Master Thesis

Multi agent room simulation for early stage building layout design
Diploma thesis

Author(s):
Voellmy, Andreas

Publication Date:
2003

Permanent Link:
https://doi.org/10.3929/ethz-a-004696422

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.
Multi Agent Room Simulation for Early Stage Building Layout Design

Diploma Thesis in Computer Science, Eidgenössische Technische Hochschule, Zürich

written by
Andreas Völlmy

Supervisors:
Prof. Kai Nagel
Bryan Raney
Karsten Droste

September 25, 2003

Abstract

In the early stages of building design, architects have to map functions to areas inside the building. These functions provide services to the future users of the building. In this work we develop two agent-based algorithms to find plausible examples of such layouts for architects to draw ideas from. These algorithms let the building’s users determine which layout is good and which one is not.

In the first algorithm we let the function areas find positions inside the building and have the users evaluate them in iteration cycles. In the second one we let the users directly decide where a function-area should be and change previous decisions by again running several iteration cycles.

We find that for a layout to be considered good by the users, the functional areas inside it need to somehow learn what the users like.

With these two algorithms we show that it is possible to find designs by letting agents make decisions in order to optimize their own happiness (or the happiness of their users), instead of having an entity outside the system, that decides how things should be and how to make all the users happy.
Contents

1 Introduction 4
  1.1 Previous Work .................................................. 4
     1.1.1 Traffic Simulation ........................................... 5
  1.2 Setup ............................................................. 5

2 First Algorithm: Spreading Qooms 6
  2.1 Previous Work .................................................. 6
     2.1.1 KaisersRot .................................................... 6
  2.2 Our Algorithm .................................................. 7
     2.2.1 Strategies ..................................................... 7
     2.2.2 Spread Qooms ............................................... 8
     2.2.3 Evaluation .................................................... 9
  2.3 Nash Equilibrium vs. Global Maximum ......................... 11
  2.4 Results .......................................................... 11
     2.4.1 Single Scoring Functions .................................. 11
     2.4.2 Multiple Scoring Functions ............................... 14
     2.4.3 Strategy Selection Factors ............................... 16
     2.4.4 Global Score vs. Individual Score ...................... 19
  2.5 Without Spreading ............................................... 19

3 Second Algorithm: Dynamic Qooms 21
  3.1 Idea ............................................................... 21
  3.2 Algorithm ........................................................ 21
     3.2.1 Next Worker Selection ..................................... 21
     3.2.2 Qoom Costs ................................................... 22
     3.2.3 Select next qoom location ................................. 23
     3.2.4 Qoom Cutting ................................................ 25
     3.2.5 Kick-Out ...................................................... 25
     3.2.6 Stabilizing the Building ................................. 25
     3.2.7 More Solutions ............................................. 26
  3.3 Improvements .................................................... 26
     3.3.1 Salaries for Qooms .......................................... 26
  3.4 Results .......................................................... 27
     3.4.1 Graphical Output ........................................... 27
     3.4.2 Cost Factors .................................................. 28
     3.4.3 Qoom-Deletion Cost Function ............................ 31
     3.4.4 Qoom-Deletion Cost Function with “Schedule” .... 31
     3.4.5 Qoom Salary Factors ...................................... 33

4 Comparison 35
  4.1 Rectangular Spreading Qooms .................................. 35
  4.2 Comparison between the Rectangular Spreading Qooms and the Dynamic Qooms . . . . 36
5 Outlook

5.1 Spreading Qooms .......................................................... 38
5.2 Dynamic Qooms .............................................................. 38
      5.2.1 Q-Learning .......................................................... 39
5.3 Rectangular Spreading Qooms .......................................... 39

6 Summary ........................................................................ 39

7 Acknowledgments ................................................................ 40

A Appendix: Implemented But Not Used .............................. 40
   A.1 Dynamic Qooms ............................................................ 40
      A.1.1 Next Worker Selection ............................................. 40
      A.1.2 Select next qoom location ........................................ 40

References ........................................................................... 42
1 Introduction

In the early stages of building design, architects have to map functions to areas inside the building. These functions provide services to the future users of the building but require certain facilities or access to facilities. E.g., the function work-area provides the service work-desk to the user, but needs light, heat and access to the hallway to do so.

In this diploma thesis we want to develop a system that can give the architect a good layout of the function areas for an office building. A good layout is one that makes the users of the building as happy as possible, where a user’s happiness depends on various parameters, like how far he has to walk, whether his office has a window or how large the office is. The parameter values can be adjusted by the architect.

The function-areas are not yet rooms, since they just say, e.g. “in this area we will need a workplace” and do not yet specify where exactly walls will be. For this reason we do not call them rooms, but quasi-rooms or just qooms.

In addition, a room can contain several qooms; for example, the manager’s office often contains a workspace and a meeting area, or normal offices contain several work-areas, so that more than one worker can work in the same office.

These qooms are used by the people inside the building, whom we will call workers (even though they may be customers, etc).

Our way to create such a layout is by letting the workers decide, what they like of a building and thus optimizing the building-design for themselves. This is contrary to trying to find a globally optimal building. In our view this is more how people are in reality, since they want to optimize things for themselves, not for everybody.

This is also simpler, since defining a global function for how good a layout is, is hard, while having people say if a layout is good for them, is easier.

In the first algorithm presented in section 2 we let the qooms try to make the workers using them as happy as possible, while in the second algorithm in sec. 3 we let the workers themselves try to optimize their happiness.

1.1 Previous Work

There are several projects to automate design, or at least help the designer with his job. First, there are various projects, working on evaluations of a design [7, 11, e.g.]. Others try to optimize parts of the building, mainly to reduce energy consumption. This is done by optimizing the heating structure [1] or the sizes of windows and other parameters of the outer walls like in [21]. Even later optimizations of this type are done by letting the finished building learn how to regulate certain parameters, which is discussed in [5].

Another topic of design automation are design agents in virtual realities (VR). In one option for this the user designs in a VR that supports him with different features like in [8] in another option the the VR that adapts to the user’s needs [13].

Even stronger in the direction of our project are projects like Agency-GP [16], where design agents try to produce a well-designed building (or structure of spaces). This is different to our project as they try to build a virtual architect, while we want to have the users of the building decide how the building should look.

A similar system is A-Design [3] which was designed mainly for mechanical and electronic designs, which sets a similar problem as architectural design, since it also tries to arrange rectangular objects in a two-dimensional space (plus several levels). A-Design again has various design agents that create a design and then get scores according to how good that design was. Again, a global view is optimized and not the satisfaction of the individual design element.

A complete package to support the early phases in building design is SEED [15], which also contains a layout generation part. This seems to work by giving goals to the rooms and then doing some optimizations on those. This again tries to create a design for the building by optimizing various constraints from outside.

Similar ways to optimize a layout, by giving specific goals for each room and for the relationships between the rooms are given can also be found in [6, 20]. [4] uses a knowledge-based system to do the same.

So the main difference of our work to most of the above mentioned ones is that we try to optimize the
happiness of individual workers, while they try to create a globally optimal solution. This is also a big difference to the bin-packing problem. Bin-packing solves the problem of how to put blocks of given size into a given space under some constraints. Additionally, we assume that our building is large enough for all rooms, so we do not have a constraint on the available space. If this is not the case, the present system cannot find a stable solution. Besides, the architects are not really interested in the best solution, but in various good solutions, that help them come up with their own good designs. A good reference for all kind of different layout search algorithms is [2]. Algorithms to search for good layouts (with global functions to define design quality), plus heuristic, rule-based, genetic, simulated annealing and extended pattern search algorithms are described there as well as some algorithms to detect geometric interference. Again, these are different from our methods, because they use global functions (or goals).

1.1.1 Traffic Simulation

A project that is related to ours is a multi-agent traffic simulation as described in [17, 18]. The framework of that project is similar to the one this project is part of [14], as it simulates the individuals using the traffic network while they try to optimize their own happiness. These individuals are represented in the system as objects with internal states who follow their own individual rules. These objects are called agents. This kind of representation is especially applicable, if the agents have complicated internal structures and all differ from each other.

The framework consists of various modules, e.g. the traffic-simulation, an agent-database or a router. Inside the traffic-simulation, the agents traveling through a traffic network are simulated. For this the agents tell the traffic-simulation through which routes they want to travel. These routes, together with information on what the agents want to do during the day and where they live, work, etc. make up an agent’s strategies, which are stored in the agent-database (which is the agents memory), together with scorings on how happy a strategy made the agent. So a strategy is obviously not so good, and will get a score to reflect this, if the agent was sitting for half a day in a traffic jam.

The strategies are executed by the traffic simulation and the output is evaluated by the agents who then score their strategies. If the agent decides to find a different way through the network, he asks the router for a new route. But he could also decide to do something else during the day, or to move to a different location, etc. So the agents can change their strategies or get new ones by asking a module for something new, but because not all agents should change their strategy at the same time, only a fraction of them may change something before the next iteration step.

But the traffic simulation first needs to know which strategy each agent is following during that day. To solve this problem, the simulation is run several times, which is called an iteration. One call to the simulation is called an iteration step.

As described in [14] we want to implement a similar framework, consisting of an agent database (storing the strategies of the workers), a layout creation module (which comes up with a layout for the building) and a pedestrian simulation (where the workers evaluate the building). In this thesis we try out two prototypes for this, that’s why all modules are built into one system.

1.2 Setup

In order to find a good layout, we first need some data: the size of the building, some information on the functional areas and what the people who will use the building want to do in it. In this work, we only work on one floor of the building. This is purely 2d but it is possible to add more floors later on as described in sec. 5.

