Analysis of the optimization used by the energy system model CEGOIA

Mourits, J.W.

J.W. Mourits

Analysis of the optimization used by the energy system model CEGOIA

Master thesis

April 15, 2022

Supervisors:
F.M. Spieksma (Leiden University)
M.A. Meyer (CE Delft)
E. Koster (CE Delft)

Universiteit Leiden
Mathematisch Instituut

Abstract

CE Delft has developed the CEGOIA-model to counsel Dutch governments and municipalities in the energy transition. CEGOIA can be used in any area, consisting of a certain number of neighbourhoods. Using an integer linear programming optimization, it computes an allocation of energy systems to the neighbourhoods, such that the total cost for the area is minimized. Different solvers have been used in CEGOIA to perform this optimization, but when the number of neighbourhoods is too large, the problem can not be solved within a reasonable amount of time. In combination with the fact that the problem is NP-hard, a heuristic has therefore been constructed. This heuristic consists of three different parts that have all been developed and implemented in the python-based CEGOIA-model currently used by CE Delft. The final goal of this thesis was to be able to run CEGOIA on the Netherlands in its entirety.

This thesis sets the mathematical framework for the optimization problem and gives a detailed description of the heuristic. Then, the results of the heuristic are shown and compared with problems that have already been optimized by CE Delft. Also, an analysis of the algorithm is given in which the complexity, existence of a solution and the general performance of the heuristic are investigated. Finally, the results are discussed and some alternative optimization methods are provided.

Contents

1 Introduction

Over the past few years the energy and sustainability consultancy company CE Delft has developed the energy system model CEGOIA. For each neighbourhood (neighbourhoods are determined by the CBS, the Dutch Centraal Bureau voor de Statistiek) in a certain area, CEGOIA computes the type of energy supply that minimizes the total cost for the entire area. The CEGOIA model has already been applied to small areas, for instance a municipality. However, for larger areas the runtime of the model is unfeasibly large or the problem can not be solved at all. The goal of this project is to analyzee the optimization method behind the CEGOIA model and to lower the runtime, so that CEGOIA can also be applied to large areas.

1.1 Framework

CEGOIA globally works as follows. An area is split into CBS-neighbourhoods, with each neighbourhoods consisting of dwellings and utilities. For each neighbourhood, the computations are executed for both the dwellings and the utilities. First, the energy demand is computed, based on data on the number of buildings, the construction year, surface of the buildings, etc. The energy demand is combined with possible insulation levels. This leads to a set of different options for each neighbourhood. Then, for each neighbourhood the different energy systems to fulfill this demand are considered. There are 12 possible energy systems. The following table shows for each energy system which energy sources are used.

	Electricity	Gas/Green gas/Hydrogen	Heat (HT)	Heat (MT)	Heat (LT)	Heat (ZLT)	Heat (WKO)	Bio mass
Condensation boiler	×	×						
Hybrid heatpump (ground)	×	×						
Hybrid heatpump (air)	×	×						
Electric heatpump (ground)	×							
Electric heatpump (air)	×							
Heat exchanger HT	×		×					
Heat exchanger MT	×			×				
Heat exchanger LT	×				×			
Heat exchanger LT with individual heatpump	×				×			
Heat exchanger ZLT with individual heatpump	×					×		
WKO	×						×	
Condensation boiler with bio mass	×							×

Table 1: Possible energy systems with their corresponding energy sources

As shown in this table, all systems use electricity. Some energy systems use heat (HT, MT,

LT and ZLT indicates the temperature from high to very low) from collective heat sources, for instance residual heat of industries. This also leads to more options with different corresponding cost, because the heat sources have a certain location and reach in which it can be used. So decisions have to be made concerning the energy system, insulation level and possibly collective heat sources.
CEGIOA then computes the total cost of all these different options for each neighbourhood. This is the sum of the cost for construction (including insulation), energy, distribution (if a network for residual heat needs to be built, it will cost a certain amount per meter) and maintenance over the period of one year.
Then there are certain constraints. Of gas, electricity, hydrogen and biomass there are limited supplies for the entire area. Furthermore, collective heat sources contain a limited amount of heat. Finally, using these computed cost and constraints, an integer programming problem (IP) is constructed and solved. This solution tells which energy system needs to be assigned to which neighbourhood in order to minimize the total cost for the entire area. The solution is usually represented in a webtool as a map with colors indicating the type of energy system (see Figure 1). When a neighbourhood is colored grey, this means that the computations are not done for this neighbourhood. Further information about for instance the insulation level and energy use, can be found by selecting a neighbourhood in the webtool.

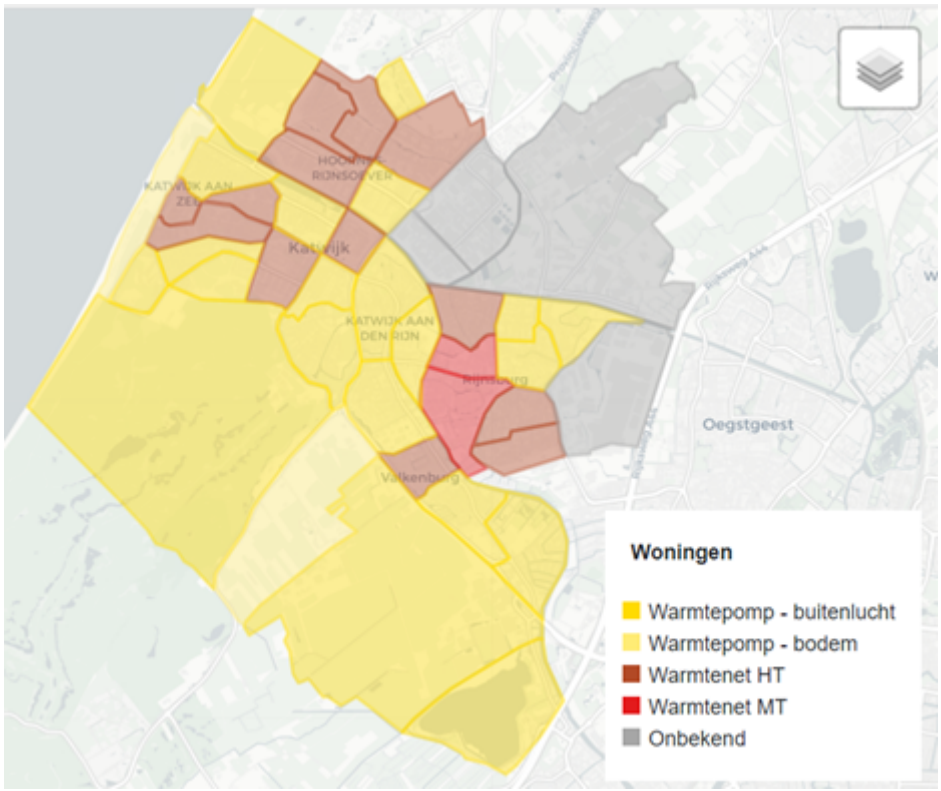

Figure 1: The output map of CEGOIA for the area of Katwijk

1.2 Problem definition

For smaller areas (less than 2000 neighbourhoods) the free solver CBC or GLPK_MI is used in the CEGOIA optimization. CBC[1] and GLPK_MI can solve small instances of the IP problem in a relatively short time, mostly within a few hours. These solvers use a combination of branch and cut algorithms , the dual simplex method and the interior point method. However, for larger areas these solvers have an unfeasibly large runtime or even fail to solve the problem. Therefore, the paid IBM solver CPLEX is used for these large areas. Due to the cost and runtime of CPLEX and due to the fact that even CPLEX cannot solve extremely large IP's (for example the Netherlands in its entirety), CE Delft wishes to find a way to solve large problems in a shorter runtime. A faster computation will also make it possible for CE Delft to perform a more accurate analysis of different scenario's. One of the final goals of this project is to run CEGOIA on the Netherlands in its entirety, which has over 13,000 neighbourhoods. This will provide insight in the distribution of heat systems over the Netherlands. With this new information, CE Delft will be able to give more accurate advice.

2 Mathematical formulation of the problem

The solution CEGOIA gives for a certain area, is currently computed using an IP-problem formulation. To illustrate the formulation of this IP-problem, we use a simple example. Consider an area consisting of three neighbourhoods. There are four possible energy systems (options) for each neighbourhood and these energy systems can use resources A and B (for example gas and electricity). The following table shows the amount of used resources and the cost for each option.

Neighbourhood	Option	A	B	Cost
1	1	10	0	10
	2	0	0	21
	3	0	7	15
	4	4	4	13
2	1	8	0	8
	2	0	0	34
	3	0	5	18
	4	6	3	17
3	1	9	0	9
	2	0	0	20
	3	0	6	16
	4	2	6	14

Table 2: Data for IP-problem example

The availability of recources A and B is set to 10 and 12 respectively. An optimal solution to this small problem can easily be computed and results in the allocation: neighbourhood 1 gets option 3, neighbourhood 2 gets option 3 and neighbourhood 3 gets option 1. This solution has total cost 42.
We will now construct an IP from this data. The variables of the IP are x_{ij} with $i \in M = \{1,2,3,4\}$ the option, and $j \in N = \{1,2,3\}$ the neighbourhood (we choose this notation as it is similar to the notation in the literature). We have

$$x_{ij} = \begin{cases} 1 & \text{if neighbourhood } j \text{ gets option } i \\ 0 & \text{otherwise.} \end{cases}$$

This leads to the following integer program.

$$\min 10x_{11} + 8x_{12} + 9x_{13} + 21x_{21} + 34x_{22} + 20x_{23} + 15x_{31} + 18x_{32} + 16x_{33} + 13x_{41} + 17x_{42} + 14x_{43} \tag{1}$$

s.t.

$$\begin{aligned} x_{11} + x_{21} + x_{31} + x_{41} &= 1 \\ x_{12} + x_{22} + x_{32} + x_{42} &= 1 \\ x_{13} + x_{23} + x_{33} + x_{43} &= 1 \\ 10x_{11} + 8x_{12} + 9x_{13} + 4x_{41} + 6x_{42} + 2x_{43} &\leq 10 \\ 7x_{31} + 5x_{32} + 6x_{33} + 4x_{41} + 3x_{42} + 6x_{43} &\leq 12 \\ x_{ij} \in \{0,1\}, \qquad & \text{for } i \in M \text{ and } j \in N. \end{aligned}$$

This problem is of the general form.

$$\min \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} \tag{2}$$

s.t.

$$\sum_{i \in M} x_{ij} = 1, \qquad \text{for } j \in N \tag{3}$$

$$\sum_{i \in M} \sum_{j \in N} a_{ij}^{(k)} x_{ij} \leq b_k, \qquad \text{for } k \in K \tag{4}$$

$$x_{ij} \in \{0, 1\}, \qquad \text{for } i \in M \text{ and } j \in N \tag{5}$$

with K the set of constraints, $M = \{1, ..., m\}$ the set of options and $N = \{1, ..., n\}$ the set of neighbourhoods. Also, we assume that $a_{ij}, c_{ij} \geq 0$ for all $i \in M, j \in N$. We call (3) the assignment constraints and (4) the energy limit constraints. The assignment constraints make sure that each neighbourhood gets exactly one option. The energy limit constraints make sure that the energy source limits are not exceeded.

