Master Thesis

Obstacles in pedestrian simulations

Author(s):
Stucki, Pascal

Publication Date:
2003

Permanent Link:
https://doi.org/10.3929/ethz-a-004696801

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Obstacles in Pedestrian Simulations

Pascal Stucki, September 12, 2003

Christian Gloor, Dept. of Computer Science, ETH Zurich
Prof. Kai Nagel, Dept. of Computer Science, ETH Zurich
Abstract

This report presents two new models to simulate the behavior of pedestrians. The first model uses a potential field model approach invented by Hoogendoorn and Bovy, but calculates the field using a wave algorithm instead of partial differential equations. The second model uses significant points in the environment to generate first a visibility graph and then a minimal spanning tree using the algorithm from Dijkstra. The walking direction and speed of the pedestrians depends on three forces similar to the social force model by Helbing. The first force leads the agent toward its destination by using either the potential field or the calculated spanning tree. A second force regulates the interactions between agents, so ensuring that agents do not walk into other pedestrians and try to keep a certain distance from each other. The third force guarantees that agents do not walk into walls and that they try to keep a certain distance from walls. The behavior of pedestrians simulated with the two models is quite similar, the main difference lies in the models’ performance. The spanning tree approach is slower than the potential field model when simple simulations are used, as it spends most time on the creation of the visibility graph. The advantages of the spanning tree model are found when more sophisticated simulations with different destinations are run, as all spanning trees can be based on the same visibility graph, where else a new potential field needs to be calculated for each destination.

The correctness of the models is proved by comparing the average pedestrian speed in a corridor with increased pedestrian density and fraction to empirical results collected by Weidmann. The models are kept rather simple to ensure fast and efficient simulations. Some large simulations of Zurich main station (700 x 200m) are used to show the usability of both models for simulations of large pedestrian infrastructures. It further presents some new aspects for pedestrian simulations: a walkability graph adapting the agents’ speed on special terrain (stairs, escalators), simulations in buildings with several floors, multiple destinations (allowing to use activities like “walk to the next exit”) and rated destinations (allowing activities like ”buy a new ticket, accept 10m of extra walking if it can be bought at a ticket desk rather than an automata”).
Contents

1 Introduction 4
  1.1 Pedestrian Simulation Models ..................................................... 4
  1.2 Two new Approaches ................................................................. 4

2 Design 6
  2.1 Geometrical Objects ................................................................. 6
  2.2 Algorithms ............................................................................. 8

3 Potential Field Model 13
  3.1 Hoogendoorn’s Concept of Utility Maximization ............................. 13
  3.2 Wave Algorithm ................................................................. 13

4 Graph Model 18
  4.1 Node Generation ................................................................. 18
  4.2 Graph Generation ................................................................. 18
  4.3 Route Generation ................................................................. 18

5 Agent Movement 20
  5.1 Potential Force ................................................................. 21
  5.2 Graph Force ................................................................. 25
  5.3 Pedestrian Pressure ................................................................. 25
  5.4 Object Pressure ................................................................. 26
  5.5 Find next Position ................................................................. 27
  5.6 Walkability ................................................................. 29

6 Testing 30
  6.1 Simulation Properties ................................................................. 30
  6.2 Adapting the Forces ................................................................. 31
  6.3 Standard Tests ................................................................. 32
  6.4 Performance ................................................................. 34
  6.5 Simulating Zurich Main Station .................................................. 39
CONTENTS

7 Summary 48

A Manual 50
A.1 Configuration .......................................................... 50
A.2 Start the Simulation .................................................. 54
A.3 Analyzing the Results ................................................. 54

B Some useful Scripts 56
B.1 Creating a Simulation Video ........................................ 56
B.2 Agent Generator ...................................................... 57
B.3 Update Coordinates in XML File .................................. 57

C Test Results 59

D Available Source Code and Documentation 61
D.1 Source Code .......................................................... 61
D.2 Documentation ....................................................... 64
1 Introduction

1.1 Pedestrian Simulation Models

The behavior of pedestrians has been empirically studied for more than four decades now. Understanding the acting of pedestrians is important for the planning, design and analysis of public pedestrian facilities (train stations, airports, shopping malls, stadiums). A number of simulation models have been proposed to predict the behavior of pedestrians in normal and panic situations.

- Queuing models(1) are based on events like entering or leaving a link or room. Waiting times are introduced due to queues building up when the traffic demand is larger than the environment’s capacity (crossing, door). Queuing models are only suitable to describe pedestrian evacuation behavior from buildings or ships, since they do not take into account the concrete geometry of buildings and obstacles.

- Cellular automata models(2) divide the area into quadratic cells, which can be in a few states (typically occupied or free). At each time step, the cell’s state is updated based on its previous state and the state of its neighbor cells. These restrictions ensure efficient simulations. On the other side, choosing the direction of the next step is limited to free neighbor cells. Therefore, cellular automata models are more adequate for car simulations, where traffic flow is mostly uni-directional.

- Continuous time models(3) (4) are based on differential equations of the form

\[
\frac{\delta x}{\delta t} = f(x(t), \beta),
\]

where the vector of state variables \( u(t) \) is a continuous function of time and \( \beta \) represents all model parameters.

- Discrete time models are similar to continuous time models, but the time is divided into small steps which results in functions of the form

\[
x_{t+1} = f(x_t, \beta).
\]

The main advantage of these discrete choice models (both continuous and discrete) is the possibility to model any direction of movement.

(5) offers a more detailed overview of existing pedestrian simulation models).

1.2 Two new Approaches

The chosen models for the simulator are influenced by already existing tools for pedestrian simulations. An early realization of a pedestrian simulation at the ETH was developed by Laurent Mauron as part of his diploma thesis(6), where he tested the behavior of simulated pedestrians in two models. The first model divided the simulated environment in square cells of different types (link, crossing, border, obstacle). The forces and walking directions for each cell were computed at the beginning of the simulation. The second model simulated the environment with a set of points and links, which the agents used to find their way toward their destination. A force simulating the interactions between agents was introduced as well (the so called pedestrian pressure). Objects were simply simulated by non-moving agents. The second model was then integrated into AlpSim(7), a project to simulate the behavior of hikers in the Alps. As an enhancement of AlpSim, but also as a basic tool for simulations in buildings, two new models were developed to integrate obstacles in pedestrian simulations. The models should at least fulfill the following requirements (concerning obstacles and walls):
Pedestrians should not walk through obstacles or walls, even if pressed by other agents.
- if there exists a way to a chosen destination, a pedestrian should be able to find the shortest path around blocking obstacles and walls toward it.
- Pedestrians should not be influenced by objects or other pedestrians hidden behind a wall (visibility check)
- Pedestrians try to keep a certain distance from obstacles.

Two approaches are used to simulate the behavior of pedestrians concerning obstacles and walls. The first model uses an idea from Hoogendoorn and Bovy(4), which divide the simulated area into cells of fixed size and calculate the cost to reach the destination from this cell. Agents will then choose their walking direction in a way to minimize the needed cost. The new model is using discrete time steps and cells instead of differential equations. The second model tries to find significant nodes in the simulated area and calculates a minimal spanning tree along which the agents will walk toward their destination. The design of the simulator is shown in chapter 2, the two models are further described in chapter 3 (Potential Field Model) and 4 (Graph Model). Chapter 5 describes the forces used by pedestrians to decide about their walking path and speed.

An often neglected point is the calibration of the models. A simple test (two crossing pedestrians, see section 6.2.1) will be used to calibrate the two models, hereby using empirical data from Weidmann(8) and collected video footage by Mauron(6). The calibration is needed to adapt the forces between pedestrians to the destination forces calculated by the two models. The standard corridor tests with increased pedestrian density and different fractions (percentage of people walking in one direction) will show whether the simulations produce realistic results (see section 6.3). A simulation of Zurich Main Station (an area of 700m x 200m) in chapter 6.5 will clarify whether the models' performance are efficient enough for large simulations.
2 Design

The design of the two models is identical as far as possible. The usual C++ main program is used to start the simulation (see section A) for a simulation manual. It first uses the XMLConfigParser to read in the properties in the configuration file (an example file is shown in section A.1.1). The file tells the simulator the filenames of the environment files with specified buildings, obstacles and destinations to be parsed (an example file is shown in section A.1.2). Each parsed object gets its representing instance from the Geometry class (see section 2.1) and is stored in the simulator. If the simulated area consists of more than one floor, several network files have to be read in and the positions of the objects have to be adapted so that different floors can be simulated as if they were lying next to each other. This includes the introduction of beam points in the middle of stairs or escalators, which will link the different floors with each other (see chapter 6.5). The simulated agents are read in from a third XML file and stored in the simulator. Their walking routes are stored as a series of way points or destination types (an example file is shown in section A.1.3). When all the needed data has been parsed in, the viewer showing the walking agents gets initialized. It will update its view by iterating over all agents and updating their position. A loop over all rounds is equal to 0.5 seconds in real time. This value is chosen as it matches the average time a pedestrian needs to do one step. The called agents will then calculate their next step depending on their actual position, destination and the nearness of other agents and obstacles (see section 5). The walking direction is chosen using either the potential field or spanning tree approach discussed in the following chapters 3 and 4. If the needed potential field or graph does not exist yet, it has to be calculated first. Calculated fields and spanning trees are stored in XML or text files for reuse in further simulations. The actual position of all agents is shown in a graphical viewer (using GDK). When all agents have reached their destination specified in the plan file the simulation gets closed. During the simulation, occurring events (actual walking positions) are logged in an XML file (see section A.3) and screen shots of the viewer are stored. The screen shots can later be used to create a video of the executed simulation (see appendix B.1).

Figure 1: Used Classes for Simulation

2.1 Geometrical Objects

The simulation uses several geometrical objects like points and lines to store the environment from the XML environment file (see figure 2 for an overview over all classes). All these classes are implemented in the file “Geometry.h” and will be further discussed in the following sections.
A **Point** instance is used to describe a position in the simulated area. It is described with Cartesian coordinates \((x, y)\).

The class **WeightedPoint** is inherited from class **Point**. In addition to the coordinates it has a defined weight as well, which is used in the potential field calculation algorithm to store cells and their calculated potential (the weight) in a queue.

The class **Node** is inherited from class **Point**. In addition to the coordinates every node has a unique id. It is used to store the way points from the XML environment file.

The class **GraphNode** is inherited from class **Node**. In addition to the coordinates and the id, a graph node has a weight and a pointer to its predecessor node. This is useful when calculating the minimal spanning tree for the graph model.

A **Line** is defined by two points with Cartesian coordinates. It is used to store walls defined in the XML environment file.

The class **Link** is inherited from class Line. In addition to the two points it has a defined weight, which is used to store the Links length when calculating the minimal spanning tree for the graph model.

To simplify the creation of XML environment files, it is possible to include polygons and rectangles. These objects get split up into lines while parsing. So there is no need to store polygons and rectangles anywhere in the simulation.
2.2 Algorithms

The simulation uses several algorithms to calculate distances or crossing points between different objects.

2.2.1 Distance between two Points

The calculation of the distance between two points \( P_1 \) and \( P_2 \) defined by their Cartesian coordinates \((x_1, y_1)\) and \((x_2, y_2)\) is done using the Pythagoras Theorem which claims that the sum of the squares of the two small sides in a right triangle equals the square of its hypotenuse. The calculation is so simple that it is faster to inline it in the code wherever it is used rather than making an extra function call whenever it is needed (see formula 3).

\[
\text{distance}_{P_1,P_2} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (3)
\]

2.2.2 Distance between Point and Line

The equation of a line \( L \) defined through two points \( P_1 \) \((x_1, y_1)\) and \( P_2 \) \((x_2, y_2)\) (the endpoints of the object) is

\[
P = P_1 + \lambda \cdot (P_2 - P_1). \quad (4)
\]

A point \( P_3 \) \((x_3, y_3)\) is closest to the line at the tangent to the line which passes through \( P_3 \), so the dot product of the tangent and the line must be zero.

\[
(P_3 - P) \cdot (P_2 - P_1) = 0 \quad (5)
\]

Inserting equation 4 into equation 5 gives

\[
(P_3 - P_1 - \lambda \cdot (P_2 - P_1)] \cdot (P_2 - P_1) = 0. \quad (6)
\]

Solving equation 6 returns the value of \( \lambda \).

\[
\lambda = \frac{(x_3 - x_1) \cdot (x_2 - x_1) + (y_3 - y_1) \cdot (y_2 - y_1)}{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (7)
\]

If \( \lambda \) is smaller than zero or bigger than one, the point of intersection is not part of the object. In this case we can just replace \( \lambda \) by zero (resp. one) and get the nearer endpoint as the point on the object that is next to \( P_3 \). Substituting \( \lambda \) in the line equation gives the point of intersection \((sx, sy)\).

\[
sx = x_1 + \lambda \cdot (x_2 - x_1) \quad (8)
\]

\[
sy = y_1 + \lambda \cdot (y_2 - y_1) \quad (9)
\]

The distance from the point to the line can now be calculated with

\[
\text{distance}_{L,P_3} = \sqrt{(x_3 - sx)^2 + (y_3 - sy)^2} \quad (10)
\]
2.2.3 Distance between Point and Rectangle or Polygon

To simplify the creation of XML environment files, it is possible to define objects as rectangles and polygons. These are split up in lines while parsing the environment file. It is therefore not necessary to calculate the distance between a point and a polygon.

2.2.4 Crossing Point of two Lines

It is normally rather easy to find the crossing point of two lines (solving the equation system of the two line equations). But there are several cases that have to be excluded before the computation to prevent divisions by zero and infinite numbers.

The first line is defined by the two points \( P_1 (x_1, y_1) \) and \( P_2 (x_2, y_2) \), the second by the points \( P_3 (x_3, y_3) \) and \( P_4 (x_4, y_4) \).

\[
L_1 = P_1 + \lambda_1 \cdot (P_2 - P_1) \quad \text{(11)}
\]
\[
L_2 = P_3 + \lambda_2 \cdot (P_4 - P_3) \quad \text{(12)}
\]

The steepness of each line is defined by

\[
m_1 = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{(13)}
\]
\[
m_2 = \frac{y_4 - y_3}{x_4 - x_3} \quad \text{(14)}
\]

A check is needed to ensure that \( x_1 \) and \( x_2 \) (or \( x_3 \) and \( x_4 \)) are not identical, as this would result in a division by zero. Now there are several special cases to distinguish:

1. Line 1 is horizontal, Line 2 is not
2. Line 1 is vertical, Line 2 is not
3. Line 2 is horizontal, Line 1 is not
4. Line 2 is vertical, Line 1 is not
5. Line 1 and Line 2 are parallel
6. Both Lines are neither horizontal nor vertical

**Line 1 is horizontal, Line 2 is not** If the first object is horizontal, the y coordinate of the crossing point is identical to the y coordinate of either \( P_1 \) or \( P_2 \) (they are identical as well). The x coordinate of the crossing point can then be calculated by including the now known y coordinate in one of the line equations.

\[
sy = y_1 \quad \text{(15)}
\]
\[
sx = x_1 + m_1 \cdot (sy - y_1) \quad \text{(16)}
\]
2.2 Algorithms

**Line 1 is vertical, Line 2 is not**  If the first object is vertical, the x coordinate of the crossing point is identical to the x coordinate of either $P_1$ or $P_2$ (they are identical as well). The y coordinate of the crossing point can then be calculated by including the now known x coordinate in one of the line equations.

\[
\begin{align*}
sx &= x_1 \\
sy &= y_1 + m_1 \times (sx - x_1)
\end{align*}
\]  
\begin{align}
(17)
\end{align}

**Line 2 is horizontal, Line 1 is not (see case 1)**

**Line 2 is vertical, Line 1 is not (see case 2)**

**Line 1 and 2 are parallel**  If the two lines are not lying on each other, there is no crossing point.

**Line 1 and Line 2 are neither horizontal nor vertical**  If all the special cases are excluded, the crossing point can be calculated normally by including the (yet unknown) crossing point in the two line equations and setting them equal to each other.

\[
\begin{align*}
x_1 + m_1 \times (sx - x_1) &= x_3 + m_2 \times (sx - x_3) \\
y_1 + m_1 \times (sy - y_1) &= y_3 + m_2 \times (sy - y_3)
\end{align*}
\]  
\begin{align}
(19)
\end{align}

These results in the following equations for the calculation of the crossing point:

\[
\begin{align*}
sx &= \frac{(m_1 - 1) \times x_1 - (m_2 - 1) \times x_3}{m_1 - m_2} \\
spy &= \frac{(m_1 - 1) \times y_1 - (m_2 - 1) \times y_3}{m_1 - m_2}
\end{align*}
\]  
\begin{align}
(21)
\end{align}

The source code of the crossing point calculation algorithm is included in listing 1.

