !wBl))) {

field.getEastElement(currentElement).getFARExtension().setExtraHorizontalAccessTo(curr
entElement);
                            }
                        } else if (flowSouthward) {
                            boolean eseBl =
field.getSouthEastElement(field.getEastElement(currentElement)).isBlocked();
                            if (!((!nBl && !neBl) || (!eeBl && !eseBl && !seBl && !sBl
&& !swBl && !wBl))) {

field.getEastElement(currentElement).getFARExtension().setExtraHorizontalAccessTo(curr
entElement);
                            }
                        }
                    } else if (flowWestward && !westBlocked) {
                        boolean wwBl =
field.getWestElement(field.getWestElement(currentElement)).isBlocked();
                        if (flowNorthward) {
                            boolean wnwBl =
field.getNorthWestElement(field.getWestElement(currentElement)).isBlocked();
                            if (!((!sBl && !swBl) || (!wwBl && !wnwBl && !nwBl && !nBl
&& !neBl && !eBl))) {

field.getWestElement(currentElement).getFARExtension().setExtraHorizontalAccessTo(curr
entElement);
                            }
                        } else if (flowSouthward) {
                            boolean wswBl =
field.getSouthWestElement(field.getWestElement(currentElement)).isBlocked();
                            if (!((!nBl && !nwBl) || (!wwBl && !wswBl && !swBl && !sBl
&& !seBl && !eBl))) {

field.getWestElement(currentElement).getFARExtension().setExtraHorizontalAccessTo(curr
entElement);
                            }
                        }
                    }
                }
            }
        }
    }

    @Override public String toString() {
        return ("Pathfinder at step " + step + ": Number of AAgents: " +
farAgentArray.length);
    }
}
```

5.1.6. Pathfinder.java:

```
package generalpackage;

import adtpackage.*;

/**
 *
 * @author Zbyněk Stara
 */
public interface Pathfinder {
    public abstract int getFailureCriterion();

    public abstract TreeSet getAgentsAtStep(int drawStep);
    public abstract TreeSet getAgentPathsUntilStep(int drawStep);
    public abstract TreeSet getAgentStartElements();
    public abstract TreeSet getAgentGoalElements();
}
```

5.1.7 FARAgent.java:

```
package farpackage;
import generalpackage.*;
import adtpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class FARAgent implements Printable {
    private Field field;
    private FARPathfinder pathfinder;
    private FAR farAlgorithm = new FAR();
    private final int FARAGENT_ID;
    public final Element START;
    public final Element GOAL;
    private List initialAgentPath; // the initial, pre-computed A* path
    private List agentPath = new List (); // the final, coordinated FAR path
    private int step = 0; // the first step of current reservation bunch
    private int reservationDepth;
    private boolean alreadyReserved = false;
    private int waitsNum = 0;
    private int currentReservationSlot = 0; // incremented when moving or waiting,
resetted at new reservation = if it got reservation_depth moves, set to 0
    private Element [] reservedElementsArray;
    private int agentGroup;
    private boolean isComplete = false;
    public FARAgent(int reservationDepth, int lastStep, Field field, FARPathfinder
pathfinder, Element start, Element goal, int id) {
        this.field = field;
        this.reservationDepth = reservationDepth;
        this.pathfinder = pathfinder;
        FARAGENT_ID = id;
        farAlgorithm.resetAExtensions(field);
        initialAgentPath = farAlgorithm.far(start, goal, field);
        for (int i = 0; i <= (lastStep + 1); i++) {
            if (i < initialAgentPath.size()) {
                agentPath.insertAtRear((Element) initialAgentPath.getNodeData(i));
            } else {
                agentPath.insertAtRear((Element)
initialAgentPath.getNodeData(initialAgentPath.size() - 1)) ;
            }
        }
        START = start;
        GOAL = goal;
    }
    public List getInitialAgentPath() {
        return initialAgentPath;
    }
    public List getAgentPath() {
        return agentPath;
    }
    public int getAgentGroup() {
        return agentGroup;
    }
    public void setAgentGroup(int agentGroup) {
        this.agentGroup = agentGroup;
    }
    public Element [] getReservedElementsArray() {
        return reservedElementsArray;
    }
    public int getAgentID() {
        return FARAGENT_ID;
    }
    public boolean isComplete() {
        return isComplete;
    }
    public boolean getAlreadyReserved() {
        return alreadyReserved;
    }
    public void reservePath(int reservationDepth, int step) {
        this.step = step;
        alreadyReserved = true;
```

```
        reservedElementsArray = new Element[reservationDepth];
        waitsNum = 0;

                                                                                // var
currentReservationSlot used there to allow hard-flagged replans
        for (int i = 0; i < reservationDepth - currentReservationSlot; i++) { // i ==
current reservation slot at element of path
            Element currentPathElement = (Element) agentPath.getNodeData((step + i) -
waitsNum); // the element we want to move to, in ideal case
            Element previousPathElement = (Element) agentPath.getNodeData(((step + i)
- waitsNum) - 1);
            Reservation destinationReservation =
currentPathElement.getFARExtension().getReservation(i);
            if ((destinationReservation == null) || (destinationReservation.getAgent()
== null)) {
                // HAS TO CHECK FOR HEAD-ON COLLISIONS:
                // if ((i - 1) > 0) {
                    // if
(!previousPathElement.getFARExtension().getReservation(i).getReservation().getAgent().
getAgentID() == currentPathElement.getFARExtension().getReservation(i -
1).getAgent().getAgentID()) {
                        // it is as usual
                    // else { // if the reservation for previousPathElement for next
slot comes from the element it wants to reserve for next slot
                        // there is a head-on collision
                        // THAT MEANS THAT IT CANNOT MOVE there AND WHAT IS MORE, IT
NEEDS TO RESERVE A PROXY PATH
                        // if (

                    //}
                // else {
                    // if
(!previousPathElement.getFARExtension().getReservation(i).getReservation().getAgent().
getAgentID() == currentPathElement.getFARExtension().getCurrentAgentID()) {
                        // it is as usual
                    // else { // if the reservation for previousPathElement for next
slot comes from the element it wants to reserve for next slot
                        // there is a head-on collision
                    //}
                //}
                System.out.println("\t\tcurrent agent " + FARAGENT_ID + " reserves
element" + currentPathElement.toString() + " at " + (step + i));
                currentPathElement.getFARExtension().setReservation(i, this,
previousPathElement, false, null, false);
                if (i + 1 != reservationDepth)
currentPathElement.getFARExtension().initReservationPriorityQueue(i + 1);
                reservedElementsArray[i] = currentPathElement;
            } else { // there is a reservation on the destination already
                if
(currentPathElement.getFARExtension().getReservation(i).getAgent().getAgentID() ==
FARAGENT_ID) {

                    continue; // it has already been reserved

                } else if
(currentPathElement.getFARExtension().isTopPriority(reservationDepth, i, this, step,
previousPathElement)
                        &&
(!pathfinder.getAlreadyFlagged().contains(currentPathElement.getFARExtension().getRese
rvation(i).getAgent().getAgentID()))) { // overwriting current reservation
                    // HAS TO CHECK FOR HEAD-ON COLLISIONS

pathfinder.addAlreadyFlagged(currentPathElement.getFARExtension().getReservation(i).ge
tAgent());

pathfinder.flagAgent(currentPathElement.getFARExtension().getReservation(i).getAgent()
, step,
(currentPathElement.getFARExtension().getReservation(i).getAgent().getAgentGroup() ==
agentGroup));
                    // after-proxy flagging moved to assign_proxy_path method
                    System.out.println("\t\tAgent " + FARAGENT_ID + " reserves element
" + currentPathElement.toString() + " at " + (step + i)
                            + ". Agent " +
destinationReservation.getAgent().getAgentID() + " flagged (lower priority)");
                    currentPathElement.getFARExtension().setReservation(i, this,
previousPathElement, false, null, false);
```