Using this matrix-notation, for our example we have:

$$(c_{ij}) = \begin{pmatrix} 10 & 8 & 9 \\ 21 & 34 & 20 \\ 15 & 18 & 16 \\ 13 & 17 & 14 \end{pmatrix}$$

$$(a_{ij}^{(1)}) = \begin{pmatrix} 10 & 8 & 9 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 4 & 6 & 2 \end{pmatrix}, (a_{ij}^{(2)}) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 7 & 5 & 6 \\ 4 & 3 & 6 \end{pmatrix}$$

$b = (10, 12)$.

The formulation (2) - (5) of the problem is a variation on the Generalized Assignment Problem (GAP), which is formulated as follows.

$$\min \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} \tag{6}$$

s.t.

$$\sum_{i \in M} x_{ij} = 1, \qquad \text{for } j \in N \tag{7}$$

$$\sum_{j \in N} a_{ij} x_{ij} \leq b_i, \qquad \text{for } i \in M \tag{8}$$

$$x_{ij} \in \{0, 1\}, \qquad \text{for } i \in M \text{ and } j \in N. \tag{9}$$

The difference between the IP-problem in CEGOIA and the GAP is in the energy limit constraints (4) and (8). In the GAP there is a constraint for every $i \in M$ and we only sum over j. In the IP-problem in CEGOIA, we have an extra variable $k \in K$ and for each k, there is a

constraint that sums over i and j. This means that the IP-problem in CEGOIA is more general than the GAP. In terms of CEGOIA: when the problem would be written as a GAP, every option ($i \in M$) would correspond to a unique energy source. By using constraint (4), we make sure that an option can use multiple energy sources ($k \in K$), for example gas and electricity. Also, as we are summing over i and j, an energy source can be used by multiple options. For instance, all options use electricity (see table 1).
Because of this relation between the IP-problem in CEGOIA and the GAP, we use the following definition.

Definition 1
The IP-problem in CEGOIA, of the form given in (2) - (5), is called the Adapted Generalized Assignment Problem (AGAP).

3 NP-hardness

Using the formulation (2) - (5) we will prove that the Adapted Generalized Assignment Problem is NP-complete. The proof is similar to the proof of NP-completeness of the Generalized Assignment Problem in [3].

Consider the NP-complete Partition problem that asks whether for a given set of n real numbers $p_1, ..., p_n$, there exists an $S \subset \{1, ..., n\}$, such that $\sum_{j \in S} p_j = \sum_{j \notin S} p_j$. The Partition problem can be reduced to the AGAP the following way.
In the AGAP, let $m = 2$ and $k = 2$. Then we take $a_{1j}^{(1)} = a_{2j}^{(2)} = p_j$ and $a_{1j}^{(2)} = a_{2j}^{(1)} = 0$ and we let $b_1 = b_2 = \sum_{j \in N} p_j/2$.
For instance, for $n = 3$, the AGAP will be as follows.

$$x_{11} + x_{21} = 1 \tag{10}$$
$$x_{12} + x_{22} = 1 \tag{11}$$
$$x_{13} + x_{23} = 1 \tag{12}$$
$$p_1 x_{11} + p_2 x_{12} + p_3 x_{13} \leq \sum_{j=1}^{3} p_j/2 \tag{13}$$
$$p_1 x_{21} + p_2 x_{22} + p_3 x_{23} \leq \sum_{j=1}^{3} p_j/2 \tag{14}$$
$$x_{ij} \in \{0, 1\} \qquad \text{for all } i \in \{1, 2\}, j \in \{1, 2, 3\} \tag{15}$$

Substituting x_{21}, x_{22} and x_{23} in the last inequality using equalities (10) - (12), leads to the following.

$$p_1(1 - x_{11}) + p_2(1 - x_{12}) + p_3(1 - x_{13}) \leq \sum_{j=1}^{3} p_j/2$$
$$\sum_{j=1}^{3} p_j - (p_1 x_{11} + p_2 x_{12} + p_3 x_{13}) \leq \sum_{j=1}^{3} p_j/2$$
$$p_1 x_{11} + p_2 x_{12} + p_3 x_{13} \geq \sum_{j=1}^{3} p_j/2$$

This substitution can be executed for inequality (13) in a similar way. Therefore, we can conclude that the inequalities (13) and (14) can be written as equalities.

Now determining the feasibility of the AGAP is equivalent with solving the Partition problem. As the Partition problem is known to be NP-hard, so is the AGAP.

To check whether a given solution of the AGAP is feasible, we simply have to check if the solution meets the constraints (3) - (5). This can be done in polynomial time, so together with the fact that the AGAP is NP-hard, we can conclude that the AGAP is NP-complete.

Because the AGAP is NP-complete, computing an optimal solution to a large instance of the problem will take an extremely long running time. Therefore, when running CEGOIA on large areas, a heuristic may have to be used to be able to find a solution within a few days. In the next chapters the requirements for such a heuristic and the tested heuristics are discussed.

4 Solution Requirements

As we have concluded in the previous chapter that problem (2) - (5) is NP-hard, we will try to find an approximation of the optimal solution. An algorithm that computes such an approximation, a so-called heuristic, must satisfy some requirements. As the results of CEGOIA are used in practice to give advice to governments on the heat supply in certain areas, some of these requirements are somewhat less mathematical and more practical. In this chapter we take a look at the requirements and wishes CE Delft has for the solution given by CEGOIA.

4.1 Applicability

The solution computed by the heuristic must still, like the original solver does for small areas, provide an allocation of energy systems to neighbourhoods. This detailed solution is necessary for the advice given to municipalities on which energy system to choose in which part of town.

CEGOIA does not take geographical clustering into account. This means that it is not possible for adjacent neighbourhoods to collaborate in order to reduce cost. For instance, suppose three adjacent neighbourhoods all use residual heat. In practice, this means that the network to transport this heat from the source to the neighbourhoods can have a shorter length (the minimal spanning tree between the source and the neighbourhoods). However, in CEGOIA, the total cost for a neighbourhood is computed individually. So for all three neighbourhoods, the cost include the entire length of the network from the source to the neighbourhood.

4.2 Quality and runtime

The given solution must be unique. This means that, when running CEGOIA twice on the same instance, the results must be the same. The total cost is allowed to deviate from the optimal cost by at most 5 percent, but of course we try to minimize this deviation as much as possible. The runtime of the optimization must not be more than two days and preferably one day for all problems.

5 Heuristic for AGAP

In Chapter 2 of his paper [6], Wilson provides a simple dual algorithm for the generalized assignment problem. As mentioned in Chapter 2, the energy carrier constraint in the AGAP is different from constraint (8) in the GAP. Because of the fact that constraint (4) in the AGAP is given for each $k \in K$, we define the sum of the amount of used resources: $a_{ij}^{(\text{tot})} = \sum_{k \in K} a_{ij}^{(k)}$. Using this sum, we have modified the algorithm by Wilson into an algorithm that fits the AGAP-formulation.

5.1 The algorithm

The steps of the algorithm, applied to a problem of the form as in (2) - (5), are as follows.

Algorithm 1 AGAP heuristic

Step 1.
Let x'_{ij} with $i \in M$ and $j \in N$ be a solution to the relaxed assignment problem

$$\min \left\{ \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \quad \middle| \quad \begin{array}{ll} \sum_{i=1}^{m} x_{ij} = 1, & \text{for } j \in N \\ x_{ij} \in \{0,1\}, & \text{for } i \in M \text{ and } j \in N. \end{array} \right\} \tag{16}$$

Step 2.
If $\sum_{i \in M} \sum_{j \in N} a_{ij}^{(k)} x'_{ij} \leq b_k$ holds for all $k \in K$, then STOP. Return x'_{ij} as a feasible solution to the problem.
Otherwise, go to step 3.

Step 3.
Let k^* be the first[1] k for which

$$\sum_{i \in M} \sum_{j \in N} a_{ij}^{(k^*)} x'_{ij} > b_{k^*}.$$

For all $j \in N$, let i_j be the unique index such that $x'_{i_j j} = 1$. Let $N_1 = \{j \in N | a_{i_j j}^{(k^*)} > 0\}$ and $M_j = \{i \in M | c_{ij} \geq c_{i_j j}\}$. For all $j \in N_1$, compute for every $i \in M_j$ the following value:

$$t_{ij} = \frac{a_{i_j j}^{(\text{tot})} - a_{ij}^{(\text{tot})}}{-c_{i_j j} + c_{ij}},$$

Where we set $t_{ij} = 0$ if $c_{i_j j} = c_{ij}$ and $a_{i_j j}^{(\text{tot})} \geq a_{ij}^{(\text{tot})}$, and $t_{ij} = -\infty$ if $c_{i_j j} = c_{ij}$ and $a_{i_j j}^{(\text{tot})} < a_{ij}^{(\text{tot})}$.
Let $(i', j') = \text{argmax}_{(i,j)} t_{ij}$.
If $t_{ij} < 0 \ \forall i \in M_j, j \in N_1$ or if $M_j = \emptyset$, STOP. Return: problem is infeasible.
Otherwise, reallocate an option by setting $x'_{i'j'} = 1$ and $x'_{i_j j'} = 0$.
Go to step 2.

The idea of this algorithm is as follows. We start with the best possible assignment of options to neighbourhoods ignoring the constraints, which means that we pick the minimal cost option

[1] After step 1 of the algorithm, the constraints are in non-increasing order of exceedance of the constraint limit in the solution to (16). During the entire heuristic, this ordering remains.

in each neighbourhood. If the energy limit constraints (4) are not met, we reallocate an option and we keep doing this until the solution is feasible. The reallocation is chosen in step 3 and is based on selecting the option with the largest relative usage decrease with respect to the total cost.

The index i_j in step 3 of the algorithm exists and is unique, due to the constraints in (16). The set M_j in step 2 of the algorithm is chosen such that infinite looping is prevented. This is done by only considering more expensive options. Testing results from CEGOIA showed that this is indeed an effective way to prevent infinite looping and to still attain a good solution.

An alternative formula for t_{ij} is to only use $a^{(k^*)}$ instead of $a^{(\text{tot})}$. However, in practice this turns out to give a worse solution. An explanation for this is that some options use an energy system that have a very small $a^{(k^*)}$, because it uses a different energy source. Due to this small $a^{(k^*)}$, the option is directly chosen with the alternative formula, because the t value is very large. But this means a lot of cheaper options are skipped and this eventually leads to a worse solution. By using $a^{(\text{tot})}$ the values of t_{ij} are more balanced and this gives a better solution.

5.1.1 Example

First we compute the sum of the constraint matrices:

$$(a_{ij}^{(\text{tot})}) = \begin{pmatrix} 10 & 8 & 9 \\ 0 & 0 & 0 \\ 7 & 5 & 6 \\ 8 & 9 & 8 \end{pmatrix}.$$

Running the described heuristic on the example given in table 2 now gives the following.
Step 1.
Solving the relaxed problem

$$\min 10x_{11} + 8x_{12} + 9x_{13} + 21x_{21} + 34x_{22} + 20x_{23} + 15x_{31} + 18x_{32} + 16x_{33} + 13x_{41} + 17x_{42} + 14x_{43}$$

s.t.

$$\begin{aligned} x_{11} + x_{21} + x_{31} + x_{41} &= 1 \\ x_{12} + x_{22} + x_{32} + x_{42} &= 1 \\ x_{13} + x_{23} + x_{33} + x_{43} &= 1 \\ x_{ij} \in \{0, 1\} \qquad & \text{for } i \in M \text{ and } j \in N. \end{aligned}$$

gives the solution $x'_{11} = x'_{12} = x'_{13} = 1$ with value 27.