2.2.5 Visibility Check

For several parts of the simulation it is necessary to find out whether two positions are visible or blocked by an obstacle. This is done by calculating the intersection between the visibility line (line from point $P_1$ to point $P_2$) and each object (see section 2.2.2).

The visibility check will then return whether the crossing point is lying on the object (on its line segment, not only on its line). The source code of the visibility check algorithm is included in listing 2.
Listing 1: Calculate Crossing Point of Line $L_1$ between $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ and Line $L_2$ between $P_3(x_3, y_3)$ and $P_4(x_4, y_4)$

```cpp
Point * s = new Point(-1, -1);

// calculate steepness P1→P2 (m1), P3→P4 (m2)
if (x1 == x2) { // Line 1 is vertical, steepness can not be calculated
    m1 = 0;
} else {
    m1 = 1.0f * (y2 - y1) / (x2 - x1);
}
if (x3 == x4) { // Line 2 is vertical, steepness can not be calculated
    m2 = 0;
} else {
    m2 = 1.0f * (y4 - y3) / (x4 - x3);
}

// 1. Case: Line 2 is horizontal, Line 1 is not
if (y3 == y4 && y1 != y2) {
    s->y = y3;
    if (m1 == 0) {
        s->x = x1;
    } else {
        s->x = x1 + (s->y - y1) / m1;
    }
}

// 2. Case: Line 2 is vertical, Line 1 is not
else if (x3 == x4 && x1 != x2) {
    s->x = x3;
    s->y = y1 + m1 * (s->x - x1);
}

// 3. Case: Line 1 is horizontal, Line 2 is not
else if (y1 == y2 && y3 != y4) {
    s->y = y1;
    s->x = x3 + (s->y - y3) / m2;
}

// 4. Case: Line 1 is vertical, Line 2 is not
else if (x1 == x2 && x3 != x4) {
    s->x = x1;
    s->y = y3 + m2 * (s->x - x3);
}

// 5. Case: Lines are parallel
else if (m1 == m2) {
    if (x3 == x4) {
        s->x = x3;
        s->y = y3;
    } else if (y3 == y4) {
        s->x = x1;
        s->y = y3;
    }
}

// 6. Case: normal, no horizontal or vertical lines
else {
    s->x = (y1 - y3 + m2 * x3 - m1 * x1) / (m2 - m1);
    s->y = y3 + m2 * (s->x - x3);
}
```
Listing 2: Visibility Check for Points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$

```cpp
// ensure that points are within field
if (x1 < 0 || x1 > width || y1 < 0 || y1 > height ||
    x2 < 0 || x2 > width || y2 < 0 || y2 > height) {
    return false;
}

// iterate over all objects
for (unsigned int i = 0; i < simulator.objects.size(); i++) {
    float x3 = simulator.objects[i].x1;
    float y3 = simulator.objects[i].y1;
    float x4 = simulator.objects[i].x2;
    float y4 = simulator.objects[i].y2;

    if (!((max(x1, x2) < min(x3, x4)) ||
          min(x1, x2) > max(x3, x4) ||
          max(y1, y2) < min(y3, y4) ||
          min(y1, y2) > max(y3, y4))) {
        Point * s = crossingPoint(x1, y1, x2, y2, simulator.objects[i]);
        float sx = s->x;
        float sy = s->y;
        delete s;

        // check whether the calculated crossing point is lying on the object:
        // if it’s not, visibility is not interrupted
        if (sx < min(x3, x4) && sx <= max(x3, x4) && sy >= min(y3, y4) && sy <= max(y3, y4)) {
            // check whether the calculated crossing point is lying between P1 and P2:
            // if it’s not, visibility is not interrupted
            if (!(sx < min(x1, x2) || sx > max(x1, x2) || sy < min(y1, y2) || sy > max(y1, y2))) {
                return false; // visibility is interrupted
            }
        }
    }
}
```

return true;
3 Potential Field Model

The first approach toward a pedestrian simulation with obstacles is inspired by an idea from Bovy and Hoogendoorn(4). They precalculate a potential field for the simulated area, which allows the pedestrians to find their destinations by walking toward the minimal potential.

3.1 Hoogendoorn’s Concept of Utility Maximization

Hoogendoorn’s concept is based on the idea that walking individuals try to maximize a utility function which increases with every performed activity and decreases with the walking costs. Hoogendoorn and Bovy specified the cost for walking (the so called running cost $L$), which depends on the importance of the activity, the expected travel time, nearness of obstacles and expected number of pedestrians on the way, the needed energy and the stimulation of the environment. A further force (called terminal cost $\theta$) is used to introduce penalties for arriving too late. The cost function can be calculated for any given time $t$, location $x$ and velocity $v$. The goal is to find the velocity path so that the cost function can be minimized, which can be described with the expected minimum perceived utility function

$$W(t, \hat{x}) = E[\int_t^{T_t} L(\tau, \hat{x}(\tau), \hat{v}(\tau)) + \theta(T_t, \hat{x}(T_t))]$$

with $\hat{v}$ being the optimal velocity. Using Bellman’s optimization principle(9), the function is calculated for a small period $[t, t + h)$. The random variate $x(t + h)$ describing the predicted location at instant $t + h$ can be substituted with a Taylor series expansion. Taking the limit $h \to 0$ results in the so called Hamilton-Bellman-Jacobi (HJB) or dynamic programming equation

$$-\frac{\delta}{\delta t}W(t, x) = H(t, x, \nabla W, \Delta W)$$

with given terminal and boundary conditions. The equation can be solved by discretizing the area into cells and approximating solutions for fixed time instants. The resulting problem is a Markov diffusion process in two dimensions.

3.2 Wave Algorithm

Instead of using partial differential equations, the potential field can be created by selecting one (or several) destinations and calculating the distance toward it for each point/cell in the area taking into consideration all the obstacles the agents have to avoid. The potential field is calculated before the simulation takes place. A separate potential field is needed for each destination. All the calculated potential fields are stored in a map for later access (see section 6.4.1). The algorithm (included in listing 3) subdivides the area into quadratic cells. The distance to the destination is always calculated for the cells’ left upper corner but will be used for the whole cell. It is initialized by setting the distance for the destination(s) (see section 6.5.3 for the usage of multiple destinations) to zero and the distance for all other cells to a maximum value. The destination cell is then added to a priority queue holding all cells to be checked sorted after their potential. The algorithm will then repeatedly pick the cell with minimal potential from the queue and try to calculate the potential for its eight neighbor cells. This is only possible if the neighbor cell is visible from the actual cell (no obstacle standing between them). The needed visibility check is described in section 2.2.5. The potential of a visible neighbor cell is then calculated by taking the potential of the just calculated cell and adding the distance between the two cells (either the cell size (for horizontal and vertical neighbor cells) or $\sqrt{2}$ times
3.2 Wave Algorithm

the cell size (for diagonal neighbor cells). If the new calculated potential is smaller than the cells’ old distance, the shortest way to reach the destination from the new cell is by walking over the old cell and the potential gets reset to the new value.

Listing 3: Potential Field Algorithm

```cpp
// STL Priority Queue with points that have to be checked
priority_queue<WeightedPoint> check;

// set potential for destination to 0 (distance is 0) and its direction to itself
potential[destx][desty] = 0;
// add destination to points to be checked
WeightedPoint temp(destx, desty, 0);
check.push(temp);

// calculate potential for points in the queue
while (!check.empty()) {
    // remove cell with minimal potential from queue
    WeightedPoint temp = check.top();
    check.pop();

    // checks all cells around actual cell whether their potential can be decreased
    for (float i = -1; i <= 1; i++) {
        for (float j = -1; j <= 1; j++) {
            // check that cell is within potential field boundaries
            if (temp.x+i >= 0 && temp.y + j >= 0 && temp.x + i < width && temp.y + j < height) {
                // calculate potential for new cell by adding up:
                // - the potential from the old cell
                // - distance between cells times the walkability
                // - the extra potential for walking near walls and obstacles
                float newPotential = potential[temp.x][temp.y] +
                    sqrt(i*i + j*j) / simulator.potential->walkability[temp.x+i][temp.y+j] +
                    simulator.potential->objPotential[temp.x+i][temp.y+j];

                // check that new potential is smaller
                if (potential[temp.x+i][temp.y+j] > newPotential) {
                    // check that visibility between the two cells is not interrupted
                    if (simulator.potential->checkVisibility(temp.x + i, temp.y + j, temp.x, temp.y, 3)) {
                        // update the walking direction of the new cell:
                        potential[temp.x+i][temp.y+j] = newPotential;

                        // add point to queue so its neighbors get checked as well
                        WeightedPoint p(temp.x + i, temp.y + j, newPotential);
                        check.push(p);
                    }
                }
            }
        }
    }
}
```

3.2.1 Performance

The order in which the cells get checked is important. A first idea may be to implement the algorithm recursively. But this would result in a lot of cells to be updated again and again (and this means checking its eight neighbor cells again and again as well). Figure 3(a) shows the first steps of the recursive algorithm.

A better solution is to use a queue to store all the cells that have to be checked. This way, the algorithm will not first check all cells in one direction, but check all the cells around the destination first, then all the cells that are two cells away from the destination, and so on. The first steps of this algorithm are shown in figure 3(b). The speedup gained when switching from the recursive approach to the usage of the queue is huge. The test data from table 1 shows that the calculation of a small potential field of 50 x 50 cells needs several million cell checks when it is done recursively, where else the second approach only needs several thousand checks. The number of cells checked does not vary a lot when objects are added to the area (but the calculation will of course need much longer as the
3.2 Wave Algorithm

visibility check needs to iterate over all objects). If a sorted queue is used\(^1\), the speedup can be increased even more, as the algorithm will then start to check the neighbors of the cells with minimal potential, which increases the chance of finding the minimal potential for a neighbor cell at first trial a lot (see figure 3(c) for an example). Table 1 shows the needed number of cell checks with the different approaches, figure 4 shows some screen shots of the created potential field after a fixed number of cell checks\(^2\). Even when using a priority queue it is not guaranteed that the first calculation of a cell’s potential is also the best (see table column “Priority Queue with Objects”). The rare cases where the first calculation does not find the minimum are found if the following happens: The potential field for cell C is first calculated from cell A (with potential \(\text{pot}_A\)) and then from cell B (\(\text{pot}_B\)). The usage of the priority queue guarantees that \(\text{pot}_A \leq \text{pot}_B\) (otherwise the calculation order would be reversed). But if cell B is lying horizontally or vertically next to cell C and cell A is a diagonal neighbor of cell C, the potential for cell C is calculated by adding the cell size to potential B (\(\text{pot}_C = \text{pot}_B + \text{cellSize}\)) and by adding \(\sqrt{2}\) times the cell size to potential A (\(\text{pot}_C = \text{pot}_A + \sqrt{2} \times \text{cellSize}\)). Therefore, the calculated potential starting with cell B may be smaller than the calculated potential starting with cell A even if \(\text{pot}_B > \text{pot}_A\). Including other aspects like walkability (see section 5.6 and object pressure (see section 5.4.2) into the potential field calculation will further increase these effects.

<table>
<thead>
<tr>
<th>Size</th>
<th>Recursive empty</th>
<th>with Objects(^3)</th>
<th>Queue empty</th>
<th>with Objects</th>
<th>Priority Queue empty</th>
<th>with Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 x 10</td>
<td>8258 (0 sec)</td>
<td>6580 (2 sec)</td>
<td>132 (0 sec)</td>
<td>121 (0 sec)</td>
<td>100 (0 sec)</td>
<td>104 (0 sec)</td>
</tr>
<tr>
<td>20 x 20</td>
<td>146794 (5 sec)</td>
<td>146922 (55 sec)</td>
<td>562 (0 sec)</td>
<td>531 (0 sec)</td>
<td>400 (0 sec)</td>
<td>460 (0 sec)</td>
</tr>
<tr>
<td>30 x 30</td>
<td>782842 (23 sec)</td>
<td>774735 (325 sec)</td>
<td>1292 (0 sec)</td>
<td>1241 (0 sec)</td>
<td>900 (0 sec)</td>
<td>1022 (0 sec)</td>
</tr>
<tr>
<td>40 x 40</td>
<td>2542990 (75 sec)</td>
<td></td>
<td>2322 (0 sec)</td>
<td>2209 (1 sec)</td>
<td>1600 (0 sec)</td>
<td>1762 (0 sec)</td>
</tr>
<tr>
<td>50 x 50</td>
<td>6290206 (195 sec)</td>
<td></td>
<td>3652 (0 sec)</td>
<td>3561 (1 sec)</td>
<td>2500 (0 sec)</td>
<td>2644 (0 sec)</td>
</tr>
</tbody>
</table>

Table 1: Needed Time and Number of Cell Checks for different Algorithms

In contrary to the graph model discussed in the next chapter, the calculated potential for a cell is not equal to its distance to the destination (so called Dijkstra metrics). First, the wave algorithm is based on updating only neighbor cells. While this allows efficient calculations, it reduced the set of possible walking directions to the eight directions of the neighbor cell (so called Manhattan metrics). This can be corrected by using some sophisticated algorithms to find the optimal walking direction (see section 5.1). Chapter 5 will show further reasons not to use Dijkstra metrics when more factors are included in the potential field calculation (walkability, object pressure).

\(^1\)The priority queue implementation from C++ Standard Template Library\(^{16}\) with overloaded operator \(<\) for class WeightedPoint

\(^2\)The whole potential field creation for the example in the figure is stored as videos in “\text{.../src/stuckip/doc/vids/PotentialFieldCreationXXX.avi}”
Figure 3: Different Approaches for Potential Field Calculation

(a) Recursion

(b) Unsorted Queue

(c) Priority Queue

Figure 3: Different Approaches for Potential Field Calculation
Figure 4: Status of the Potential Field created using Recursion (left), Queue (middle) and PriorityQueue Approach (right) after 1000, 5000 and 10000 Cell Checks
4 Graph Model

An alternative to the discussed potential field approach is to let the agents walk along the edges of a graph. The implementation of this idea is done in three steps:

1. Node generation: Starting with the given layout from the XML file, useful nodes have to be chosen which mark important points in the landscape.
2. Link generation: Pairs of nodes with uninterrupted visibility (no objects between them) get linked with a line.
3. Route generation: The final step is to choose the shortest route depending on the agent’s start position and its destination (generating a spanning tree using Dijkstra’s shortest path algorithm (10)).