```
                    if (i + 1 != reservationDepth)
currentPathElement.getFARExtension().rotateReservationPriorityQueue(i + 1);
                    reservedElementsArray[i] = currentPathElement;
                } else { // there is a reservation on the destination & this agent is
not allowed to override it
                    if ((previousPathElement.getFARExtension().getReservation(i) ==
null)
                            ||
(previousPathElement.getFARExtension().getReservation(i).getAgent() == null)) {
                        previousPathElement.getFARExtension().setReservation(i, this,
previousPathElement, true, previousPathElement, false);
                        if (i + 1 != reservationDepth)
previousPathElement.getFARExtension().initReservationPriorityQueue(i + 1);
                        System.out.println("\t\tAgent " + FARAGENT_ID + " fails to
reserve element " + currentPathElement.toString() + " at " + (step + i)
                                + " due to agent " +
destinationReservation.getAgent().getAgentID() + " (lower priority) and waits at
element " + previousPathElement.toString());
                        reservedElementsArray[i] = previousPathElement;
                        waitsNum += 1;
                    } else { // there is a reservation on the destination which this
agent cannot override & there is a reservation on this agent's current position

pathfinder.addAlreadyFlagged(previousPathElement.getFARExtension().getReservation(i).g
etAgent());

pathfinder.flagAgent(previousPathElement.getFARExtension().getReservation(i).getAgent(
), step,
(previousPathElement.getFARExtension().getReservation(i).getAgent().getAgentGroup() ==
agentGroup));
                        System.out.println("\t\tAgent " + FARAGENT_ID + " fails to
reserve element " + currentPathElement.toString() + " at " + (step + i)
                                + " due to agent " +
destinationReservation.getAgent().getAgentID() + " (lower priority) and waits at
element " + previousPathElement.toString()
                                +
((previousPathElement.getFARExtension().getReservation(i) != null)
                                        ?
(previousPathElement.getFARExtension().getReservation(i).getAgent() != null)
                                                ? (". Agent " +
previousPathElement.getFARExtension().getReservation(i).getAgent().getAgentID() + "
wanting to go there was flagged" )
                                                : (". There is no reservation at this
element")
                                        : (""))
                                );
                        previousPathElement.getFARExtension().setReservation(i, this,
previousPathElement, true, previousPathElement, false);
                        if (i + 1 != reservationDepth)
previousPathElement.getFARExtension().initReservationPriorityQueue(i + 1);
                        reservedElementsArray[i] = previousPathElement;
                        waitsNum += 1;
                    }
                }
            }
        }
    }
    public void moveToReservation(int reservationDepth, int step) { // actually move
them - increments step (by definition) + the currentReservationSlot (if res_depth,
change it to zero) + move back all reservations
        this.step = step;
        System.out.println("Current step for agent " + FARAGENT_ID + ": " + step);
        Element reservedElement = reservedElementsArray[0];
        Element nextPathElement = (Element) agentPath.getNodeData(step);
        Element waitPathElement = (Element) agentPath.getNodeData(step - 1);
        if (reservedElement.isEqual(waitPathElement)) { // waiting
            agentPath.insertAsNode(waitPathElement, step); // waiting - adding that to
the path
            agentPath.removeLast(); // keeping the agent path the same distance
        } else if (reservedElement.isEqual(nextPathElement)) { // moving out
            reservedElement.getFARExtension().setCurrentAgent(null);
            nextPathElement.getFARExtension().setCurrentAgent(this);
            if (reservedElement.isEqual(GOAL)) {
                isComplete = true;
            }
        } else {
```

```
            System.out.println("Error");
        }
        if (alreadyReserved) {
            currentReservationSlot += 1;
            if (currentReservationSlot == reservationDepth) {
                currentReservationSlot = 0;
            }
       }
        for (int i = 0; i < reservedElementsArray.length; i++) {
            if ((i + 1) == reservedElementsArray.length) {
                reservedElementsArray[i] = null;
            } else {
                reservedElementsArray[i] = reservedElementsArray[i + 1];
            }
        }
    }
    public void preplanningWaitReservation(int step) {
        reservedElementsArray = new Element[reservationDepth];
        if ((START.getFARExtension().getReservation(0) != null) &&
(START.getFARExtension().getReservation(0).getAgent() != null)) {
            pathfinder.flagAgent(START.getFARExtension().getReservation(0).getAgent(),
step, (START.getFARExtension().getReservation(0).getAgent().getAgentGroup() ==
agentGroup));
            START.getFARExtension().setReservation(0, this, START, true, START,
false);
            reservedElementsArray[0] = START;
        } else {
            START.getFARExtension().setReservation(0, this, START, true, START,
false);
            reservedElementsArray[0] = START;
        }
    }
    public void assignProxyPath(int step, int reservationSlot, Element proxyStart) {
        Element previousElement = (Element) agentPath.getNodeData(step - 1);
        Element postponedElement = (Element) agentPath.getNodeData(step);
        List proxyPath = farAlgorithm.far(proxyStart, postponedElement, field);
        for (int i = 0; i < (proxyPath.size() - 1); i++) {
            Element currentProxyPathElement = (Element) proxyPath.getNodeData(i);
            agentPath.insertAsNode(currentProxyPathElement, step + i);
            agentPath.removeLast(); // keeping the agentPath (lastStep + 1) long
        }
    }
    @Override public String print() {
        return toString();
    }
    @Override public String toString() {
        String string = "";
        string += "FARAgent ID: ";
        string += FARAGENT_ID;
        string += "; Start: ";
        string += START.print();
        string += "; Goal: ";
        string += GOAL.print();
        string += "; Path: [";
        string += agentPath.print();
        string += "]";
        return string;
    }
}
```

5.1.8. Agent.java:

```
package generalpackage;
import adtpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class Agent {
    private Field field;
    private int agentID = -999;
    public int startX = -999;
    public int startY = -999;
    public int goalX = -999;
```