Step 2.
We have $\sum_{i \in M} \sum_{j \in N} a_{ij}^{(1)} x'_{ij} = 10 + 8 + 9 > 10 = b_1$, so the solution x'_{ij} is not feasible and we go to step 3.

Step 3.
Consider the constraint $k^* = 1$. Then we have $N_1 = \{1, 2, 3\}$ and $M_j = \{2, 3, 4\}$ for all $j \in N_1$.

Now we compute t_{ij} for all $j \in N_1$, $i \in M_j$, which results in the following.

$$\begin{aligned} t_{21} &= \tfrac{10}{11}, \quad t_{22} = \tfrac{8}{26}, \quad t_{23} = \tfrac{9}{11}, \\ t_{31} &= \tfrac{3}{5}, \quad t_{32} = \tfrac{3}{10}, \quad t_{33} = \tfrac{3}{7}, \\ t_{41} &= \tfrac{2}{3}, \quad t_{42} = -\tfrac{1}{9}, \quad t_{43} = \tfrac{1}{5}. \end{aligned}$$

The largest $t_{ij} \geq 0$ is t_{21}, so $(i', j') = (2, 1)$. We now reallocate the option chosen for neighbourhood 1, so $x'_{21} = 1$ and $x'_{11} = 0$.

Step 2.
We have $\sum_{i \in M} \sum_{j \in N} a_{ij}^{(1)} x'_{ij} = 8 + 9 > 10 = b_1$, so the solution x'_{ij} is not feasible and we go to step 3.

Step 3.
We have $N_1 = \{2, 3\}$ and $M_j = \{2, 3, 4\}$ for all $j \in N_1$. The t_{ij} for $i \in M$, $j \in N_1$ remain the same, so

$$\begin{aligned} t_{22} &= \tfrac{8}{26}, \quad t_{23} = \tfrac{9}{11}, \\ t_{32} &= \tfrac{3}{10}, \quad t_{33} = \tfrac{3}{7}, \\ t_{42} &= -\tfrac{1}{9}, \quad t_{43} = \tfrac{1}{5}. \end{aligned}$$

Now the largest $t_{ij} \geq 0$ is t_{23}, so $(i', j') = (2, 3)$. We reallocate the option chosen for neighbourhood 3, so $x'_{23} = 1$ and $x'_{13} = 0$.

Step 2.
The solution $x_{21} = x_{12} = x_{23} = 1$ is feasible and has total cost 49.

5.2 One step improvement (AGAP reversed)

To improve the result of the AGAP heuristic, we have developed an algorithm that considers one step improvements. This means that in one neighbourhood we choose a different option and all other allocations remain the same. As the value of the objective function must be improved, the cost of this new option must be lower and the solution must still be feasible. The implemented one step improvement is similar to the algorithm described in Section 5.1, but now in reversed order (this is why the algorithm is called "AGAP reversed"). This means that we consider only the cheaper options for each neighbourhood that are still feasible and we choose the reallocation that results in the largest relative cost decrease with respect to the total usage.

Algorithm 2 AGAP reversed

Step 1.
Let x'_{ij} with $i \in M$ and $j \in N$ be the solution computed by the AGAP heuristic.

Step 2.
For all neighbourhoods $j \in N$ let i_j be the unique index such that $x'_{i_j j} = 1$. For all $j \in N$ and for all $i \in M$ for which $c_{ij} < c_{i_j j}$, we compute the relative cost decrease:

$$s_{ij} = \frac{c_{i_j j} - c_{ij}}{-a^{(\text{tot})}_{i_j j} + a^{(\text{tot})}_{ij}} = \frac{1}{t_{ij}}$$

(if $a^{(\text{tot})}_{i_j j} \geq a^{(\text{tot})}_{ij}$, let $s_{ij} = \infty$).

Step 3.
Let (i', j') be the pair with the largest relative cost decrease, such that after reallocating $x'_{i'j'} = 1$ and $x'_{i_{j'}j'} = 0$, the solution x'_{ij} for all $i \in M, j \in N$ is still feasible. Perform this reallocation and go back to step 2. If such a pair (i', j') does not exist: STOP.

After testing the algorithm with different formulas for s_{ij} it turned out that $s_{ij} = 1/t_{ij}$ led to the best results. An explanation can be that we choose the option with the largest relative usage decrease with respect to the total cost in the AGAP heuristic. Reversing this, means we choose the option with the largest relative cost decrease with respect to the total usage. Therefore, this formula for s_{ij} has the desired effect.

If $a^{(\text{tot})}_{i_j j} \geq a^{(\text{tot})}_{ij}$, there is less usage of resources while the cost decrease. This means reallocating $x_{i_j j} = 0$ and $x_{ij} = 1$ would be advantageous. Therefore, when $a^{(\text{tot})}_{i_j j} \geq a^{(\text{tot})}_{ij}$ in step 2 of AGAP reversed, we let $s_{ij} = \infty$.

Suppose that after this algorithm has been executed there is still a possible one step improvement. This means that in one neighbourhood a cheaper option can be chosen while all other neighbourhoods keep the same option. So there exists a pair (i, j) for which $c_{ij} < c_{i_j j}$ and such that reallocating $x_{ij} = 1$ and $x_{i_j i} = 0$ still gives a feasible solution. However, if such a pair exists, the algorithm cannot have stopped in step 3. So we can conclude that after performing the AGAP reversed algorithm there are no possible one step improvements.

The idea to reverse the AGAP heuristic and the steps of the algorithm, are not based on anything found in the literature, but are original contributions. This also applies to the multiple step improvement that is described later in this thesis.

5.2.1 Example (continued)

Performing the greedy improvement on the example in Section 5.1.1 gives the following.

Step 1.
The solution computed by the AGAP heuristic is $x'_{21} = x'_{12} = x'_{23} = 1$ and has total cost 49.

Step 2.
For all $j \in N$ we compute the relative cost decreases for all $i \in M \setminus \{i_j\}$ for which $c_{i_j j} \geq c_{ij}$:

$$s_{11} = \tfrac{11}{10}, \quad s_{13} = \tfrac{11}{9},$$
$$s_{31} = \tfrac{6}{7}, \quad s_{33} = \tfrac{4}{6},$$
$$s_{41} = \tfrac{8}{8}, \quad s_{43} = \tfrac{6}{8}.$$

Step 3.
Reallocating $x'_{11} = 1$, $x'_{13} = 1$ and $x'_{41} = 1$ all lead to an infeasible allocation, so the feasible reallocation with the largest cost decrease is: $x'_{31} = 1$ and $x'_{21} = 0$. Therefore the new allocation is: $x'_{31} = x'_{12} = x'_{23} = 1$ and we go back to step 2.

Step 2.
The relative cost decreases are:

$$s_{11} = \tfrac{5}{3}, \quad s_{13} = \tfrac{11}{9},$$
$$s_{33} = \tfrac{4}{6},$$
$$s_{41} = \tfrac{2}{1}, \quad s_{43} = \tfrac{6}{8}.$$

Step 3.
All corresponding reallocations are infeasible, so the algorithm stops. The allocation is: $x_{31} = x_{12} = x_{23} = 1$ and has total cost 43.

5.3 Results

The algorithms AGAP and AGAP reversed have been implemented and tested on some problems CE Delft has been working on. These problems have already been optimized using CBC or GLPK_MI, so the results of the heuristic can be compared to the optimal solution. In the following table the column “Heuristic solution” shows the testing results of the entire heuristic. So this is the AGAP heuristic given in section 5.1, followed by AGAP reversed.
In this table the first column contains the number of neighbourhoods for each problem. The column “cost diff” contains the difference in total cost between the optimal solution and the heuristic solution and the column ”% deviation” shows this difference as a percentage of the total cost of the optimal solution.

	Optimal solution		Heuristic solution			
# nbh	Total cost	Runtime (sec)	Total cost	Runtime (sec)	Cost diff.	% deviation
11	68522319	0.279	68522319	0.016	0	0
42	121478550	1.683	121489071	0.200	10521	0.009
67	233752194	7.352	234171297	0.454	419103	0.179
178	262379424	18.612	263068512	0.570	689088	0.263
404	822442810	5,139.389	823395458	11.324	952648	0.116
1141	2825574770	> 10000	2835505746	39.130	9930976	0.351
1807	4888654668	> 10000	4910666315	414.695	22011647	0.450

Table 3: Results of the heuristic compared to the optimal result

After a slight change in step 2 of AGAP reversed, the runtime of the algorithm can be reduced. This can be done by not computing all s_{ij} in every iteration. Instead only the $s_{i,j'}$, with j' the neighbourhood in which the most recent reallocation was performed in step 3, have to be recalculated. However, this can result in a different solution. It is possible that an option (i, j) that was not feasible, becomes feasible after a reallocation that reduces the use of an energy source. When all s_{ij} are computed in every iteration, this option will become available. But in the alternative, faster algorithm, this is not the case. The following table shows the results of the faster AGAP reversed algorithm.

	Optimal solution		Heuristic solution			
# nbh	Total cost	Runtime (sec)	Total cost	Runtime (sec)	Cost diff.	% deviation
11	68522319	0.279	68522319	0.030	0	0
42	121478550	1.683	121489071	0.047	10521	0.009
67	233752194	7.352	234171297	0.200	419103	0.179
178	262379424	18.612	263068512	0.464	689088	0.263
404	822442810	5,139.389	823395458	1.954	952648	0.116
1141	2825574770	> 10000	2835687824	7.958	10113054	0.358
1807	4888654668	> 10000	4910666315	60.710	22011647	0.450

Table 4: Results of the heuristic compared to the optimal result using the faster version of AGAP reversed

It turns out that for the calculated problems, only the problem with 1141 neighbourhoods gives a different solution. Instead of a 0.351% deviation, it is now 0.358%. However, the runtimes have decreased significantly. Therefore, the faster version of AGAP reversed is used.

A small note: there was a certain problem consisting of 137 neighbourhoods where the deviation from the optimal solution was more than 1%. However, this problem was quite unique due to some demands made by the municipality. For example, the electricity limit had to be very low and there was a minimum insulation level requirement. The reason for the deviation came from this combination of demands and would not occur in other problems. Therefore, the results of this problem are disregarded.

5.4 Multiple step improvement

After the improvement described in Section 5.2, there are no neighbourhoods in which a cheaper option can be chosen such that the solution is still feasible. In other words, there are no one step improvements. Hence, the only way to reach a better overall solution is to select a more expensive option for certain neighbourhoods after which some other neighbourhoods can use cheaper options. We call this multiple step improvements. The difficulty in this approach is which neighbourhood should get a more expensive option and by how much the cost in this neighbourhood must be raised, in order to decrease the total cost of the overall solution.
In the algorithm described in this section, we solve this in the following way. We define two sets. The first set, P, contains rows corresponding to cheap options. The second set, Q contains rows corresponding to expensive options. Then, for all neighbourhoods where an option in P is selected in the current solution, we try replacing this option by an option from Q. Subsequently, we perform AGAP reversed and if the total cost declines, this becomes the new solution.
But before we explain the algorithm in detail, a mention about the structure of the cost matrix has to be made.

Note 1: Sorting the cost matrix
In the implementation of CEGOIA, the cost matrix is sorted such that $c_{ij} \leq c_{i'j}$ if and only if $i \leq i'$ for all $i, i' \in M$. So for all neighbourhoods, the options are sorted such that the cost is ascending. Of course, the matrices $(a_{ij}^{(k)})$ are sorted in sync with the cost matrix. However, this means that a row of a matrix does not necessarily correspond to an option. To still be able to retrieve the final solution of CEGOIA, every matrix element in every column gets an identificator that tells to which option it corresponds. The heuristic is implemented in the same way, so using these sorted matrices. For Algorithms 1 and 2, it is not important whether the matrices are sorted. However, the algorithm described in this paragraph uses the fact that the matrices in CEGOIA are sorted.