4.1 Node Generation

The first step is to choose a set of nodes over which the agents should walk to reach their destination. The set should be large enough to allow realistic walking paths but small enough for fast simulations (the dependency of the graph model performance on the number of nodes is shown in section 6.4.6). The generation of the nodes is directly done by the parser when reading the environment file. The first set of nodes added to the graph are all possible endpoints defined in the XML file. Then all the objects have to be included. The simplest solution is to add all endpoints of the objects to the set of nodes. But as agents have a body, they are not able to reach nodes that are lying directly on the object, as their shape does not allow them to get within a distance of 0.23m (see section 6.3.4 for an explanation of this value) to obstacles. A better approach is to add several points in a certain distance from the objects endpoint (defined in the XML configuration file under “ObjectRange”, see section 6.1.3). With the second approach, agents will not walk directly toward the endpoint, but take a small curve around all edges. If the object is a line with endpoints (x1, y1) and (x2, y2) and x1 < x2, the four nodes (x1 - range, y1 - range), (x1 - range, y1 + range), (x2 + range, y2 - range), (x2 + range, y2 + range) are added to the graph’s node set. Objects stored as rectangles are read into the simulation as four lines. The simulation stores the four nodes outside the rectangle. The set of nodes can be further reduced by ensuring that no node is inserted twice.

4.2 Graph Generation

Once the nodes are selected, it is time to create the graph. All nodes that can “see” each other get linked. The visibility check is done with the same algorithm that is used to calculate the potential field (see section 2.2.5). It is important to minimize the number of nodes included in the graph, as the algorithm has to go over all objects and will be called for each pair of nodes (o(m*n*n) for m objects and n nodes, see section 6.4.6 for a performance evaluation). Every edge of the so created graph gets weighted with the distance between its end points (section 5.6 shows how the introduction of walkability will change the weight for special links).

4.3 Route Generation

The shortest path from the agent’s start point to its destination is found using Dijkstra’s shortest path algorithm (10) (see listing refsource:graph:rougegeneration:spanningtree). It is initialized by setting the distance to the destination for itself to zero and to a maximum value for all other nodes. All nodes are then stored in a sorted container. One after another, the node with minimal distance gets removed from the container and the weights of all its neighbors is updated depending on the weight of the link between them. If the container is empty, all nodes know their distance to the destination node and the next node on the path toward it (minimal spanning tree, see figure 5).

\[\text{The C++ set implementation of the Standard Template Library (16) with overloaded operator <}\]
4.3 Route Generation

Figure 5: Graph and Spanning Tree with Line, Rectangle and Polygon

Listing 4: Spanning Tree Generation Algorithm (Dijkstra)

```cpp
// link generation with visibility check
for (unsigned int i = 0; i < simulator.graphNodes.size(); i++) {
    for (unsigned int j = i + 1; j < simulator.graphNodes.size(); j++) {
        if (checkVisibility(simulator.graphNodes[i]->x, simulator.graphNodes[i]->y, simulator.graphNodes[j]->x, simulator.graphNodes[j]->y)) {
            Link l(simulator.graphNodes[i], simulator.graphNodes[j]);
            simulator.graphNodes[i]->links.push_back(l);
            simulator.graphNodes[j]->links.push_back(l);
        }
    }
}

set<GraphNode *> unsettledPoints;
SpanningTree * tree = new SpanningTree();
// initialisation: set node weight for destination node to zero
for (int i = 0; i < graphNodes.size(); i++) {
    if (graphNodes[i]->x == sx && graphNodes[i]->y == sy) {
        graphNodes[i]->weight = 0;
    }
}
// add all nodes to unsettled points
tree->nodes.push_back(graphNodes[i]);
unsettledPoints.push_back(graphNodes[i]);

while (!unsettledPoints.empty()) {
    GraphNode * g = *unsettledPoints.begin();
    unsettledPoints.erase(g);
    for (unsigned int i = 0; i < g->links.size(); i++) {
        // reduce weight of node at other end (if possible) and set its predecessor
        // (its next node on the way towards the destination)
        if (g->links[i].gn1 == g && g->links[i].gn2->weight > g->weight + g->links[i].weight) {
            tree->nodes[g->links[i].gn2->id]->weight = g->weight + g->links[i].weight;
            tree->nodes[g->links[i].gn2->id]->pre = g;
            unsettledPoints.insert(g->links[i].gn2);
        } else if (g->links[i].gn2 == g && g->links[i].gn1->weight > g->weight + g->links[i].weight) {
            // reduce weight of node at other end (if possible) and set its predecessor
            // (its next node on the way towards the destination)
            tree->nodes[g->links[i].gn1->id]->weight = g->weight + g->links[i].weight;
            tree->nodes[g->links[i].gn1->id]->pre = g;
            unsettledPoints.insert(g->links[i].gn1);
        }
    }
}
```
5 Agent Movement

Given a calculated potential field or spanning tree, an agent is now able to calculate its optimal walking path for any position and destination as long as there exists such a path. The simulation is iterating in loops over all agents. In every round, an agent can do exactly one step. The average speed of an agent is 1.34m/s and the step frequency around 2Hz (see section 6.1.1 and 6.1.2 for a derivation of these values). This means that an agent will walk maximally half the length of its desired speed per round and that he will not do any turns during one round (as one round is equal to one step. Of course agents do change their walking direction between two steps). To determine the next step (speed and direction), the simulation will first calculate the different forces influencing the agent. Included forces are:

- The force leading the agent toward its destination (either calculated using the potential or the spanning tree approach).
- The pedestrian pressure, which ensures that agents try to keep a certain distance from other agents.
- The object pressure, which ensures that agents try to keep a certain distance from obstacles and walls (section 5.4.2 will show that this force can be included directly into the potential field calculation).

The calculation of these forces is further described in section 5.1 to 5.4. Adding up all these forces leads to an optimal next step. Several checks are needed to ensure that this step can actually be done, as a position is only reachable if:

1. The position is within the field boundaries (simulated area).
2. No obstacles or walls are blocking the new step (check visibility).
3. No walls or obstacles are within the agents radius (check object pressure).
4. No other agents are standing within a range of twice the agents radius (check pedestrian pressure).

Figure 6 shows some examples of possible and impossible next steps.

![Figure 6: Some possible and impossible next Steps](image)

If the position can not be reached, an alternative has to be found. This part is discussed in section 5.5.
5.1 Potential Force

Using the potential field approach, agents will try to walk toward regions with minimal potential. There are different solutions how to determine the walking direction from the calculated potential field. The goal is to find an optimal trade-off between the needed number of cell checks and the exactness of the found walking direction.

5.1.1 Choosing the Neighbor Cell with minimal Potential

A simple approach to find the walking direction for a given position and destination is to check all its neighbor cells whether they are reachable and to walk into the direction of the neighbor cell with minimal potential. This solution is very fast, as the check will only include the eight neighbor cells, but the agent only knows about eight possible walking directions. So instead of walking on the shortest path toward its destination, the agent will only walk along its eight possible directions (see figure 7 for an example). If the cell with minimal potential is chosen among cells within a radius of twice the cell size, the number of possible walking directions gets doubled as well (the same results can be reached when summing up the potentials of all neighbor cells multiplied with their distance to the actual cell and walking toward the average direction). But further increasing the searched area will not enlarge the number of possible walking directions any further (except if the area is chosen large enough to include the next object blocking the direct way to the destination), as the potential field calculation is only based on eight possible directions (the directions to each neighbor cell) rather than on real distances.

5.1.2 Follow the minimal Potential until a Wall or Waypoint is reached

To enlarge the set of possible directions, the agent should not limit its search for the minimal potential to its neighbor cells. But checking all cells within a certain distance from the actual cell is expensive. On the other side, when the neighbor cells have been checked and a neighbor with minimal potential has been found, many cells within the next circle around the actual cell can be excluded (the agent already knows approximately in which direction it should start walking). It is therefore enough to check the cells around the just found cell to find the minimal potential within the next circle. This procedure is repeated until the precalculated way reaches a corner or the next way point (see source code in listing 5). As figure 8 shows, the agent is now taking the shortest path to the next corner or way point.

This approach lets the agent walk directly toward its destination or toward one of the corners of the next object blocking its direct way to the next destination. On the other hand, precalculating a long way toward the destination is rather slow and expensive. Some speed-up could be gained if the agent would store its precalculated way. An agent would then precalculate its way to the next corner or destination, store the so calculated way point and walk toward it until it is reached. But the agent may be forced away from its way by other agents (see section 5.3 about agent interactions). It would therefore be necessary to recalculate the agent’s way whenever it is pushed too far away from its precalculated path. On the other hand, the direction an agent is choosing when standing on one point is the same as long as the destination is the same. As a separate potential field is needed for each destination, the next waypoint can be precalculated for each cell while calculating the potential field and stored with the field.

5.1.3 Precalculating the next Waypoint

The precalculation of the next waypoint (next point where the agent has to change its direction) can be merged with the calculation of the potential field. Whenever the potential for a cell can be minimized in the algorithm described under 5.1.1, the waypoint should be recalculated. The waypoint from the old cell is already known. If it can be seen from the actual cell as well, it is the waypoint for the new cell as well. If on the other hand an object is blocking the view from the new cell to the old cell’s direction, it is still possible to precalculate the waypoint with the algorithm used to calculate the agent’s way in Listing 5. The source code can be found in listing 6.
problems by including new terminating conditions (stop when near a wall, stop when walkability changes). But the algorithm has to be updated whenever new criteria are added.

The precalculation algorithm is no longer useable when the potential field is no longer simply based on distances (e.g., when including the object pressure or the walkability). It is still possible to adapt this algorithm to these problems by including new terminating conditions (stop when near a wall, stop when walkability changes). But the algorithm has to be updated whenever new criteria are added.
5.1 Potential Field

### 5.1.4 Follow the minimal Potential along minimal Direction(s)

The walking direction found with the algorithm described in 5.1.1(checking only the neighbor cells) is correct, but it offers only one of eight possible directions. But the knowledge of the best direction can be used to limit further checks to cells lying in this direction. The algorithm is therefore initialized by searching the two neighbor cells with minimal potential. Now it is ensured that the optimal walking direction is lying somewhere between the directions pointing toward these cells. The algorithm will now follow these two directions until both directions do no longer offer cells with smaller potential or are not reachable. If the potential field includes other factors (object pressure, walkability), the algorithm has to be slightly adapted. It is now not enough for checked cells to offer a smaller potential, but the difference between the potential of the checked cell and its predecessor must be always the same. This ensures that the algorithm stops as soon as it reaches a region near a wall (potential is influenced by the included object pressure) or a region with another walkability factor (e.g. stair).

---

<table>
<thead>
<tr>
<th>1.)</th>
<th>2.)</th>
<th>3.)</th>
<th>4.)</th>
<th>5.)</th>
<th>6.)</th>
<th>7.)</th>
<th>8.)</th>
<th>9.)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Potential Field Grid" /></td>
<td><img src="image2" alt="Potential Field Grid" /></td>
<td><img src="image3" alt="Potential Field Grid" /></td>
<td><img src="image4" alt="Potential Field Grid" /></td>
<td><img src="image5" alt="Potential Field Grid" /></td>
<td><img src="image6" alt="Potential Field Grid" /></td>
<td><img src="image7" alt="Potential Field Grid" /></td>
<td><img src="image8" alt="Potential Field Grid" /></td>
<td><img src="image9" alt="Potential Field Grid" /></td>
</tr>
</tbody>
</table>

1.) Destination cell (3, 4) is set to 0.
2.) Neighbor cells for (3,4) get checked.
   - Cells (2,3), (3, 3), (4, 3), (2, 4), (4, 4) queued.
3.) Neighbor cells for (3, 3) get checked.
   - Cells (2, 2), (3, 2), (4, 2) blocked by wall.
4.) Neighbor cells for (2, 3) get checked.
   - Cells (1, 3), (1, 4) added to queue.
5.) Neighbor cells for (2, 4) get checked.
6.) Neighbor cells for (2, 3) get checked.
7.) Neighbor cells for (4, 3) get checked.
   - Cells (3, 2), (4, 2) added to queue.
8.)...

Figure 8: Example of the Potential Field Algorithm with a Wall.

---

---
Listing 5: Improved Walking Direction Algorithm

```c
float xx = x;
float yy = y;
bool done = false;
float minPot = potential[x][y];

while (!done) {
    int ii = 0;
    int jj = 0;

    // find the neighbor cell with minimal potential
    for (float i = -1; i <= 1; i++) {
        for (float j = -1; j <= 1; j++) {
            // ensure that new cell is within potential field boundaries
            if (xx + i >= 0 && xx + i < width && yy + j >= 0 && yy + j < height) {
                tempPotential = potential[xx + i][yy + j];
                // check whether the new cell has a smaller potential and is visible from old cell
                if (!(i == 0 && j == 0) && tempPotential < minPotential && simulator.potential->checkVisibility(xx, yy, xx + i, yy + j, 1)) {
                    ii = i;
                    jj = j;
                    minPotential = tempPotential;
                }
            }
        }
    }

    // if we are near an obstacle or no neighbor cell has a smaller potential or
    // the cell with minimal potential is not visible from the old cell, we are done
    if (simulator.potential->objPotential[temp.x + ii][temp.y + jj] > 0 ||
        (ii == 0 && jj == 0) ||
        !simulator.potential->checkVisibility(temp.x + ii, temp.y + jj, xx + ii, yy + jj, 1)) {
        done = true;
    }

    // otherwise, repeat the algorithm for the neighbor cell with minimal potential
    else {
        xx += ii;
        yy += jj;
    }
}
```

5.1.5 Using Dijkstra instead of Manhattan metrics

The problem to find the optimal walking direction can of course as well be solved when using the correct distances to the destination instead of Manhattan distances. Nishinari (2) has presented such an approach. A similar solution could easily be implemented by combining the potential field model with the graph model. The graph model would then first be used to generate a number of significant nodes in the network and calculate their distance to the destination using a spanning tree (see chapter 4 about graph model for details). The potential field is then calculated for each cell by adding the distance to the next visible graph node and the distance from this graph node to the destination along the spanning tree. This approach is not realized in the potential field model \(^5\), because it does not allow to include other factors in the potential field calculation (walkability, object pressure). Not including these factors means loosing the most important advantages of the potential field model, and it is better to use the graph model, which is using Dijkstra metrics and more efficient than the potential field approach (see section 6.4.6 for a comparison of the two models).

