Design To Cost with the Aid of Numerical Optimization Techniques

Thorsten M. Loemker [1], Albrecht Degering
Computer Science in Architecture and Landscape Architecture (CALA), Technical University of Dresden, Germany
http://www.arch.tu-dresden.de/cala/
[1] thorsten.loemker@tu-dresden.de, [2] albrecht.degering@mailbox.tu-dresden.de

Abstract. *This paper discusses the use of optimization methods in the architectural design process. It points out the possible scope of integration with regard to building costs in the design of new buildings.*
Keywords: *Optimization; Layout-Planning; Design to Cost.*

Introduction

Aside architecture numerous disciplines exist where computer-supported planning and construction play an important role in the design of a product. In the automotive industry, aircraft industry or in mechanical engineering the designing engineers surely obtain support during the form finding process through the use of computer programs. The use of optimization techniques from within these programs is obligatory and as almost every real problem has one or more optimal solutions, it is the cardinal goal in most sectors of industry to strive for these solutions. But as common as the use of these techniques is in banking corporations, insurance companies or the manufacturing industry, as scarce is their use in the architectural domain. Even if optimization techniques are used for specific tasks in the building sector, e.g. static calculations, almost no architect ever touched an optimization engine to calculate a range of design solutions that meet his or her needs. Unlike the products of the industries mentioned above the form of buildings to be developed seem not to follow clearly defined design-variables and principles. However, economic efficiency, profitability and usability determine architectural principles and therefore the aesthetics of the building product as well. Architecture is therefore not free of principles that can clearly be labeled and that influence the appearance of a building. In contrast to other disciplines though it appears to be more difficult to determine the important variables in the architectural design process and to set up the rules these variables have to comply with. Our research demonstrates the use of optimization engines and their scope of integration in the design process with regard to building costs. We point out predominant factors that influence building costs of domestic architecture and integrate these factors in mathematical calculations that result in various geometrical building envelopes and floor plans with quantifiable performance.

Commonly, building a house is regularly limited by the budget of its prospective owner. Decisions made during the design process clearly depend on this budget. Usually architects start with determining rough building costs in the very early stages of the design process to get an overview of the expected expenses. Computing power, which nowadays can be found in every single office in form of personal computers, plays an important role in this process. With the use of this power, more detailed estimation of costs is possible. With the aid of optimization techniques, right in the first steps of the design process it is possible to find cost-saving design variants, disclosing more flexibility during further work progress.

How can these optimizations techniques look like? Of course an exact analysis of costs during the preliminary design steps is not possible - and obviously not even necessary. However, some decisions made very early have great impact on the expenses. The outer shape of a building influences the area of outer walls, which are more expensive then common inner walls and thus influence costs to a greater extend. Arrangements and sizes of rooms, together with necessary walls that for example protect bathrooms from insights, are reflected in a similar way in our calculations. Finally, static and building climatic factors are instrumental in pricing the house. Well thought static structures cause less material use, well-designed window areas lead to a better quality of rooms and consequently to reduced costs.

This document provides an overview about our research regarding cost reduction during the early stages of design. We demonstrate a way to design low priced buildings with the aid of numerical optimization techniques.

Previous research

As shown in previous research (Loemker, 2006a, 2006b), it is possible to solve layout-planning problems through the use of optimization algorithms that search for optimal solutions in a given set of rooms. The results of this research are integrated in an improved form to provide the initial room sets for our new optimization searches that try to find cost effective layout solutions. The algorithms were simplified; additional constraints have been added for increased performance and flexibility.

According to these optimization models, a java program for easy constraint creating and editing was developed. Users are now provided with the possibility to interactively formulate special requirements, which are parsed and passed to the optimization engine. The results are constrained satisfying room arrangements, which easily can be interpreted and displayed. These room arrangements are the basis for the further optimization process, which our research is referencing.

Basic principles

To reach the main goal, cost reduction, we decided to concentrate on optimizing the count of elements within a building and their arrangement. Elements are walls and windows, floors and ceilings as well as stairs. At the moment, our results are limited to one storey buildings. Thus, stairs and floors, respectively ceilings, are momentarily fixed in size. Therefore, our main task was to find an optimization algorithm for walls within a given pattern of spaces. With regard to the evaluation of existing buildings we defined the term 'room' to be obsolete and to replace it with the term 'space'. The classical concept of rooms implies boundaries. For a given arrangement of spaces, which determines location and size of walls, cost optimization would be restricted to minimal changes. With the equivalent definition of a space allocation program, spaces do not have to be separated. This definition is not new due to the fact that contemporary architecture often makes use of this principle – open and floating spaces do not depend on specific room usages anymore.