In this section we assume that the matrices are sorted as explained in Note 1. The algorithm will then be as follows.

Algorithm 3 Multiple step improvement

Step 1.
Let x'_{ij} with $i \in M$ and $j \in N$ be the solution computed by the AGAP heuristic followed by the AGAP reversed heuristic. Fix $p, q \geq 1$ such that $p + q \leq m$.
Let $P = \{1, ..., p\} \subset M$. Let $R = \{i \in M | x'_{ij} = 1 \text{ for some } j \in N\} = \{r_1, ..., r_s\}$ and let $Q = \{r_{s-q+1}, ..., r_s\}$. Note that q must be chosen such that $q \leq s + 1$.

Step 2.
For all neighbourhoods $j \in N$ let i_j be the unique index such that $x'_{i_j j} = 1$. Let $x^*_{ij} = x'_{ij}$ for $i \in M, j \in N$ and let c^* be the corresponding total cost.
For all $j \in N$ with $i_j \in P$ do:
    For all $i \in Q$ do:
        Let $x'_{i_j j} = 0$ and $x'_{ij} = 1$. If $\sum_{i \in M} \sum_{j \in N} a_{ij}^{(k)} x'_{ij} \leq b_k$ holds for all $k \in K$, perform AGAP reversed with this solution as starting solution.
        If the resulting total cost is less than c^*:
            $x^*_{ij} = x'_{ij}$ for $i \in M, j \in N$, let c^* be the corresponding total cost.
        Otherwise: $x'_{ij} = x^*_{ij}$ for $i \in M, j \in N$.
STOP. Return x^*_{ij} as a feasible solution.

As the matrices are sorted as mentioned in note 1, the set Q contains the q most expensive chosen rows. We only consider the chosen rows, because in CEGOIA there is usually a limited set of selected rows that correspond to practically advantageous combinations of energy systems and insulation levels. Be aware that p and q must be chosen such that $P \cap Q = \emptyset$. To illustrate how this algorithm works, we again take a look at the example.

5.4.1 Example (continued)

After sorting the options, we get the following cost matrix and corresponding constraint matrices.

$$(c_{ij}) = \begin{pmatrix} 10 & 8 & 9 \\ 13 & 17 & 14 \\ 15 & 18 & 16 \\ 21 & 34 & 20 \end{pmatrix},$$

$$(a_{ij}^{(1)}) = \begin{pmatrix} 10 & 8 & 9 \\ 4 & 6 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, (a_{ij}^{(2)}) = \begin{pmatrix} 0 & 0 & 0 \\ 4 & 3 & 6 \\ 7 & 5 & 6 \\ 0 & 0 & 0 \end{pmatrix}, (a_{ij}^{(\text{tot})}) = \begin{pmatrix} 10 & 8 & 9 \\ 8 & 9 & 8 \\ 7 & 5 & 6 \\ 0 & 0 & 0 \end{pmatrix},$$

$b = (10, 12)$.

Step 1.
The starting solution is now $x'_{31} = x'_{12} = x'_{43} = 1$. This is the same solution as the one obtained by AGAP reversed, but as the matrices are sorted, the order of the indices changes. For this example, we fix $p = 1$ and $q = 2$, so $P = \{1\}$. From the starting solution it follows that $R = \{1, 3, 4\}$. As q is equal to 2, this means that $Q = \{3, 4\}$.

Step 2.
We fix $x^*_{31} = x^*_{12} = x^*_{43} = 1$ and $c^* = 43$. The only neighbourhood $j \in N$ with $i_j \in P$ is $j = 2$. For all $i \in Q$ we now have:

- For $i = 3$, we let $x'_{12} = 0$ and $x'_{32} = 1$. The solution is still feasible, so we perform AGAP reversed. This results in the solution $x'_{31} = x'_{12} = x'_{43} = 1$, which has total cost $43 \not< 43 = c^*$. This means x'_{ij} is automatically restored to x^*_{ij}, so $x'_{31} = x'_{12} = x'_{43} = 1$.

- For $i = 4$, we let $x'_{12} = 0$ and $x'_{42} = 1$. The solution is still feasible, so we perform AGAP reversed. This results in the solution $x'_{31} = x'_{12} = x'_{43} = 1$, which has total cost $43 \not< 43 = c^*$. This means x'_{ij} is automatically restored to x^*_{ij}, so $x'_{31} = x'_{12} = x'_{43} = 1$.

The algorithm finishes and there has been no improvement. The solution is: $x_{31} = x_{12} = x_{23} = 1$ with total cost 43.

5.5 Results after the multiple step improvement

After applying the AGAP heuristic and the AGAP reversed heuristic on the problems shown in Section5.3, the multiple step improvement has been executed with different parameters p and q. For $p = q = 3$ it turned out that the total cost was the same for all problems as for $p = q = 2$. Logically, for $p = q = 2$ the runtime was less than for $p = q = 3$ (for 1807 neighbourhoods it is 2127 seconds against 4349 seconds) as fewer alternative options had to be considered. The cases

$p = 1$, $q = 2$ and $p = 2$, $q = 1$ have also been tested, but they both had higher total cost for some of the problems and the running time did not differ much from the case of $p = q = 2$ (all running times were between 2000 and 3000 seconds). Therefore taking $p = q = 2$ led to the best results, which are shown in the following table.

	Optimal solution		Heuristic solution			
# nbh	Total cost	Runtime (sec)	Total cost	Runtime (sec)	Cost diff.	% deviation
11	68522319	0.279	68522319	0.098	0	0
42	121478550	1.683	121489071	1.181	10521	0.009
67	233752194	7.352	234171297	1.403	419103	0.179
178	262379424	18.612	262614249	6.676	234825	0.089
404	822442810	5139.389	823092468	38.253	649658	0.079
1141	2825574770	10000+	2834971564	389.347	9150783	0.333
1807	4888654668	10000+	4910484399	1,713.640	21829731	0.447

Table 5: Results of the heuristic after the multiple step improvement with $p = q = 2$ compared to the optimal result

If these results are compared to the results shown in Section 5.3, it can be noticed that for the problems with 11, 42 and 67 neighbourhoods there is no difference, but the other problems have a better solution after the multiple step improvement. However, the running time also increases significantly: from 61 seconds for 1807 neighbourhoods to 1714 seconds. A running time of 1714 seconds is still acceptable, but when running CEGOIA on the Netherlands in its entirety, the runtime is too large. As the heuristic is only used in very large problems, the multiple step improvement is therefore usually not used. This means that the heuristic stops after the AGAP reversed algorithm.

5.6 Results for the Netherlands in its entirety

The final goal of this project was to run CEGOIA on the Netherlands in its entirety. Unfortunately, it turned out that the memory space of the input matrices for the constraints was too large and overloaded the RAM. Especially for the constraints regarding the residual heat sources, there were too many large matrices: more than 2000 13952×448-matrices. This problem did not occur in the optimization itself, but in the computations before the optimization that were already implemented by CE Delft. After unsuccessfully trying several methods to reduce this memory space, we had to conclude that the input for the Netherlands was simply too large and a simplification of the problem had to be used. This problem did not contain a constraint matrix for each residual heat source, but one matrix for each municipality containing the combined quantities of all heat sources within the municipality. Also some heat systems that are not frequently used were left out, resulting in less rows for each matrix. Due to these changes in the problem, the optimization could be performed and this resulted in the following map.

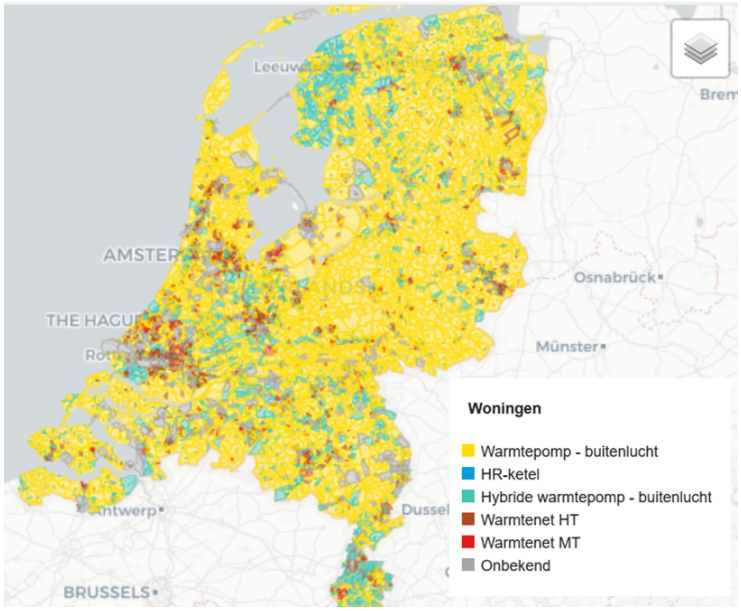

Figure 2: Result of CEGOIA for the Netherlands

Of course, we could not compare these results to the optimal solution. However, we were able to compare these results of the heuristic to the optimal solution of certain parts of the Netherlands. For example, for Zuid-Holland

Also, it seems that in practice, these results are quite logical. In the large cities, such as Rotterdam, Den Haag, Amsterdam and Utrecht, a heat exchanger is used. This is because a network needs to be installed to use residual heat and in densely populated areas this is cheaper. Also, in these cities, there is much residual heat available from industries. In the less populated areas, such as the east of the Netherlands, a heatpump is used. This is because a heat network is too expensive in these areas and for a heatpump only the heatpump itself is needed for each building. In Friesland and the south of Limburg, a hybrid heatpump is often selected. We suspect that this is because the buildings in these areas are quite old. This means it is more expensive to

increase the insulation. A hybrid heatpump is an option that does not need the buildings to be well insulated, so therefore a hybrid heatpump is often the best option in these areas.

6 Analysis of the heuristic

The results found in Table 3 meet the requirements set in chapter 4, but it is also interesting to know something about the performance of the heuristic on general integer programming problems. Therefore, in this section we investigate whether the algorithm always gives a unique solution and we take a look at the quality of the solution of general integer programs. Also, the complexity of the algorithm is computed and the results of a small sensitivity analysis are given.

6.1 Existence and uniqueness of a solution

The solution given by the heuristic is unique. That is, when running the heuristic several times on the same data, the result is always the same. The only part of the algorithm where it is not immediately clear that this is the case, is in the beginning of step 3 where a constraint k^* is chosen. However, in the programming code the constraints are checked one by one and k^* is the first constraint that is exceeded, so this always results in the same solution for each run.

To investigate the existence of a solution, we distinguish two cases: when there is only one constraint and when there are at least two constraints.

6.1.1 Existence of a solution for one constraint

In this paragraph, we assume that there is only one constraint, so $k = 1$. This means we can write $a_{ij}^{(tot)} = a_{ij}$. We will show that, if a feasible solution exists, the AGAP heuristic will return a feasible solution. Consider the following lemma.

Lemma
Suppose that $k = 1$. If an AGAP has a feasible solution, the AGAP heuristic (Algorithm 1) will result in a feasible solution.