\(^5\)But there exists a test version of it, see Appendix D
Listing 6: Precalculating the next Waypoint

```java
// if the potential from the new cell \((x+i, y+j)\) can be decreased, do it
if (potential[x + i][y + j] > tempdata) {
    potential[x+i][y+j] = tempdata;
    // if the walking direction from the old cell \((x,y)\) can be seen from the
    // new cell this is the waypoint for the new cell as well
    if (checkVisibility(x+i, y+j, direction[x][y]->x, direction[x][y]->y)) {
        direction[x+i][y+j]->x = direction[x][y]->x;
        direction[x+i][y+j]->y = direction[x][y]->y;
    }
}
else {
    // calculate waypoint \((xx, yy)\) using algorithm from section 5.1.2
    direction[x+i][y+j]->x = xx;
    direction[x+i][y+j]->y = yy;
}
```

// checks all cells around actual cell whether their potential can be decreased
for (float i = - 1; i <= 1; i++) {
    for (float j = - 1; j <= 1; j++) {
        // if the potential from the new cell \((x+i, y+j)\) can be decreased, do it
        if (...) {
            potential[x + i][y + j] = ...;
            simulator.potential->objPotential[temp.x + i][temp.y + j] == 0 &&
            simulator.potential->checkVisibility(temp.x + i, temp.y + j,
            direction[temp.x][temp.y]->x,
            direction[temp.x][temp.y]->y, 1)) {
                direction[temp.x + i][temp.y + j]->x = direction[temp.x][temp.y]->x;
                direction[temp.x + i][temp.y + j]->y = direction[temp.x][temp.y]->y;
            }
        }
    }
}
```

5.2 Graph Force

Given a calculated spanning tree with the agents destination as root, an agent can simply follow the tree’s links up
to the root to find its way to the destination. As an agent will normally not start directly on one of the graphs node,
the first step is to find the node over which the distance to the destination is minimal and which is not hidden behind
an object. By including all the objects endpoints (or some points next to them), it is ensured that there will always
be some visible nodes for each cell. After the agent has reached this node, it will walk along the spanning tree
until it reaches its destination. If there are more walking points in the agent’s plan, another spanning tree has to be
calculated and the whole procedure gets repeated.

5.3 Pedestrian Pressure

So far an agent choosing its walking direction is only concerned with finding the shortest path toward its next des-
tination without walking through walls. But there may be other agents blocking the shortest path. It is therefore
necessary to include a second force which leads the agent around other agents. To include this second force, it is
necessary to adapt its size to the size of the (normalized) potential field or graph force (using the factor "Pedestri-
anPressureFactor" from the XML config file, the calibration of its value is described under 6.2.1). The pedestrian
pressure force may push the agent away from its shortest path toward its next destination. When passing a crowded
area, the pedestrian pressure may even try to push an agent through walls. To prevent this it is necessary to add
up the forces and check whether the calculated position is reachable (not blocked by a wall). The functionality to regulate the interactions between different agents is already included in the AlpSim project. Before an agent is doing a step, a check is done whether some other agents are standing next to it (within a certain radius stored in the XML config file as "PedestrianInteractionRange", its derivation is explained in section 6.1.2). A visibility check is included here as agents are only influenced by other agents near them if they can actually see them. An exponential force decay is used and described by Mauron(6).

\[ f_{ij} = e^{-\frac{|r_i - r_j|}{r_p}} \times \frac{r_i - r_j}{|r_i - r_j|} \]  \hspace{1cm} (25)

\( f_{ij} \) is the force contribution of agent j to agent i; \( r_i \) is the position of agent i. The pedestrian pressure will be adapted to the force from the potential field or the graph when determining the agents next step. If the pedestrian pressure is pointing opposite to the favorite walking direction, it would only slow down the agent, but not help him to avoid the other pedestrians (e.g. when two pedestrians are walking directly toward each other). In this case, the force will be adapted to lead the agent to the left or right at random.

### 5.4 Object Pressure

The force model so far enables agents to find their shortest path to any chosen destination. The visibility checks done before every step ensure that agents are not walking through walls or into other agents.

As the nodes selected for the graph creation are all within a certain distance from the endpoints of an obstacle, agents will automatically keep their distance to obstacles (if they are not pushed toward them by other agents). As a node is reached as soon as the agent is within a certain distance from its position, the agents will automatically walk in curves around obstacles (instead of walking along a link until the node is reached). An example path is shown in figure 10(e).

But the potential field as it is calculated so far leads the agents from one corner of an object to the next one, always choosing the shortest path to the destination (see figure 10(a)). But in reality, pedestrians would keep a certain distance of about 0.2 to 0.5m (see section 6.1.3) from walls and walk in small curves around corners instead of straight lines. There are several ways to keep agents away from objects. One approach is to use a force similar to the pedestrian pressure described in section 5.3. Another possibility is to increase the potential field near objects, so agents will automatically keep a certain distance from obstacles (if possible). Finally, the walkability approach discussed in section 5.6 can be used to slow down agents walking to near to a wall or obstacle. They will then automatically stay away from walls when trying to find the fastest way.

#### 5.4.1 Object Pressure

To ensure that pedestrians keep a certain distance to obstacles, Helbing has suggested in his social force model(3) to use a force similar to the pedestrian pressure described in section 5.3. The force decreases monotonic with increasing distance to the next obstacle. As all obstacles are assumed to be static (no cars or similar), these distances can be precalculated. The object pressure is precalculated for each cell by iterating over all objects and calculating the distance from this cell to the object (see section 2.2.2). If the distance is larger than 0.5m (see section 6.1.3 for an explanation of this value), the object can be ignored. Otherwise, the object pressure influencing agents standing in the actual cell is calculated depending on the distance between the agent and the object. There are several possibilities to calculate the force: Helbing uses a monotonic decreasing potential, where else Hoogendorn (4) is suggesting a negative exponential function. These forces can then be summed up and stored in an array. The problems with this approach: There is always a clear border between cells where the agent is not influenced by an object and cells where he is pushed away from the wall. This leads to some strange behavior when the agent is pushed toward a wall by the potential field. When he gets too near the wall, he will be pushed away by the object.
pressure, when he has reached a certain distance he will be pushed nearer to the wall again by the force from the potential field. Picture 10(b) shows the trail of an agent jumping along the walls.

5.4.2 Adapt Potential Field

Another possibility which will not lead to the problems described in 5.4.1 is to store the information about the distance to the next object directly in the potential field. The potential of all cells within a certain distance (0.5m, see section 6.1.3 for an explanation of this value) from an obstacle get increased. The added potential is depending on the distance to the next object (see section 6.2.2 for the exact value). One problem that occurs when the potential field is adapted is that the next way point can no longer be precalculated the way it is described in section 5.1.3, as the algorithm would find its way around the area next to an obstacle. Figure 9 shows such an example. The solution is to stop the precalculation algorithm as soon as a cell with increased object potential is reached. Cells which are near an obstacle and have an increased potential themselves will therefore only choose the neighbor cell with minimal potential as their walking direction.

<table>
<thead>
<tr>
<th>Uncorrected</th>
<th>Adapted Potential</th>
<th>Adapted Potential and Directions</th>
</tr>
</thead>
<tbody>
<tr>
<td>stops as corner is reached</td>
<td>stops as visibility is interrupted</td>
<td>stops as cell with increased potential is reached</td>
</tr>
<tr>
<td>8 7.6 7.2 6.8 6.4 6</td>
<td>10.4 9.4 8.4 7.4 6.4 6</td>
<td>10.4 9.4 8.4 7.4 6.4 6</td>
</tr>
<tr>
<td>7.6 6.6 6.2 5.8 5.4 5</td>
<td>10.8 9.8 9.8 8.4 7.8 6.4 5</td>
<td>10.8 9.8 9.8 7.8 6.4 5</td>
</tr>
<tr>
<td>7.2 6.2 5.2 4.8 4.4 4</td>
<td>11.8 12.2 10.8 9.8 7.4 5.4 4</td>
<td>11.8 12.2 10.8 7.8 6.4 4</td>
</tr>
<tr>
<td>7.6 6.6 6.2 5.8 5.4 5</td>
<td>11.8 12.2 13.2 13.8 6.8 4.4 3</td>
<td>11.8 12.2 13.8 6.8 4.4 3</td>
</tr>
<tr>
<td>8 7.6 7.2 6.8 6.4 6</td>
<td>11.8 12.2 16.6 16.6 5.8 4.4 2</td>
<td>11.8 12.2 16.6 5.8 4.4 2</td>
</tr>
<tr>
<td>9 8.6 8.2 2.4 1.4</td>
<td>11.8 16.2 18.6 5.4 2.8 1</td>
<td>11.8 16.2 18.6 5.4 2.4 1</td>
</tr>
<tr>
<td>10 9.6 9.2 2 1</td>
<td>11.8 17.2 19.6 5 2</td>
<td>11.8 17.2 19.6 5 2</td>
</tr>
</tbody>
</table>

Figure 9: Walking Directions without any Corrections (left), with adapted Potential Field (middle) and with adapted Walking Direction Algorithm (right)

5.5 Find next Position

When all the forces are calculated and added up, several checks are needed to find out whether the desired position is reachable:

- The position must be within the field boundaries (within the simulated area).
- The position must hold a minimal distance to obstacles and walls (minimal distance is equal to the radius of an agent’s body, which is defined in the XML config file under "AgentRadius". Statistical values can be found...
5.5 Find next Position

Figure 10: Walking Directions without any Corrections(a), with adapted Potential Field(b), with adapted Walking Direction Algorithm(c), with minimal Directions Approach(d) and with Graph Model(e)

- The position must hold a minimal distance to other agents (minimal distance is equal to twice the radius of an agent’s body).
- No obstacles and walls should stand between the actual and the desired position. This can be ensured with the same visibility check used for the potential field and the graph calculation (see section 2.2.5).
- A check whether other agents are standing between the actual and the desired position is not necessary. Agents can not jump over other agents as they do only one step per round (for higher speeds, e.g. in evacuation simulations, the step frequency should be adapted to the agents velocity). With normal speeds around the average of 1.34m/s, a single step will have a length of 0.67m. As agents are simulated with a radius of 0.23m, an agent would have to do a step of 0.23m (its own shape at the old position) + 0.46m (the other agents body shape) + 0.23m (again its own body shape at the new position) = 0.92m (see figure 11).

5.5.1 Find alternative position

If the step can not be done, an alternative position has to be found. This is done by checking the whole area around the agent which he could reach in one step and choosing the position with minimal potential (minimal distance for the graph model) that fulfills all criterias described above. In crowded areas, it may become difficult to find a free place and only free cells behind the agent are found (so the agent would have to walk backward). This is not totally unrealistic, as pedestrians stuck in a crowd may really choose to do some back steps to avoid the crowded section. But it is not realistic that an agent is doing a $180^\circ$ turn without losing some speed. It is therefore better to limit the area with possible positions for the next step. If the speed is less than $\frac{1}{4}$ of the agent’s maximal speed, all directions are allowed. If the speed is between $\frac{1}{4}$ and $\frac{1}{2}$ of the desired speed, all directions that do not differ by
5.6 Walkability

Buildings may have areas where pedestrians move faster or slower than their normal velocity. An example realized in the Zurich Mainstation simulation (see section 6.5.1 are stairs and escalators. When thinking of the AlpSim project(7), people may leave the path and take a short cut over a field or through a forest. Or they may be slowed down by the steepness of the path. For these situations, both models allow to introduce a walkability factor. In the potential model, the walkability has to be set for each cell. The pedestrians speed is then multiplied with the cells walkability. The default value is \( \text{walkability} = 1 \), which will not influence the agents velocity. The potential model automatically sets the cells belonging to stairs and escalators. Similar approaches can be realized in the AlpSim project. In the graph model, walkability is included by adapting the length of the belonging graph links. This works rather good for stairs and escalators, as they mostly have small links ending not far from the end of the object. If the approach should be used for the AlpSim model as well, the correct solution would be to check which part of the link is actually going over an area with limited walkability. Note that this is not (yet) realized in the graph model, as the described approximation works rather well in buildings and especially for stairs and escalators.

5.6.1 Using Walkability instead of an Object Pressure

Instead of an object pressure or an adapted potential field (see section 5.4) it is as well possible to use the walkability to ensure that agents keep a certain distance to obstacles. This is done by reducing the walkability for cells within a certain radius of an obstacle (the property “ObjectRange” in the XML configuration file) depending on their distance to the obstacle. Only agents calculating their next step using the approach described in section 5.1.4 (following the direction(s) with minimal potential until a cell with unexpected potential (e.g. because it has a special walkability) is reached) include the walkability into their decision.
6 Testing

Before running some tests, the simulation has to be calibrated by setting the properties in the XML configuration file. Several parameters can be set by taking measured empirical values from field trials. Section 6.1 presents some empirical facts about pedestrians and their walking behavior and explains the chosen values for the test simulations. Adapting the different forces (potential field force, graph force, pedestrian pressure, object pressure) to each other is more difficult. A simple test scenario, two pedestrians crossing each other, is used to adapt the pedestrian pressure and the object pressure to the potential or the graph force (see section 6.2). When the simulation system is calibrated, some standard tests are run (corridor with increased pedestrian density and fraction) and the results are compared to the results from the predecessor models and empirical data from field measurements (see section 6.3). The performance of the simulation models is tested and compared in section 6.4 and some hints are shown how some speed-up could be gained. Finally, a larger scenario (Zurich Main Station) is run to show the usability of the simulator for real world environments.

6.1 Simulation Properties

The amount of collected data about pedestrian behavior under various conditions is huge. Weidmann (8) has analyzed and collected the results of several hundred studies. The collection offers results from field measurements for most of the properties needed in the configuration file (see section A.1.1).

6.1.1 Desired Speed

The speed a pedestrian is choosing when he is not influenced by other pedestrians and obstacles depends on attributes concerning the agent itself (sex, age, size, health, mood, stress, baggage, handicaps) and its environment (purpose of walking, time of day and year, weather, way length, underground, steepness, attractiveness, danger). The speed depends further on the pedestrian density in the area (see section 6.1.2) and the distance to obstacles (see section 6.1.3). Collecting the results from more than a hundred different field measurements, Weidmann has calculated a normally distributed average speed of 1.34 m/s (4.83 km/h) and a standard deviation of 0.26 (19.3%). If the pedestrian is walking on a stair, the speed is decreased by 54% (upstairs) or 48% (downstairs) (this is implemented in section 6.5.1). Typical escalators have a steepness of $30^\circ$ and an angular velocity of $0.5m/s$ (section 6.5.2). The simulation includes these facts by defining an average speed of $1.34m/s$ and a standard deviation of 0.26 (see section B.2 for details on the script used to create agents with random position and normally distributed speed). Agents walking on a stair have their velocity reduced by 50% (these is realized with the walkability approach discussed in section 5.6). Assuming that a majority of the pedestrians is walking on escalators, a velocity of $0.8m/s$ is chosen.

6.1.2 Distance to other Pedestrians

When pedestrians are confronted with other people near them, they will try to keep a certain distance. First of all, the distance between two pedestrians is limited physically. Weidmann states an average (maximal) body diameter of 0.46m and a body depth of 0.23m. These results in a calculated rectangular floor space of $0.11m^2$ per person. Weidmann assumes a higher value of $0.15m^2$ per person, which results in a maximal density of 6.6 persons/m$^2$. With densities above 3 persons/m$^2$, physical contact is inevitable. In queues and similar waiting simulations, the measured densities range between 2.0 and 2.9 persons/m$^2$. When walking, agents need some extra space of around 30cm on each side and some free space in front of them depending on their speed. These explains the slower movement in crowded areas. The dependency of speed and density is shown in formula 28.

$$steplength = \frac{speed_{desired}}{frequency_{steps}}$$

(26)
Weidmann is working with an average step frequency of 2.1 Hz and a needed space width of 0.71m, which results in a possible density of 2.2 persons/m² for the desired average speed of 1.34m². The possible speed decreases with increasing density (-10% for 0.5 persons/m², -50% for 1.5 persons/m²) and reaches zero at a density of 5.4 persons/m². In reality, the needed space increases even faster with increasing speed, which is interpreted as a result of larger safety margins around the pedestrian. As Weidmann does not make any statements about the distance within which pedestrians are reacting on each other, field measurements from Mauron(6) are used to specify a pedestrian interaction radius of 1.71m. Several possible shapes (rectangles, circles) are tested in section 6.3.4.

6.1.3 Distance to Objects

Pedestrians will not only keep a certain distance to other people (see section 6.1.2) but to objects as well. Weidmann has collected the following distances for pedestrians walking on sidewalks: 0.45m to walls, 0.35m to fences, 0.35m to the road and around 0.3m for small obstacles like street lights, signals, trees or benches. For pedestrians passing a corridor, the distances are smaller: 0.20 to 0.25m. In curves, pedestrians keep an extra distance of 0.15m. In the test scenarios, the simulation is using the same distance of 0.5m for all objects.

6.2 Adapting the Forces

6.2.1 Adapting the Pedestrian Pressure

The constant “PedestrianPressureFactor” from the configuration file (see section A.1.1) is used to adapt the pedestrian pressure to the (normalized) forces from the potential field or the graph. To find the most realistic value, results from Mauron’s field measurements are used, which state that pedestrians will normally keep a distance of 1.03m when crossing each other. The test simulates two crossing agents and measures the crossing distance for different pedestrian pressure factors. Test results from table 2 show that the simulated distance is next to the distance measured by Mauron when using a pedestrian pressure factor of 1.5 for the potential model and a factor of 0.55 for the graph model.

<table>
<thead>
<tr>
<th>PedestrianPressureFactor</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>1.4</th>
<th>1.5</th>
<th>1.6</th>
<th>1.7</th>
<th>1.8</th>
<th>1.9</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximal Distance Potential[m]</td>
<td>0.73</td>
<td>0.79</td>
<td>0.85</td>
<td>0.91</td>
<td>0.97</td>
<td>1.02</td>
<td>1.07</td>
<td>1.12</td>
<td>1.17</td>
<td>1.21</td>
<td>1.25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PedestrianPressureFactor</th>
<th>0.5</th>
<th>0.55</th>
<th>0.6</th>
<th>0.65</th>
<th>0.7</th>
<th>0.75</th>
<th>0.8</th>
<th>0.85</th>
<th>0.9</th>
<th>0.95</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximal Distance Graph[m]</td>
<td>0.83</td>
<td>1.02</td>
<td>1.17</td>
<td>1.32</td>
<td>1.46</td>
<td>1.59</td>
<td>1.72</td>
<td>1.83</td>
<td>1.93</td>
<td>2.01</td>
<td>2.08</td>
</tr>
</tbody>
</table>

Table 2: Maximal Distance between two crossing Agents for different Pedestrian Pressure Factors.

Figure 12: Screen Shot of two Agents and their Trails walking toward each other (blue Agent is coming from the left, yellow Agent from the right side)
6.2.2 Adapting the Object Pressure

As the object pressure is directly included in the potential field (see section 5.4.2), there is no such thing like an “ObjectPressureFactor” for the potential model. What has be calibrated is the value with which the potential field is increased next to obstacles (depending on the distance from the cell to the next object). If the additional value is higher than the normal addition between two neighbor cells (which is either 1 for horizontal or vertical neighbors or \( \sqrt{2} \) for diagonal neighbor cells), it is ensured that an agent (if he is not pushed by other agents) will always choose cells that are not lying within the defined range of an obstacle. On the other side, agents that can not walk into their favorite direction (because it is blocked by other agents) will still accept cells next to obstacles, as they are not concerned about the cells exact potential value as long as it is the (reachable) neighbor cell with minimal potential. The factor “AddedPotentialObjects” is set to 1.5, so ensuring that it is higher than the normal addition between two neighbor cells even if a large cell size of 1m is used.

The graph model ensures that pedestrians keep a certain distance to objects when choosing the graph nodes around (and not on top) of an objects corner (see section 4.1). The chosen distance in the property “ObjectRange” is equal to the distance an agent will keep from objects if it is not influenced by other agents.

6.3 Standard Tests

This section includes two standard tests to show the correct functionality of the two models. The first test measures the speed and the flux within a corridor with an increased number of pedestrians walking into the same direction. In the second test, agents are passing the corridor in both directions. A small Perl script is used to create the needed number of agents with random positions within the corridor area (it is listed in Appendix 11).

6.3.1 Simulation Parameters

**Corridor Layout** Mauron has modeled its corridor as a rectangle with length L = 10m and width W = 2m. The area S of the corridor is L*W = 20\( m^2 \). To exclude the influences from walls (object pressure) and boundary conditions, the simulated corridor is chosen twice as wide and deep and only the results from the inner part of the corridor (10m x 2m) were included. These is similar to the testing environment chosen by Hoogendoorn(4) for this test.

**Density** The density \( \rho \) is defined as the number of pedestrians N divided by the surface of the corridor S (pedestrians \( /m^2 \)).

**Fraction** The parameter \( 0 \leq C \leq 0.5 \) quantifies the fraction of pedestrians going in one direction. \( C = \frac{1}{3} \) means that a third of the pedestrians goes in one direction and the rest walks in the other direction.

**Pedestrian Interaction** The pedestrian interaction range (1.71m, measured by Mauron) as well as the pedestrian and the object pressure factor (see section 6.2) are the same for all tests.

**Boundary Conditions** The simulation uses periodic boundary conditions. Agents reaching one end of the corridor reappear on its other side and forces may affect agents on the other end of the corridor. If random boundary conditions are used (for every agent leaving one end of the corridor, another agent is placed at a random position at the other end of the tunnel), some typical pedestrian behavior like line formation and direction separation can not be observed.
6.3.2 Measured Quantities

**Mean forward Speed**  The mean forward speed \( \langle v_f \rangle \) describes the average speed of the pedestrians relative to the desired velocity \( v_0 \), i.e.

\[
\langle v_f \rangle = \frac{1}{N} \sum_{\text{pedestrians}_i} v \cdot e_0.
\]  

(29)

**Pedestrian Flux**  The pedestrian flux \( \Phi \) describes the number of pedestrians passing a gate perpendicular to the corridor walls in one second divided by the gate width.

6.3.3 Simulation Procedure

3 simulations with different random seeds are run for each set of parameters. A simulation run consists of 100 thermalization time steps to “warm up” the simulation. These are followed by 250 time steps with logged speed and pedestrian flux. The simulations are run for both the potential field and the graph model.

6.3.4 Results uni-directional Flow in Corridor

The first test is used to show the behavior of agents walking under different pedestrian densities. An important factor is the chosen ground plan by which an agent is represented. According to Weidmann, an average person has a body width of 46cm and a depth of 23cm, which results in an average space requirement of 0.085m\(^2\) (density: 9.3 persons/m\(^2\)). With clothes, baggage and some extra space for the feet, the needed space increases to 0.15m\(^2\) (density: 6.6m\(^2\)). These facts offer different approaches for simulations; a first idea is to represent agents by circles, as it is easy to calculate the distance between two agents and to find overlapping persons. One tested approach is simulating the pedestrians with a radius of 0.23m, which matches with the average maximal pedestrian radius (0.5 times the body width). A second approach is to calculate the radius from the average space requirement, which results in a radius of 0.19m (\( \sqrt{\frac{0.15}{\pi}} \)). But the agent can as well be represented by a rectangle. Again, it is possible to take the values from Weidmann (width = 0.46m, depth = 0.23m) or to calculate them from the average space requirement (by keeping the typical ratio of 2:1), which results in a width of 0.56m and a depth of 0.28m.

The measured values are compared with the collected values from Weidmann, who cites the following formula by Kladek(8) to calculate the speed for a given pedestrian density.

\[
v_{F,hi} = v_{FH,i} * (1 - e^{-\gamma * (\frac{1}{D_{max}} - \frac{1}{D})})
\]  

(30)

\[
v_{F,hi} = 1.34 * (1 - e^{-1.913 * (\frac{1}{D_{max}} - \frac{1}{D})})
\]  

(31)

\( v_{F,hi} \) is the velocity at full freedom of movement, \( \gamma \) is a calibration constant, \( D \) denotes the Density and \( D_{max} \), the maximal density at which movement gets impossible. With \( v_{F,hi} = 1.34 \text{m/s} \) and \( D_{max} = 5.4 \text{P/m}^2 \) Weidmann calculates a calibration constant \( \gamma = 1.913 \text{P/m}^2 \).

The expected and simulated velocities (for different shapes) are shown in figure 13, the exact results can be found in Appendix C.

The measured values show that shapes calculated from the average space requirement offer the better result than those taken directly from the average body width (which is only stated for naked persons). As the best approximation
6.4 Performance

is obtained when representing the agents by circles with a radius of 0.23m, these settings will be used for all further testings. Even though the model was kept as simple (and as fast) as possible, none of the simulated values (for the selected radius r=0.23m) differs by more than 10% or 0.05m from the values stated by Weidmann. According to Weidmann, a maximal flux of about 1.2 to 1.5 P/sm (the results differ from study to study) is reached at a density of 1.7 to 2.0 P/m². The tests with the potential field show a maximal flux of 1.47P/sm at a density of 1.75 P/m², the values for the graph model are 1.80 P/sm at a density of 1.5 P/m².

6.3.5 Results bi-directional Flow in Corridor

The second test shows the behavior of pedestrians passing a corridor in both directions. The mean average forward speed and the flux are calculated for different densities and fractions. The speed loss caused by the pedestrians walking in opposite direction varies (according to Weidmann) between 4 and 15%. The loss is not bigger as the two pedestrian flows separate from each other. Similar to the results from Hoogendoorn, the agents in the test are slowed down up to 50% and the heaviest losses are found for equal pedestrian flow sizes (fraction C = 0.5). Flow separations and line formations are only found for densities up to 1.5 P/m². Some screen shots of the test simulations for different fractions and pedestrian densities are shown in figure 15.