About the functional areas we first define what kinds are available. In this thesis there will only be work, meeting and leisure areas, but more could be added later. Additionally there is one outside area with one entrance into the building.
From the workers we need to know, what they will be doing during the day. In other words, we need their plans describing their activities (work, meet, drink a coffee, ...) and the times from when to when they follow which activity and where.

We assume these plans are fixed and given. They will not be optimized by our program and so the building can only be as good as the plans. To optimize them would be outside the scope of this work and belongs to the field of operations research.

A plan for one worker would look like this:

<table>
<thead>
<tr>
<th>Activity</th>
<th>From</th>
<th>To</th>
<th>At</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outside</td>
<td>00:00</td>
<td>08:00</td>
<td>home</td>
</tr>
<tr>
<td>Work</td>
<td>08:00</td>
<td>10:00</td>
<td>my office</td>
</tr>
<tr>
<td>Coffee</td>
<td>10:00</td>
<td>10:30</td>
<td>the coffee area</td>
</tr>
<tr>
<td>Work</td>
<td>10:30</td>
<td>12:30</td>
<td>my office</td>
</tr>
<tr>
<td>Outside</td>
<td>12:30</td>
<td>13:30</td>
<td>restaurant</td>
</tr>
<tr>
<td>Work</td>
<td>13:30</td>
<td>14:00</td>
<td>my office</td>
</tr>
<tr>
<td>Meet</td>
<td>14:00</td>
<td>15:30</td>
<td>meeting room 1</td>
</tr>
<tr>
<td>Work</td>
<td>15:30</td>
<td>17:00</td>
<td>my office</td>
</tr>
<tr>
<td>Outside</td>
<td>17:00</td>
<td>24:00</td>
<td>home</td>
</tr>
</tbody>
</table>

We assume that all workers start and end their day outside the office building.

2 First Algorithm: Spreading Qooms

2.1 Previous Work

This algorithm was inspired by the project KaisersRot [9] and the traffic-simulation project described in sec. 1.1.1.

2.1.1 KaisersRot

In the KaisersRot project, the goal was to design a village, which should be similar to a naturally “grown” one. The algorithm tries to position lots belonging to buildings inside an area of land. For this the steps in the following list are done. Each step is implemented as an infinite iterative loop, so to proceed from one step to the next, this needs an interactive decision by the user. For a visualization of these steps, see fig. 1.

1. The buildings with their lots are put into the area and get some parameters like size, attraction to other buildings, etc, which they should try to fulfill as well as possible. Attraction can for example be that some houses want to be near the church, while others hate the church bells and so want to be away from it.

2. The buildings try to defend their lot size by applying repulsive forces to all neighboring buildings that lie too close, causing them to spread apart. At this stage the lots are circular. They try to push away buildings with overlapping lots, but also to move closer to certain buildings by applying attractive forces to those buildings.

3. The area is divided into polygons with edges between the circles. For this a triangulation between the buildings is used, which gives the neighbors of every building. From that the lot-polygons are calculated. In the resulting polygon-mesh three polygons touch each other at every intersection point.

4. Now the buildings once more try to adjust their lot size, this time by moving the corners of their polygons. The polygon-centers (where the buildings are) are fixed at this stage.

5. Streets are added along the polygon edges and grown to their correct sizes.
2.2 Our Algorithm

Our algorithm combines the KaisersRot-algorithm with the framework of the traffic simulation. It was decided, that for office buildings it is enough to work with circles or ellipses, as the methods to put in walls are part of a different work at a later design step. Thus, we only need the second step of the KaisersRot-algorithm doing the spreading out, replacing lots with qooms. In fact, because the calculations for ellipses would be much more complex we decided to only use circles at the moment. We also replaced the attraction of qooms with strategies and scoring functions.

This gives us a simple agent database like in the traffic simulation where our qoom-agents are stored with their strategies which consist of an initial position in the building. Also, we let 10% of the qoom-agents decide on a new strategy between each iteration step. From the activities of the workers we get the needed qooms with types and IDs. All these qooms are added in the beginning (like in step 1 of KaisersRot).

As already mentioned in sec. 1.2, in this algorithm we assume that the worker’s plans are optimized, and so all qooms listed in the plans-file are needed. If two workers have the same qoom-ID, this means they want to go to the same qoom, for example to a meeting. Not trying to optimize the plans means, that we do not try to put qooms together, e.g. if one is only used in the morning, while the other is only used in the afternoon.

In addition a qoom has a radius which is given by the minimum size defined for its function area. At the moment we use these radii directly, but later on we could add that the qooms also try to grow.

So our algorithm looks like this:

\[
\text{for } \text{nrSteps do} \\
\text{select strategies} \\
\text{spread-out qooms} \\
\text{evaluate qooms} \\
\text{end for}
\]

where the qooms first select their initial position (see sec. 2.2.1), then spread out over the building, which is described in sec. 2.2.2, and in the end are evaluated and the strategy is scored, which is described in sec. 2.2.3

2.2.1 Strategies

The qooms have strategies, which at the moment are only their initial positions. In later works, this could be extended, e.g. to include the size of the qoom as well. So a strategy consists of an initial position for this qoom and a score to it. The score says how good this strategy was up to now. It is normally a negative number, since it mainly consists of a penalty due to the distance the workers walked to this qoom. The qooms want to maximize their score, therefore a better strategy has a higher (or less negative) score. How the score is calculated is shown in sec. 2.2.3.

Each qoom stores a fixed amount of strategies, which for our tests was set to 5.

In the beginning of every timestep, a qoom selects its strategy to be used. To do this it first decides, if it should use one of its already known strategies or generate a new one. See fig. 2 for the pseudocode of this.

\textbf{Use an existing strategy: } If the qoom decides to use an existing strategy, it selects one of them randomly, with higher selection probabilities on the better strategies.

Since our scores are values in \([-\infty, \infty]\) and the worse score are towards \(-\infty\), we need to normalize them first to get probabilities from them. For the ith score \(S_i\) this is done with:

\[
u_i = e^{\beta S_i} \tag{1}\]
for all qoom in qoomList do
    if decide to reuse strategy then
        randomly select strategy
    else
        { decided to create a new strategy }
        create a new strategy
    end if
end for

Figure 2: strategy selection algorithm

From (1) the probability \( p \) of the \( i \)th strategy is calculated by

\[
p_i = \frac{u_i}{\sum_j u_j}
\]  

(2)

The factor \( \beta \) in this formula sets how strongly we weight the score, if \( \beta = \infty \) the best score is always selected, while if \( \beta = 0 \) all scores have the same probability. So \( \beta \) sets the level of exploring versus exploiting good results.

For results with different \( \beta \)'s see sec. 2.4.3.

Create a new strategy: A qoom decides to create a new strategy with a certain probability. If it does so the qoom generates a new initial position randomly within the building. For this we pull a uniform distributed value for \( x \) from \([0..size_x]\) and for \( y \) from \([0..size_y]\).

Later on, these values could also be influenced by the already existing strategies and their scores.

The new strategy is then tried out and evaluated (see sec. 2.2.3). If it is worse (i.e. has a lower score) than all existing strategies, it’s discarded. If not, it replaces the strategy with the worst score (which in return is discarded).

2.2.2 Spread Qooms

This is the main part of this algorithm since it determines the effectual positions of the qooms. To spread the qooms out, we let them push each other away, and repeat this, until no qoom moved more than a certain threshold value or until we have done more than a fixed maximum number of steps, in case the solution did not stabilize. This method is shown in fig. 3

while something moved do
    { calculate forces }
    for all qoom in qoomList do
        Calculate forces from walls and special objects
        for all neighbor in qoomList/qoom do
            Calculate force from qoom onto neighbor
        end for
    end for
    { apply forces }
    for all qoom in qoomList do
        apply force to velocity and position
    end for
end while

Figure 3: Algorithm to spread-out the qooms

Calculation of the forces: If the two qooms do not overlap there is no force between the two qooms. If they are too near (i.e. they do overlap) they repulse each other. For that qoom \( q_i \) applies the following force onto qoom \( q_j \):

---

For results with different \( \beta \)'s see sec. 2.4.3.
\[ f_{q_i,q_j} = \frac{1}{2} \left( rad_{q_i} + rad_{q_j} - \text{dist}(q_i, q_j) \right) \cdot \mathbf{u}_{q_i,q_j} \]  

(3)

where \( \text{dist}(q_i, q_j) \) is the distance between the two qooms, \( rad_{q_i} \) is the desired radius of the qoom i, and \( \mathbf{u}_{q_i,q_j} \) is the unit vector from qoom i to qoom j.

The factor \( \frac{1}{2} \) was added to ensure that the sum of the forces from \( q_i \) onto \( q_j \) and from \( q_j \) onto \( q_i \) would push the qooms just far enough apart so that they no more overlap, but no further.

If the centers of two qooms are at the same position, the above method does not work since \( \mathbf{u}_{q_i,q_j} \) is not defined. In this case they push each other with a small random force so that in the next step they can apply the forces as defined in (3).

If a qoom is partially or completely outside the building, the outer walls apply a force onto it. Since the qoom cannot push the walls away, the full force is applied to it. The force pushes the qoom inside, with a force equal to

\[ f = (rad_{q_i} - \text{dist}(q_i, \text{wall})) \]  

(4)

where \( \text{dist}(q_i, \text{wall}) \) is the distance of the qoom center to the wall. It is positive, if the center is inside the building, and negative if it is outside.

There is also the possibility to add additional areas, which push with the same kind of force. This could for example be a hallway.