Proof
Let an AGAP be given such that a feasible solution x^* exists. Now suppose the heuristic returns that the AGAP is infeasible. This means that in the last iteration we have an infeasible solution x. For all neighbourhoods $j \in N$ let i_j be the unique index such that $x_{i_j j} = 1$. As x is returned as an infeasible solution, we have:

$$\sum_{i \in M} \sum_{j \in N} x_{ij} a_{ij} > b, \tag{17}$$

and for all i with $c_{ij} \geq c_{i_j j}$ (if any) we have:

$$t_{ij} = \frac{a_{i_j j} - a_{ij}}{-c_{i_j j} + c_{ij}} < 0, \text{ so } a_{i_j j} - a_{ij} < 0. \tag{18}$$

As a feasible solution x^* exists, there must exist a $j \in N$ with $x^*_{i^*_j j} = 1$ and:

a) $a_{i^*_j j} < a_{i_j j}$ (because x^* is feasible and x is infeasible),

and therefore:

b) $c_{i^*_j j} < c_{i_j j}$ (follows from property a and (18)).

Note that from property b, it automatically follows that in this neighbourhood j, option i_j can not already be selected in the first step of the AGAP heuristic. So, suppose that for this neighbourhood j, the option i_j in the infeasible solution is selected in iteration $h > 1$. Then in iteration $h-1$, we have $x_{lj} = 1$ for $l \in M$, $l \neq i_j, i_j^*$.
So this means that in iteration $h-1$ by property a, we have:

$$a_{lj} - a_{i_j^* j} > a_{lj} - a_{i_j j}.$$

And by property b, we have:

$$-c_{lj} + c_{i_j^* j} < -c_{lj} + c_{i_j j}.$$

So combining this leads to the following.

$$t_{i_j^* j} = \frac{a_{lj} - a_{i_j^* j}}{-c_{lj} + c_{i_j^* j}} > \frac{a_{lj} - a_{i_j j}}{-c_{lj} + c_{i_j j}} = t_{i_j j}. \tag{19}$$

However, in step 3 of the AGAP heuristic, we select the option with the largest positive t-value. So this means that for neighbourhood j, we do not choose option i_j in iteration $h-1$. This contradicts what we previously stated. Therefore, we can conclude that the AGAP heuristic will return a feasible solution, if there exists one. □

We can ask ourselves if, for $k = 1$, this feasible solution is also the optimal solution. However, this does not have to be the case and we can show this by the following counterexample. Consider the IP-problem with one constraint:

$$(a_{ij}) = \begin{pmatrix} 8 & 7 \\ 2 & 3 \end{pmatrix}, \qquad (c_{ij}) = \begin{pmatrix} 1 & 2 \\ 6 & 6 \end{pmatrix}$$
$$b = 11.$$

In this example, the AGAP heuristic selects the second option for the first neighbourhood in the first iteration. This means that the feasible solution given by the heuristic is $x_{21} = x_{12} = 1$ with total cost 8. However, the optimal solution is $x_{11} = x_{22} = 1$ with total cost 7.

6.1.2 Existence of a solution for at least two constraints

When there are at least two constraints, an example can be constructed in which a feasible solution exists, but is not found by the AGAP heuristic described in Section 5.1. Consider the following problem.

$$\min 7x_{11} + 6x_{12} + 9x_{21} + 12x_{22} \tag{20}$$

s.t.

$$\begin{aligned} x_{11} + x_{21} &= 1 \\ x_{12} + x_{22} &= 1 \\ 4x_{11} + 5x_{12} &\leq 4 \\ 3x_{21} + 4x_{22} &\leq 5 \\ x_{ij} &\in \{0,1\} \qquad \text{for } i \in \{1,2\} \text{ and } j \in \{1,2\}. \end{aligned}$$

This problem has a feasible solution (which is also the optimal solution): $x_{11} = x_{22} = 1$. However, when performing the algorithm on this problem, it concludes that it is infeasible. The starting solution is $x_{11} = x_{12} = 1$ (which is infeasible due to the first inequality) and in the first iteration the gains are $t_{21} = 1/2$ and $t_{22} = 1/6$. So after reallocating, the new solution is $x_{21} = x_{12} = 1$, which is still infeasible. In the next iteration, as only options with higher cost are considered, the only gain is $t_{22} = 1/6$. After reallocating, the final solution $x_{21} = x_{22} = 1$ is infeasible as the constraint $4x_{21} + 5x_{22} \leq 4$ is exceeded.
In the heuristic for the Generalized Assignment Problem given in [6], this is prevented by chosing $t_{i'j'}$ such that $\sum_{j \in N} a_{i'j} x_{i'j} + a_{i'j'} \leq b_{i'}$. However, as the integer programming problem in CEGOIA is not an actual GAP, this does not work.
An alternative way to simulate this step, has been tested by only choosing reallocations that do not have a negative effect on other constraints. However, it turns out that the algorithm described in the previous sections has led to the best results. Also, it very rarely happens that an infeasible solution is returned, while there exists a feasible solution; for all tested problems in CEGOIA, this never occurred. Therefore, these versions of the AGAP heuristic and AGAP reversed heuristic have been used in CEGOIA.

6.2 Performance of the algorithm on general integer programming problems

To test the performance of the heuristic on general integer programming problems, a simulation has been implemented. In the simulation a $m \times n$ cost matrix is generated with c_{ij} drawn from a uniform distribution over the interval $[100, 500)$. Also a predetermined number of constraints is generated where each constraint consists of an $m \times n$ constraint matrix with $a_{ij}^{(k)}$ drawn from a uniform distribution over the interval $[1, 20)$ and a corresponding constraint limit b_k drawn from a uniform distribution over the interval $[a, b)$ where $a = 10 \cdot \frac{n}{2}$ and $b = 10 \cdot n$. The bounds of the intervals of the uniform distribution have been chosen in such a way that in most cases the problem is neither infeasible, nor choosing the cheapest option for each neighbourhood is immediately a feasible solution.

6.2.1 Results for different values of m and n

In the following table the percentage deviation from the optimal total cost of a simulation with 3 constraints is shown for different m (number of options) and n (number of neighbourhoods). Due

to running time issues, these simulations only use the AGAP heuristic and the AGAP reversed heuristic, so not the multiple step improvement. The results are obtained by taking the average of the percentage deviation from the total cost of the optimal solution of 500 simulations.

m \ n	10	20	30	40	50	60	70
10	3.883	2.720	2.611	2.348	2.422	2.281	2.446
20	3.925	3.554	3.210	2.821	3.431	3.273	3.337
30	4.509	3.210	2.528	2.648	2.701	2.521	2.522
40	3.107	2.331	2.259	1.969	2.418	2.044	2.010
50	2.823	1.958	1.950	1.680	1.861	1.607	1.716
60	2.488	1.955	1.573	1.579	1.294	1.392	1.345

Table 6: Percentage diviation of the heuristic from the optimal solution for different m and n, based on 500 simulations with 3 constraints, using the AGAP and AGAP reversed heuristic.

We see that for these randomly generated integer programming problems, the percentage deviation from the optimal solution is larger than for the CE-problems. These problems are smaller than the CE-problems; in the simulation there are at most 70 neighbourhoods and 60 options. This is because otherwise the simulation takes too long (for 70 neighbourhoods and 60 options it already takes several hours). However, from the table it seems that the larger the problem, the better the solution. Therefore, we did some simulations in which only the number of options and the number of neighbourhoods varied.
First, the number of options is fixed on 30 and we vary the number of neighbourhoods. Again, there are 500 simulations and 3 constraints. The following figure shows the percentage deviation from the optimal solution for different numbers of neighbourhoods.

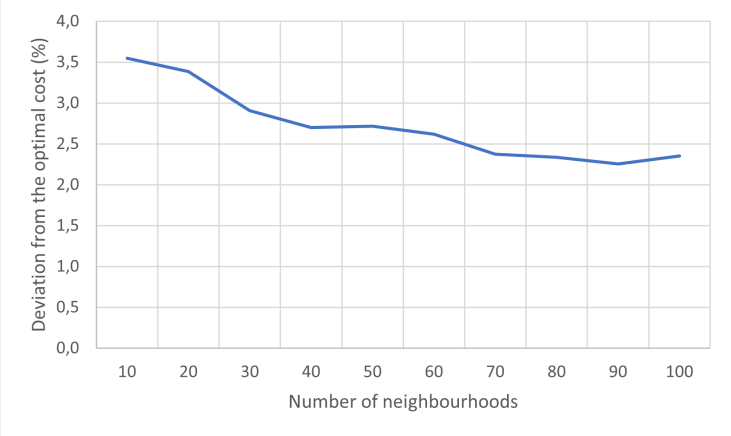

Figure 3: Percentage deviation from the optimal cost for a varying number of neighbourhoods with 30 options and 3 constraints based on 500 simulations.

Indeed, a larger number of neighbourhoods usually results in a better solution. However, at more than 70 neighbourhoods, the decrease is not significant anymore.
When the number of neighbourhoods is fixed on 42 (we chose 42 because it is the number of

neighbourhoods of an existing CE-problem) and we vary the number of options, the results are as follows.

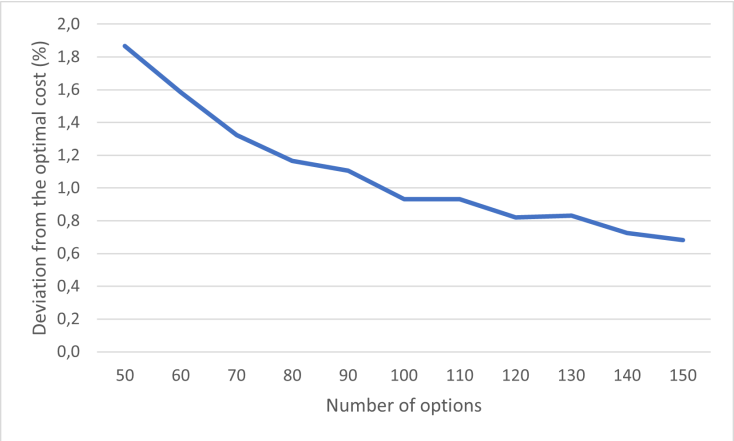

Figure 4: Percentage deviation from the optimal cost for a varying number of options with 42 neighbourhoods and 3 constraints based on 500 simulations.

In this figure it is clear that the deviation from the optimal solution decreases, when the number of options increases. This leads to the conclusion that the results of the heuristic are slightly worse on general linear programming problems compared to CE-problems. However, when a general problem is large enough, the deviation of the heuristic from the optimal solution reaches the same value as for CE-problems.
A somewhat unexpected result is that the results in the simulation are better when the problems are larger. This is in contradiction with the results of the CE-problems, where the larger problems have slightly worse results. An explanation can be that in this simulation we only consider different m and n, but the number of constraints is always 3. This does not match with CEGOIA, because a larger area (more neighbourhoods) usually contains more residual heat sources, so more constraints. Therefore, we take a look at the results for different numbers of constraints.

6.2.2 Results for different numbers of constraints

First, we consider the simulation with an increased number of constraints. However, increasing the number of constraints from 3 to 4, leads to a lot more simulations in which the heuristic does not find a solution, while there does exist one. To investigate how often this happens, a simulation has been performed in which the number of constraint varied. Table 7 shows the number of infeasible solutions for 1 to 5 constraints. The size of the problems ranged from 10 to 50 and we let $m = n$.