6.4 Performance

A pedestrian simulation is measured on two points: realistic results and performance. Both models are designed to allow effective simulations concerning time and memory consumption. Sections 6.4.1 to 6.4.5 show some measures taken to speed up the simulation. Note that many performance improvements are already described in the prior chapters. The results of the executed performance tests for different environments are shown in section 6.4.6. Section 6.4.7 offers some tips how the speed-up could be increased even more (note that the actions discussed in this section have not been implemented).
6.4 Performance

(a) Potential

(b) Graph

Figure 14: Mean Forward Speed and Flux for different Fractions and Pedestrian Densities

6.4.1 Caching calculated Data

In the potential model, a separate potential field has to be calculated for each destination. The calculation is done when an agent requests the potential field for a specific destination. As most simulations include several agents with the same destination, it is useful to cache the calculated potential fields. This is done using an associative container with keys\(^6\). The key used is a concatenation of the destinations’ x and y coordinates. To ensure that the key for destination (1, 23) and (12, 3) is not the same, the key generation algorithm is extending all coordinates to the same length:

\(^6\)The map implementation of the Standard Template Library(16)
The same approach is used to store the calculated spanning trees in the graph model.

### 6.4.2 Store calculated Data for further Simulations

Testing a simulation often means to run the same (or similar) scenarios again and again. To speed up repeated simulations, calculated potential fields (for the potential model) and visibility graphs (for the graph model) are stored in XML or text files. Before calculating a new potential field or graph, the simulation will first check whether
6.4 Performance

the desired potential or spanning tree is already cached. If this is not the case, the simulation will try to load the requested data from a file. A calculation is only needed if both trials fail.

6.4.3 Ignore small Obstacles in Spanning Tree

The performance of the graph model depends mostly on the number of included graph nodes, which increases linearly with the number of objects. But it is not necessary to include small obstacles like benches and pillars when generating the spanning tree, as agents will automatically avoid these objects when they are confronted with them. A good optimization is therefore to load all objects from the XML file, but to base the spanning tree only on important objects (this is easily implemented in the XML file parser, as every object has a specified type like "building", "wall" or "pillar". A small check to exclude types with small objects is enough). Section 6.4.6 shows the gained speed-up when reducing the number of objects and nodes. Figure 16 shows that in rare cases, this approach may lead to wrong trails. In the example (from Zurich Main Station environment), the four pillars in the middle of the picture are not included in the visibility graph. As the direct way for the agent on the right is blocked by the pillars, it has to walk over a near node. This results in the strange path shown in the figure, as no nodes are found near the pillar.

![Figure 16: Wrong Trail (red) because smaller Objects (Pillars) are ignored. Correct Trail is indicated in green.](image)

6.4.4 Order Conditions in if Statements

There are two functions in the code that are called again and again and consume a lot of time. The first is the visibility check, which has to iterate over all objects, the second the calculation of the pedestrian pressure, which loops over all agents. There are several places in the code where the execution of a statement depends on multiple conditions. In those cases, it should always be ensured that the most expensive checks are done last. If one of the previous conditions fails, the expensive visibility check or pedestrian pressure calculation can be dropped. This small changes reduced the calculation time for the potential field by a factor of ten.

6.4.5 Pre- versus Just-In-Time Calculation

Data calculated for the whole area (e.g. the walking direction) can be either precalculated before the actual simulation takes place or calculated only when it is needed by an agent. Precalculation spends a lot of time at the beginning of the simulation, where else just-in-time calculations slow down the performance while the agents are walking around. The best solution is to combine the advantages of both approaches by calculating the walking direction for a cell when it is needed for the first time and then store it for further agents walking over this cell.
6.4 Performance

6.4.6 Performance Results

The performance tests are run on SIM2, a computer with a Pentium III 700MHz processor and 256MB of RAM. The performance measurements will concentrate on the needed simulation time, as the memory usage of both models is rather low (<10MB). The tests are run with the data from the Zurich Main Station environment (see section 6.5) with more than 3000 Objects and the standard settings proposed in section A.1. They compare the dependence of the needed calculation time on various aspects.

**Cell Size (Potential Model)** The potential field model can be calculated using different cell sizes. As could have been expected, the calculation time grows linear with the number of cell checks. If the cell size is halved, the number of cells gets multiplied by a factor of four and so does the calculation time. Table 3 shows the needed calculation time and the number of cell checks for different cell sizes. The same results can be found by increasing the size of the area (which is another way of changing the number of cells). The importance of calculating the cells in the right order has already been discussed in section 3.2.1. The performance of the graph model does neither depend on the size of the area nor the size of the single cell.

<table>
<thead>
<tr>
<th>Cell Size (m)</th>
<th>Cell Checks</th>
<th>Calculation Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1'325'459</td>
<td>1554</td>
</tr>
<tr>
<td>0.5</td>
<td>325'331</td>
<td>386</td>
</tr>
<tr>
<td>1</td>
<td>78'324</td>
<td>102</td>
</tr>
</tbody>
</table>

Table 3: Influence of the Cell Size on the Calculation Time.

**Number of Objects (Potential Model)** Before the potential of a new cell can be set, a visibility check has to be done to ensure that the new cell is visible from the old one. As the check has to iterate over all objects, the calculation time will increase linear with the number of objects included in the simulation. The results shown in table 4 affirm this. Visibility checks are a real performance killer, they should therefore only be done when they are really needed, as is further discussed in section 6.4.4.

<table>
<thead>
<tr>
<th>#Objects</th>
<th>Calculation Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>65</td>
</tr>
<tr>
<td>500</td>
<td>219</td>
</tr>
<tr>
<td>1000</td>
<td>420</td>
</tr>
<tr>
<td>1500</td>
<td>643</td>
</tr>
<tr>
<td>2000</td>
<td>866</td>
</tr>
</tbody>
</table>

Table 4: Influence of the number of simulated Objects on the Calculation Time.

**Number of Nodes (Graph Model)** The graph model spends most of its simulation time on the calculation of the links. Again, the most time consuming operation is the visibility check needed to find out whether two nodes can be linked (there exist better visibility graph algorithms. They are discussed in section 6.4.7). As each pair of nodes has to be checked, the performance will increase quadratically with the number of nodes. Table 5 shows the expected quadratical dependency between calculation time and the number of simulated nodes.

**Number of Agents (Potential and Graph Model)** A comparison of the two models with increased number of agents is done for the Zurich Main Station environment in section 6.5.5.
### 6.5 Simulating Zurich Main Station

The standard tests described in section 6.3 have shown that the behavior of the simulated pedestrians is similar to measured real world results. A larger simulation is needed to show the performance of the simulations. The Main Station of Zurich was chosen as the final challenge for both models. It includes an area of about 700m x 200m and more than 3000 walls and obstacles. Several enhancements had to be added to allow realistic simulations. Stairs and escalators had to be introduced (see sections 6.5.1 and 6.5.2). For several tasks it was necessary to include multiple destinations in one potential field or spanning tree (see section 6.5.3). Section 6.5.4 describes the used solutions to generate the needed pedestrians and their walking plans. The presented results give an overview about the various possibilities of the simulation.

#### 6.5.1 Stairs

People walking on stairs do not have the same speed as people walking on even ground. Weidmann(8) has measured that pedestrians loose about 48% of their speed when walking downstairs and 54% of their speed when walking upstairs. In the simulation, all pedestrians walk with only 50% of their speed when on a stair (this is realized with the walkability approach discussed in section 5.6). When walking down or up a stair, pedestrians reach another floor. As the simulation is only 2-dimensional, floors are simulated next to each other. This can be done by including so called beam points in the middle of the stair. In the graph model, it is sufficient to add a special beam node in the middle of a stair. All agent walking over the stair will have to pass this point and get beamed to the other floor. In the potential field model, more beam points have to be added to ensure that all agents passing the stir get beamed.

#### 6.4.7 Some more Ideas

All the tested performance dependencies seem realistic. The simulation could gain a lot of speed if the simulated area is split up in blocks, so the visibility check only has to care about objects and agents within the same or one of the neighbor blocks. Agents crossing the border between two blocks are erased from the old block and added to the new one. This strategy is already included in the AlpSim project(7).

There are a lot of more sophisticated algorithms to create visibility graphs(11). The graph model uses the naive approach of going over all objects for all pair of nodes (o(n² * m) for n nodes and m links). Better algorithms use binary search trees (o(n * log n * m)) or arrangements (o(n²)). Optimal solutions have complexity (o(n * log n + m)). Instead of using complex algorithms, the graph model offers the opportunity to store once calculated graphs in an XML file. Therefore the visibility graph for a building must only be calculated once. This is a useful approach as long as the simulations are based on static environments. If the models should once be used to find optimal buildings by using genetical algorithms, it is certainly worth the effort to include one of the standard visibility graph algorithms, as the simulated environment will change for each simulation. Similar approaches could be used to speed up the algorithm used when initializing new agents to find the best graph node on the spanning tree to walk towards the root.

#### 6.5 Simulating Zurich Main Station

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>#Links</th>
<th>Calculation Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Links</td>
</tr>
<tr>
<td>100</td>
<td>1'301</td>
<td>1</td>
</tr>
<tr>
<td>500</td>
<td>13'349</td>
<td>51</td>
</tr>
<tr>
<td>1000</td>
<td>25'558</td>
<td>458</td>
</tr>
<tr>
<td>1500</td>
<td>39'868</td>
<td>1225</td>
</tr>
<tr>
<td>2000</td>
<td>50'587</td>
<td>2660</td>
</tr>
</tbody>
</table>

Table 5: Influence of the number of simulated Nodes on the Calculation Time.
The solution is to add a row of beam cells across the stair. The simulation has to take care of beam points in the following situations:

**Potential Field Calculation**  When the wave algorithm calculating the potential field reaches a beam point, the potential for the point at the other end of the beam (the same stair on the upper or lower floor) gets set with the same potential and added to the queue as well.

**Spanning Tree Generation**  To simulate beam points in the graph model, extra nodes in the middle of a stair are introduced. They are linked with the other end of the beam point with a graph link of length zero. The spanning tree can then be calculated as usual.

**Pedestrian Pressure**  The pedestrian pressure has to take care of beam points in so far as the distance in the simulation is no longer equal to the distance in the real world, as agents standing on the same stair may be at different ends of a beam point. The solution is easiest explained in an example: The Zurich Main Station simulation represents an area of 700 x 200m. To simulate subways, the same area is attached again below the ground floor, so creating an area of 700 x 400m. If a stair from the main floor to the subway has a middle point (its beam point) at \((x_1 = 100, y_1 = 200)\), the same stair in the subway (to the ground floor) will be represented at point \((x_2 = 100, y_2 = 300)\). The check whether two agents are influencing each other is now no longer based on the distances \(dx = x_2 - x_1\) and \(dy = y_2 - y_1\) but the distances \(dx = x_2 - x_1\) and \(dy = (y_2 - y_1)\%200\).

**Walking**  Pedestrians reaching a beam point should disappear on one floor and reappear on the other floor. In the graph model, this is done by checking whether the agent is standing on a beam point whenever he reaches a graph node. If this is the case, the agent gets beamed to the other end by updating its position according to the coordinates of the beam point at the other end. In the potential field model, it is necessary to include a flag so that agents know whether or not they have just been beamed to another floor. Otherwise they may find out that they’re still on a beam cell in the next round and get beamed back where they have come from. Figures 17 shows a small stair example with walking trails for both models.

### 6.5.2 Escalators

Escalators are handled like stairs, but with a walkability of 0.8. What should be included as well (but is not yet) is that movement on escalators is directed (only one way). The direction of the escalator would have to be specified in the environment file and beam points would have to be directed.

### 6.5.3 Multiple Destinations

Walking destinations and way points can not always be specified with fixed coordinates. A pedestrian may only be interested to get to the next exit, not to a specific one. For these situations, both models allow to specify multiple destinations. Nodes are allowed to have a type like “exit” in the environment file and way points may be specified with a type rather than its coordinates. In the graph model, multiple destinations are handled by generating a special node with coordinates (-1, -1) and weight zero. All destination nodes are then linked with this special node with links of weight zero. Thus the spanning tree generation algorithm will start with the special node and move on to all destination nodes and from there to all other nodes. Of course agents have reached their destination as soon as they reach one of the destination nodes (no need to walk to the special node).

Multiple destinations in the potential field approach are modeled by setting all destinations to zero before starting the wave algorithm. The destination is reached as soon as the reached cell is pointing to itself. For special problems it may be useful to rate the destinations. A pedestrian may be interested to read the train arrival times. he would
6.5 Simulating Zurich Main Station

Figure 17: Example stair with Walking Trail for both Models

take into account a detour of 10m if he could get the information from a screen rather than an old-fashioned arrival board. These can be simulated by rating the destinations. In the graph model, links from arrival screen to the special node (-1, -1) are weighted with zero as hitherto, where else links from arrival boards get a weight of 10. A similar solution exists for the potential model by setting the potential or destination cells with arrival boards to 10 instead of zero.

6.5.4 Agent Generation

The simplest solution is to generate agents at random within the simulated area and include a check whether or not they are standing on a walkable area (not within a building, not on the railways...). This check can be done by ensuring that the potential field for the starting position is set or that one of the graph nodes is visible. Note that this can not be done within the generation Perl script, as it does not know anything about the potential field or the spanning tree.

The simplest scenario is the simulation of an evacuation. All agents will walk to the next possible exit (multiple destination approach, see section 6.5.3). More sophisticated scenarios are the generation of a crowd of people at a platform (simulation of a train arrival, as used in section 6.5.6). Complex scenarios can only be simulated using a plan generator, which would go beyond the scope of this report (see (12) for actual studies on this topic).

6.5.5 Performance

The performance tests for the Zurich Main Station area were run on the same machine as the the Corridor tests (P3 700MHz, 256MB RAM). A simple scenario was used with agents generated at random over the whole area. If an agent has to start on an impossible position (within an obstacle, on the railway), it gets simply removed. Using this simple agent generation approach, 43% of the created agents are already removed in the first round (so 57% of the area are accessible). All agents have the same aim, which is to leave Zurich Main Station as soon as they can. This
is realized by setting their first and only way point to "exit" and using the multiple destination approach discussed in section 6.5.3 with seven possible exits.

The first measurements stop the time needed until the simulation can start. Parsing the XML environment and plan files is done in a few seconds. So the preparation time depends mostly on the time needed to generate the potential field or the visibility graph and spanning tree. Six approaches are tested:

1. Potential field approach with cell size 0.5m.
2. Potential field approach with cell size 0.25m.
3. Potential field approach with cell size 0.5m and precalculated walking directions (see section 5.1.3).
4. Potential field approach with cell size 0.25m and precalculated walking directions.
5. Graph model (3136 nodes, 91262 links).
6. Graph model ignoring small obstacles like pillars or benches for spanning tree calculation (1379 nodes, 12557 links, see section 6.4.3).

Table 6 shows an overview over the needed precalculation times.

<table>
<thead>
<tr>
<th>a)</th>
<th>b)</th>
<th>c)</th>
<th>d)</th>
<th>e)</th>
<th>f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>386</td>
<td>1'554</td>
<td>9230</td>
<td>10h</td>
<td>2'134</td>
<td>10'582</td>
</tr>
</tbody>
</table>

| Table 6: Needed precalculation times (s) for a) Potential(0.5cm cells), b) Potential(0.25cm cells), c) Potential with Precalculation (0.5cm cells), d) Potential with Precalculation (0.25cm cells), e) Graph with ignored small Obstacles and f) Full Graph.

The next investigated point is the time needed to simulate the walking agents. Table 7 gives an overview over the needed total time and time per round to simulate 10, 100 and 500 agents leaving Zurich Mainstation. It includes as well the average time and distance needed by the agents to leave the terminal. 5 variants are tested, all potential fields use a cell size of 0.5m:

1. Potential field with minimal neighbor approach (see section 5.1.1).
2. Potential field with minimal distances approach (see section 5.1.4).
3. Potential field with precalculated walking directions (see section 5.1.3).
4. Graph model (3136 nodes, 91262 links).
5. Graph model ignoring small obstacles like pillars or benches for spanning tree calculation (1379 nodes, 12557 links, see section 6.4.3).

In the graph model, the first simulation round takes much longer than all the other rounds as each agent has to find the node on the optimal node on the spanning tree to walk to the destination. Therefore, the time needed for the first round is shown as well. In the potential field model, about a minute is needed to load the already calculated potential field.

The results of the simulations are quite similar for all approaches. The creation of the potential field is up to five times faster than the graph model (if a cell size of 0.5m is used, which is reasonable for most simulations, but may cause problems when small passages are used (escalators)). With a cell size of 0.25m, both models need about the same time to do their precalculations. Precalculated walking directions speed up the simulation time per round, but the time needed before the simulation start is so huge that it is almost never worth the effort (only if simulations
for the same environment are run again and again). The potential field model is faster while the simulation takes place. The graph model shows its advantages when more complex simulations with several different destinations are simulated. The visibility graph has to be calculated only once and can then be reused for the creation of the different spanning trees (spanning tree creation is done within seconds). The potential model on the other hand has to calculate a new potential field for each destination. As a rule of thumb we can state that potential fields are faster for small simulations and more flexible for the inclusion of new ideas, while graph models are the faster solution for complex simulations.

A screen shot of Zurich Main Station simulated with the graph model is presented in figure 18, one using the potential field approach in 19.

### 6.5.6 Results

The creation of a potential field or spanning tree for given destinations can already produce interesting results without having to simulate any agents. The creation of a potential field or spanning tree with all ticket automatas as destinations can for example be used to find places where passengers have to walk a long way to get their ticket and where it could be worth to place another automat. Another possibility is to use the potential field with all possible exits as destinations to find the locations where people would have to run the longest way to leave the building in case of an emergency\(^7\)(when looking at the results from section 6.5.5, even agents walking at normal speed do not need much longer than one minute and 250 meters to leave the building from any position within the simulated area.

\(^7\)pictures of these potential fields can be found under "/src/stuckip/doc/PotentialFieldExit.png", "/PotentialFieldTicket.png" and "/PotentialFieldTrolleys.png"
(if they do not care which exit they take)).

Some results using walking pedestrians have already been shown in section 6.5.5. All results (measured times and walking distances) have to be treated with care, as the creation of complex scenarios with realistic activity plans would go beyond the scope of this diploma work. The following screen shots of some simulations should just give an impression of the useability of both models for large simulations. Figure 20 shows a jam of pedestrians before a stair. The simulation used 100 agents created at random on platform 4 and 5 leaving the terminal through the subway.

Figure 21 shows pedestrian flow separation and line formation in the subways. The subway used 100 agents arriving on platform 6 and 7 leaving the terminal toward Bahnhofstrasse (downwards) and 100 agents arriving at platform 4 and 5 leaving toward Landesmuseum (upwards). The two flows have to cross in the subway to reach their destinations.
Figure 18: Spanning Tree for Zurich Main Station with Trails of 500 leaving Agents (Multiple Destinations Approach: seven possible Exits). Ground Floor is modeled on the left Side, Subway on the right Side. Lines between left and right Side are Beam Links for Stairs and Escalators.
Figure 19: Potential Field for Zurich Main Station with Trails of 500 leaving Agents (Multiple Destinations Approach: eight possible Exits). Ground Floor is modeled on the left Side, Subway on the right Side.
6.5 Simulating Zurich Main Station

(a) Potential

(b) Graph

Figure 20: Pedestrian Crowd before Stairs

Figure 21: Flow Separation and Line Formation in the Subway.
7 Summary

The report presented two models for pedestrian simulations. Both models offer realistic results similar to empiric data about pedestrian behavior in a corridor with increased density and fraction. Typical pedestrian behavior like line separation and trial formation are found in both models. In comparison to other pedestrian simulation models, both models are kept lean and allow efficient simulations of large areas with several thousand agents, as was shown by simulating Zurich Main Station. The main difference of the two models is found in their performance. The potential field approach is faster when all agents are walking towards the same destination (e.g. evacuation simulations, where the goal is to find the next exit). For more complex simulations with different destinations, the graph model is more efficient, as it can reuse the visibility graph for the calculation of different minimal spanning trees. New factors influencing the agents walking direction and speed are easier included in the potential field approach, as is shown with the implementation of the walkability idea and the introduction of stairs and beam points.

One of the future goals is to include one of the models into AlpSim to simulate hikers walking through a village or entering a cable railway. It would certainly be interesting to enhance the models with more details and more realistic activity plans and test them on other buildings (stadium, airport, ETH). The performance of both models can still be improved by realizing one or several of the adaptions suggested in the report (blocks).

I can honestly say that I liked most of the work included in my diploma thesis, as I was free to follow the directions I was interested in. I hope the models will be of some use in the AlpSim project or further diploma works. Looking back I can say that I probably spent to much time trying to find better algorithms to calculate the walking direction from a given potential field. Especially the development of the solution precalculating the directions during the calculation of the potential field needed a lot of time and is probably just too slow to be of much use in the AlpSim project. But I still think that it is the most original idea included in this work, as it bases the solution of finding the best walking direction on the way the potential field gets created. This is (but that is just my humble oppinion) a really elegant way to include Dijkstra metrics in the potential field approach. I haven’t found any similar approaches in related models, and it sure gives the best walking trails. I finally like to thank all persons helping me during these four month, first of all my assistant Christian Gloor and Professor Kai Nagel. Further thanks go to Michael Balmer for arranging the contacts with SBB (Swiss Railway Organisation) and Thomas Streit from SBB, who showed huge interest in my work and offered me all the Zurich Mainstation Maps for free.
References


A Manual

The source code for the simulation is written in C++. It is stored in the local CVS directory under

- ...src/stuckip/src/potential and
- ...src/stuckip/src/graph.

A.1 Configuration

Before running a simulation, some simulator properties should be set and the simulator must be told about the environment (obstacles) and the agents (activity plans) it should simulate. All these configurations are done using XML files.

A.1.1 XML Configuration File

All configuration properties for the simulator are stored in the XML configuration file. It can be found under

- ...src/stuckip/potential/config.xml or
- ...src/stuckip/graph/config.xml.

If another name for the file is used, the Makefile or the starting command (see section A.2) has to be changed as well. A typical XML configuration file is shown under listing 7.

The following list describes the purpose of all properties.

Number of Network Files The property “NumberOfNetworkFiles” states how many network files have to be read in. The simulator will then try to read in the filenames of the network files under the properties “NetworkFile0” ...“NetworkFileN”. Multiple environment files are used when simulating a building or area with several floors. The simulation will then automatically adapt the coordinates of points on different floors so they do not interfere with objects on other floors (see section 6.5.1).

Network File 0..n The property “NetworkFile” holds the path and the name of the XML file with the environment settings (obstacles and nodes). The usage of this file is described in section A.1.2.

Plan File The property “PlanFile” holds the path and the name of the XML file with the agents’ activity plans. The usage of this file is described in section A.1.3.

Width The width of the simulated area (in m). Ensure that it matches the environment described in the network file (see A.1.2).

Height The height of the simulated area (in m). Ensure that it matches the environment described in the network file (see A.1.2).
Listing 7: XML Config File