For every such space program, an algorithm can find solutions fitting structural and practical requirements. These arrangements can be optimized to cost, to allow good floor plan solutions at an optimized level of building costs.

Respectively we defined four steps, which assure diversity of results. Decomposing the process in smaller steps helps to lower combinatorial diversity. Furthermore the user can change or cancel the progress if it is predictable that the results will not fit the task.

Space arrangement program selection

The basis is a pre-defined space program. As mentioned in our earlier research, the methods for creating a bandwidth of solutions are known and used in this research. Selecting a solution fitting the problem best from an economic point of view, but also under architectural aspects, is the first step to a good design. For this, it is necessary to select the solution manually, since a computer is not able to capture all edge conditions yet. Whereas manual selection means that the optimizer sometimes produces hundreds of different solutions that all fulfill the constraints and objectives defined during the previous optimization run. The manual selection process provides the option that the selected variant can obtain other subjective features that were not specified in the calculations.

Determination and definition of constraints

Some of these conditions are important for the task performed by the optimization engine. E.g., it is necessary to pass data anent adjacent buildings, not just for preventing windows in concerning walls, but also to prevent the calculation of redundant walls for static reasons. Other constraints might be necessary to meet requirements of sound insulation or to screen residents from view. The user can define these constraint definitions interactively. We marked out three different possibilities - to close a room, to close a room in special directions and to close a room to the outside of the building. This convention covers most of the cases relevant in the design process. For all other cases, the algorithm is flexible enough to handle them.

Optimization

Flexible but fast algorithms to generate optimized and multifarious solutions are the main aspect of our research. These algorithms will be described within the next chapters.

Performance and output

Finally, an evaluation of results allows a comparison of the solutions from different points of view. In the first instance this will be done with regard to building costs. The user selects one of the solutions found, which are displayed on a board and for which the expected price is calculated.

Optimization Techniques

The cost calculation is based on the element method described in DIN 276. For every element we detailed prices that are used in the calculations. The price of an element is dependent of its specific size, i.e. width, height and depth. The optimization engine which is ILOG OPL (Van Hentenryck, 1999) can therefore operate with abstract data. To provide the data, it was necessary to develop an interactive input program including a parser. As a programming language Java was chosen, whose class concept provides the possibility to reuse the functions.

For performance reasons the parser overlays the ground plan with a grid pattern, with a square side length of one meter. The emerging squares act as a coordinate system for the communication with the optimizer. So, every wall can be described as a tuple of integers, two of them representing the coordinates, the other two referring to the direction of the wall. Thus, typical wall sets consist of four integer arrays with the same length according the count of walls. Four fields at the same index in the corresponding array represent a one meter long part of a wall. Furthermore, the parser already separates the outer walls from the inner ones.

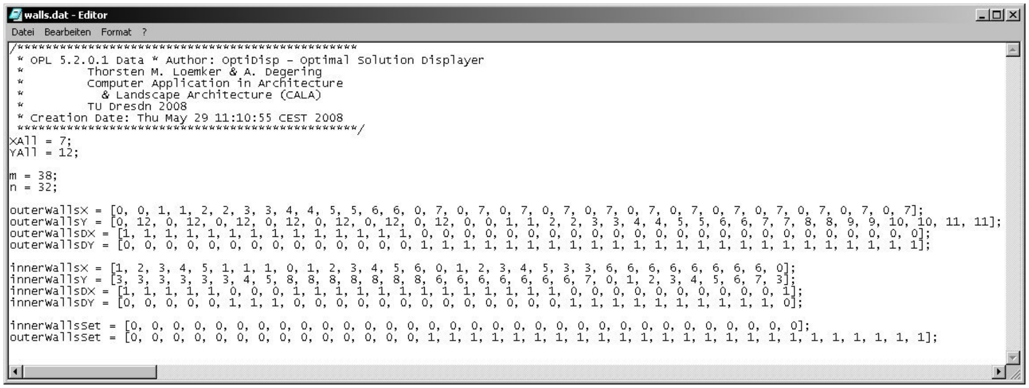
This separation in types of walls was made for more flexible cost calculation on the one hand and on the other to provide different types of walls, according to their location. An outer wall can be either closed or open, which leads to other different features. A closed wall will carry the ceiling, which has an influence on statics. A window is necessary for lighting and airing. Inner walls can be of two different types; by now, they are closed and either non-supporting or supporting walls.