```
    public int goalY = -999;
    public Agent(Field field, HashSet freeSet, int id) throws Exception {
        this.field = field;
        agentID = id;
        if (freeSet.size() > 1) {
            do {
                startX = (int) ((Math.random() * (field.X_DIMENSION - 2)) + 1);
                startY = (int) ((Math.random() * (field.Y_DIMENSION - 2)) + 1);
            } while (!freeSet.contains(field.getElement(startX,
startY).getKey(field.Y_DIMENSION)));
            field.getElement(startX, startY).setStartForAgent(id);
            freeSet.remove(field.getElement(startX,
startY).getKey(field.Y_DIMENSION));
            do {
                goalX = (int) ((Math.random() * (field.X_DIMENSION - 2)) + 1);
                goalY = (int) ((Math.random() * (field.Y_DIMENSION - 2)) + 1);
            } while (!freeSet.contains(field.getElement(goalX,
goalY).getKey(field.Y_DIMENSION)));
            field.getElement(goalX, goalY).setGoalForAgent(id);
            freeSet.remove(field.getElement(goalX, goalY).getKey(field.Y_DIMENSION));
            /*AAgent newAAgent = new AAgent(field.getElement(startX, startY),
field.getElement(goalX, goalY), field, newAAgentID);
            aAgentSet.add(newAAgent, newAAgentID);
            newAAgentID++;*/
        } else { // not enough free spaces to make the starts and goals
            throw new Exception();
        }
    }
    public int getStartX() {
        return startX;
    }
    public int getStartY() {
        return startY;
    }
    public Element getStart() {
        return field.getElement(startX, startY);
    }

    public int getGoalX() {
        return goalX;
    }
    public int getGoalY() {
        return goalY;
    }
    public Element getGoal() {
        return field.getElement(goalX, goalY);
    }
    public int getAgentID() {
        return agentID;
    }
}
```

5.1.9. FAR.java:

```
package farpackage;

import apackage.*;
import adtpackage.*;
import generalpackage.*;

/**
 *
 * @author Zbyněk Stara
 */
public class FAR {
    public FAR() {

    }

    public void resetAExtensions(Field field) {
        for (int i = 0; i < field.X_DIMENSION; i++) {
            for (int j = 0; j < field.Y_DIMENSION; j++) {
                field.getElement(i, j).setAExtension(new AExtension());
            }
```

```
        }
    }

    public List far(Element start, Element goal, Field field) {
        HashSet closedSet = new HashSet((field.X_DIMENSION - 2) * (field.Y_DIMENSION -
2));  // The set of nodes already evaluated.
                                        // ORDERED BY ELEMENT KEYS
        MinimalQueue openSet = new MinimalQueue();    // The set of tentative nodes to
be evaluated, initially containing the start node
                                        // ORDERED BY F_SCORES

        double gScore = 0;                                  // Cost from start along
best known path.
        double hScore = field.manhattanDistanceBetween(start, goal); //
distanceBetween == heuristic_cost_estimate
        double fScore = gScore + hScore;                    // Estimated total cost
from start to goal through y.

        start.getAExtension().setScores(gScore, hScore, fScore);

        openSet.enqueue(start, start.getAExtension().getFScore());

        while (!openSet.isEmpty()) {
            Element current = openSetChooser(field, openSet, closedSet);
            List neighborList = current.getFARExtension().getAccessibleElements();

            if (current.isEqual(goal)) {
                return reconstructPath(start, goal);
            }

            while (!neighborList.isEmpty()) {
                Element currentNeighbor = (Element) neighborList.removeFirst();
                double tentativeGScore = -999;

                if ((currentNeighbor.isBoundary()) || (currentNeighbor.isObstacle()))
{
                    continue;
                } else if (!field.isLegalMove(current, currentNeighbor)) {
                    continue;
                } else if
(closedSet.contains(currentNeighbor.getKey(field.Y_DIMENSION))) {
                    continue;
                } else {
                    tentativeGScore = current.getAExtension().getGScore() +
field.manhattanDistanceBetween(current, currentNeighbor);
                }

                if (!openSet.contains(currentNeighbor.getAExtension().getFScore())) {
// if current neighbor is not in open set

                    currentNeighbor.getAExtension().setCameFrom(current);

                    gScore = tentativeGScore;
                    hScore = field.manhattanDistanceBetween(currentNeighbor, goal);
                    fScore = gScore + hScore;

                    currentNeighbor.getAExtension().setScores(gScore, hScore, fScore);

                    openSet.enqueue(currentNeighbor,
currentNeighbor.getAExtension().getFScore());
                }
                else if (tentativeGScore <
currentNeighbor.getAExtension().getGScore()) { // else if it is in openSet but new
GScore is better
                    openSet.remove(currentNeighbor.getAExtension().getFScore());

                    currentNeighbor.getAExtension().setCameFrom(current);

                    gScore = tentativeGScore;
                    hScore = currentNeighbor.getAExtension().getHScore(); // hScore is
already set for this one
                    fScore = gScore + hScore;

                    currentNeighbor.getAExtension().setScores(gScore, hScore, fScore);

                    openSet.enqueue(currentNeighbor,
```

```
currentNeighbor.getAExtension().getFScore());
                } else {
                } // else nothing happens
            }

            closedSet.add(current, current.getKey(field.Y_DIMENSION));
        }
        return null;
    }

    private Element openSetChooser(Field field, MinimalQueue openSet, HashSet
closedSet) {
        Element currentElement = null;
        Element firstElement = null;

        int i = 0;

        do {
            do {
                currentElement = (Element) openSet.dequeue();
            } while (closedSet.contains(currentElement.getKey(field.Y_DIMENSION)));

            if (i == 0) {
                firstElement = currentElement;
            }

            if (currentElement.getAExtension().getCameFrom() != null) {
                if
(currentElement.getAExtension().getCameFrom().getAExtension().getCameFrom() != null) {
                    if
(currentElement.getAExtension().getCameFrom().isEqual(field.getNorthElement(currentEle
ment))
                            &&
currentElement.getAExtension().getCameFrom().getAExtension().getCameFrom().isEqual(fie
ld.getNorthElement(field.getNorthElement(currentElement)))) {
                            // if element's cameFrom and its cameFrom are from the
same direction

                        return currentElement;
                    } else if
(currentElement.getAExtension().getCameFrom().isEqual(field.getEastElement(currentElem
ent))
                            &&
currentElement.getAExtension().getCameFrom().getAExtension().getCameFrom().isEqual(fie
ld.getEastElement(field.getEastElement(currentElement)))) {

                        return currentElement;
                    } else if
(currentElement.getAExtension().getCameFrom().isEqual(field.getSouthElement(currentEle
ment))
                            &&
currentElement.getAExtension().getCameFrom().getAExtension().getCameFrom().isEqual(fie
ld.getSouthElement(field.getSouthElement(currentElement)))) {

                        return currentElement;
                    } else if
(currentElement.getAExtension().getCameFrom().isEqual(field.getWestElement(currentElem
ent))
                            &&
currentElement.getAExtension().getCameFrom().getAExtension().getCameFrom().isEqual(fie
ld.getWestElement(field.getWestElement(currentElement)))) {

                        return currentElement;
                    } else { // if element's cameFrom and its cameFrom are not from
the same direction
                        openSet.enqueue(currentElement,
currentElement.getAExtension().getFScore());
                    }
                } else {
                    return firstElement;
                }
            } else {
                return firstElement;
            }

            i++;
```

```
        } while ((!currentElement.isEqual(firstElement)) || (i == 1));

        return firstElement;
    }

    private List reconstructPath(Element startNode, Element currentNode) {
        List path;
        if (currentNode != startNode) { //came_from[current_node] is set {
            List tempPath = reconstructPath(startNode,
currentNode.getAExtension().getCameFrom());
            path = tempPath;
            path.insertAtRear(currentNode);
        }
        else {
            path = new List();
            path.insertAtRear(currentNode);
        }
        return path;
    }
}
```

5.1.10. Reservation.java:

```
package farpackage;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class Reservation implements Printable {
    private FARAgent agent = null;
    private Element cameFrom = null;
    private boolean isWaitReservation = false;
    private Element waitingFor = null;
    private boolean isProxyReservation = false;
    public Reservation() {
    }
    public Reservation(FARAgent agent, Element cameFrom) {
        this.agent = agent;
        this.cameFrom = cameFrom;
    }
    public Reservation(FARAgent agent, Element cameFrom, boolean isWaitReservation,
Element waitingFor, boolean isProxyReservation) {
        this.agent = agent;
        this.cameFrom = cameFrom;
        this.isWaitReservation = isWaitReservation;
        this.waitingFor = waitingFor;
        this.isProxyReservation = isProxyReservation;
    }
    public FARAgent getAgent() {
        return agent;
    }
    public Element getCameFrom() {
        return cameFrom;
    }
    public boolean isWaitReservation() {
        return isWaitReservation;
    }
    public Element getWaitingFor() {
        return waitingFor;
    }
    public boolean isProxyReservation() {
        return isProxyReservation;
    }
    public String print() {
        return toString();
    }
    @Override
    public String toString() {
        return ("Request from agent " + agent.getAgentID() + " coming from " +
cameFrom.print());
    }
}
```

5.1.11. FreeGroup.java:

```
package generalpackage;
import adtpackage.TreeSet;
/**
 *
 * @author Zbyněk Stara
 */
public class FreeGroup implements Printable {
    private Field field;
    private int freeGroupNumber = -999;
    private int numberElements = -999;
    private TreeSet elementSet;
    public FreeGroup(int freeGroupNumber, Element element, Field field) {
        this.field = field;
        this.freeGroupNumber = freeGroupNumber;
        elementSet = new TreeSet();
        numberElements = 1;
        elementSet.add(element, element.getKey(field.Y_DIMENSION));
    }
    public void addElement(Element element) {
        numberElements += 1;
        elementSet.add(element, element.getKey(field.Y_DIMENSION));
    }
    public void removeElement(Element element) {
        numberElements -= 1;
        elementSet.remove(element.getKey(field.Y_DIMENSION));
    }
    public Element removeMinElement() {
        numberElements -= 1;
        return (Element) elementSet.removeMin();
    }
    public int getFreeGroupNumber() {
        return freeGroupNumber;
    }
    public int getNumberElements() {
        return numberElements;
    }
    @Override public String print() {
        return toString();
    }
    @Override public String toString() {
        return "FreeGroup number: " + freeGroupNumber + "; Number of elements: " +
numberElements + "; Contents: {" + elementSet.print() + "}";
    }
}
```

5.1.12. AExtension.java:

```
package apackage;

import generalpackage.*;

/**
 *
 * @author Zbyněk Stara
 */
public class AExtension {
    private double gScore = -999; // cost along best known path = distance
    private double hScore = -999; // heuristics
    private double fScore = -999; // distance + heuristics combined

    private Element cameFrom = null;

    public AExtension() {

    }

    public void setScores(double gScore, double hScore, double fScore) {
        this.gScore = gScore;
```

```
        this.hScore = hScore;
        this.fScore = fScore;
    }

    public void setGScore(double gScore) {
        this.gScore = gScore;
    }
    public void setHScore(double hScore) {
        this.hScore = hScore;
    }
    public void setFScore(double fScore) {
        this.fScore = fScore;
    }

    public double getGScore() {
        return gScore;
    }
    public double getHScore() {
        return hScore;
    }
    public double getFScore() {
        return fScore;
    }

    public void setCameFrom(Element element) {
        cameFrom = element;
    }
    public Element getCameFrom() {
        return cameFrom;
    }
}
```

5.1.13. APathfinder.java:

```
package apackage;
import adtpackage.*;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class APathfinder implements Pathfinder {
    private int FAILURE_CRITERION = 100; // not used, just for compatibility with
table drawing methods
    private Field field;
    private AAgent[] aAgentArray;
    int step = 0;
    public APathfinder(Field field) {
        this.field = field;
        aAgentArray = new AAgent[field.getAgentArray().length];
        for (int i = 0; i < field.getAgentArray().length; i++) {
            Agent agent = field.getAgentArray()[i];
            AAgent newAAgent = new AAgent(field, field.getElement(agent.getStartX(),
agent.getStartY()), field.getElement(agent.getGoalX(), agent.getGoalY()), i);
            aAgentArray[i] = newAAgent;
        }
    }
    public int getFailureCriterion() {
        return FAILURE_CRITERION;
    }
    public TreeSet getAgentsAtStep(int drawStep) {
        TreeSet elementSet = new TreeSet();
        for (int i = 0; i < aAgentArray.length; i++) {
            AAgent currentAgent = aAgentArray[i];
            if (drawStep < currentAgent.getAgentPath().size()) {
                elementSet.add((Element)
currentAgent.getAgentPath().getNodeData(drawStep), currentAgent.getAgentID());
            } else {
                elementSet.add(currentAgent.GOAL, currentAgent.getAgentID());
            }
        }
        return elementSet;
    }
    public TreeSet getAgentPathsUntilStep(int drawStep) {
```

```
        TreeSet elementSet = new TreeSet();
        for (int i = 0; i < aAgentArray.length; i++) {
            AAgent currentAgent = aAgentArray[i];
            for (int j = 1; j < drawStep; j++) {
                if (j < (currentAgent.getAgentPath().size() - 1)) {
                    Element currentElement = (Element)
currentAgent.getAgentPath().getNodeData(j);
                    elementSet.add(currentElement,
currentElement.getKey(field.Y_DIMENSION));
                }
            }
        }
        return elementSet;
    }
    public TreeSet getAgentStartElements() {
        TreeSet elementSet = new TreeSet();
        for (int i = 0; i < aAgentArray.length; i++) {
            AAgent currentAgent = aAgentArray[i];
            elementSet.add(currentAgent.START, currentAgent.getAgentID());
        }
        return elementSet;
    }
    public TreeSet getAgentGoalElements() {
        TreeSet elementSet = new TreeSet();
        for (int i = 0; i < aAgentArray.length; i++) {
            AAgent currentAgent = aAgentArray[i];
            elementSet.add(currentAgent.GOAL, currentAgent.getAgentID());
        }
        return elementSet;
    }
    @Override
    public String toString() {
        return ("Pathfinder at step " + step + ": Number of AAgents: " +
aAgentArray.length);
    }
}
```

5.1.14. AAgent.java:

```
package apackage;
import adtpackage.*;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class AAgent implements Printable {
    private AStar aStarAlgorithm = new AStar();
    private final int AAGENT_ID;
    private List agentPath;
    public final Element START;
    public final Element GOAL;
    public AAgent(Field field, Element start, Element goal, int id) {
        AAGENT_ID = id;
        aStarAlgorithm.resetAExtensions(field);
        agentPath = aStarAlgorithm.aStar(start, goal, field);
        START = start;
        GOAL = goal;
    }
    public List getAgentPath() {
        return agentPath;
    }
    public int getAgentID() {
        return AAGENT_ID;
    }
    @Override public String print() {
        return toString();
    }
    @Override public String toString() {
        String string = "";
        string += "AAgent ID: ";
        string += AAGENT_ID;
        string += "; Start: ";
        string += START.print();
```