```xml
<config>
  <module name="pedsim">
    <!--Parameters specifying the simulated area-->
    <param name="NumberOfNetworkFiles" value="2"/>
    <param name="NetworkFile0" value="mainstation0.xml"/>
    <param name="NetworkFile1" value="mainstation-1.xml"/>
    <param name="PlanFile" value="plan.xml"/>
    <param name="Width" value="700"/>
    <param name="Height" value="200"/>
    <param name="CellSize" value="2"/>
    <param name="Timeout" value="10"/>
    <param name="PedestrianPressureFactor" value="1.5"/>
    <param name="ObjectPressureFactor" value="0"/>
    <param name="AgentRadius" value="0.23"/>
    <param name="ObjectRange" value="0.25"/>
    <param name="AddedPotentialObjects" value="1.5"/>
    <param name="PedestrianInteractionRange" value="1.71"/>
    <!--Parameters specifying the calculated forces-->
    <param name="ZoomViewer" value="5"/>
    <param name="ZoomPicture" value="5"/>
    <param name="DarknessPotential" value="1"/>
    <param name="DrawDirections" value="false"/>
    <param name="DrawPotentialField" value="false"/>
    <param name="DrawObjectPressure" value="false"/>
    <param name="CreateVideo" value="true"/>
    <param name="CreateFinalPicture" value="true"/>
    <param name="DrawTrailsViewer" value="true"/>
    <param name="DrawTrailsFinalPicture" value="true"/>
    <param name="DrawTrailsVideo" value="true"/>
    <param name="DrawSpanningTree" value="true"/>
  </module>
</config>
```

**Cell Size**  For the calculation of the potential field, but also for the object pressure, the simulated area is divided into squares of fixed size. The exactness, but also the performance of the simulation, depends on the chosen cell size. The cell size is stored in meters, tested cell sizes are 1, 0.5 and 0.25.

**Time Out**  The time in milliseconds after which the viewer is updated. As the simulation is usually run as fast as possible, the timeout property can just be kept on a low interval (e.g. 10ms) which ensures that the viewer will not unnecessarily slow down the simulation.

**Pedestrian Pressure Factor**  The factor to adapt the calculated pedestrian pressure (see section 5.3 for the potential field and section 5.2 for the graph model) to the force from the potential field or the graph. The calculated pressure is multiplied with the stored factor. If the factor is set to zero, the pedestrian pressure is ignored. The tests shown in section 6.2.1 show the derivation of the used factors for both models (1.5 for the potential model, 0.55 for the graph model).

**Agent Radius**  This parameter ensures that agents are not only simulated as points but have an actual volume with the specified radius. Tests while calculating the agent’s next step ensure that the agent keeps a minimal distance from other agents and walls equal to the stated radius. Weidmann(8) states that the average (maximal) body diameter is 0.46m and the average body depth 0.23m. Several body shapes (rectangle, circle) and sizes are tested in section 6.3.4. A circular body with a radius of 0.23m offers the best results.
Object Range  The radius around each object within which pedestrians are reacting on the object (tying to avoid it). This is the distance the agents will keep from walls when they are not forced to walk nearer. Weidmann has collected measured distances for different kinds of obstacles, which range between 0.2 and 0.5m. The simulation uses the same distance of 0.5m for all objects.

**Added Potential Objects**  The force that keeps agents away from objects is included directly in the potential field (see section 5.4.2). The value to be added to the potential is calculated by subtracting the distance from the cell to the object from a constant maximal value, which is stored in the configuration file as property “AddedPotentialObjects”. If agents can not walk into their desired direction as their way is blocked, they will walk into the direction of the (reachable) neighbor cell with minimal potential (only interested in the relative value to other cells, not the absolute potential). The chosen additional value is therefore not that important. If it is higher than the maximal difference between two neighbor cells (which is $\sqrt{2}$ times the CellSize for diagonal cells), it is ensured, that an agent will only choose to walk near the object if all other ways are blocked. The used value is therefore 1.5, as it is higher than $\sqrt{2}$ * CellSize even if the cell size is set to 1.

Pedestrian Interaction Range  The radius within which pedestrians react and try to avoid each other. In his diploma work, Laurent Mauron(6) has done some field trials and calculates a pedestrian interaction range of 1.71m.

Zoom Viewer  The factor to enlarge the viewer showing the moving agents. A factor of 10 means that one simulated square meter is shown as 10 x 10 pixels.

Zoom Picture  The factor to enlarge the created screen shots and the calculated potential or graph. A factor of 10 means that one simulated square meter is stored with 10 x 10 pixels.

Darkness Potential  This parameter is only used for the potential field model. If the parameter “DrawPotentialField” is set to true, the simulation will create a screen shot of the calculated potential field. The background color of each cell depends on its potential. There are 256 different shades of a color that can be used for the background color (0 is black, 255 is the brightest color). The background color for each cell is calculated by subtracting the product of its potential and the property “DarknessPotential” from the maximal value (255). So the higher the property, the darker the drawn potential. A dark potential will emphasize the differences between the cells, but some parts of the area that are too far away from the destination may remain black. For most simulations it is best to use a value between 1 and 5.

Draw Directions  This parameter is only used for the potential field model. The property “DrawDirections” tells the simulator whether it should draw the precalculated walking directions for each cell (see section 5.1.3) in the screen shot drawn after the potential field is calculated. Allowed values for this property are “true” and “false”.

Draw Potential Field  This parameter is only used for the potential field model. Allowed values are “true” and “false”. If the property is set to “true”, the simulation will create a screen shot of the calculated potential field (otherwise it will only draw the obstacles). The background color of each cell depends on its potential (see section A.1.1 for details).

Create Video  This parameter tells the simulator to create screen shots after each iteration over all agents. The screen shots can then be used to create a video of the simulation (see appendix B.1 for details). Allowed values are “true” and “false”.
Create Final Picture  This parameter tells the simulator to create a screen shot from the viewer after the last agent has reached its destination. Allowed values are “true” and “false”.

Draw Trails Viewer  This parameter tells the simulator to include the trails of all agents (path they have walked so far) in the viewer. Allowed values are “true” and “false”.

Draw Trail Final Picture  This parameter tells the simulator to include the trails of all agents (path they have walked so far) in the final picture taken after the last agent has reached its destination (see section A.1.1). Allowed values are “true” and “false”.

Draw Trails Video  This parameter tells the simulator to include the trails of all agents (path they have walked so far) in the screen shots taken after each iteration over all agents (see section A.1.1). Allowed values are “true” and “false”.

A.1.2 XML Environment File

The simulated environment (destination points, obstacles) is read in from the XML file(s) specified in the configuration file under “NetworkFile0” to “NetworkFileN” (see section A.1.1). The parsing of these files is described in section 2. An example of such an XML file with several nodes and obstacles is shown under listing 8. Note that nodes can be classified with a type. This is important when generating the agents’ plan files. Agent can be instructed to walk toward a specific way point (by specifying the points id) or to walk toward the next way point of a specified type (e.g. walk toward the next exit, see section 6.5.3).

Listing 8: XML Environment File