**Calculation of velocities and new positions:** To now calculate the new positions, velocities and accelerations we use the Euler-Cromer method with the starting velocities set to 0.

Given the force from the neighbors and the walls, and that the mass of each qoom is 1, we can calculate the acceleration as:

\[ \mathbf{a}(t + \delta t) = -\frac{1}{\tau} \mathbf{v}(t) + \int_{t_0}^{t} \mathbf{f} (r(t), \mathbf{v}(t), t) \, dt \]  

(5)

where the term \(-\frac{1}{\tau} \mathbf{v}(t)\) stands for the deceleration done by the qoom, because it wants to stand still in the end. For all the tests \( \tau = 2.0 \) was used.

These forces are similar to those in a system of springs. The two masses have springs attached to themselves, and if they are too near to each other, the springs touch and push the qooms away. This is the force \( \int_{t_0}^{t} \mathbf{f} (r(t), \mathbf{v}(t), t) \, dt \). At the same time we have a frictional force that slows the qooms down again, this is the deceleration \(-\frac{1}{\tau} \mathbf{v}(t)\).

Like in the physical version the qooms continue moving even after the force from another qoom has stopped, until the friction has decreased to a halt.

Having the acceleration we can calculate the speed

\[ \mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t \mathbf{a}(t + \delta t) \]  

(6)

and then the position

\[ \mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \Delta t \mathbf{v}(t + \Delta t) \]  

(7)

Inside the program \( \Delta t \) was set to 1.

**2.2.3 Evaluation**

As mentioned in sec. 2.2.1 each qoom tries out one of its strategies at a certain iteration step. After the final positions have been calculated, the qooms are evaluated and get a new score for the strategy they used.

For this, every qoom calculates some sub-scores during the evaluation phase. At the moment the following scores are implemented:

- \( S_{\text{walk}} \): a score from the workers walking to and from it
- \( S_{\text{move}} \): a score for how far away from its initial position it was moved during spreading
- $S_{\text{overlap}}$: a score for how much overlap it has with its neighbors after the pushing. This kind of overlaps happen either because there is not enough space or because we stopped the spreading too early.

- $S_{\text{facility}}$: a score from special facilities, which can be something positive (like a qoom being near a window) giving a positive score, or something negative (like a qoom being beside the toilet) giving a negative score (or a penalty)

- $S_{\text{out}}$: a score for being outside the building. The qooms always start inside the building but it can happen that they get pushed outside.

The calculation for these is shown in fig. 4. Initial and the final positions refer to the centers of the qooms.

```
{ calculate $S_{\text{walk}}$ }
for all worker in workerList do
  for all activities in worker.actList do
    walkCost = distance(oldQoom, newQoom)
    oldQoom.walk += walkCost/2
    newQoom.walk += walkCost/2
  end for
end for
for all qoom in qoomList do
  { calculate $S_{\text{move}}$ }
  qoom.move = distance(qoom.finalPosition, qoom.initialPosition)
  { calculate $S_{\text{overlap}}$ }
  for all otherqoom in (qoomList - qoom) do
    dist = distance(qoom.finalPosition, otherqoom.finalPosition)
    wantedDist = qoom.rad + otherqoom.rad
    if dist < wantedDist then
      qoom.overlap = (wantedDist - dist) / wantedDist
    end if
  end for
  { calculate $S_{\text{facility}}$ }
  for all facilityArea in facilityList do
    if qoom overlaps facilityArea then
      qoom.facility += 1 {this is scaled, if not complete overlap}
    end if
  end for
  { calculate $S_{\text{out}}$ }
  if qoom.finalPosition is outside building then
    qoom.out = 1
  end if
end for
```

Figure 4: Algorithm to evaluate qooms

These scores are then weighted by factors and summed up and thus give the complete score for this iteration:

$$S_{\text{new}} = F_{\text{walk}} \cdot S_{\text{walk}} + F_{\text{move}} \cdot S_{\text{move}} + F_{\text{overlap}} \cdot S_{\text{overlap}} + F_{\text{facility}} \cdot S_{\text{facility}} + F_{\text{out}} \cdot S_{\text{out}}$$

Note that at the moment $S_{\text{walk}}$ is the only component of the score that is directly related to the workers’ happiness and $S_{\text{facility}}$ may be influenced by workers’ wishes. The other scores are only related to the qooms own happiness.

The score $SS$ of the used strategy $j$ is then adapted by averaging with the previous value of the score ($SS_{j,\text{old}}$) to stabilize it.

$$SS_{j,\text{new}} = (1 - \alpha) \cdot SS_{j,\text{old}} + \alpha \cdot S_{\text{new}}$$

This is better than simply replacing the score, since a strategy can once be worse, because a neighbor tried out something new.
2.3 Nash Equilibrium vs. Global Maximum

In an algorithm trying to find a global maximum, every qoom would try to position itself in the best way for the building, not caring if it itself gets a bad position. But in our algorithm, the qooms try to optimize their own scores, in order to make their users (a subset of all workers) as happy as possible, while they do not take the other qooms into consideration. As the best solution for one qoom is often not the best solution for the whole system, this does not necessarily lead to a global maximum.

So we do not try to reach a global maximum and it can even happen, that the global score of the building gets worse again with later iterations, because some qooms find better solutions for themselves. This will eventually lead to a Nash Equilibrium. In this state, no qoom is able to improve on its own. While it might be possible, that two or more qooms changing at the same time could get better results.

For more on Nash Equilibria and evolutionary game theory see [12]

On the other hand, as already stated in the introduction, if we want to optimize a global solution, someone has to first provide a function that gives us a global score. This is not trivial, as people tend to want to optimize their own happiness not the happiness of everyone.

In section 2.4.4 we tried out a global score by just averaging all individual scores. As shown there, this does not get better than optimizing the individual scores.

2.4 Results

This algorithm was tested with the scoring functions mentioned in sec. 2.2.3. The test was run with a one-story building of size 70 x 21 meters having a hallway in the middle, with an entrance at one end.

For the workers using the building we used a randomly generated plan-file for 75 persons using 100 different qooms.

The workers have plans like the one described in sec. 1.2. The algorithm produces an output like the one in fig. 5. The qooms are shown as circles with their ID’s. Note the hallway in the middle, which pushes qooms away.

For some tests described below, we added an area along the north wall of the building, in which work-qooms get a bonus. This could for example be a window on the not so hot side of the building. This area is a rectangle along the whole width and one third of the height deep. Non-office-qooms (i.e. meeting- and leisure-qooms) do not get any bonus (or punishment) in this area.

In all tests the qooms have 5 strategies. If not stated otherwise, the selection temperature $\beta$ was set to 0.5, the probability to create a new strategy was 0.1 and the factor $\alpha$ to weight the previous score was 0.1.

In the following sections we will show plots with different settings. These plots show the average walking distance to and from a qoom during a day since it is easier to understand than the scores and since the scores differ strongly depending on the scoring factors. Also with our settings, the walking distance is the main reason for our workers to be unhappy, as opposed to qooms.

With the distance plots the worse results have higher walking distances, and the target is to minimize this. As noted in sec. 2.2.1 and 2.2.3 this distance is added to the score as a negative value punishing strategies by multiplying it with $F_{walk}$.

2.4.1 Single Scoring Functions

First a set of tests were run using only one of the scoring functions (e.g. $S_{out}$, $S_{overlap}$, etc.) from formula (8) in sec. 2.2.3. The goal was to see what impact they have on the walk distance and thus on the walk score. So we ran the program with only one scoring function activated for the strategy selection but output the distances walked by the workers to compare their impact. For this we turn all the factors ($F_{out}$, ...) to 0 except for one.

If the scoring is used, then the factors are set to the following values:
Figure 5: Typical output from the spreading qoom algorithm. The qooms are shown as circles with their center and their qoom-ID.

![Typical output from the spreading qoom algorithm. The qooms are shown as circles with their center and their qoom-ID.](image)

Figure 6: average walking distance vs. iteration step, for different scoring functions

<table>
<thead>
<tr>
<th>Scoring</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{walk}$</td>
<td>$F_{walk} = -1.0$</td>
</tr>
<tr>
<td>$S_{move}$</td>
<td>$F_{move} = -2.0$</td>
</tr>
<tr>
<td>$S_{overlap}$</td>
<td>$F_{overlap} = -25.0$</td>
</tr>
<tr>
<td>$S_{facility}$</td>
<td>$F_{facility} = -200.0$</td>
</tr>
<tr>
<td>$S_{out}$</td>
<td>$F_{out} = -200.0$</td>
</tr>
</tbody>
</table>

These values are also used for the rest of the tests, except for the score from the facility area, which was only used where specifically noted.

These tests gave the following results for the total-walking distance on the 10’000th iteration:

<table>
<thead>
<tr>
<th>applied scoring</th>
<th>walking distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>walk score</td>
<td>31’236m</td>
</tr>
<tr>
<td>move score</td>
<td>37’039m</td>
</tr>
<tr>
<td>outside building</td>
<td>33’307m</td>
</tr>
<tr>
<td>overlap score</td>
<td>35’316m</td>
</tr>
<tr>
<td>facility score</td>
<td>35’077m</td>
</tr>
</tbody>
</table>

In figures 6 and 7 we plot the average walking distance to and from a qoom versus the iteration steps for the different scoring functions.

12
The first result of this is, that the walking distance is optimized when its scoring is activated, as expected, but the results are far from stable. This is so because one qoom changing its strategy potentially changes the score for all other qooms, as all qooms end up in different positions after the spreading. This instability can be reduced by changing the factor that weights old scores versus new ones, which we show in sec. 2.4.3 and fig. 14.