So indeed, the number of infeasible solutions increases for a larger number of constraints. Also, the smaller the problems, the more infeasible solutions there are. Intuitively, this is a logical result; a problem with many constraints and few options and neighbourhoods is hard to solve. As the percentage of infeasible solutions is already quite large for 4 constraints, this simulation is less representative. Therefore, we performed a simulation with less than 3 constraints. This led to the results for two constraints shown in Table 8.

m \ k	1	2	3	4	5
10	0	18	137	181	193
20	0	0	26	138	265
30	0	0	3	50	180
40	0	0	0	17	123
50	0	0	0	2	81

Table 7: Number of times the heuristic returns an infeasible solution, while there does exist a feasible solution. Based on 500 simulations with $m = n$ and different numbers of constraints.

m \ n	10	20	30	40	50	60	70
10	3.088	2.473	2.052	1.860	1.964	1.786	1.526
20	1.911	1.631	1.359	1.223	1.254	1.150	1.049
30	1.468	1.188	1.107	0.923	0.912	0.934	0.862
40	1.227	0.867	0.764	0.778	0.697	0.622	0.675
50	0.926	0.792	0.657	0.639	0.543	0.558	0.536
60	0.821	0.592	0.596	0.504	0.508	0.501	0.493

Table 8: Percentage diviation of the heuristic from the optimal solution for different m and n, based on 500 simulations with 2 constraints, using the AGAP and AGAP reversed heuristic.

Comparing these results to Table 6, shows that less constraints leads to a smaller deviation from the optimal solution. For larger problems, the percentage deviation of the simulation even reaches the percentage deviation of the CE-problems. When there is only one constraint, the results of the simulation are as shown in Table 9.

m \ n	10	20	30	40	50	60	70
10	0,268	0,173	0,112	0,082	0,065	0,049	0,038
20	0,207	0,107	0,069	0,048	0,043	0,029	0,027
30	0,162	0,072	0,047	0,036	0,026	0,022	0,018
40	0,131	0,055	0,043	0,030	0,022	0,017	0,015
50	0,097	0,046	0,029	0,021	0,017	0,013	0,011
60	0,082	0,036	0,025	0,017	0,016	0,011	0,010

Table 9: Percentage diviation of the heuristic from the optimal solution for different m and n, based on 500 simulations with 1 constraint, using the AGAP and AGAP reversed heuristic.

The deviation for 1 constraint is clearly less than for 2 or more constraints. It is even better than some of the results of the CE-problems.

Now we can take a look at the question mentioned in Section 6.2.1: Why does increasing the number of neighbourhoods lead to better results in the simulation and worse results in CEGOIA? In the simulations for $1, 2$ and 3 constraints, we saw that less constraints leads to better solutions. In the CE-problems the number of constraints can be quite large. Table 10 shows for some problems how many constraints there are.

Number of neighbourhoods (n)	Number of constraints (k)
42	9
67	13
178	12
404	35
1141	43
1807	47

Table 10: Number of constraints for some CE-problems

So the number of constraints in CE-problems increases for a larger number of neighbourhoods. As we saw in Tables 6, 8 and 9, the deviation from the optimal solution is smaller when there are less constraints. This explains why the results in CEGOIA are worse for larger problems.

Now another question remains: Why is the deviation from the optimal solution in CEGOIA always less than 0.5%? Considering the fact that these problems have many constraints, this seems rather low compared to the simulation results. An explanation for that is the following.

In CEGOIA the number of constraints corresponds to the number of energy sources. So this can be electricity or gas, but also collective heat sources like residual heat of industries. The number of heat sources given in the table above appear to be too much to find a solution with the heuristic, based on the results of the simulation. However, only electricity is used in every option. All other constraints correspond to energy sources that can only be applied in a limited area (for instance residual heat) or in a few options (for instance bio mass). In conclusion, only one constraint limits all combinations (i, j). The remaining constraint matrices contain mainly zeroes.
In addition, the constraint matrices in CEGOIA have a certain structure. This structure is based on the fact that rows correspond to options. As a result, the numbers in the constraint matrices are in the same range when they correspond to the same heat system. Also, there is an interrelationship between the constraints as they correspond to certain energy sources. This means that the information in Table 1 is reflected in the constraint matrices. Even after sorting the matrices as explained in Note 1, this structure remains roughly the same.

In conclusion, problems in CEGOIA have many interconnected constraints that all have a certain structure. In Table 4 we can see that the heuristic works well for these types of problems. In the simulation, we do not have this structure; the matrix elements are chosen randomly from a certain distribution. Here the number of neighbourhoods, options and constraints all affect the quality of the heuristic solution. This leads to a certain balance between the characteristics of an AGAP and the quality of the solution. So for some simulations the heuristic works as good as it does for CEGOIA. For other simulations, usually simulations with many constraints and few options and neighbourhoods, the heuristic performs worse.

6.2.3 Comparison with the results found by Wilson

Finally, the simulation has also been executed with the variable values given by Wilson in Section 3 of [6]. The matrices $(a_{ij}^{(k)})$ have been implemented in such a way that they match the form of the GAP-constraints as in (8). This is done by letting k be equal to the m in [6] and setting

$(a_{ij}^{(k)}) = 0$ for all $i \neq k$. With this implementation, the simulation matches the design of the test performed in [6] and the results can be compared. However, the simulation results are a factor 100 larger than the results given in Table 1 of [6]. By computing some ranges of values for p_{ij} given the formulas in the article, we found that an optimal value of 10422 for $n = 500$ is too small. So this raises the question whether the data in Table 1 of [6] is correct.

6.3 Results after different stages of the heuristic

As the heuristics solves the IP-problem in different steps, it is interesting to take a look at the total cost of the solution after each step. We distinguish the following steps:

1. The solution without constraints (the solution of (16) in step 1 of algorithm 1);
2. The solution when the AGAP algorithm has been completed;
3. The solution when AGAP reversed has been completed.

For the total cost of these solutions, we have in general: $c(1) \leq c(\text{OPT}) \leq c(3) \leq c(2)$ with $c(\text{OPT})$ the total cost of the optimal solution.
For problems with many neighbourhoods and constraints, the difference between $c(1)$ and $c(\text{OPT})$ is usually relatively large. This is because for $c(1)$ there are no constraints, so in each neighbourhood the cheapest option can be selected. The difference between $c(\text{OPT})$ and $c(3)$ is equal to the differences found in Table 4, so these differences are relatively small. The difference between $c(3)$ and $c(2)$ are larger, because AGAP reversed reduces the total cost significantly.

6.4 Existence of a solution for infinitely many options

Paragraphs 6.1 and 6.2.1 raise the following question: For a fixed number of neighbourhoods, does the heuristic always find a feasible solution when m tends to infinity (assuming a feasible solution exists)? Unfortunately, this is not the case. This can easily been shown by expanding the options in Example 20 of Paragraph 6.1. In this example, the matrices defining the problem are as follows.

$$(a_{ij}^{(1)}) = \begin{pmatrix} 4 & 5 \\ 0 & 0 \end{pmatrix}, (a_{ij}^{(2)}) = \begin{pmatrix} 0 & 0 \\ 3 & 4 \end{pmatrix}, b = (4, 5), (c_{ij}) = \begin{pmatrix} 7 & 6 \\ 9 & 12 \end{pmatrix}$$

If we now let the number of options tend to infinity and we simply copy the last rows of each matrix, we get the following problem.

$$(a_{ij}^{(1)}) = \begin{pmatrix} 4 & 5 \\ 0 & 0 \\ 0 & 0 \\ \vdots & \vdots \end{pmatrix}, (a_{ij}^{(2)}) = \begin{pmatrix} 0 & 0 \\ 3 & 4 \\ 3 & 4 \\ \vdots & \vdots \end{pmatrix}, b = (4, 5), (c_{ij}) = \begin{pmatrix} 7 & 6 \\ 9 & 12 \\ 9 & 12 \\ \vdots & \vdots \end{pmatrix}$$

The only feasible solutions are $x_{11} = x_{i2} = 1$ with $i \geq 2$. But when performing the AGAP heuristic on this problem, the first iteration results in a reallocation such that $x_{i1} = x_{12} = 1$ with $i \geq 2$. And as the AGAP heuristic is formulated such that only more expensive options are considered in next iterations, we can never have $x_{11} = 1$ after the first iteration. Therefore, a feasible solution will not be found using the heuristic.

6.5 Complexity

It is useful to analyze the complexity of the algorithms. The complexity of the AGAP heuristic is as follows.

- Step 1 has complexity $\mathcal{O}(m \cdot n)$, because for each neighbourhood $j \in N$ we have to find the minimum cost option out of the m options.
- Step 2 has complexity $\mathcal{O}(k \cdot n)$, because all k constraints have to be checked and each check can be done by adding up the n chosen options.
- Step 3 has complexity $\mathcal{O}(m)$, because for every column in which a reallocation has been executed in the previous iteration, the t_{ij} have to be computed (only in the first iteration all t_{ij} have to be computed).

Step 1 is only performed in the first iteration. Step 2 and 3 have to be executed at most $(m-1) \cdot n$ times, because every iterations ends with a reallocation and there are $(m-1) \cdot n$ possible reallocations in total. This means that the total complexity of the AGAP algorithm is:

$$C(AGAP) = \mathcal{O}(m \cdot n + m \cdot n(k \cdot n + m)) = \mathcal{O}(m \cdot n(k \cdot n + m))$$

Now we compute the complexity of AGAP reversed (the fast version).

- Step 2 has complexity $\mathcal{O}(m)$, because for every column in which a reallocation has been executed in the previous iteration, the t_{ij} have to be computed (only in the first iteration all t_{ij} have to be computed).
- Step 3 has complexity $\mathcal{O}(n \cdot k)$, because the constraints have to be checked. As we use the fast version of AGAP reversed, the feasibility of every combination (i, j) is checked at most once.

These steps are executed at most $(m-1) \cdot n$ times, because that is the number of possible reallocations. Therefore, the total complexity of the algorithm is:

$$C(AGAPr) = \mathcal{O}(m \cdot n(m + n \cdot k)).$$

The complexity of the multistep improvement is as follows.

- Step 1 has complexity $\mathcal{O}(n(m \log m))$, because for each neighbourhood all options have to be sorted.
- Step 2 has complexity $\mathcal{O}(n \cdot q \cdot C(AGAPr))$, because for each neighbourhood with an option in P (which are all neighbourhoods in the worst case), we perform AGAP reversed for all options in Q (which consists of q options)

As step 1 is only executed once, the total complexity of the multistep improvement is:

$$C(MI) = \mathcal{O}(n(m \log m) + n \cdot q \cdot C(AGAPr)) = \mathcal{O}(n \cdot q \cdot C(AGAPr))$$

6.6 Sensitivity analysis

To investigate the influence of the constraint limits, a sensitivity analysis has been performed. In this analysis, we ran CEGOIA on different scenarios of one problem (Katwijk, with 42 neighbourhoods) and compared the total cost of the heuristic and the optimal solution. These scenarios are defined by different parameters, such as the target year and constraint limits, that are determined before running CEGOIA. These parameters can have any number of possible values,

depending on what scenarios are useful to investigate by CE Delft. If a certain parameter is not taken into account, the limit can be set to infinity. Table 11 shows which parameters can be modified for the Katwijk project and their corresponding different values.