```xml
<network>
  <nodes>
    <node id="0" type="meetingpoint" x="535" y="95"/>
    <node id="1" type="exit" x="505" y="25"/>
    <node id="2" type="exit" x="589" y="25"/>
    <node id="3" type="exit" x="655" y="67"/>
  </nodes>
  <objects>
    <rectangle type="trolleys"> <!--Baggage Trolleys behind Snack Bar (Side Landesmuseum)-->
      <coord x="506.6" y="31"/>
      <coord x="507.2" y="35"/>
    </rectangle>
    <rectangle type="building"> <!--Snack Bar (Side Landesmuseum)-->
      <coord x="513.4" y="25.8"/>
      <coord x="524.6" y="30"/>
    </rectangle>
    <line type="info"> <!--Info Map (Side Landesmuseum)-->
      <coord x="540.4" y="32.6"/>
      <coord x="542.6" y="32.6"/>
    </line>
    <polygon type="escalator"> <!--Escalator from 2.UG (Main hall)-->
      <coord x="545.4" y="82.6"/>
      <coord x="562.6" y="86.2"/>
      <coord x="562.6" y="87.4"/>
      <coord x="545.4" y="83.8"/>
    </polygon>
  </objects>
</network>
```
A.2 Start the Simulation

A.1.3 XML Plan File

The activity plans of the simulated agents are read in from the XML file specified in the configuration file under “PlanFile” (see section A.1.1). The parsing of this file is described in section 2. An example of such an XML file with several agents is shown under listing 9. Note that agents can be instructed to walk toward a specific way point (by specifying the points id) or to walk toward the next way point of a specified type (e.g. walk toward the next exit, see section 6.5.3).

Listing 9: XML Plan File with Agent Activities

```xml
<agents>
  <agent id="0" starttime="0" x="10" y="90" speed="1">
    <waypoint type="exit"/>
  </agent>
  <agent id="1" starttime="0" x="20" y="90" speed="1">
    <waypoint type="trolley"/>
    <waypoint id="1"/>
  </agent>
</agents>
```

If the simulation should include several hundred or thousand agents, it is useful to create the plan file with a script. An example is shown in section B.2.

A.2 Start the Simulation

The simulation can be started with the Makefile included with the source code under...

- ...src/stuckip/potential/Makefile or
- ...src/stuckip/graph/Makefile.

The command “make run” will compile all code and start the simulation. The command “./sim config.xml” is used to start the simulation without using the Makefile.

A.3 Analyzing the Results

Depending on the settings in the xml configuration file (see section A.1.1 for details), the simulation will log occurring events, store calculated data for further reuse and create screen shots of the calculated potential field or spanning tree and the moving agents and their trails.

A.3.1 Output

A running simulation writes to the standard output what it is doing at the moment and how long it needed for every action it takes (file parsing and loading, field or graph generation, round simulation...). This is useful for comparing the performance of the different models (as it is done in section 6.5.5).

A.3.2 Events

The position of all pedestrians is logged after each round (equal to one step) in the file...
A.3 Analyzing the Results

- ...src/stuckip/src/potential/data/events.xml or
- ...src/stuckip/src/graph/data/events.xml.

A typical event is of the following form:

```xml
event type="walk" id="0" time="10" x="399.331" y="6.02819"/>
```

so logging the position of agent zero ten seconds after the start of the simulation. If a waypoint is reached, the following event is logged:

```xml
event type="waypoint" id="0" time="10" x="399.331" y="6.02819" extra="5.76"/>
```

The “extra” marks how far the distance the agent has walked so far. This allows to calculate the average time and distance an agent needs to reach its destination (as it is done in section 6.5.5). It may help a lot to analyze the log file with grep.

A.3.3 Stored Results

To prevent the simulation from recalculating already known data (see section 6.4.2 for details), once calculated data gets stored in different files:

- A calculated potential field for the destination x = 50, y = 20 gets stored under “../src/potential/data/PotentialField.x50.y20.tx”.
- The calculated graph links for the environment file “network.xml” get stored under “../src/graph/data/GraphLinksnetwork.xml”.
- The calculated object distances for the environment file “network.xml” get stored under “../src/graph/data/DistancesToNextObjectnetwork.xml.txt”.

A.3.4 Created Pictures

Analyzing text files to check whether the simulation has calculated a correct potential field or spanning tree is rather exhausting (so it can not always be prevented). But often a picture of the calculated potential field is already enough to locate the problems. The simulator is therefore producing several pictures and screenshots:

- A picture of the calculated potential field for destination x = 50 and y = 20 is stored under “../src/potential/data/PotentialField.x50.y20.png”.
- A picture of the calculated graph with all visible links is stored under “../src/graph/data/Graph.png”.
- A picture of the calculated spanning tree for destination x = 50 and y = 20 is stored under “../src/graph/data/SpanningTree.x50.y20.png”.
- Screen Shots of the walking agents taken after each simulation round are stored under “../src/potential/data/movie/xxx.png” or “../src/graph/movie/xxx.png”. They can be used for video creation (see section B.1).
B Some useful Scripts

During this work I have created and used several small tasks where it would simply have to be to much work to type the same commands again and again. This section is therefore not really part of the report, but may be quite useful when using the simulator.

B.1 Creating a Simulation Video

Some time is needed to simulate large buildings with many agents. It is therefore useful to have a possibility to store already run simulations for demonstration purposes. The application offers a possibility to create a video of the run simulation. As the storing of all the simulated events needs its time, the video creation process can be switched on and off with the property “CreateVideo” in the XML config file.

B.1.1 Creating Simulation Screenshots

The video is created from a series of pictures. These pictures have to be created and stored during the simulation. The way this is done for the moment is by using the function “main.draw()”, which iterates over all agents and marks their actual position. Later on, it will eventually be possible to switch to GDK2.0 and use the “pixbuf” package to create screen shots directly from the viewer. The created screen shots have to be numerated, so that the video application knows in which order the pictures have to be included. Simply enumerating the pictures with 1, 2, 3 ... will not work, as a string comparison is used and therefore “11.png” ¡ “2.png”. A simple solution is to start the enumeration with a high value (e.g. “10000.png”). The GDK package(14) creates png images with 8 bits per pixel. The video application on the other side needs images with 24 or 32 bits per pixel. The conversion can be done with the Linux function “convert” and is best done with a small script iterating over all pictures (see listing 10). Ensure that the script “convert8bpp24bpp” is in the same directory as the screenshots and run it with the command

Listing 10: Shell Script converting all Images in the Directory from 8bpp to 24bpp

```bash
#!/bin/bash
for f in *.png; do
  convert -geometry 800x800 $f $f
  convert -type TrueColor $f `basename $f .png`_24.png
  rm $f
done
```

B.1.2 Creating the Video

The video is created with the function “mencoder” of the mplayer package(15). Width (800 in the example), height (800) and the number of frames per second (10) can be passed as argument.