With the given wall set the optimizer can determine the type of the wall, which will influence costs. In two arrays, each of them representing one type of wall, the type of wall can be turned on or off. A Boolean value for each wall segment is set to represent the walls of the building. Through the use of a simple multiplication operation the values of walls not relevant can be masked. With the aid of this technique, we found a simple way performing the price calculation and defining static constraints, which easily can be interpreted by the optimization engine.

In the following examples, we will use the following notation, taken from OPL Model syntax.

n, m	Integer, indicating the number of outer/ inner walls
XAll, YAll	Integer, declaring the building size in two directions
supOutWall[i]	supporting outer wall at index [i]
nonsupOutWall[i]	non-supporting outer wall at index [i]
supInnWall[i]	supporting outer wall at index [i]
nonsupInnWall[i]	non-supporting inner wall at index [i]
forall(i in 1..n)	for every element i within the range from 1 to n
sum(i in 1..n)	sum of all elements i within the range from 1 to n
sIWallDX[i], sIWallDY[i]	supporting inner wall x/y direction length
isOutWallSet[i], isInnWallSet[i]	Decision variables containing the outer wall's state
sOWallDX[i], sOWallDY[i]	supporting outer wall x/y direction length

Special constraints are necessary to meet static needs and user defined requirements. We figured out seven necessary constraints, three of them are design constraints, four to meet static requirements. Below, each constraint of the optimization routine is explained and an abstract formulation is specified, showing the basic principle.



```
/**************************************************
 * OPL 5.2.0.1 Data * Author: OptiDisp - Optimal Solution Displayer
 *         Thorsten M. Loemker & A. Degering
 *         Computer Application in Architecture
 *           & Landscape Architecture (CALA)
 *         TU Dresdn 2008
 * Creation Date: Thu May 29 11:10:55 CEST 2008
 **************************************************/
XAll = 7;
YAll = 12;

m = 38;
n = 32;

outerWallsX = [0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7];
outerWallsY = [0, 12, 0, 12, 0, 12, 0, 12, 0, 12, 0, 12, 0, 12, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11];
outerWallsDX = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
outerWallsDY = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];

innerWallsX = [1, 2, 3, 4, 5, 1, 1, 1, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 3, 3, 6, 6, 6, 6, 6, 6, 6, 6, 0];
innerWallsY = [3, 3, 3, 3, 3, 3, 4, 5, 8, 8, 8, 8, 8, 8, 8, 6, 6, 6, 6, 6, 6, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 3];
innerWallsDX = [1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1];
innerWallsDY = [0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0];

innerWallsSet = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
outerWallsSet = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
```

Figure 1
A data set, written in simple ASCII-Data format

First, we have to define the possibilities of the type both of inner and outer walls. It is self-explanatory, that an outer wall can only be either closed or unclosed. This is expressed by adding the two corresponding array values, comparing the result to one. Therefore, either one or the other value has to be set to satisfy the constraint

```
forall(i in 1..m)
( supOutWall[i] + nonsupOutWall[i] == 1 )
```

For inner walls the constraint is nearly the same, except that an inner wall does not necessarily have to be turned on – it is also possible not to set it. The addition can result in zero, which is expressed by the less-equal sign.

```
forall(i in 1..n)
( supInnWall[i] + nonSupInnWalls[i] <= 1 )
```

The third and last design constraint defines the relation between the area of opened and unopened, thus supporting and non-supporting, outer walls. The factor can be expressed with a variable; in the example, it is fixed to 2/3.

```
( sum(i in 1..m) supOutWall[i] ) < 2/3*m
```

For static constraints, we decided to use a simple form of definition. For performance reasons the optimization engine makes no proof of static regulations, but the parser performs this task. However, relevant rules are included, preventing the optimizer from generating invalid solutions.