We can also see, that the different scoring functions have different influences on the walking distance. So the walking score is obviously reduced, if the qoom is punished for being outside the building. On the other hand, if the qooms have to stick to their initial position, the walking distance is influenced negatively. How this will change with combinations of the scoring functions, we will see in sec. 2.4.2.
2.4.2 Multiple Scoring Functions

Figure 8 shows results of combining either the penalty for being pushed far or the penalty for overlap with the one for the walking. This gives a slightly better result than only the walking. This probably happens, because now the score says to what extend the qoom was following the strategy, and how far the people had to walk.

Figure 9 shows the result of mixing more scores together. As one can see from the figure, the combination of scores does not improve the walking-distance more than using only the walking score.

In figure 10 we plot the scores for the run using all scores. With these factors the walking score is the most influential one. As we progress, more work-qooms move into the facility-area, and thus the corresponding score increases. But this and the urge to improve the walking also lead to more pushing, and so the push-score gets worse.

The score at the end of the plot at timestep 2000 is out of competition, since in the end of the run we apply the best strategies of every qoom (no new strategies generated) in order to produce the final layout.

Note that in fig. 10 the average scores are plotted. The qooms try to maximize their scores, contrary to other plots, where we plotted the walking distance that should be minimized.
Figure 9: average walking distance vs. iteration step, for different mixes of scoring functions

Figure 10: the sub-scores vs. iteration step when using all scoring functions
2.4.3 Strategy Selection Factors

Selection Temperature  As described in section 2.2.1, we select a strategy for each qoom with a certain probability relative to $e^{\beta \cdot \text{score}}$.

We did tests with different $\beta$s and plotted the average distance that the workers had to walk although we used all scoring functions for the tests.

In fig. 11 the average walking distance of all workers during one day is plotted versus the iteration steps. As one can see, the score is improved in the beginning but then stays more or less at the same level. This is done with all scoring functions activated as described in sec. 2.4.1.

The value used there was also set to 0.5, as in fig. 11.

In figure 12 we tried the same with different values for $\beta$. There does not seem to be any significant change in the behavior. This is so, because the strategies of one worker all have similar scores after a few iteration steps. This means that the scores differ at maximum by 10%. (Strategies from different workers may have stronger differences though.) If the strategies have similar scores, they are all selected with about the same probability, no matter what the $\beta$ value is.

Assuming that the scores are negative and knowing that our scores lie between -130 and -350, larger values for $\beta$ do not work either, because of a problem with double values: because the smallest storable number in IEEE-double is 5e-324, the smallest values for the normalization is around $e^{-744}$. So if we select a larger $\beta$ than about 2, we get that all strategies have a normalized value of 0, and so always the first strategy is selected.

A way around this would have been to select the best strategy, if $\sum e^{\beta \cdot \text{score}_i} = 0$ and in all other cases to use machine epsilon when $e^{\beta \cdot \text{score}_i}$ = 0. This has not been implemented though.

This problem could disappear for different plans, but this was not tested.

Probability to Select a New Strategy  We described above how changing the probabilities of the already existing strategies should change the outcome of the result, but that it actually has almost no influence on the result.

Now we want to see how the probability to generate a new strategy changes the result. This is the probability for the decision if we should reuse an existing strategy or generate a new one, in sec. 2.2.1.
If we set this value to too low values, the qooms hardly ever try out something new, and as the new things often will be worse than the old strategy, they probably stick to their old strategies. But on the other hand, when this probability is too high, the qooms do not learn how good and stable a solution is, because some qooms constantly try out new things.

In fig. 13 one can see how the average walked distance changes with different probabilities for a new strategy. The optimum value seems to be around 0.1. Also note, that with \( p = 0.001 \) the fluctuations in the walking distance are very small, while the two higher values change more. This is so because with \( p = 0.001 \) there are almost no new strategies tried out. This was also shown for traffic problems in [19].

**Factor of Previous Score** One more factor in this area is how strongly we weight a new score. This is described in formula (9) in sec. 2.2.3.

With five strategies, this does not seem to have any influence though, as fig. 14 shows. This is probably the case, because the strategies are replaced too fast, and so they do not really have any old values.
Figure 13: average walking distance vs. iteration step for different probabilities to create a new strategy

Figure 14: average walking distance vs. iteration step for different values of $\alpha$
2.4.4 Global Score vs. Individual Score

As described in 2.3 every qoom-agent gets its own score. In this test we wanted to see what would happen, if we gave every agent the global score instead of its local one.

For this we added up the scores for all qooms during the evaluation phase (see 2.2.3) and in the end gave average score (=sum divided by number of qooms) back to the qooms, who updated it like the normal score and then stored it in their strategies. So now the qoom-agents store both the global and the normal score with each of their strategies. During the selection phase they mix the local and the global score, so for the normalized score in sec. 2.2.1 we use instead of (1)

\[ u_t = e^{\beta \left( f_{\text{global}} \cdot S_{\text{local}} + (1 - f_{\text{global}}) \cdot S_{\text{local}}\right)} \]  

which is afterwards used for the calculation of the selection probability (see formula (2)).

From figure 15 and figure 16 one can see that using a global score only changes things very little in the outcome. Global fact = 0 means we only use the local score (as up to now), while global fact = 1 means that we only use the global score and ignore the local one.

2.5 Without Spreading

We also ran one test where we turned the spreading off. This means that the strategies also have to find out, where there are many overlaps and try to find good positions on their own.

In figure 17 again the average walking distance is plotted. As one can see the system learns similarly to the system with spreading in the beginning, but then the learning becomes slower. This is also the case, because now another qoom trying something else has even stronger impact on our strategies, since on the one hand the walking distance can decrease strongly, but on the other hand the overlap score (which had to be strongly increased for this to work) gets a lot worse, when the two qooms just occupy the same space.
Figure 16: average walk score vs. iteration step using global scores

Figure 17: average walk score vs. iteration step with only a few spreading steps
3 Second Algorithm: Dynamic Qooms

3.1 Idea

In this algorithm the workers themselves create their individual spaces. Thus the qooms are now created during the day on the contrary to the first algorithm, where we created all qooms at the beginning.

Therefore a worker coming into the building or changing to another activity first checks if the qoom he wants to go to exists already. If it does not, he has to create this qoom before he is able to use it.

In order for a worker to create a qoom, he needs to know how big a qoom he needs and where there is space free for it. For this our building is divided into a grid of bars. A bar is a "Building Atomic Range," meaning it is an area of space that cannot be subdivided. A qoom is built up from several of these bars.

The size of those bars has a practical meaning in architecture: because of the width of the windows and the spacing of columns between the parking lots in the basement, a wall in a building can only be on a grid of about 1.35 meters in one direction. In the other direction there is the constraint, that a real room (not qoom) cannot be deeper than 7.5 meters (else there is not enough light) and there should not be more than two functional areas (=our qooms) in this direction.

So we get a grid of 1.35m x 3.75 m bars on which to place the qooms. This grid also means, that in this algorithm we will use rectangular qooms and no more circles (opposite to sec. 2). A qoom consists of one or more of these bars.

This has one more big difference to the spreading qooms algorithm from sec. 2, because we no more necessarily assume that the qoom usage is optimized in the plans. This algorithm tries to find a stable solution to use bars, but this may include changing the qoom-ID or even the qoom-type during the day. This means, that a qoom can be an office in the morning and a meeting space in the afternoon. Such problems are left to the architect to decide if they make sense. Some of them can also be discouraged by setting some parameters differently as will be described in sec. 3.2.2.

3.2 Algorithm

Thus the algorithm described in sec. 3.1 looks like the following:

\begin{verbatim}
for number of iteration steps do
  repeat
    select next worker { see 3.2.1 }
    (nextWorker.oldQoom).nrWorkers--
    find qoom-location { see 3.2.3 }
    (nextWorker.newQoom).nrWorkers++ { go there }
  until all workers done
end for
\end{verbatim}

3.2.1 Next Worker Selection

We need to process the workers in the order of their activity changes. For this we use an event based list, where a worker changing his next activity is an event with a starttime. This event list is ordered by the times at which the workers change their activities. At an event the corresponding worker is processed and in the end a new event is generated with the time when the worker ends his activity. This new event is then put back into the event list. The activities and their times are given by the worker’s plan, as described in sec. 1.2. The pseudocode for this is shown in fig. 18.

Inserting workers into this list means we put them between the last worker with a smaller event time and the first worker with a larger event time. But if there are several workers with the same event time, we need to decide on their ordering in the event list. This happens if several workers want change their activity at the same time. In the current implementation, we always add workers with the same starting time as another worker already in the list after this worker/these workers. We could also add them in a more random order,
while eventList not empty do
   event = eventList.popFront
   worker = event.worker
   process worker
   if worker has next time (according to plan) then
      new event at worker.nextTime
   end if
end while

Figure 18: event list handling to process workers

but this was not tested. That method is described in appendix A.1.1

3.2.2 Qoom Costs

The general idea of the costs is to make the workers build qooms in good locations and let the building slowly grow more stable, so that in the end all qooms stay around all day.

In this algorithm we use costs instead of scores. It costs the worker to build a qoom or to break one down. This is different from the scores used in sec. 2.2.1 since the workers do not get feedback later on, how good their decisions were, but they have to pay in advance a fee for things that are expected by the architect to be not so good choices for them or for other workers.

In section 3.2.3 we will describe how a worker creates a qoom. Here we discuss the costs for the different parts of this. The following list contains the possible costs a worker has to pay. The total fee for a specific qoom is the sum of all applied costs.

