university of groningen

faculty of mathematics and natural sciences

Heuristics for the Quadratic Assignment Problem

Willemieke van Vliet

Bachelor Thesis in Mathematics

July 2009

Heuristics for the Quadratic Assignment Problem

Summary

The quadratic assignment problem we can see as a facility location problem. Assume we have n facilities and n locations. We know the flows between each pair of facilities and the distances between each pair of locations. To each location we want to assign a facility such that the distance times the flow is minimized.

The quadratic assignment problem is an NP-hard optimization problem. This means that there is no algorithm that can solve this problem in polynomial time. Heuristics are algorithms which try to find the optimal solution or a solution that is close to the optimal solution. There are a lot heuristics for the quadratic assignment problem and in my thesis I discuss a few of them. Further I compare a new heuristic of Zvi Drezner with another heuristic, tabu search.

Bachelor Thesis in Mathematics
Author: Willemieke van Vliet
Supervisor: Dr. Mirjam Dür
Date: July 2009

Institute of Mathematics and Computing Science
P.O. Box 407
9700 AK Groningen
The Netherlands

Contents

Chapter 1

The Quadratic Assignment Problem

The quadratic assignment problem (QAP) is an interesting combinatorial optimization problem. Koopmans and Beckmann [14] introduced this problem in 1957 as an economic location problem. But nowadays the QAP has also a lot of applications in other fields and there are many real life problems which can be modeled by QAP's. Moreover, many other combinatorial problems, such as the travelling salesman problem, can be formulated as a QAP.

Furthermore, the QAP is one of the great challenges in combinatorial optimization. This is because the problem is very hard to solve and also hard to approximate.

This chapter follows the outline of [4, 2]

1.1 Problem Formulation

We can describe the QAP mathematically as follows. There are n facilities and n locations. We denote the distance between location i and location j as d_{ij} and the flow or cost between facility i and facility j as c_{ij}. To each location we will assign a facility such that the distance times the flow, this we call the total costs, are minimized. So the QAP is to find a permutation p of the set of facilities which minimizes the objective function (1.1).

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} d_{p(i)p(j)} \tag{1.1}$$

Here is $p(i)$ the location of facility i.

An alternative way of formulating the QAP is the Koopmans-Beckmann formulation. This formulation is more useful than the formulation presented here for some solution methods. Koopmans and Beckmann define for this new formulation the matrix X. X is an $n \times n$ matrix and its entries fulfill the following conditions.

$$\begin{aligned} \sum_{i=1}^{n} x_{ij} &= 1, \qquad 1 \le j \le n \\ \sum_{j=1}^{n} x_{ij} &= 1, \qquad 1 \le i \le n \\ x_{ij} &\in \{0,1\}, \qquad 1 \le i,j \le n \end{aligned} \tag{1.2}$$

Now we can reformulate (1.1) as

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} c_{ij}d_{kl}x_{ik}x_{jl}, \tag{1.3}$$

where c_{ij} are still the costs between facility i and facility j and d_{ij} the distances between location i and location j. Here the elements of matrix X satisfy the conditions of (1.2) and

$$x_{ik} = \begin{cases} 1 & \text{if facility } i \text{ is located at } k. \\ 0 & \text{otherwise} \end{cases}$$

It is our goal to minimize (1.3) with respect to x_{ik} and x_{jl}.

1.2 Applications

The QAP has applications in many fields. The major application is the QAP as facility location problem, but a few other examples are scheduling, wiring problems in electronics, transportation and design of controlpanels and type writer keyboards. We illustrate two of these examples of the QAP.

Hospital Layout[6]

Hospital layout is an example of a facility location problem. In a hospital there are a number of different departments and some patients have to travel from one department to another. The QAP in this case is to minimize the total distance travelled by patients in a year.