Parameter	Possible values	Explanation
Target year	2030, 2050	The Dutch government stated that in 2030 at least 1.5 million dwellings should be carbon neutral, in 2050 all buildings should be carbon neutral
Electricity limit	753750, 850000, ∞	The amount of available electricity for the entire area
Gas limit	0, ∞	The amount of available gas for the entire area. This is a combination of green gas and fossil gas; the target year (2030 or 2050) determines the ratio.
Geothermal energy	518000, ∞	Geothermal energy is a collective heat source that can be used in Katwijk. The amount can be set to the estimated available amount or infinity.
Insulation level	Current, 70 kWh/m2, 50 kWh/m2	The minimum insulation level determines the maximum amount of energy that can be used in buildings per square meter. So a level of 50 kWh/m2 means that all buildings must have an insulation level such that at most 50 kWh energy per square meter is used. When the insulation level is "current", this means that there is no requirement on the insulation level.

Table 11: The different parameters in the sensitivity analysis and their possible values

All possible combinations of values for the parameters have been tested, so a total of 72 scenarios. As the table containing the results of all these scenarios was very large, it has not been added to this report. However, the results will be discussed here.
The first thing that stood out was that for an electricity limit of infinity, the heuristic solution and the optimal solution were the same. All options for a neighbourhood use electricity and the cheapest options are usually all-electric heat systems. So when there is an unlimited amount of electricity, the optimal solution is simply selecting the cheapest option in each neighbourhood. The AGAP heuristic starts by taking the cheapest option in each neighbourhood and then checks the feasibility. As these cheapest options are all-electric and the electricity limit is infinite, this starting solution is already feasible and therefore the heuristic solution is the same as the optimal solution.
When only scenarios with an electricity limit of 753750 and 850000 are considered, it is difficult to draw conslusions from the results. When the amount of gas is infinite, most scenarios have heuristic cost equal to the optimal cost. However, for some scenarios (6 out of 24) this is not the case and it is unclear what causes this difference. This is even more complicated as the available gas is a mix of green gas and fossil gas, which have different cost. For the other parameters, there

does not seem to be a connection between the value of the parameter and the cost difference between the heuristic and optimal solution.
Another reason to perform a sensitivity analysis, was to check the performance of the heuristic on different scenarios. It turns out that all 72 tested scenarios result in a difference of less than 0.5% between the heuristic and the optimal solution. In 63 scenarios the difference is even less than 0.05%. So we can conclude that the heuristic performs well on all different scenarios.

6.7 System limit

In CEGOIA it is also possible to set a limit on the number of dwellings using a certain heat system. For instance, at most 50% of the dwellings can use a hybrid heatpump. CE Delft has been working on one problem with 910 neighbourhoods that has such limits. In these problems the limits (as a percentage of the total number of dwellings) are as follows

System	Limit (%)
Green gas	15
Heat exchanger LT or MT	40
Heat exchanger HT	20
Electric heatpump	10
Condensation boiler	45

The heuristic has been tested on this problem and a feasible solution has been found with total cost 1655 million. The total cost of the optimal solution is 1641 million, so the deviation from the optimal solution is a bit more than 0.5%. This is a larger deviation than for the problems in Table 4. However, it is still quite small and it is useful to know that the heuristic can even solve problems with these limits. Note that in this problem the sum of all limits is more than 100%. This means there is a flexibility that is probably needed to find a feasible solution. The expectation is that the closer the sum of the limits gets to 100%, the smaller the probability that a feasible solution is found.

7 Alternative methods

The results of the AGAP heuristic meet all requirements set in Chapter 4, but other methods have also been tested. These methods and their results are described in this chapter.

7.1 The LP-relaxation

A way to gain more knowledge about the linear programming problem and to find a lowerbound for the problem is to solve the LP-relaxation. This means that the integrality constraint 5 is changed into $x_{ij} \geq 0$ and $x_{ij} \leq 1$ for all $i \in M, j \in N$. In other words, the variables do not have to be integer anymore. However, solving the LP-relaxation still takes a long time, sometimes even longer than the original problem. An explanation for this, is that the solvers used for the IP-problem have special tricks for IP-problems that result in a faster computation. Therefore other solvers have been tested on the LP-relaxation, but none of them found a solution to the large problems (more than 1000 neighbourhoods) within a reasonable time.

7.2 Alternatives within the heuristic

In the AGAP heuristic and the AGAP reversed heuristic, a formula is used to decide which reallocation to perform. For both algorithms some alternative formulas have been tested. Furthermore, as mentioned in Paragraph 5.1, step 3 of the AGAP heuristic uses the sum of the constraint coefficients, $a^{(\text{tot})}$. The same algorithm has also been tested with only the coefficients of the exceeded constraint instead of the sum. AGAP reversed has also been tested with only the coefficients of the exceeded constraint in the formula instead of $a^{(\text{tot})}$. However, for both the AGAP heuristic and AGAP reversed, the current method led to the best results and has therefore been used for CEGOIA.

Furthermore, in AGAP reversed, a different method has been tested in which the slack of each constraint is considered to decide which reallocation to perform. This means that after the AGAP heuristic, for all constraints the used amount of resource has been substracted from the limit. The result of this computation is called the slack of the constraint and this slack shows how much of the resource is still available. Then, in the formula in step 2 of AGAP reversed, different formulas have been tested in which options using resources with large slack are more likely to be selected than options using resources with a low slack. However, this method also led to worse results than the method that was eventually used in CEGOIA.

7.3 Solvers

To compute the optimal solution, CE Delft has been using the CBC and GLPK_MI solvers for small problems and CPLEX for larger problems. These solvers are part of CVXPY (see [4]), a modeling language for optimization problems. There are a number of solvers that can be called by CVXPY and some of them have been tested on the IP problem in CEGOIA. The current solvers led to the best results.

In [2] (Chapter 11) and [5], column generation is mentioned as a solving method for large linear programming problems. To implement this method independently in CEGOIA is not practical, as it is very complex. However, column generation can already be used in CPLEX. As large

problems, for instance the Netherlands in its entirety, cannot be solved using CPLEX, a heuristic has more chance of success than column generation.

8 Discussion

After analyzing the results of the heuristic, we can conclude that a major improvement has been made. It is now possible to run CEGOIA on the Netherlands in its entirety, which has been the main purpose of this project and which is something that can be very useful for CE Delft. However, the heuristic does not lead to the optimal solution, but an approximation of the optimal solution. Although the deviation from the optimal cost has never exceeded 0.5 percent of the total cost, there is still a risk that for the Netherlands in its entirety, the deviation is larger. An attempt to improve the results of the heuristic has already been made in the multiple step improvement. However, this improvement increased the runtime significantly and has therefore not been used on very large problems. This means that in further research to find solutions closer to the optimum, a different approach is probably necessary.

For CE Delft, it is important to know when to use the heuristic. When a problem is small, and the optimal solution can therefore been found in a reasonable time, there is no use in running CEGOIA with the heuristic. However, on a large problem with many neighbourhoods, the heuristic can be used to find a good solution. Also, the heuristic can be used to perform an analysis on different scenarios of a large problem. Of course, there usually still is a small deviation from the true optimal solution. This is generally not a problem, as the solution found in CEGOIA is not directly interpreted as a blue print for which energy system should be used in which neighbourhood. For small areas this sometimes is the case, but for large areas, CEGOIA is used more as a general idea for the distribution of systems. This means that the goal of running CEGOIA on large areas is to decide roughly where the different systems are used. Not on a neighbourhood level, but more on a town level or even a larger area. This then gives insight in, for instance, where the green gas should be supplied. It is very useful for CE Delft to be able to answer such questions.

9 References

[1] Robin Lougee-Heimer John Forrest. Cbc user guide, 2005. https://www.coin-or.org/Cbc/cbcuserguide.html#heuristics.

[2] M. Schouten-Straatman L.C.M. Kallenberg. *Geavanceerde onderwerpen in de besliskunde*. Universiteit Leiden, 2019.

[3] Luk N. Van Wassenhove Marshall L. Fisher, R. Jaikumar. A multiplier adjustment method for the generalized assignment problem. *Management science*, 32(9), 1986.

[4] Stephen Boyd Steven Diamond. Cvxpy user guide, 2016. https://www.cvxpy.org/tutorial.html.

[5] Thomas Stützle Vittorio Maniezzo, Marco Antonio Boschetti. *Matheuristics: Algorithms and Implementations*. Springer, 2021.

[6] John M. Wilson. A simple dual algorithm for the generalized assignment problem. *Journal of Heuristics*, 2:303–311, 1997.

Figure 5: See Donald Duck nr. 5 2022

Appendix A

This appendix contains the code of the implementation of the heuristic in CEGOIA. At the start of this graduation project, there was already a Python-based model for CEGOIA. All code shown in this chapter is self developed and added to the existing CEGOIA-model. For the implementation of the heuristic, one function and one file were added. The function, heuristic_solve, is added to call the different algorithms: AGAP, AGAP reversed and possibly the multiple step improvement. These algorithms are called Wilson and Wilson reversed in the programming code. The added file, heuristic.py, then contains the code of the seperate algorithms.

The Python-code for the function is as follows.

```
# Input: characteristics of optimization problem (constraints, limits and costs).
# Computes an approximation of the optimal solution using the Wilson heuristic, which consists of
# Wilson, Wilson reversed and, if enabled, the multiple-step improvement.
# Output: a feasible solution (result matrix, selected options) and total costs
def heuristic_solve(all_constraints_unsorted, constraint_limits_unsorted, cost, neighbourhood_options_matrix):
    ce_log("start heuristic solve")
    start_timestamp = time.time()
    neighbourhood_options_ids_selected = []
    costs_no_constraints = np.sum(cost[0, :])
    # sort all_constraints_unsorted such in order of exceedance
    all_constraints = []
    constraint_limits = []
    number_of_constraints = len(constraint_limits_unsorted)
    slack = np.zeros(number_of_constraints)
    for i in range(0, number_of_constraints):
        slack[i] = constraint_limits_unsorted[i] - np.sum(all_constraints_unsorted[i][0, :])
    slack_order = np.argsort(slack)
    for i in range(0, number_of_constraints):
        all_constraints.append(all_constraints_unsorted[slack_order[i]])
        constraint_limits.append(constraint_limits_unsorted[slack_order[i]])
    # start algorithm
    result_matrix = wilson(all_constraints, constraint_limits, cost)  # Wilson heuristic
    result_matrix = wilson_reversed(result_matrix, all_constraints, constraint_limits, cost, 1)  # Wilson reversed
    # the multiple-step improvement (meerstapsverbetering): increases running time significantly, can be commented
    # result_matrix = multiple_step_improve(result_matrix, all_constraints, constraint_limits, cost)
    # end algorithm
    end_timestamp = time.time()
    diff_time = end_timestamp - start_timestamp
    ce_log('end heuristic solve in ' + str(diff_time) + ' seconds')
    chosen_options_costs = np.multiply(result_matrix, cost)
    total_costs = int(round(chosen_options_costs.sum()))
```

```
32      rows = result_matrix.shape[0]
33      cols = result_matrix.shape[1]
34      # # print per column the row of the chosen option
35      # for column in range(0,cols):
36      #     for row in range(0,rows):
37      #         if result_matrix[row][column] == 1:
38      #             print("(" + str(row) + ", " + str(column) + ")")
39      # write result matrix to file result_matrix.txt
40      with open('result_matrix_heur.txt', 'w') as testfile:
41          for row in result_matrix:
42              testfile.write(' '.join([str(a) for a in row]) + '\n')
43      ce_log('RESULT MATRIX')
44      print(result_matrix)
45      # ce_log(result_matrix)
46
47      # Format result_matrix
48      #  [
49      #     [buurt1optie1    buurt2optie1]
50      #     [buurt1optie2    buurt2optie2]
51      # ]
52
53      for option_row_count, option_row in enumerate(range(0, rows)):
54          for neighbourhood_column_count, neighbourhood_column in enumerate(
                range(0, cols)):
55              if result_matrix[option_row, neighbourhood_column] == 1:
56                  neighbourhood_options_id = neighbourhood_options_matrix[
                        neighbourhood_column_count][
57                      option_row_count]
58                  neighbourhood_options_ids_selected.append(neighbourhood_
                        options_id)
59
60      ce_log('result count ' + str(len(neighbourhood_options_ids_selected)))
61      print('total costs {}, neighbourhood_options_ids_selected {}'.format(
            total_costs, neighbourhood_options_ids_selected))
62
63      return neighbourhood_options_ids_selected, total_costs, result_matrix
```