```
        string += "; Goal: ";
        string += GOAL.print();
        string += "; Path: [";
        string += agentPath.print();
        string += "]";
        return string;
    }
}
```

5.1.15. AStar.java:

```
package apackage;
import adtpackage.*;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class AStar {
    public AStar() {
    }
    public void resetAExtensions(Field field) { // used in aagent
        for (int i = 0; i < field.X_DIMENSION; i++) {
            for (int j = 0; j < field.Y_DIMENSION; j++) {
                field.getElement(i, j).setAExtension(new AExtension());
            }
        }
    }
    public List aStar(Element start, Element goal, Field field) {
        HashSet closedSet = new HashSet((field.X_DIMENSION - 2) * (field.Y_DIMENSION -
2));  // The set of nodes already evaluated.
                                        // ORDERED BY ELEMENT KEYS
        MinimalQueue openSet = new MinimalQueue();    // The set of tentative nodes to
be evaluated, initially containing the start node
                                        // ORDERED BY F_SCORES
        double gScore = 0;                                  // Cost from start along
best known path.
        double hScore = field.distanceBetween(start, goal); // distanceBetween ==
heuristic_cost_estimate
        double fScore = gScore + hScore;                    // Estimated total cost
from start to goal through y.
        start.getAExtension().setScores(gScore, hScore, fScore);
        openSet.enqueue(start, start.getAExtension().getFScore());
        while (!openSet.isEmpty()) {
            Element current = (Element) openSet.dequeue();
            List neighborList = field.getNeighborElements(current);
            if (current.isEqual(goal)) {
                return reconstructPath(start, goal);
            }
            while (!neighborList.isEmpty()) {
                Element currentNeighbor = (Element) neighborList.removeFirst();
                double tentativeGScore = -999;
                if ((currentNeighbor.isBoundary()) || (currentNeighbor.isObstacle()))
{
                    continue;
                } else if (!field.isLegalMove(current, currentNeighbor)) {
                    continue;
                } else if
(closedSet.contains(currentNeighbor.getKey(field.Y_DIMENSION))) {
                    continue;
                } else {
                    tentativeGScore = current.getAExtension().getGScore() +
field.distanceBetween(current, currentNeighbor);
                }
                if (!openSet.contains(currentNeighbor.getAExtension().getFScore())) {
// if current neighbor is not in open set
                    currentNeighbor.getAExtension().setCameFrom(current);
                    gScore = tentativeGScore;
                    hScore = field.distanceBetween(currentNeighbor, goal);
                    fScore = gScore + hScore;

                    currentNeighbor.getAExtension().setScores(gScore, hScore, fScore);
                    openSet.enqueue(currentNeighbor,
currentNeighbor.getAExtension().getFScore());
```

```
                }
                else if (tentativeGScore <
currentNeighbor.getAExtension().getGScore()) { // else if it is in openSet but new
GScore is better
                    openSet.remove(currentNeighbor.getAExtension().getFScore());

                    currentNeighbor.getAExtension().setCameFrom(current);
                    gScore = tentativeGScore;
                    hScore = currentNeighbor.getAExtension().getHScore(); // hScore is
already set for this one
                    fScore = gScore + hScore;
                    currentNeighbor.getAExtension().setScores(gScore, hScore, fScore);
                    // it already is in openset
                    openSet.enqueue(currentNeighbor,
currentNeighbor.getAExtension().getFScore());
                } else {
                } // else nothing happens
            }
            closedSet.add(current, current.getKey(field.Y_DIMENSION));
        }
        return null;
    }
    private List reconstructPath(Element startNode, Element currentNode) {
        List path;
        if (currentNode != startNode) { //came_from[current_node] is set {
            List tempPath = reconstructPath(startNode,
currentNode.getAExtension().getCameFrom());
            path = tempPath;
            path.insertAtRear(currentNode);
        }
        else {
            path = new List();
            path.insertAtRear(currentNode);
        }
        return path;
    }
}
```

5.1.16. BinarySearchTree.java:

```
package adtpackage;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class BinarySearchTree implements ADT {
    private class Node {
        private Object data;
        private double key;
        private Node left = null; // smaller
        private Node right = null; // bigger
        private Node(Object data, double key) {
            this.key = key;
            this.data = data;
        }
        @Override public String toString() {
            return printHelper(this, "");
        }
        private String printHelper(Node node, String outputString) {
            if ((node.left == null) && (node.right == null)) {
                outputString = outputString + node.key;
            } else {
                outputString = outputString + "(";
                if (node.left != null) {
                    outputString = printHelper(node.left, outputString);
                } else {
                    outputString = outputString + "x";
                }
                outputString = outputString + "-";
                outputString = outputString + node.key;
                outputString = outputString + "-";
                if (node.right != null) {
                    outputString = printHelper(node.right, outputString);
```

```
                } else {
                    outputString = outputString + "x";
                }
                outputString = outputString + ")";
            }
            return outputString;
        }
    }
    private class Del {
        private Node kappa; // storing the new organization of nodes
        private Node sigma; // storing the deleted node
        public Del (Node kappa, Node sigma) {
            this.kappa = kappa;
            this.sigma = sigma;
        }
    }
    private Node root = null;
    private int size = 0;
    private Object [] traverseInOrderArray;
    private int positionCounter = 0;
    private boolean arrayChanged = true;
    public BinarySearchTree() {
    }
    public <T extends Printable> ReturnCode insert(T data, double key) {
        if (isEmpty()) root = new Node(data, key);
        else root = insertHelper(data, key, root);
        size += 1;
        arrayChanged = true;
        return ReturnCode.SUCCESS; // showing that the insertion went without problems
    }
    private Node insertHelper(Object data, double key, Node currentNode) {
        if (key < currentNode.key) {
            if (currentNode.left == null) {
                currentNode.left = new Node(data, key);
                return currentNode;
            } else {
                currentNode.left = insertHelper(data, key, currentNode.left);
                return currentNode;
            }
        } else { // if key is the same or bigger than currentNode.key
            if (currentNode.right == null) {
                currentNode.right = new Node(data, key);
                return currentNode;
            } else {
                currentNode.right = insertHelper(data, key, currentNode.right);
                return currentNode;
            }
        }
    }
    public Object search(double key) { // gives data of searched node
        Node searchNode = searchHelper(key, root);
        if (searchNode != null) return searchNode.data;
        else return null;
    }
    private Node searchHelper(double key, Node currentNode) { // returns null if node
doesn't exist
        if (currentNode == null) {
            return null;
        } else {
            if (key == currentNode.key) {
                return currentNode;
            } else if (key < currentNode.key) {
                return searchHelper(key, currentNode.left);
            } else {
                return searchHelper(key, currentNode.right);
            }
        }
    }
    public Object delete(double key) { // gives data of deleted nodes
        Del del = deleteHelper(key, root);
        Node deleteNode = del.sigma;
        root = del.kappa;
        size -= 1;
        arrayChanged = true;
        if (deleteNode != null) return deleteNode.data;
        else return null;
```