We will model this problem as follows. We have n locations and n facilities, in this case the facilities are the different departments, and we will place one facility at each location. We know the yearly flow f_{ik} of patients between facility i and k and the distance d_{jq} between the locations j and q. We define I as the set of facilities and J as the set of locations. We can now formulate the problem as follows.

$$\begin{aligned} \text{Minimize} \quad & \sum_{i,j}\sum_{k,q} f_{ik}d_{jq}y_{ij}y_{kq} \\ \text{subject to} \quad & \sum_{j\in J} y_{ij} = 1 \qquad \forall i \in I \\ & \sum_{i\in I} y_{ij} = 1 \qquad \forall j \in J \\ & y_{ij} = \begin{cases} 1 & \text{if facility } i \text{ is located at } j. \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{1.4}$$

The hospital layout-problem is now formulated as the Koopmans-Beckmann formulation. So we can see (1.4) as a QAP.

Design of Type Writer Keyboards [17]

The design of type writer keyboards is still the same since 1873. We will allocate the letters to keys in such a way that the typing time is minimal.

In this example the letters are the facilities and the keys of the keyboard are the locations. We define t_{kl} as the elapsed time between typing two letters, indexed by k and l. This time depends only on the previous letter pressed, here letter k, and it is independent of the letters which are pressed before pressing letter k. Further, f_{ij} is the relative frequency of the i-th j-th letter pair and $\varphi(i)$ denotes the key with letter i. So the objective function we want to minimize with respect to φ is

$$\sum_k \sum_l t_{kl} f_{\varphi(k)\varphi(l)}. \tag{1.5}$$

As we can see, this problem is now formulated in the same way as a QAP in formulation (1.1).

Many other combinatorial problems can be formulated as QAP's. For example the travelling salesman problem and the maximum clique problem. We will now illustrate how we can see the travelling salesman-problem as a QAP.

Travelling Salesman Problem Formulated as a QAP

There are n cities and we know the distances between each of these cities. In the travelling salesman problem we want to visit each of these cities once, but in the mean time we want to minimize the travel-distance.

We will formulate this problem as a QAP. Therefore we will see the travelling salesman problem as a facility location problem. The cities we want to visit are now the locations and our facilities are the numbers 1 till n. We assign to each city a number, a facility, and these numbers represent when we visit this city. For example, when a particular city gets number 2, this city will be the second location we visit. We define c_{ij} as follows

$$c_{ij} = \begin{cases} 1 & \text{if } i = j - 1 \\ 0 & \text{otherwise} \end{cases}$$

So $c_{ij} = 1$ if we travel from city i to j, because then $i = j - 1$, and otherwise zero. We define the distances between the cities i and j as $d_{p(i)p(j)}$, where $p(i)$ is the location of number i. This means the i-th city we visit, is city $p(i)$. In this way we can see the travelling salesman problem as a QAP.

1.3 Computational Complexity

The QAP is one of the most difficult combinatorial optimization problems and in 1976 Sahi and Gonzalez [19] proved that the QAP is strongly NP-hard. This means that there is no algorithm for solving the QAP in polynomial time. An algorithm can solve a problem in polynomial time, when its runningtime is bounded by a polynomial function of the problem size. It is when the execution time is $O(n^k)$, where k is a constant.

Furthermore Sahi and Gonzalez [19] proved that the QAP is also NP-hard to approxiate. Problems of size larger than about 20 can generally not be solved to optimality in reasonable time. Also problems bigger than something like $n = 30$ are very hard to approximate.

1.4 Heuristics

Exact algorithms are used to solve optimization problems to optimality. But exact algorithms can only solve small problems and even for these small problems the most exact algorithms have long runningtimes. Therefore we look at heuristics for QAP's. Heuristics are algorithms which try to find a solution that is the optimal or is at least close to the optimal solution. The QAP is NP-hard and heuristics are needed in order to find an approximate solution for this problem in reasonable time. Because of the need for heuristics, there is a lot of research to find new heuristics for solving the QAP. We describe here a few heuristics.