The Python-code for the file is as follows (above each function, a block of comments explains what that function does).

```
1  # Checks if current solution (result_matrix) satisfies all constraints.
2  # Output: 0 if solution is feasible, index of exceeded constraint if
       solution is unfeasible
3  def check_feasibility(all_constraints, constraint_limits, result_matrix,
       cost):  # returns rows and columns that cause the constraints to be
       exceeded
4      number_of_constraints = len(constraint_limits)
5      # check if constraints are met, if not: return index of constraint that
           is exceeded
6      for i in range(0, number_of_constraints):
7          matrix_product = np.multiply(all_constraints[i], result_matrix)
8          if matrix_product.sum() > constraint_limits[i]:
9              return np.nonzero(matrix_product)
10     return 0
```

```
11
12 # Returns a feasible solution (result_matrix) using matrix T that contains
       the relative gain of each option
13 def wilson(all_constraints, constraint_limits, cost):
14     ce_log("start wilson")
15     number_of_constraints = len(constraint_limits)
16     sum_constraints = all_constraints[0]  # matrix containing the sum of
           all constraint-matrices
17     for i in range(1, number_of_constraints):
18         sum_constraints = np.add(sum_constraints, all_constraints[i])
19     result_matrix = np.zeros([all_constraints[0].shape[0], all_constraints
           [0].shape[1]], dtype=np.int)
20     rows = result_matrix.shape[0]
21     cols = result_matrix.shape[1]
22     # create initial relaxation, columns of cost matrix must be sorted
           ascendingly!
23     for i in range(0, cols):
24         result_matrix[0, i] = 1
25     # fill T (proportional gain matrix) for the first time
26     T = np.zeros([rows, cols])
27     for j in range(0, cols):
28         for i in range(1, rows):
29             if cost[0][j] == cost[i][j]:
30                 if sum_constraints[0][j] - sum_constraints[i][j] >= 0:
31                     T[i][j] = 0
32                 else:
33                     T[i][j] = -99
34             else:
35                 T[i][j] = (sum_constraints[0][j] - sum_constraints[i][j]) /
                       (-cost[0][j] + cost[i][j])
36     # while no feasible solution: reallocate
37     feasible = False
38     while not feasible:
39         slack = np.zeros(number_of_constraints, dtype=float)
40         for i in range(0, number_of_constraints):
41             matrix_product = np.multiply(all_constraints[i], result_matrix)
42             slack[i] = constraint_limits[i] - matrix_product.sum()
43         constr_exceed = check_feasibility(all_constraints, constraint_
               limits, result_matrix, cost)
44         if constr_exceed == 0:  # solution is feasible
45             feasible = True
46         else:
47             if np.all(T <= 0):  # if all elements in T are negative, the
                   problem is infeasible
48                 print("Problem is infeasible!")
49                 return 0
50             else:  # find the best reallocation
51                 max_in_T = 0
52                 for j in range(0, cols):  # find max T over all columns
                       where constraint is exceeded
53                     if j in constr_exceed[1]:
54                         max_in_column = np.max(T[:, j])
55                         if max_in_column > max_in_T:
```

```
56                              max_in_T = max_in_column
57                              max_row = np.argmax(T[:, j])
58                              max_col = j
59                  # reallocate
60                  for i in range(0, rows):
61                      if result_matrix[i][max_col] == 1:
62                          result_matrix[i][max_col] = 0
63                  result_matrix[max_row][max_col] = 1
64                  # update T (cost matrix must be sorted!)
65                  for i in range(0, max_row + 1):  # after reallocation don't
                        compute t again for cheaper options
66                      T[i][max_col] = -99
67                  for i in range(max_row + 1, rows):
68                      if T[i][max_col] != -99:
69                          if cost[max_row][max_col] == cost[i][max_col]:
70                              if sum_constraints[max_row][max_col] - sum_
                                    constraints[i][max_col] >= 0:
71                                  T[i][max_col] = 0
72                              else:
73                                  T[i][max_col] = -99
74                          else:
75                              T[i][max_col] = (sum_constraints[max_row][max_
                                    col] - sum_constraints[i][max_col]) / (
76                                          -cost[max_row][max_col] + cost[i][
                                              max_col])
77      return result_matrix
78
79  # Improves solution obtained by function "wilson" (result_matrix) using
        matrix better_options (S in report).
80  # Output: improved result_matrix
81  def wilson_reversed(result_matrix, all_constraints, constraint_limits, cost
        , fraction):
82      ce_log("start wilson_reversed")
83      rows = result_matrix.shape[0]
84      cols = result_matrix.shape[1]
85      number_of_constraints = len(constraint_limits)
86      sum_constraints = all_constraints[0]
87      for i in range(1, number_of_constraints):
88          sum_constraints = np.add(sum_constraints, all_constraints[i])
89      # fill chosen_rows with index of the chosen row for each column
90      chosen_rows = np.zeros(cols, dtype=np.int)
91      for j in range(0, cols):
92          for i in range(0, rows):
93              if result_matrix[i, j] == 1:
94                  chosen_rows[j] = i
95      # fill better_options with relative gain for the first time
96      better_options = np.zeros([rows, cols], dtype=float)
97      for j in range(0, cols):
98          i = 0
99          while result_matrix[i, j] != 1:
100             if sum_constraints[chosen_rows[j], j] == sum_constraints[i, j]:
101                 better_options[i, j] = 99
102             else:
```

```
103                 if fraction:
104                     better_options[i, j] = (-cost[chosen_rows[j], j] + cost
                            [i, j]) / (
105                             sum_constraints[chosen_rows[j], j] - sum_
                                constraints[i, j])
106                 else:
107                     better_options[i, j] = -cost[chosen_rows[j], j] + cost[
                            i, j]
108             i += 1
109     finished = False
110     while not finished:
111         # compute slack for each constraint (how much of each constraint is
                still available)
112         slack = np.zeros(number_of_constraints, dtype=float)
113         for i in range(0, number_of_constraints):
114             matrix_product = np.multiply(all_constraints[i], result_matrix)
115             slack[i] = constraint_limits[i] - matrix_product.sum()
116         # try to reallocate
117         reallocation_feasible = False
118         while not reallocation_feasible:
119             # find maximal s
120             max_in_better_options = np.argmax(better_options)
121             max_row = math.floor(max_in_better_options / cols)
122             max_col = max_in_better_options % cols
123             # if there are no positive s: stop
124             if better_options[max_row, max_col] == 0:
125                 finished = True
126                 reallocation_feasible = True
127             else:
128                 result_matrix[max_row, max_col] = 1  # try reallocation
129                 result_matrix[chosen_rows[max_col], max_col] = 0
130                 check_constraints = True
131                 for constraint in range(0, number_of_constraints):
132                     constraint_difference = all_constraints[constraint][max
                            _row, max_col] - all_constraints[constraint][
133                             chosen_rows[max_col], max_col]
134                     if constraint_difference > slack[constraint]:
135                         check_constraints = False
136                 if not check_constraints:  # not feasible -> undo
                        reallocation
137                     better_options[max_row, max_col] = 0
138                     result_matrix[max_row, max_col] = 0
139                     result_matrix[chosen_rows[max_col], max_col] = 1
140                 else:
141                     # update chosen_rows
142                     chosen_rows[max_col] = max_row
143                     # update better_options
144                     for i in range(0, max_row):
145                         if sum_constraints[max_row, max_col] == sum_
                                constraints[i, max_col]:
146                             better_options[i, max_col] = 99
147                         else:
148                             if fraction:
```

```
149                                 better_options[i, max_col] = (-cost[max_row
                                        , max_col] + cost[i, max_col]) / (
150                                                 sum_constraints[max_row, max_
                                                    col] - sum_constraints[i,
                                                    max_col])
151                             else:
152                                 better_options[i, max_col] = -cost[max_row,
                                        max_col] + cost[i, max_col]
153                     for i in range(max_row, rows):
154                         better_options[i, max_col] = 0
155                     reallocation_feasible = True
156     return result_matrix
157
158 # Improves solution obtained by "wilson" and "wilson_reversed" (result_
        matrix). Tests different reallocations
159 # and accepts them if total costs are reduced. For large projects this can
        take a long running time with no large
160 # improvement, so this function is usually turned off. Output: improved
        result_matrix
161 def multiple_step_improve(result_matrix, all_constraints, constraint_limits
        , cost):
162     ce_log("start multiple step improvement")
163     rows = result_matrix.shape[0]
164     cols = result_matrix.shape[1]
165     cheap_rows_range = 2  # number of tested cheap rows
166     exp_rows_range = 2  # number of tested expensive rows
167     number_of_constraints = len(constraint_limits)
168     sum_constraints = all_constraints[0]
169     for i in range(1, number_of_constraints):
170         sum_constraints = np.add(sum_constraints, all_constraints[i])
171     chosen_options_costs = np.multiply(result_matrix, cost)
172     best_total_costs = int(round(chosen_options_costs.sum()))
173     best_result_matrix = copy.deepcopy(result_matrix)
174     for j in range(0, cols):  # for each column test more expensive option
175         chosen_rows = np.zeros(cols, dtype=np.int)
176         # fill chosen_rows with index of the chosen row for each column
177         for k in range(0, cols):
178             for i in range(0, rows):
179                 if result_matrix[i, k] == 1:
180                     chosen_rows[k] = i
181         # make copy of chosen_rows that is sorted in descending order
182         chosen_rows_sorted = copy.deepcopy(chosen_rows)
183         chosen_rows_sorted.sort()
184         chosen_rows_sorted = chosen_rows_sorted[::-1]
185         # if exp_rows_range = x, we test row indices of the first x+1
            elements of chosen_rows_sorted
186         if chosen_rows[j] <= cheap_rows_range:  # if cheap option has been
            chosen -> try more expensive option
187             for new_row in range(0, exp_rows_range+1):
188                 # try reallocation
189                 result_matrix[chosen_rows[j], j] = 0
190                 result_matrix[chosen_rows_sorted[new_row], j] = 1
191                 # check feasibility
```

```
192                 constr_exceed = check_feasibility(all_constraints,
                        constraint_limits, result_matrix, cost)
193                 if constr_exceed == 0:  # feasible -> do Wilson reversed
194                     result_matrix = wilson_reversed(result_matrix, all_
                            constraints, constraint_limits, cost, 1)
195                     chosen_options_costs = np.multiply(result_matrix, cost)
196                     total_costs = int(round(chosen_options_costs.sum()))
197                     if total_costs < best_total_costs:  # better solution
198                         best_result_matrix = copy.deepcopy(result_matrix)
199                         best_total_costs = total_costs
200                     else:  # worse solution -> make reallocation undone
201                         result_matrix = copy.deepcopy(best_result_matrix)
202                 else:  # unfeasible solution -> make reallocation undone
203                     result_matrix = copy.deepcopy(best_result_matrix)
204     return best_result_matrix
```