On the other hand, this list only names the costs. The exact values are to be set by the user since he is the one to decide what is important and what not. He could for example allow (forbid) workers to reuse the office of other workers by lowering (raising) the price to reuse a qoom. If this price is too high, the worker either has to destroy the office and build a new one there, or find another location for his office.

- **Cost to walk to the qoom**: So that the workers decide on qooms near their previous work location, they have to pay for walking. Another meaning of this is that their walking time is lost work time and time costs.

- **Cost to use an existing qoom**: To use a qoom that exists already, i.e. that has the right type and qoom-ID has a low cost for the worker.

- **Cost to re-use an unused qoom**: If the worker wants to use a qoom with the right type but a different id, he has to pay a fee for this since afterwards the original worker who built this qoom needs to change something else again. Reusing qooms also goes against the original idea, that the plans are optimized for qoom usage.

- **Cost to build a qoom**: To build a qoom on empty space costs a small fee proportional to the size of the qoom.

- **Cost to build a qoom with empty bars around**: If we want to force the system into a tighter setup, we can put in a cost per empty bar adjacent to the new qoom. This means that it is cheaper to build next to an existing qoom than to build in a free space.

- **Cost to delete an unused qoom**: To delete a qoom with no workers inside it, should be costly, since it is quite unrealistic and at least in later iterations we do not want this to happen anymore. This cost and how we can raise it is discussed in detail in sec. 3.2.6. To delete a used qoom is not possible, unless the workers are first kicked-out.

- **Cost to cut a qoom**: Since a worker can decide to build a bigger qoom than he needs, the moment another worker needs the space he can decide to cut the too large qoom to take some space away from it. This costs depending on how many bars the cut-qoom has left afterwards but it is cheaper than the cost to delete the qoom. This is described in detail in sec. 3.2.4.
• **Cost to kick out another worker:** If there is absolutely no other option, a worker can also decide to kick out another worker. This is described in sec. 3.2.5 and is clearly very expensive since it disrupts other people’s work. This is only the cost to kick the other one out, the cost to destroy the qoom is extra.

• **Size bonus:** Beside these costs there is also one bonus a worker can get: if he takes a qoom larger than the minimum size he would need, he gets a bonus (i.e. the cost is reduced since he is very happy to have so much space). This is needed so that the workers consider this possibility at all (since as we see in 3.2.3 we always pick the cheapest option).

• **Cost for no qoom:** If a worker cannot find any qoom, he gets a heavy penalty and is forced to wait outside the building for his next activity change. It is possible to let the workers have this as an option (meaning they also consider the penalty beside all the qooms) in 3.2.3. This does not make a difference, since as we stated in the introduction we assume that we always have enough space for all qooms.

More cost and bonus functions could be added easily. For example we could add a bonus for special qoom-types in certain areas, like for example offices near the outer walls, which corresponds to the facility area of the first algorithm.

Another thing that could be added later is to let the workers go on paying qooms, after they leave them, so that it becomes more expensive to delete or reuse this qooms. This would make sense for example for offices, which the workers want to keep, even if they are not there. This is not implemented, and it could (at least partially) also be achieved by raising the price to reuse a qoom.

### 3.2.3 Select next qoom location

After the next worker has been selected and he has left his previous qoom, he needs to find out where to go. In his plan it says, that he should start to work now in a qoom of a specific type and a given ID (which is mainly needed to go to the right meeting).

If several people want to go to the same qoom, the first one to go there decides on the placing. It could also be specified in the plans file, who decides, but at the moment it’s just the first one.

The worker tries to find a cheap qoom to go to, since using a qoom costs. The different costs have been described in 3.2.2.

To find his qoom he first checks if it already exists and whether it has people inside. This would mean that he has to go to a meeting with other workers, and someone else has already decided where to put the qoom. In this case he just goes there.

If the qoom exists, but is empty, the worker found the qoom of e.g. last day (meaning the last iteration step). This is possible, since the building does not change over night. He then checks, if he finds a cheaper solution. Also if the qoom does not yet exist, the worker needs to find a location to put it at.

This is shown in figure 19.

```python
qoomId = worker.nextActivity.qoomId
if (∃ qoomId ∈ building.qoomList) and (building.qoomList.qoomId).nrWorkers > 0) then
    building.qoomList.qoomId.addWorker(worker)
else
    qoom = find new location for qoom
    building.qoomList.add(qoom)
    qoom.addWorker(worker)
end if
```

Figure 19: location selection algorithm

For the selection of a new qoom-location the worker has two possibilities: either he finds a qoom of the right type that is free at the moment and he just uses it, or he finds enough free space to create a new qoom of the right type.

In the first case, the worker changes the ID of the qoom and then uses it.
In the second case the worker has to first destroy old qooms if there are some in his way and then build his new qoom in the now empty space. But a worker can only destroy an old qoom, if no-one is in it at the moment.

An additional feature is that workers can decide to make their qooms larger than needed. If another worker now wants to use this space, he can also just cut away a part of the other qoom. This is not only cheaper than destroying it but also possible, even when the qoom is in use.

<table>
<thead>
<tr>
<th>needed</th>
<th>action</th>
<th>initial building</th>
<th>new building</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, work</td>
<td>use existing qoom</td>
<td><img src="image" alt="initial" /></td>
<td><img src="image" alt="new" /></td>
<td>0€</td>
</tr>
<tr>
<td>2, work</td>
<td>reuse free qoom</td>
<td><img src="image" alt="initial" /></td>
<td><img src="image" alt="new" /></td>
<td>1€</td>
</tr>
<tr>
<td>2, work</td>
<td>use free space</td>
<td><img src="image" alt="initial" /></td>
<td><img src="image" alt="new" /></td>
<td>4*25€ = 100€ (25€ per new bar)</td>
</tr>
<tr>
<td>2, work</td>
<td>cut old qoom(s)</td>
<td><img src="image" alt="initial" /></td>
<td><img src="image" alt="new" /></td>
<td>2<em>30€ + 4</em>25€ = 160€ (30€ cut cost, 25€ per bar)</td>
</tr>
<tr>
<td>2, work</td>
<td>destroy old qoom(s)</td>
<td><img src="image" alt="initial" /></td>
<td><img src="image" alt="new" /></td>
<td>200€ + 4*25€ = 300€ (200€ for deletion, 25€ per new bar)</td>
</tr>
</tbody>
</table>

Figure 20: Table of the different possible actions to get a qoom

The different options a worker has to create a qoom are shown in figure 20.

To decide where to put his qoom, a worker has two pairs of options: he could either enumerate all possibilities, or sample n and select from those. From the set of possibilities he then can either select the best option, or he can select them according to some probabilities set by how good the location would be. This is done similarly to the strategy selection method described in formulas (1) and (2) in sec. 2.2.1.

How good an option is is determined by the cost of the qoom. The lower the cost, the better the option. This differs to the strategies we had in sec. 2, where the better strategy was the one with the higher score.

In this version, only the method to select the best qoom (i.e. the cheapest one) is implemented. The other options are described in appendix A.1.2.

The algorithm to select the best option is shown in figure 21. The list of available locations is created by going through all bars and checking if it is possible (and how much it costs) to build this qoom there.

```java
mincost = ∞
selpos = NULL
for all location in availableLocations do
  if location.cost < mincost then
    selpos = location
    mincost = location.cost
  end if
end for
```

Figure 21: Select cheapest qoom location
3.2.4 Qoom Cutting

Cutting away parts of a qoom is only possible if the qoom is larger than its minimal size, and it can only be cut so much that it still has at least its minimal size after the cut.

The cost of cutting a qoom depends on the number of extra bars the qoom had before ($N_E$) and the number of bars to be cut away ($N_C$).

The cost function for cutting is:

$$C_{cut} = base\text{CutPrice} \times \frac{N_C^2}{N_E + 1}$$

(11)

Because a qoom always has to be rectangular, we possibly need to cut away more bars than the new qoom really needs, as a cut in a qoom can only be one line. This is shown in fig. 22.

The worker creating the new qoom decides on the cutting direction, by picking the option that is cheaper for him.

3.2.5 Kick-Out

With just being able to build on unused spaces, we can run into a blocked situation where there would be a better solution, but because other workers are still using the space, we cannot get it. As an example for this consider the following test-case:

We have eleven workers, four of them come a bit earlier in the morning. All workers work in their offices in the morning, but in the afternoon, the four early arrivers go to a meeting.

With the present method, the workers put their offices in the morning near to the entrance and so when the first worker in the afternoon tries to get a meeting qoom, he can only put it at the end of the hallway.

To correct this by allowing the workers to kick other workers out of their qooms, and then use those to build something new. This should be quite expensive if the other workers want to still stay a long time, but it should be almost free if the kicked-out worker anyway leaves at the same step and is just a bit later in the event queue.

A worker that has been kicked out of his qoom has to go outside and his next event is handled as if he had no qoom for the previous activity (but he does not have to pay the penalty for this). This may have some negative effects on the distance optimization, since we have more workers coming in from outside, than there really are.

In the test case, the worker can now select a space for the meeting knowing that the other workers will also leave soon (and so using their offices to build the meeting qoom).

3.2.6 Stabilizing the Building

So far the only difference between two iteration steps was the initial layout in the morning, which is carried over from the previous day (or iteration step). This means, that if one solution is good in the morning and one in the afternoon we will constantly switch between them, as the prices are constant.

To solve this, we added the possibility to raise the price to destroy a qoom with subsequent days. This works by raising the qoom deconstruction cost during the night, when no one is in the building. This means
that the next day the workers will have to pay more, if they want to destroy a qoom. This price is at the moment a quadratic function of the iteration step. This is a function which looks like the following:

\[ C_{\text{delet}_t} = 250\€ + 0.5\€ \times t^2 + 5.0\€ \times t \]  

(12)

where \( t \) is incremented every night.