First, it has to be assured that walls are able to carry the load of the roof, respectively the load of the ceiling. For this a convention was made, that the sum of the lengths of all supporting walls is more than a third of the area of the roof. This constraint is sufficient to grant the building not to collapse.

```
( sum(i in 1..m) supOutWall[i]
+ sum(i in 1..n) supInnWall [i] ) > ( XAll*Yall )/3
```

Furthermore it has to be constrained that the building is supported in both directions. To provide adequate walls it was necessary to add the following constraint, requesting that 40% of each direction has to be supported by walls, either inner or outer. For simplification of the statements, a definition of two Integer variables was made, combining each direction's wall data with the decision variables and adding them to one value.

```
SupOutWallsX
= sum(i in 1..m) sOWallDX[i]*isOuterWallSet[i]

SupOutWallsY
= sum(i in 1..m) sOWallDY[i]*isOuterWallSet[i]

SupInnWallsX
= sum(i in 1..n) sIWallDX[i]*isInnerWallSet[i]

SupInnWallsY
= sum(i in 1..n) sIWallDY[i]*isInnerWallSet[i]
```

With these definitions the statement for the constraint definition given above is

```
SupOutWallsX + SupInnWallsX > 0.4*XAll;

SupOutWallsX + SupInnWallsY > 0.4*YAll;
```

The last definition was made to prevent that all walls can point in the same direction, which also would cause the building to collapse. A formulation determining the bandwidth of relation between the walls oriented in each direction assures this.

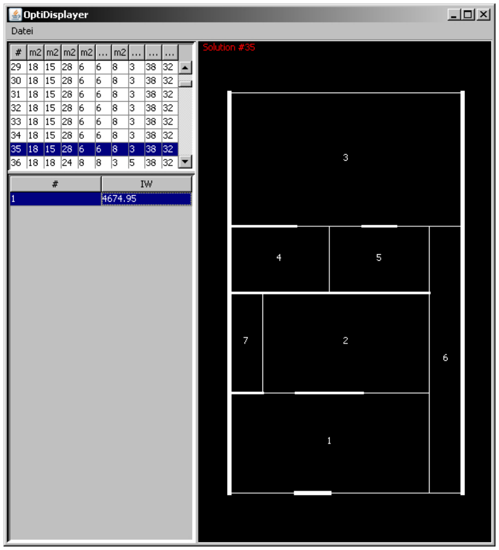
```
SupOutWallsX+SupInnWallsX
< 2*(SupOutWallsY+SupInnWallsY) &&

(SupOutWallsY+SupInnWallsY)
< 2*( SupOutWallsX+SupInnWallsX)
```

The optimization goal is expressed with a simple formula, multiplying the walls turned on with the correlating price value.

```
minimze sum(i in 1..m) (pSOW*supOutWall[i]
+pNSOW*nonSupOutWall[i])
+sum(i in 1..n) (pSIW*supInnWall[i]
+pNSIW*nonSupInnWall[i])
```

With the aid of these five simple constraints, connected to the minimization goal, it is already possible to create buildable floor plans with quantifiable performance.



#	m2	m2	m2	m2	...	m2	...	...	...
29	18	15	28	6	6	8	3	38	32
30	18	15	28	6	6	8	3	38	32
31	18	15	28	6	6	8	3	38	32
32	18	15	28	6	6	8	3	38	32
33	18	15	28	6	6	8	3	38	32
34	18	15	28	6	6	8	3	38	32
35	18	15	28	6	6	8	3	38	32
36	18	18	24	8	8	3	5	38	32

#	IW
1	4674.95

Figure 2
Screenshot of the parser program with an output of the optimization process

Prospects

Optimized architectural designs will get increasingly important in the future. We have demonstrated a basic method to design to cost with the aid of numerical optimization techniques. The example given shows the direction of our further research. The potential of adding complex constraints, improving especially the software's performance for larger buildings and floor plans and to remove the one storey restrictions are in progress. Furthermore connecting existing BIM-software to our system, offers great possibilities for importing material data into the optimization process and writing it back to the BIM-model, thus relieving the user of entering redundant data in different programs.

References

Loemker, T. M.: 2006a, Plausibilität im Planungsprozess, Umbau und Umnutzung als Optimierungs-aufgabe (Dissertation), Bauhaus-Universität Weimar.

Loemker, T. M.: 2006b, Revitalization of Existing Buildings through Sustainable Non-Destructive Floor Space Relocation, in M. Mourshed, Proceedings of the GBEN 2006 conference, Global Built Environment Network, Preston, pp. 181-189.

Van Hentenryck, P. and Lustig, I.: 1999, The OPL Optimization Programming Language, MIT Press, Cambridge, Mass.