```
    }
    private Del deleteHelper(double key, Node currentNode) { // returns null if node
being deleted doesn't exist
        if (currentNode == null) {
            Del delSent = new Del(null, null);
            return delSent;
        } else {
            if (key == currentNode.key) {
                Del delReceived = deleteMover(currentNode); // removing & fixing
algorithm;
                return delReceived;
            } else if (key < currentNode.key) {
                Del delReceived = deleteHelper(key, currentNode.left);
                currentNode.left = delReceived.kappa;
                Del delSent = new Del(currentNode, delReceived.sigma);
                return delSent;
            } else {
                Del delReceived = deleteHelper(key, currentNode.right);
                currentNode.right = delReceived.kappa;
                Del delSent = new Del(currentNode, delReceived.sigma);
                return delSent;
            }
        }
    }
    public Object deleteMin() { // returns data of the smallest node
        Del del = deleteMinHelper(root);
        Node deleteMinNode = del.sigma;
        root = del.kappa;
        size -= 1;
        arrayChanged = true;
        if (deleteMinNode != null) return deleteMinNode.data;
        else return null;
    }
    private Del deleteMinHelper(Node currentNode) { // returns null if the tree is
empty
        if (currentNode == null) {
            Del delSent = new Del(null, null);
            return delSent;
        } else {
            if (currentNode.left != null) {
                Del delReceived = deleteMinHelper(currentNode.left);
                currentNode.left = delReceived.kappa;
                Del delSent = new Del(currentNode, delReceived.sigma);
                return delSent;
            } else if (currentNode.right != null) { // if curentNode.left is null but
currentNode.right is not
                Del delReceived = deleteMover(currentNode.right); // need to fix
currentNode.right
                currentNode.right = delReceived.kappa;
                Del delSent = new Del(currentNode, delReceived.sigma);
                return delSent;
            } else { // if both left and right are null
                Node send = currentNode;
                currentNode = null;
                Del delSent = new Del(currentNode, send);
                return delSent;
            }
        }
    }
    private Del deleteMover(Node node) {
        if (node.right != null) {
            if (node.right.left != null) { // going to exchange node with
node.right.left
                Node tempLeft = node.left;
                Node tempRight = node.right;
                Node send = node;
                Del delReceived = deleteMover(node.right.left); // fixing
node.right.left
                node.right.left = delReceived.kappa;
                node = delReceived.sigma;
                node.left = tempLeft;
                node.right = tempRight;
                Del delSent = new Del(node, send);
                return delSent; // sending the deleted node
            } else { // node.right is not null but it has nothing to the left
                Node tempLeft = node.left;
```

```
                Node send = node;
                node = node.right;
                node.left = tempLeft;
                Del delSent = new Del(node, send);
                return delSent; // sending the deleted node
            }
        } else if (node.left != null) { // node.right is null but node.left is not
            Node send = node;
            node = node.left;
            Del delSent = new Del(node, send);
            return delSent; // sending the deleted node
        } else { // both nodde.right and node.left are null
            Node send = node;
            node = null;
            Del delSent = new Del(node, send);
            return delSent; // sending the deleted node
        }
    }
    public boolean isEmpty() {
        return root == null;
    }
    public int size() {
        return size;
    }
    public Object getNodeData(int node) {
        if (!arrayChanged) {
            return traverseInOrderArray[node];
        } else {
            traverseInOrder();
            return traverseInOrderArray[node];
        }
    }
    public Object [] traverseInOrder() {
        traverseInOrderArray = new Object [size];
        positionCounter = 0;
        traverseInOrderHelper(root);
        arrayChanged = false;
        return traverseInOrderArray;
    }
    private void traverseInOrderHelper(Node node) {
        if (node.left != null) traverseInOrderHelper(node.left);
        traverseInOrderArray[positionCounter] = node.data;
        positionCounter += 1;
        if (node.right != null) traverseInOrderHelper(node.right);
    }
}
```

5.1.17. TreeSet.java:

```
package adtpackage;
import generalpackage.Printable;
/**
 *
 * @author Zbyněk Stara
 */
public class TreeSet extends BinarySearchTree implements Printable, Set {
    public TreeSet() {
        super();
    }
    public <T extends Printable> ReturnCode add(T data, double key) {
        ReturnCode insertionResult;
        if (!contains(key)) {
            insertionResult = insert(data, key);
        } else {
            insertionResult = ReturnCode.SKIP;
        }
        return insertionResult;
    }
    public Object remove(double key) { // will return null if it doesn't exist
        return delete(key);
    }
    public Object removeMin() {
        return deleteMin();
    }
```

```
    public boolean contains(double key) {
        return search(key) != null;
    }
    @Override public String print() {
        return toString();
    }
    @Override public String toString() { // THIS MIGHT NOT WORK BECAUSE KEYS ARE NOT
GOING IN ORDER - FIXED?
        if (!isEmpty()) {
            String printedNodes = "";
            for (int i = 0; i < (size() - 1); i++) {
                printedNodes = (printedNodes + "{" + printNode(i) + "}, ");
            }
            printedNodes = (printedNodes + "{" + printNode(size() - 1) + "}");
            return ("Set size: " + size() + "; Contents: " + printedNodes);
        } else {
            return "Set is empty";
        }
    }
    private <T extends Printable> String printNode(int node) {
        return ((T) getNodeData(node)).print();
    }
}
```

5.1.18. HashTable.java:

```
package adtpackage;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class HashTable implements ADT, Printable {
    protected class TableElement <T extends Printable> {
        protected Object data = null ;
        protected double key = -999;
        protected TableElement(T data, double key) {
            this.data = data;
            this.key = key;
        }
    }
    protected TableElement [] dataArray;
    protected int size = 0;
    private Object [] traverseArray;
    private int positionCounter = 0;
    protected boolean arrayChanged = true;
    public HashTable(int dimension) {
        dataArray = new TableElement[dimension];
        for (int i = 0; i < dataArray.length; i++) {
            dataArray[i] = null;
        }
    }
    public boolean isEmpty() {
        return (size == 0);
    }
    public int size() {
        return size;
    }
    public <T extends Printable> ReturnCode insert(T data, double key) {
        int checkCounter = 0;
        boolean positionFound = false;
        int hashValue = hashCalculation(key);
        while (checkCounter < dataArray.length) {
            if (dataArray[hashValue] == null) {
                dataArray[hashValue] = new TableElement(data, key);
                positionFound = true;
                size += 1;
                arrayChanged = true;
                break;
            } else {
                checkCounter += 1;
                hashValue += 1;
                if (hashValue == dataArray.length) hashValue = 0;
            }
```