This is similar to simulated annealing, where the state space is explored by allowing the algorithm to select inferior solutions with a probability depending on a temperature. With later steps this temperature is lowered, until inferior solutions are never selected. This is described e.g. in [2]. The temperature from simulated annealing compares to the inverse of our \( t \).

So with later days it gets more expensive to tear down unused qooms. This forces the workers to either build new ones further in the back or to reuse the already existing qooms.

3.2.7 More Solutions

The method from sec. 3.2.6 stabilizes the building, but the solution it gives does not have the same quality everywhere. The qooms near the entrance are built early and have adapted and improved for a long time, while the qooms further in the back are only created, when the price to tear down qooms is already too high. This means, the back is just a wild ordering of qooms.

In order to fix this, we use different methods to alternately lower and raise the deconstruction-prices. Again this has an analogy in simulated annealing, where raising the temperature again to find more solutions is called an “annealing schedule”.

Our first method is, to tell all the workers, that it’s cheaper again to destroy qooms so the workers can make changes again. But this may just lead to changes in the front area if the price is reduced to far.

In the second method, we only want to let one specific worker make changes. For this we lower the deletion cost for only this worker and run all the workers again. This should lead to that one worker improving his day. Afterwards we reduce the costs a bit for all workers, so that the system can relax again. This does not work though if the selected worker does not take decisions, e.g. if he comes late to a meeting he is not able to decide on where to put the meeting-qoom.

An alternative to this is to make one run with only the selected worker by executing only his plan, so he has no competition and can change the location of any qoom he wants. But this has a problem too since with a high probability the other workers immediately build back his changes, when they get into the building on the next iteration step.

How to select such a worker is another question. We can either select the one with the worst score or we select one randomly (probably weighted with the score).

These methods mix the solution layout newly and often come up with a different solution. But the new solution is often not better, and it would be difficult to implement a method to let the system go back, if the previous solution was better (because we would have to store the locations of all the existing qooms at that time).

3.3 Improvements

With the workers only selecting the momentary best solution, there is almost no learning. The only bit of information kept is the current layout. But there is no learning if it was good or bad.

3.3.1 Salaries for Qooms

The current setup still has some problems, mainly with qooms used by several workers at the same time. The main concern is that if there are many people who want to use the same qoom, but only one decides where it will be, the others have no way to tell him if his choice was good.
But we do not want to add full scale learning since we could not come up with a meaningful way to do it. Two possible ways using q-learning are described in sec. 5.2.1 of the outlook.

We did not find a good way to add scoring to the workers, first because as they at the moment decide deterministically where to put the qooms, they cannot have strategies for that, and second because these scores would have to be updated by other workers using the qoom – possibly much later, maybe even the next day.

So we add salaries to the qooms, which will influence their destruction costs, since this is the step where a worker changes the decision of another worker. These salaries correspond to scores as they depend on how good the qoom is, but they are not scores, since the qoom cannot decide on anything, and so a qoom’s only option to change is, to decide to destroy itself. The decision to destroy itself is taken in the night. If the qoom does not have a positive salary, it decides it was too bad and disappears.

The salaries consist of several parts:

- a fixed payment from each worker entering the qoom
- a salary per timestep while a worker is inside the qoom
- a penalty to walk to the qoom
- a penalty to walk away from the qoom

This means, when a worker arrives at a qoom, he pays it the fixed entry cost minus an amount corresponding to his walking there (times a factor).

Beside this, we divide the day into time slices, and at the end of every slice, every qoom gets a salary depending on the number of workers inside it at that moment. When a worker leaves the qoom, the qoom pays him back depending on his walking cost to the next qoom.

The salary should not raise constantly, so at the same time, the qooms get paid for the usage, their salary is diminished with a fraction. This is similar to formula (9) in sec. 2.2.3, but in difference to there we only apply the fraction to the old salary. So the formula is:

\[
Sal_{q; t+\Delta t} = \alpha \cdot Sal_{q; t} + \left( \#\text{Workers}_{q; t+\Delta t} \cdot \text{UseCost/ sec } \cdot \Delta t \right)
\]  

(13)

This is integrated into the event based mechanism described in sec. 3.2.1 by adding special events at every timeslice (after all workers with this time have been processed). During those events the updates for all qooms are done.

For our tests, we set the length of the timeslices to five minutes, since this is also the granularity of our test plans.

To then apply these salaries to the cost for the qooms, the salary is added to the qoom deletion cost.

### 3.4 Results

#### 3.4.1 Graphical Output

In figure 23 we show a typical output from the dynamic qooms algorithm. It contains in the upper part the layout at this moment with numbers denoting how many days the qoom has already been there, and in the lower part a plot of the cumulative usage of each qoom type at each bar. This plot is created by summing up at every time slice which qoom type was at this location but only if it was used. Then we divide each sum by the maximum possible number and assign each value to the appropriate color channel of the image (see figure caption for color explanations).

The entrance to the building is on the left side in the middle. Note that there is no hallway in this version. The white space in the middle is separating the two plots.

From the lower part the architect can decide, in which area he needs to put a certain qoom type, while the upper part gives him a possible layout.

From the text output one can also retrieve the exact qoom-IDs:
3.4.2 Cost Factors

In section 3.2.2 we saw the different parts of the qoom-costs. Now we want to see, how different values for these influence the outcome. For that we compare the average walking distance (per worker) and the number of qooms, that were deleted during a day.

The goal is that after a certain time, no qooms are deleted anymore and the walking distance should be minimized.

All tests are compared with the same run. In that run we set the cost factors in the following way:

- The cost for walking is 30€ per meter walked.
- The cost to leave a neighbor bar free is 25€ per bar.
- To destroy a qoom costs 250€.
- To reuse a qoom costs 200€.
- To kick out a worker costs 0.1€ per second he still wants to be in this qoom.

Note the difference of these plots to the plots in sec. 2.4: There we plotted the average walking distance to and from a qoom, while here we plot the average walking distance of a worker per day. Since the sum of the walking distances to and from a qoom (actually the halved walking distances to and from a qoom) is the same as the sum of the walking distances per worker, the only difference between these two values is, that in sec. 2.4 we divided by the number of qooms, while here we divide by the number of workers.

In figure 24 we can see the results with different costs for walking. As one can see, if it is more expensive to walk, the workers try to put their next qooms nearer, but thus they also delete more qooms.

In figure 25 we can see how the walking distance changes with higher costs to delete a qoom. We can see that with a very high cost, the building eventually stabilizes at a solution, though one with a high walking distance. While with lower values again the walking distance becomes better, but there are more qooms deleted.

In figure 26 we can see that raising the price to create a qoom with free bars around it, does some changes in the beginning, but then leads to the same solution.

In figure 27 we see how the cost to kick out other workers changes the result. This cost was described in 3.2.5. These plots show, that with a higher cost factor, the workers kick each other less often out, and thus have to walk more. But with a too low value, the walking distance becomes more chaotic, as the workers often find themselves without a qoom, when they want to go to their next workplace.
Figure 24: average walking distance vs. iteration step and number of deleted qooms vs. iteration step; for different cost factors for the walking

Figure 25: average walking distance and number of deleted qooms vs. iteration step, with different costs to delete a qoom

Figure 26: average walking distance and number of deleted qooms vs. iteration step, with different costs for leaving bars free
Figure 27: average walking distance and number of deleted qooms vs. iteration step, with different costs to kick out other workers
3.4.3 Qoom-Deletion Cost Function

In section 3.2.6 we described how to add a time-dependent cost function to the deletion cost. Here we now want to test, how different functions for this affect the results.

In figure 28 we used functions linear in the iteration step (denoted by $t$ in the plots). Again for comparison the result with a fixed deletion cost is plotted. As one can see, with a slowly growing function, the workers adapt better to the building, resulting in a lower average walking than with the quickly growing one. This also means, that the building stabilizes slower.

Now we do the same with a function quadratic in the iteration step. This is shown in figure 29. This shows, that even though this converges faster than the linear function shown in fig. 28, it goes to a better result.

And finally we made a run with two functions containing a quadratic and a linear term. This is shown in figure 30. This leads only to small changes compared to the two previous options.

3.4.4 Qoom-Deletion Cost Function with “Schedule”

For a final test with the methods from sec. 3.2, we tried to add schedules to our runs, as described in sec. 3.2.7. This means that we lowered the costs to delete qooms after every 300 iteration-steps of normal running.

For this we applied the methods mentioned in sec. 3.2.7. At step 300, 600, 900 and 1200 we reduced the price for all workers back to the initial values and let it grow again. At step 1500, 1800, 2100, 2400 and 2700 we set the price for one worker during one step back to the initial value, and then from that step afterwards set the prices for all workers back to the values of the 5th iteration step. At step 3000, 3300 and...
3600 finally we only let one worker into the building. Afterwards we again set the prices for all workers back to the values of the 5th iteration step.

As we saw in section 3.4.3, the solution stabilizes after some iterations if we use a growing deletion cost function. After the mixing steps the solution stabilizes in the same way but as fig. 31 shows they may stabilize at better or worse solutions. We can see, that our solutions sometimes are better, but there is no “learning” which configuration was good, and so the program always continues with the latest one. Also we see that the solution does not depend on which method to mix the solution up we used. The different layouts are written into the output file, and so the architect can retrieve the solution he liked best. Note that in figure 31 the basic plot is not shown. It would not make sense, since there is no increasing deletion cost there, and so it stays in the same area.
Figure 32: average walking distance and number of deleted qooms vs. iteration step, varying the decreasing factor \( \alpha \) of the qoom salary.