```
mencoder -mf on:w=800:h=800:fps=10:type=png -ovc copy -o test.avi \*.png
```

The encoder creates a video in the .avi format, which can be watched with mplayer or other video viewers.
B.2 Agent Generator

The tests explained in section 6 included up to several hundred agents. Small Perl scripts are used to generate the needed number of agents with random positions and normally distributed speed and store them in an appropriate XML format. The script included in listing 11 was used to create the needed number of agents. The speed is calculated using a mean value and a standard deviation.

Listing 11: Perl Script creating Agents for Corridor Simulation

```perl
#!/usr/bin/perl

print "<agents>

$nOfAgents = $ARGV[0];
$id = 0;

srand($$+time);

while ($id < $nOfAgents) {
    $starttime = 0;
    $x = sprintf "%2f", rand(700);
    $y = sprintf "%2f", rand(200);

    $mean = 1.34;
    $deviation = 0.26;
    $low = $mean - $deviation * 1.75;
    $high = $mean + $deviation;

    $speed = rand($high);
    last if ($speed >= $low);

    $waypointid = 0;
    print "	<agent id="$id" starttime="$starttime" x="$x" y="$y" speed="$speed">
        		<waypoint id="$waypointid"/>
    </agent>

    $id++;
}

</agents>
```

The script is started with the following command:

```
perl PlanGeneratorCorridor.pl 100 0.5 > plan.xml
```

The first argument ("100") stands for the number of generated agents, the second ("0.5") for the used fraction and "plan.xml" is the XML plan file where all the agents are stored. Similar scripts are used for the different mainstation simulations. A second script (see listing 12) is used in the corridor tests to start a whole series of simulations with increased pedestrian density and fraction. It writes all the results into the specified log file.

B.3 Update Coordinates in XML File

After having read in more then 3000 objects from different plans of Zurich Mainstation, new objects above all others made it necessary to update the coordinates of all objects. This can easily be done using awk(13), a pattern matching program included in most Linux distributions. The used awk script is shown in listing 13.

The script is only updating the y coordinates, but can be easily adapted for x coordinates. The script is used with the following command:

```
awk -F "" -f CoordinatesAdaptor.awk infile.xml > outfile.xml
```
Listing 12: Perl Script starting a whole series of Corridor Simulations with increased Pedestrian Density and Fraction

```perl
#!/usr/bin/perl

system("make");

 logfile = "results.txt";

 for ($fraction = 0; $fraction <= 0.5; $fraction += 0.1) {
     open (LOG, ">$logfile") || die "$logfile could not be opened: $!
     print LOG "Simulation: $fraction Fraction:\n"
     close LOG;
     system("perl PlanGeneratorCorridor.pl 4 $fraction > plan.xml");
     system("./sim config.xml");
     for ($agents = 20; $agents <= 400; $agents+=20) {
         system("perl PlanGeneratorCorridor.pl $agents $fraction > plan.xml");
         system("./sim config.xml");
     }
 } for ($fraction = 0; $fraction <= 0.5; $fraction += 0.1) {
     open (LOG, ">$logfile") || die "$logfile could not be opened: $!
     print LOG "Simulation: $fraction Fraction:\n"
     close LOG;
     system("perl PlanGeneratorCorridor.pl 4 $fraction > plan.xml");
     system("./sim config.xml");
     for ($agents = 20; $agents <= 400; $agents+=20) {
         system("perl PlanGeneratorCorridor.pl $agents $fraction > plan.xml");
         system("./sim config.xml");
     }
 }
```

Listing 13: AWK Script updating all Coordinates in an XML file

```awk
BEGIN{}
if ($3=="y"){
    print "$1 " "$2 " "$3 " "$4+15 " "$5
} else {print}
```
### C Test Results

<table>
<thead>
<tr>
<th>Speed (m/s)</th>
<th>Weidmann / Kladek</th>
<th>Circle 0.19m</th>
<th>Circle 0.23m</th>
<th>Rectangle 0.46 x 0.23m</th>
<th>Rectangle 0.56 x 0.28m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.34</td>
<td>1.32</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
</tr>
<tr>
<td>0.50</td>
<td>1.30</td>
<td>1.22</td>
<td>1.22</td>
<td>1.22</td>
<td>1.22</td>
</tr>
<tr>
<td>0.75</td>
<td>1.20</td>
<td>1.15</td>
<td>1.14</td>
<td>1.14</td>
<td>1.15</td>
</tr>
<tr>
<td>1.00</td>
<td>1.06</td>
<td>1.06</td>
<td>1.04</td>
<td>1.06</td>
<td>1.04</td>
</tr>
<tr>
<td>1.25</td>
<td>0.93</td>
<td>0.97</td>
<td>0.93</td>
<td>0.98</td>
<td>0.92</td>
</tr>
<tr>
<td>1.50</td>
<td>0.81</td>
<td>0.87</td>
<td>0.79</td>
<td>0.87</td>
<td>0.80</td>
</tr>
<tr>
<td>1.75</td>
<td>0.70</td>
<td>0.77</td>
<td>0.66</td>
<td>0.79</td>
<td>0.66</td>
</tr>
<tr>
<td>2.00</td>
<td>0.61</td>
<td>0.68</td>
<td>0.57</td>
<td>0.69</td>
<td>0.55</td>
</tr>
<tr>
<td>2.25</td>
<td>0.52</td>
<td>0.61</td>
<td>0.50</td>
<td>0.60</td>
<td>0.46</td>
</tr>
<tr>
<td>2.50</td>
<td>0.45</td>
<td>0.55</td>
<td>0.42</td>
<td>0.54</td>
<td>0.38</td>
</tr>
<tr>
<td>2.75</td>
<td>0.39</td>
<td>0.48</td>
<td>0.35</td>
<td>0.47</td>
<td>0.32</td>
</tr>
<tr>
<td>3.00</td>
<td>0.33</td>
<td>0.43</td>
<td>0.30</td>
<td>0.43</td>
<td>0.26</td>
</tr>
<tr>
<td>3.25</td>
<td>0.28</td>
<td>0.39</td>
<td>0.24</td>
<td>0.38</td>
<td>0.23</td>
</tr>
<tr>
<td>3.50</td>
<td>0.23</td>
<td>0.35</td>
<td>0.19</td>
<td>0.35</td>
<td>0.20</td>
</tr>
<tr>
<td>3.75</td>
<td>0.19</td>
<td>0.31</td>
<td>0.15</td>
<td>0.31</td>
<td>0.16</td>
</tr>
<tr>
<td>4.00</td>
<td>0.15</td>
<td>0.27</td>
<td>0.12</td>
<td>0.28</td>
<td>0.14</td>
</tr>
<tr>
<td>4.25</td>
<td>0.12</td>
<td>0.24</td>
<td>0.09</td>
<td>0.25</td>
<td>0.12</td>
</tr>
<tr>
<td>4.50</td>
<td>0.10</td>
<td>0.21</td>
<td>0.06</td>
<td>0.22</td>
<td>0.09</td>
</tr>
<tr>
<td>4.75</td>
<td>0.06</td>
<td>0.18</td>
<td>0.04</td>
<td>0.20</td>
<td>0.08</td>
</tr>
<tr>
<td>5.00</td>
<td>0.03</td>
<td>0.16</td>
<td>0.02</td>
<td>0.18</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 8: Tables with speed (m/s) and flux (pedestrians/sm) for different densities for uni-directional flow in corridor.
### Table 9: Tables with speed (m/s) and flux (pedestrians/sm) for different densities and fractions for bi-directional flow in corridor.
D Available Source Code and Documentation

All files belonging to this work can be found in the "...src/stuckip" cvs repository. The files are arranged in two main branches: "...src/stuckip/src" is containing all the source code of the two models while "...src/stuckip/doc" is holding the LaTeX files for this report and additional documentation and videos.

D.1 Source Code

Different runnable versions for both models are available:

- **src/potential** The final version of the potential field model. It is configured to simulate Zurich Main Station using the files "mainstation0.xml" (ground floor), "mainstation-1.xml" (subways) and "plan.xml" (agents activity plans). The minimal directions approach described in 5.1.4 is used to find the shortest paths to the destination and object forces are included with the walkability approach described in section 5.6.

- **.../src/stuckip/src/potential/corridor** The version used for the corridor tests described in section 6.3.

- **.../**

- **src/stuckip/src/graph** The final version of the graph model. It is configured to simulate Zurich Main Station using the files "mainstation0.xml" (ground floor), "mainstation-1.xml (subways) and "plan.xml" (agents activity plans).

- **.../src/stuckip/src/graph/corridor** The version of the spanning tree model used for the corridor tests described in section 6.3.

All versions have the same code structure and use the following files:

- "main.h", "main.cpp" The usual C++ main file starting the simulation.
- "Simulator.h", "Simulator.cpp" The class storing all objects and agents and logging occurring events.
- "Parser.h", "Parser.cpp" The XML parser to read in the environment and plan files.
- "ConfigParser.h", "ConfigParser.cpp" The XML parser to read in the simulation properties.
- "Geometry.h", "Geometry.cpp" File with all object classes (node, line, point...).
- "Agent.h", "Agent.cpp" Class simulating a walking agent.
- "Potential.h", "Potential.cpp" Class generating the potential field.
- "Graph.h", "Graph.cpp" Class generating the spanning tree.
- "Makefile" Makefile used for compiling and starting the simulator.
- "sim" The executable simulator.
- "config.xml" The configuration file with the simulator settings.
- "mainstation0.xml", "mainstation-1.xml" XML environment files Zurich Main Station (ground floor and subways).
- "stairs.xml" XML environment for a stairs example.
- "corridor.xml" XML environment file used for the corridor testings.
- "plan.xml" XML plan file with agents’ activities.
- "PlanGeneratorCorridor.pl" Perl script to create agents for corridor simulations (see section B.2).
- "PlanGeneratorMainstation.pl" Perl script to create agents for mainstation simulations.
- "Testing.pl" Perl script to run a series of corridor simulations (see section B.2).
- CoordinatesAdaptor.awk AWK script to adapt the coordinates of an environment file (see section B.3).
- "data/" The folder where all output (logfile, pictures, XML and text files) will be stored.

Table 10 shows an overview of all functions and where they are described in the code (functions labeled with “P” are used in the potential field model, functions with a “G” in the graph model).

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Section</th>
<th>P</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Constructor</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Agent Destructor</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Agent.move</td>
<td>5</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Agent.addNode</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Agent.pedpressure</td>
<td>5.3</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Agent.objpressure</td>
<td>5.4</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Agent.potpressure</td>
<td>5.1</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Agent.graphpressure</td>
<td>5.2</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ConfigParser.initXMLConfigParser</td>
<td></td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ConfigParser.getPropertyAsString</td>
<td></td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ConfigParser.getPropertyAsInt</td>
<td></td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ConfigParser.getPropertyAsFloat</td>
<td></td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ConfigParser.getPropertyAsBool</td>
<td></td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Point Constructor</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Point.print</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WeightedPoint Constructor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WeightedPoint.print</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WeightedPoint.operator:</td>
<td></td>
<td>3.2.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Node Constructor</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Node.print</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GraphNode Constructor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphNode.print</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphNode.operator:</td>
<td></td>
<td>4.3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Line Constructor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line.print</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link Constructor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link.print</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpanningTree Constructor</td>
<td></td>
<td>4.3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SpanningTree.draw</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpanningTree.findNextGraphNode</td>
<td></td>
<td>5.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
<td>Section</td>
<td>P</td>
<td>G</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-------------</td>
<td>---------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Graph Constructor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph Destructor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph.generateLinks</td>
<td></td>
<td>4.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.getSpanningTree</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph.crossingPoint</td>
<td></td>
<td>2.2.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.checkVisibility</td>
<td></td>
<td>2.2.5</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.isCorner</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph.draw</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph.setNextWall</td>
<td></td>
<td>5.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.getNextWall</td>
<td></td>
<td>5.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.calculateMapKey</td>
<td></td>
<td>6.4.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.getBeamPoint</td>
<td></td>
<td>6.5.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.printLinks</td>
<td></td>
<td>6.4.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.loadLinks</td>
<td></td>
<td>6.4.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Graph.calculateDistancesToNextObject</td>
<td></td>
<td>5.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>main</td>
<td></td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Parser.xmlstart</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Parser.xmlend</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Parser.DataHandler</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Parser.parseNetwork</td>
<td></td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Parser.parsePlan</td>
<td></td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Parser.parseGraphLinks</td>
<td></td>
<td>6.4.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PotentialField Constructor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PotentialField.calculatePotentialField</td>
<td></td>
<td>3.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PotentialField.draw</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PotentialField.print</td>
<td></td>
<td>6.4.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential Constructor</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential Destructor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Potential.load</td>
<td></td>
<td>6.4.2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.calculateMapKey</td>
<td></td>
<td>6.4.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.calculateDistancesToNextWall</td>
<td></td>
<td>5.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.calculateWalkability</td>
<td></td>
<td>5.6</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.getPotentialField</td>
<td></td>
<td>5.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.crossingPoint</td>
<td></td>
<td>2.2.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.checkVisibility</td>
<td></td>
<td>2.2.5</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.isCorner</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Potential.isEndPoint</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.getBeamPoint</td>
<td></td>
<td>6.5.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.setNextWall</td>
<td></td>
<td>5.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Potential.getNextWall</td>
<td></td>
<td>5.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Simulator Constructor</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Simulator Destructor</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Simulator.initPotential</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulator.initGraph</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulator.openLogfile</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Simulator.closeLogfile</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Simulator.logEvent</td>
<td></td>
<td>A.3.2</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Table 10: Table with all Functions and where they are described in the Report.

D.2 Documentation

The \LaTeX files used to create this report are found in the folder ”...src/stuckip/doc/template”. It holds the following files and directories:

- DA.pdf The final version of this report.
- DAsmall.pdf A smaller version without huge screen shots.
- DA.tex The main \LaTeX file used to create the report.
- *.tex All \LaTeX files included in the report.
- */eps” The images included in the report.
- */fig” Original versions of Xfig graphics.
- */jpg” Original jpg images.
- */png” Original png images (screen shots of simulations).
- */xls” Excel sheets of the corridor testing results.

The slides for the presentation are stored under ”...src/stuckip/doc/slides”.

Videos of executed simulations can be found in the ”/vids” folder.

Finally there are two old documentations of the pedestrian pressure and the object pressure:

- ”...src/stuckip/doc/PedestrianPressureExample.doc”
- ”...src/stuckip/doc/ObjectPressureExample.doc”