```
    }
    if (positionFound) return ReturnCode.SUCCESS;
    else return ReturnCode.OVERFLOW;
}
public Object search(double key) {
    int checkCounter = 0;
    boolean tableElementFound = false;
    int hashValue = hashCalculation(key);
    while (checkCounter < dataArray.length) {
        if (dataArray[hashValue] != null) {
            if (dataArray[hashValue].key == key) {
                tableElementFound = true;
                break;
            } else {
                checkCounter += 1;
                hashValue += 1;
                if (hashValue == dataArray.length) hashValue = 0;
            }
        } else {
            checkCounter += 1;
            hashValue += 1;
            if (hashValue == dataArray.length) hashValue = 0;
        }
    }
    if (tableElementFound) return dataArray[hashValue].data;
    else return null;
}
public Object delete(double key) {
    int checkCounter = 0;
    boolean tableElementFound = false;
    Object dataReturned = null;
    int hashValue = hashCalculation(key);
    while (checkCounter < dataArray.length) {
        if (dataArray[hashValue].key == key) {
            dataReturned = dataArray[hashValue].data;
            size -= 1;
            arrayChanged = true;
            dataArray[hashValue] = null;
            tableElementFound = true;
            break;
        } else {
            checkCounter += 1;
            hashValue += 1;
            if (hashValue == dataArray.length) hashValue = 0;
        }
    }
    if (tableElementFound) return dataReturned;
    else return null;
}
protected int hashCalculation(double key) {
    return ((int) key % dataArray.length);
}
public Object getNodeData(int node) {
    if (!arrayChanged) {
        return traverseArray[node];
    } else {
        traverse();
        return traverseArray[node];
    }
}
public Object [] traverse() {
    traverseArray = new Object[size];
    positionCounter = 0;
    for (int i = 0; i < dataArray.length; i++) {
        if (dataArray[i] != null) {
            traverseArray[positionCounter] = dataArray[i].data;
            positionCounter += 1;
        }
    }
    arrayChanged = false;
    return traverseArray;
}
public String print() {
    return toString();
}
@Override public String toString() {
```

```
        String string = "";
        if (!isEmpty()) {
            string = "Size: " + size +"; Contents: [";
            int i = 0;
            for (; i < (size - 1); i++) {
                if (dataArray[i] != null) {
                    string += "{";
                    string += dataArray[i].data.toString();
                    string += "}, ";
                }

            }
            while (dataArray[i] == null) {
                i++;
            }
            string += "{";
            string += dataArray[i].data.toString();
            string += "}";
            string += "]";
        } else {
            string = "Hash table is empty";
        }
        return string;
    }
}
```

5.1.19. HashSet.java:

```
package adtpackage;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class HashSet extends HashTable implements Set {
    public HashSet(int dimension) {
        super(dimension);
    }
    public <T extends Printable> ReturnCode add(T data, double key) {
        int checkCounter = 0;
        boolean positionFound = false;
        ReturnCode additionResult = ReturnCode.UNDEFINED;
        int hashValue = hashCalculation(key);
        while (checkCounter < dataArray.length) {
            if (dataArray[hashValue] == null) {
                dataArray[hashValue] = new TableElement(data, key);
                positionFound = true;
                size += 1;
                arrayChanged = true;
                additionResult = ReturnCode.SUCCESS;
                break;
            } else if (dataArray[hashValue].key == key) {
                positionFound = true;
                additionResult = ReturnCode.SKIP;
                break;
            } else {
                checkCounter += 1;
                hashValue += 1;
                if (hashValue == dataArray.length) hashValue = 0;
            }
        }
        if (!positionFound) additionResult = ReturnCode.OVERFLOW;

        return additionResult;
    }
    public Object remove(double key) { // will return null if it doesn't exist
        return delete(key);
    }
    public boolean contains(double key) {
        return search(key) != null;
    }
}
```

5.1.20. Set.java:

```
package adtpackage;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public interface Set extends ADT {
    abstract <T extends Printable> ReturnCode add(T data, double key);
    abstract Object remove(double key);
    abstract boolean contains(double key);
}
```

5.1.21. MaximalQueue.java:

```
package adtpackage;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class MaximalQueue implements Printable {
    private class Node <T extends Printable> {
        private T data = null;
        private double key = -999;
        private Node next = null;
        private Node(T data, double key) {
            this.data = data;
            this.key = key;
        }
        private Node(T data, double key, Node next) {
            this.data = data;
            this.key = key;
            this.next = next;
        }
    }
    private Node head = null;
    private int size = 0;
    public MaximalQueue() {

    }
    public boolean isEmpty() {
        return (head == null);
    }
    public int size() {
        return size;
    }
    public boolean contains(double key) {
        if (isEmpty()) return false;
        else {
            Node current = head;
            while (current.key > key && current.next != null) {
                current = current.next;
            }
            if (current.key == key) {
                return true;
            } else return false;
        }
    }
    public <T extends Printable> void enqueue(T data, double key) { // adds to the
list according to node keys
        if (size() == 0) {
            head = new Node(data, key);
        } else if (size() == 1) {
            if (head.key >= key) {
                head.next = new Node(data, key);
            } else {
                Node temp = head;
                head = new Node(data, key);
                head.next = temp;
            }
        } else {
```

```
            if (head.key >= key) { // if adding the same key, the new one goes behind
the old one
                Node current = head.next;
                Node previous = head;
                while (current.key >= key) {
                    if (current.next == null) {
                        previous = current;
                        current = current.next;
                        break;
                    } else {
                        previous = current;
                        current = current.next;
                    }
                }
                Node temp = current;
                previous.next = new Node(data, key, temp);
            } else {
                Node temp = head;
                head = new Node(data, key, temp);
            }
        }
        size += 1;
    }
    public Object dequeue() { // removes the first element from the queue - the
largest one
        if (isEmpty()) {
            return null;
        } else {
            Node temp = head;
            head = head.next;
            size -= 1;
            return temp.data;
        }
    }
    public Object remove(double key) { // removes the first of given key
        if (isEmpty()) return null;
        else if (size == 1) {
            if (head.key == key) {
                Node temp = head;
                head = null;
                size -= 1;
                return temp.data;
            } else return null;
        } else {
            if (head.key != key) {
                Node current = head.next;
                Node previous = head;
                while (current.key != key) {
                    if (current.next == null) {
                        return null;
                    } else {
                        previous = current;
                        current = current.next;
                    }
                }
                Node temp = current;
                previous.next = current.next;
                return temp.data;
            } else {
                Node temp = head;
                head = head.next;
                size -= 1;
                return temp.data;
            }
        }
    }
    public Object getNodeData(int node) {
        if ((!isEmpty()) && (node < size)) {
            Node current = head;
            for (int i = 0; i < node; i++) {
                current = current.next;
            }
            return current.data;
        } else {
            return null;
        }
```

```
    }
    @Override public String print() {
        return toString();
    }
    @Override public String toString() {
        if (!isEmpty()) {
            String printedNodes = "";
            for (int i = 0; i < (size - 1); i++) {
                printedNodes = (printedNodes + "{" + printNode(i) + "}, ");
            }
            printedNodes = (printedNodes + "{" + printNode(size - 1) + "}");
            return ("List size: " + size + "; List contents: [" + printedNodes + "]");
        } else {
            return ("List is empty");
        }
    }
    private <T extends Printable> String printNode(int node) {
        return ((T) getNodeData(node)).print();
    }
}
```

5.1.22. MinimalQueue.java:

```
package adtpackage;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public class MinimalQueue implements Printable {
    private class Node <T extends Printable> {
        private T data = null;
        private double key = -999;
        private Node next = null;
        private Node(T data, double key) {
            this.data = data;
            this.key = key;
        }
        private Node(T data, double key, Node next) {
            this.data = data;
            this.key = key;
            this.next = next;
        }
    }
    private Node head = null;
    private int size = 0;
    public MinimalQueue() {

    }
    public boolean isEmpty() {
        return (head == null);
    }
    public int size() {
        return size;
    }
    public boolean contains(double key) {
        if (isEmpty()) return false;
        else {
            Node current = head;
            while (current.key < key && current.next != null) {
                current = current.next;
            }
            if (current.key == key) {
                return true;
            } else return false;
        }
    }
    public <T extends Printable> void enqueue(T data, double key) { // adds to the
list according to node keys
        if (size() == 0) {
            head = new Node(data, key);
        } else if (size() == 1) {
            if (head.key <= key) {
                head.next = new Node(data, key);
```

```
            } else {
                Node temp = head;
                head = new Node(data, key);
                head.next = temp;
            }
        } else {
            if (head.key <= key) { // if adding the same key, the new one goes behind
the old one
                Node current = head.next;
                Node previous = head;
                while (current.key <= key) {
                    if (current.next == null) {
                        previous = current;
                        current = current.next;
                        break;
                    } else {
                        previous = current;
                        current = current.next;
                    }
                }
                Node temp = current;
                previous.next = new Node(data, key, temp);
            } else {
                Node temp = head;
                head = new Node(data, key, temp);
            }
        }
        size += 1;
    }
    public Object dequeue() { // removes the first element from the queue - the
smallest one
        if (isEmpty()) {
            return null;
        } else {
            Node temp = head;
            head = head.next;
            size -= 1;
            return temp.data;
        }
    }
    public Object remove(double key) { // removes the first of given key
        if (isEmpty()) return null;
        else if (size == 1) {
            if (head.key == key) {
                Node temp = head;
                head = null;
                size -= 1;
                return temp.data;
            } else return null;
        } else {
            if (head.key != key) {
                Node current = head.next;
                Node previous = head;
                while (current.key != key) {
                    if (current.next == null) {
                        return null;
                    } else {
                        previous = current;
                        current = current.next;
                    }
                }
                Node temp = current;
                previous.next = current.next;
                return temp.data;
            } else {
                Node temp = head;
                head = head.next;
                size -= 1;
                return temp.data;
            }
        }
    }
    public Object getNodeData(int node) {
        if ((!isEmpty()) && (node < size)) {
            Node current = head;
            for (int i = 0; i < node; i++) {
```

```
                current = current.next;
            }
            return current.data;
        } else {
            return null;
        }
    }
    @Override public String print() {
        return toString();
    }
    @Override public String toString() {
        if (!isEmpty()) {
            String printedNodes = "";
            for (int i = 0; i < (size - 1); i++) {
                printedNodes = (printedNodes + "{" + printNode(i) + "}, ");
            }
            printedNodes = (printedNodes + "{" + printNode(size - 1) + "}");
            return ("List size: " + size + "; List contents: [" + printedNodes + "]");
        } else {
            return ("List is empty");
        }
    }
    private <T extends Printable> String printNode(int node) {
        return ((T) getNodeData(node)).print();
    }
}
```

5.1.23. List.java:

```
package adtpackage;
import generalpackage.Printable;
/**
 *
 * @author Zbyněk Stara
 */
public class List implements Printable {
    private class Node <T extends Printable> {
        private T data = null;
        private Node next = null;
        private Node(T data) {
            this.data = data;
        }
    }
    private Node head = null;
    private int size = 0;
    public List() {
        head = null;
    }
    public boolean isEmpty() {
        return (head == null);
    }
    public int size() { // size is 1 when there is 1 element – which the 0th element
        return size;
    }
    public <T extends Printable> void insertAtFront(T data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
        size += 1;
    }
    public <T extends Printable> void insertAtRear(T data) {
        if (isEmpty()) {
            head = new Node(data);
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = new Node(data);
        }
        size += 1;
    }
    public <T extends Printable> void insertAsNode(T data, int node) {
        if (node == 0) {
```

```
        insertAtFront(data);
    } else if((!isEmpty()) && (node < size)) {
        Node current = head;
        for (int i = 1; i < node; i++) {
            current = current.next;
        }
        Node temp = current.next;
        current.next = new Node(data);
        current.next.next = temp;
        size += 1;
    } else {
        insertAtRear(data);
    }
}
public Object removeFirst() {
    if (isEmpty()) {
        return null;
    }
    Node first = head;
    head = head.next;
    size -= 1;
    return first.data;
}
public Object removeLast() {
    if (isEmpty()) {
        return null;
    }
    Node current = head;
    if (current.next == null) {
        head = null;
        return current.data;
    }
    Node previous = null;
    while (current.next != null) {
        previous = current;
        current = current.next;
    }
    previous.next = null;
    size -= 1;
    return current.data;
}
public Object removeNode(int node) {
    if ((!isEmpty()) && (node < size)) {
        Node current = head;
        if (node == 0) {
            return removeFirst();
        }
        current = current.next;
        Node previous = head;
        for (int i = 1; i < node; i++) {
            current = current.next;
            previous = previous.next;
        }
        Node temp = current;
        previous.next = current.next;
        return temp.data;
    } else {
        return null;
    }
}
public Object getNodeData(int node) {
    if ((!isEmpty()) && (node < size)) {
        Node current = head;
        for (int i = 0; i < node; i++) {
            current = current.next;
        }
        return current.data;
    } else {
        return null;
    }
}
@Override public String print() {
    return toString();
}
@Override public String toString() {
    if (!isEmpty()) {
```

```
            String printedNodes = "";
            for (int i = 0; i < (size - 1); i++) {
                printedNodes = (printedNodes + "{" + printNode(i) + "}, ");
            }
            printedNodes = (printedNodes + "{" + printNode(size - 1) + "}");
            return ("List size: " + size + "; List contents: [" + printedNodes + "]");
        } else {
            return ("List is empty");
        }
    }
    private <T extends Printable> String printNode(int node) {
        return ((T) getNodeData(node)).print();
    }
}
```

5.1.24. ADT.java:

```
package adtpackage;
import generalpackage.*;
/**
 *
 * @author Zbyněk Stara
 */
public interface ADT {
    abstract boolean isEmpty();
    abstract int size();
    abstract <T extends Printable> ReturnCode insert(T data, double key);
    abstract Object search(double key);
    abstract Object delete(double key);
}
```

5.1.25. ReturnCode.java:

```
package adtpackage;
/**
 *
 * @author Zbyněk Stara
 */
public enum ReturnCode {
    SUCCESS("The addition was successful."),
    FAILURE("The addition was unsuccessful."),
        SKIP("The addition was skipped."),
        OVERFLOW("The container overflowed during addition process."),
    UNDEFINED("The addition status is undefined.");
    private String description;
    private ReturnCode(String description) {
        this.description = description;
    }
    public String getDescription() {
        return description;
    }
}
```

5.1.26. Printable.java:

```
package generalpackage;

/**
 *
 * @author Zbyněk Stara
 */
public interface Printable {
    abstract String print();
}
```