Figure 33: average walking distance and number of deleted qooms vs. iteration step, varying the salary factor for the walking to and from the qoom.

### 3.4.5 Qoom Salary Factors

As mentioned in section 3.3.1 we tried to improve the walking distances and the qoom-deletions by adding salaries to the qooms. Testing this showed, that the walking distances increase strongly, while the qoom deletion rates decrease. For this reason, the following plots of the average walking distance have a different range.

For all tests, the same default values as in the previous sections were used. For the salary functions the defaults were:

- the salary decreasing factor \( \alpha = 0.99 \)
- the factor to penalize the walking distance: \( F_{\text{way penalty}} = 5.0 \text{€/m} \)
- the use cost (or rent) per second: \( S_{\text{use}} = 0.5 \text{€/sec} \)
- the fixed cost when entering a qoom: \( S_{\text{enter}} = 500 \text{€} \).

In figure 32 we varied the factor \( \alpha \). Note that if the qooms can acquire enough salary, the solution becomes stable, even though there are still 30-50 qooms deleted every day. In this case the same qooms are deleted every day, which means that the space should be used for different functions during the day. This is a valid solution for our program, which then the architect has to figure out how to use. On the other hand if the salary is decreased too much, the solution does not stabilize.

In figure 33 the factor \( F_{\text{way penalty}} \) for the walking was changed. If this factor is too high, the qooms can no more accumulate salary, and so the system does not reach a stable solution.
Figure 34: average walking distance and number of deleted qooms vs. iteration step, varying the rent per second.

The amount of rent $S_{use}$ paid to the qooms per second by every worker is varied in fig. 34. With lower amounts for the rent, the solution stabilizes slower, but after some time the qooms acquired enough salary to stabilize. Then only some qooms are switched every day.

And finally we varied the amount $S_{enter}$ a worker gives a qoom when he enters it. This is plotted in fig. 35. Note that here even though the qooms get a higher salary the result is less stable. This is probably so, because the qooms now have too early a high salary and all qooms have a similar high value. This forces the workers to destroy even qooms with high values.

Only after many steps the version with the entry cost set to 1000€ gets stable. The solution with only 250€ does not get completely stable during the time of these runs.

Figure 35: average walking distance and number of deleted qooms vs. iteration step, varying the entry cost.
4 Comparison

In sections 2 and 3 we have described two different algorithms to find good layouts. From the setup, they are quite different, since in the first one the qooms decide on the space to use, while in the second one the workers decide this. Additionally there are the following differences:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>space division</th>
<th>qoom-forms</th>
<th>optimization in time</th>
<th>global optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spreading</td>
<td>continuous</td>
<td>only circles</td>
<td>over whole day</td>
<td>local, global possible</td>
</tr>
<tr>
<td>Dynamic</td>
<td>discrete</td>
<td>rectangular, free forms possible</td>
<td>only next action (local)</td>
<td>local, global hard</td>
</tr>
</tbody>
</table>

Comparing the run times of the two algorithms gives the following results:

The spreading qooms algorithm needed about 4.5 minutes for 500 iteration steps, using 100 qooms and 75 workers and allowing at most 1000 spread out steps (these were almost never all used). The dynamic qooms algorithm needed about 4.75 hours for 500 iteration steps, again using 100 qooms and 75 workers. Both tests were run on a dual CPU machine with two 1GHz Intel Pentium III Processors and 1GB RAM. Obviously the two results are not directly comparable, since as we saw in the results sections 2.4 and 3.4 the time until a solution stabilizes strongly depends on the settings. But generally the spreading qooms algorithm reaches its best solution in shorter time than the dynamic qooms algorithm.

Another big difference is that in the spreading qooms the qooms try to find a good solution for a group of workers (the workers using this qoom), while in the dynamic qooms every worker tries to find a good solution for himself.

But as we have seen in section 3.3, the second algorithm is also getting better, if we add salaries to the qooms, which is some form of scoring, while it would be difficult to score the workers.

So in the end we arrive in both cases at a similar method.

4.1 Rectangular Spreading Qooms

There is a big problem with comparing the results of the two algorithms, since one is trying to position circles, while the other one tries to put rectangles onto a grid.

In order to be able to compare the two, a special version of the spreading qooms algorithm was implemented, which uses rectangles instead of circles. With this we need to change the calculation of the forces from equation (3) in sec. 2.2.2.

Since we no more have radii, the forces are changed to the following function, if the two qooms overlap. If not, the forces stay 0. The forces are split up now in a force in the x, and a force in the y direction:

\[ f_{x:q_1,q_2} = \frac{1}{2} \cdot o_x \cdot u_x \]  \hspace{1cm} (14)

and

\[ f_{y:q_1,q_2} = \frac{1}{2} \cdot o_y \cdot u_y \]  \hspace{1cm} (15)

where the center of the two qooms are at \( (cx_1, cy_1) \) and at \( (cx_2, cy_2) \), \( o_x \) and \( o_y \) are the overlap’s in the x and y direction and \( u_x = \frac{cx_1 - cx_2}{o_x(s(x_1-x_2) + s(x_1-x_2))} \) and \( u_y = \frac{cy_1 - cy_2}{o_y(s(y_1-y_2) + s(y_1-y_2))} \). These values are also shown in fig. 36.

But this gives a very unstable result, because now two qooms only overlapping a little bit in one direction, but having a lot of overlap in the other one, mostly push each other along the long way, instead of the short one. This is shown also in fig. 37.

To solve this, we apply only the force in the direction which is smaller, i.e. if \( f_x < f_y \) then we use \( \vec{f} = f_x \cdot \vec{u}_x \). Only if \( f_x = f_y \) we use the full force, which is \( \vec{f} = f_x \cdot \vec{u}_x + f_y \cdot \vec{u}_y \). This still can lead to unwanted results, as now qooms that are in a row cannot move out of the way to the side. To solve this, we could change the force to push along the line connecting the qoom-centers. This has not been implemented though.
4.2 Comparison between the Rectangular Spreading Qooms and the Dynamic Qooms

Figure 38 shows the three different outputs, the one from the spreading qoom algorithm from sec. 2, the one from the rectangular spreading qoom algorithm from sec. 4.1 and the one from the dynamic qoom algorithm described in sec. 3. The rectangular spreading qooms output has the same scale as the dynamic qooms, but since in the spreading qooms algorithms we always have all qooms inside the system, there are more qooms there than in the dynamic qooms version. On the other hand the two spreading qooms algorithms have different outputs, since also the test layout was different (different qoom sizes, the hallway, etc).

In figure 39 we see the behavior of the rectangular spreading qooms versus the dynamic qooms. Note that in difference to the plots in sec. 2.4, here the average walking distance per worker is plotted, as was done in sec 3.4. This is not what the rectangular spreading qooms tries to optimize, since it tries to optimize the walking distance to and from a qoom. But the averages are still very close to those seen in sec. 2.4, since the only difference is the number of qooms vs the number of workers, through which we divide.

So in figure 39 we can see, that the rectangular spreading qooms get to a worse solution than the dynamic qooms. Note that the dynamic qooms first have smaller walking distances, but as the number of deleted qooms is reduced, it raises, while the spreading algorithm uses all the qooms at all times, and thus the walking improves over time. Also, since the spreading algorithm at all times has all the qooms in the building, the walking distances are automatically longer than in the dynamic qoom algorithm, which allows the workers to reuse qooms with different id’s.

Also, since we know our test plans are not necessarily optimal, the dynamic qooms algorithm is able to find solutions that require less walking. The spreading qooms algorithm must use as many qooms as specified in the plans, even if not all of that space is always used, so the walking distance can’t go down as far.
Figure 38: The three different visual outputs: (1) the spreading qooms, (2) the rectangular spreading qooms and (3) the dynamic qooms.

Figure 39: average walking distance per worker vs. iteration step using the rectangulated spreading qooms algorithm and the dynamic qooms algorithm.
5 Outlook

In this chapter we summarize again, what all could be done in future works on this project. First the general additions, that could be added to all three algorithms, then come the specific additions.

- **add more qoom-types**: at the moment there are only work-, meet- and leisure-qooms.
- **add more floors**: more floors could be simulated easily, but the workers would have to know, how to go to a different floor, and in the spreading algorithms the qooms would have to be able to decide on different floors (this could be added to their strategies).
- **detect space problems**: at the moment we assume, the building is large enough, a method should be added, which can check, if this is so, if we are in a critical space area (where packing the qooms could give a solution) or if there is no possibility to put all qooms into the building.
- **optimize plans**: A module to optimize the plans could be added, which tries to put qooms together prior to running our programs.
- **individualize evaluation**: A further step would be, to let the agents set the parameters to evaluate a layout. E.g. one worker could decide, that for him it’s not so important to walk a long way in the morning, but he wants to have a short way between his office and the office of his secretary.
- **add module that puts in walls**: the output of these algorithms could be used as the input for another module, that would then try to make actual rooms out of our qooms. After this, a pedestrian simulation such as the one in [10] can be run to evaluate the layout.

5.1 Spreading Qooms

In section 2 several possible additions were already mentioned, here is a collection of them:

- **varying sizes**: qooms could decide to try to get a bigger size, and get a reward if this is successful. For this we would have to add the qoom-size to the strategies.
- **new strategies depend on old ones**: In sec. 2.2.1 we describe how to get a new strategy. This could also depend on the existing strategies, so that a qoom would try more strategies in an area, where there are already good ones.
- **fix problem with too large $\beta$s**: As we saw in sec. 2.4.3 we have a problem with too large $\beta$s in the strategy selection. This could be improved by applying the there mentioned solution.

5.2 Dynamic Qooms

In the dynamic qooms, several extensions were mentioned.

- **Keep office when not there**: At the moment, when a worker leaves his office, he never knows, if his office will still be there, when he comes back. To change this, we could add the possibility for workers to go on paying a qoom even while they are not there, and thus reserving it for later use.
- **Meeting-organizer**: As stated in sec. 3.2.3 at the moment the first worker decides on where to hold a meeting. It could also be set in the plans, who is the organizer.
- **Next worker selection**: The selection of the next worker described in sec. 3.2.1 could be done more randomly. This is described in appendix A.1.1.
- **Next qoom location selection**: The location for the next qoom could also be selected more randomly, this is described in appendix A.1.2.
- **Test with combinations of salaries and raising deletion costs**: We tested the dynamic qooms algorithm with raising qoom deletion costs (as described in sec. 3.2.6 and tested in sec. 3.4.3) and with salaries (as described in sec. 3.3.1 and tested in sec. 3.4.5) but did not try to combine the two. With good settings, this could lead to even better results.
5.2.1 Q-Learning

Besides those, there was the idea to let workers look ahead during the day and try to not only optimize their local qoom choice (now), but their whole day. For this it was discussed if q-learning could help.

In Q-learning an agent remembers his states during the day and the reward he got for them. When he moves through this state space, he propagates some of this reward to earlier states, so he learns which way through the state space was good.

This could be added to the dynamic qoom algorithm in various ways.

The first idea is to let the workers learn which locations for their qooms were good. For this they would need to remember where they placed which qoom, this would be their states.

This has a complication though, since we need to propagate punishments from other workers who also use this qoom to the worker who created it.

Also this seems rather difficult to add to the present algorithm since it differs strongly from the present cost based system.

Another way would be to let the bars remember, which qoom-types were there at which times and how good they were. They could then adapt the costs according to the qoom-type. Also the bars would be punished, if the qoom-type changes during the day.

This has the problem that now the bars learn instead of the workers or the qooms, adding one more type of agents to the system.

5.3 Rectangular Spreading Qooms

The rectangular spreading qooms was least tested, since it was only introduced for the comparison. Still there are some improvement ideas.

- **Improve Forces:** As we saw already in sec. 4.1, the forces make some problems with this algorithm. We should test whether forces along the line connecting the qoom centers are better suited.

  - **Snap to grid:** In the end of the algorithm, the qooms could be pushed into a grid. This would lead to an even more similar result as the dynamic qoom algorithm.

  - **Cumulative output:** Like in the dynamic qoom in sec. 3.4 we could create a cumulative output of the qooms at a location. This would be easier with the snap to grid option.

6 Summary

We have shown two prototype algorithms to find layouts for big office buildings by letting the future users decide what they like instead of defining a global fitness for the building. To allow users to make individualized decisions, both algorithms are agent-based.

In the first algorithm, the qooms are agents, trying to find good positions inside the building. They are then evaluated by the users. In the second algorithm the users themselves are the agents deciding on the positioning of the functional areas.

But in both algorithms the qooms learn what is good in the end, while the workers evaluate them. In the second algorithm the workers decide on the positioning, but they do so based on knowledge which the qooms gain. So basically both algorithms in the end have qooms learning and users evaluating them.

For both algorithms many parameters were introduced, but the results of the first one showed that many of the parameters do not, or only marginally change the results. We also found that using the average scores of the qooms as a global scoring function does not lead to a better solution than letting the qooms optimize their local scores individually. So in this case the Nash-Equilibrium is optimized as well as a global score.

In the second algorithm we showed that only correct tuning of the parameters leads to a stable solution. Unfortunately for the second algorithm it is harder to understand what really is being optimized.
With these two algorithms we showed that it is possible to find designs by letting agents make decisions to optimize their own happiness (or the happiness of their users) instead of having an entity outside the system that decides how things should be and how to make all the users happy.

7 Acknowledgments

I would like to thank Riko Jacob at the Institute for Theoretical Computer Science, ETH for his input on bin-packing and other theory aspects of this work; Markus Braach at the Chair for Computer Aided Architectural Design, ETH for his demonstration and help on KaisersRot; and Oliver Fritz also at the Chair for Computer Aided Architectural Design, ETH for his input on how architects see the project.

Further I would like to thank Karsten Droste at the Chair for Computer Aided Architectural Design, ETH; and Prof. Kai Nagel at the Institute for Computational Science, ETH for their input and ideas how to do this project. And especially I would like Bryan Raney at the Institute for Computational Science, ETH not only for his input but also for the endless time he invested to make this text understandable. I would also thank my friend Robert for his constructive critique and my wife, Vassiliki, for her support, help and understanding.

A Appendix: Implemented But Not Used

The following additions were implemented but have not been used for any test runs.

A.1 Dynamic Qooms

A.1.1 Next Worker Selection

As stated in section 3.2.1, at the moment we process the workers with the same starting time in the order in which they are added to the eventlist. But we could also process them more randomly.

In order to let the workers with the same starting time be processed in random order, we would need a method to randomly insert them inside the list of workers with the same starting time. This could be achieved by going through the list of workers and drawing a random number every time we encounter a worker with the same starting time. If this number is smaller than \( \frac{\text{#workers with this time so far}}{\text{workers with this time so far}} \), we put the new worker’s event before this event in the list.

The pseudo code for this is shown in fig. 40.

With this algorithm, all workers with the same time have the same probability to be in a specific place. This can be shown by a simple induction proof (done here for the first position in the list): in the beginning, when there is only one worker he is the first with probability one, and as long as he stays the only one, this is correct.

If we have already \( n \) workers each being the first with probability \( p_{(i,n)} = \frac{1}{n} \) and one more worker is added now, the probability to put the new worker onto the first position is \( \alpha = \frac{1}{n+1} \) so all the old ones now have the probability

\[
p_{(i,n+1)} = (1 - \alpha) \cdot p_{(i,n)} = (1 - \frac{1}{n+1}) \cdot \frac{1}{n} = \frac{1}{n+1}
\]

(16)

The new one is gets in front with the probability

\[
p_{(n+1,n+1)} = \sum_{i=0}^{n} p_{(i,n)} \cdot \alpha = \alpha \cdot \sum_{i=0}^{n} p_{(i,n)} = \frac{1}{n+1} \cdot 1
\]

(17)

So both the old ones and the new one now have the probability \( p_{(i,n+1)} = \frac{1}{n+1} \). QED

A.1.2 Select next qoom location

In section 3.2.3 we saw the method to find a location for a new qoom. But that was just selecting the solution with the minimal cost. In the following we show two more methods, to select a location.
if not worker.nextTime in eventList.times then
    add worker.event to list
else
    set nWorker to the first worker with time = worker.nextTime
    inserted = false
    nrWorkers = 1
    while (nWorker.nextTime == worker.nextTime) and not inserted do
        nrWorkers++
        if uniformRand(0,1) <= 1/nrWorkers then
            insert worker before nWorker
            inserted = true
        else
            nWorker = next worker in list
        end if
    end while
    if not inserted then
        insert worker at end of same time list
    end if
end if

Figure 40: random next worker selection

**Probabilistic Location Selection**  In this case we again need to be able to adapt a probability on the fly when we see new data. This is almost the same as in A.1.1 but here we also have different probabilities. In order for this to work, we first need to transform the cost into a value that is in $[0, \infty]$. For this we set $C_i = e^{-\beta \cdot C_i}$. Where $\beta$ is a constant factor that decides how exploratory our workers are. If $\beta = 0$ all locations have the same probability, is $\beta = \infty$ we will select the best location.

Now as we “see” the locations one after the other, we need something to calculate the change probability. So the first location has probability 1 to be selected (it’s the only one). But after $n$ steps, when we find the $(n+1)$th possible location we have the following situation: each one of the old locations is at the moment selected with probability

$$P_{i,n} = \frac{C_i}{\sum_{j=0}^{n} C_j}$$

The sum of all these probabilities is 1, as it should be. So to get a probability of

$$P_{n+1, n+1} = \frac{C_{n+1}}{\sum_{j=0}^{n+1} C_j}$$

for the $(n+1)$th location, we need to select it with this probability. With that all the old locations are now selected with the probability:

$$P_{i, n+1} = P_{i,n} \cdot (1 - P_{n+1, n+1}) = \frac{C_i}{\sum_{j=0}^{n} C_j} \cdot \frac{\sum_{j=0}^{n+1} C_j - C_{n+1}}{\sum_{j=0}^{n+1} C_j} = \frac{C_i}{\sum_{j=0}^{n+1} C_j}$$

With this our algorithm becomes the following:

```python
sumC = 0.0
selpos = NULL
for all location in availableLocations do
    c = exp(-beta * location.cost)
    sumC += c
    if uniformRand(0,1) <= c/(sumC) then
        selpos = location
    end if
end for
```
Sampled Location Selection  As it was already mentioned in 3.2.3, with this method every worker samples a certain amount of random locations for his qoom and then selects one. If he does not get a valid location he does not go to his next activity but leaves the building to smoke a cigarette and calm down.

To select which one of the sampled options he will use, he could again either take the one with the smallest cost or select one randomly as described above.

The number of locations a worker tries out could be his strategy, he could even be able to set various numbers for different activities, meaning some things are more or less important to him.

Local Search  Another similar idea would be, to let the worker search for a good qoom near his present qoom by spiraling around it, and having a threshold, when to accept a solution and stop searching.

References