1.4.1 Construction Methods

Construction methods are the oldest heuristics for QAP's and were introduced by Gilmore [10] in 1962. Construction methods are relatively simple to implement and have a short running time, but unfortunately they have often poor results.

Construction methods start with an empty solution and iteratively we approach the optimal solution. A not yet located facility is assigned to a not yet occupied location. There are different rules which can be used for choosing an assignment and a location. For example we can use a local view-rule. This rule selects a facility i which has maximum costs with an already placed facility j. This facility is assigned to a location such that $c_{ij}d_{p(i)p(j)}$ is minimized. Another example is a global view-rule. Which facility and which location are selected here depends not only on the already located facilities, but also on the other facilities.

An example of a construction method is the CRAFT [1], this heuristic is one of the oldest heuristics in use. Another construction method that gets relatively good results is the method of the increasing degree of freedoms [16].

1.4.2 Tabu Search Algorithms

In 1989 Tabu search was introduced by Glover [11, 12] as a local search technique for combinatorial optimization problems. Therefore Tabu search algorithms are not only heuristics for QAP's, but tabu search is also used for other hard combinatorial optimization problems.

Tabu search makes use of the neighbourhood structure of solutions. We will call the operation, which gives a neighbour of a solution, a move. In the case of QAP's a move is a pair exchange. The basic idea of tabu search is to remember which solutions already have been visited, therefore we make use of a tabu list. The tabu list is a list of forbidden moves. This list avoids cycling, which means that one or more solutions are visited more than once. During the iterations the tabu list is updated, some moves come in the list. Some moves are also cancelled from the list, because otherwise the length of the list becomes too large. Which moves are cancelled from the list is determined in different ways. For example we can use the first-in-first-out rule.

Tabu search starts with an initial solution and we call this solution the current solution. During an iteration we search for a best-quality solution. We look at all neighbours of the current solution, except the ones which can not be reached, because its move is on the tabu list. The best-quality solution is now the neighbour-solution with the best objective value. This best-quality solution becomes the current solution, the tabu list is updated and the iteration starts again. The best-quality solution is not always better than the current solution. The

algorithm stops when the stop criterion is satisfies, often this stop criterion is a maximum running time or a maximum number of iterations.

1.4.3 Simulated Annealing Approaches

Simulated annealing approaches are based on the analogy between combinatorial optimization problems and many particle physic systems. We can see the feasible solutions of the optimization problem as the states of the physical system and the objective function values correspond to the energy of the states of this systems. In the physic model we want a low energy state, so we want to minimize the energy.

We have a solid in a heath bath and we will use condensed matter physics annealing to get this solid in a low energy state. This method consist of two phases. First, the temperature of the bath is increased to a maximum value and secondly the temperature is carefully decreased and we get the solid in the ground state, the lowest energy state. Simulated annealing process is like this process. We want to decrease the objective value very slowly and carefully.

Let E_i be the energy level of the current state i, this is the same as the objective value of the current solution i. With the neighbourhoodstructure of QAP's we find a new state j, the energy level of this state we call E_j. If $E_j - E_i$ is negative, j becomes our new current state. Otherwise there is a probability of $\exp \dfrac{E_i - E_j}{K_B t}$ that j is accepted as the current state. Here K_B is the Boltzmann constant and t the temperature. If j does not become the current state then i is still the current state and we will look at another neighbour of i.

Here we see that if the solution j is better than the current solution, we replace the current solution by solution j. In this way we get a lower current objective value, that is a energy state. But if the solution j is not better than the current solution there is still a change that the center solution is updated. If this happens the current objective value increases, but this is not bad. Because if we sometimes let the current objective value increase a bit, the probability to get stuck in a local minimum with a poor quality becomes smaller. That we some times replace the current solution by a worse one is part of carefully decreasing of energy.

Simulated annealing approaches are usefull for a lot of optimization problems and for the QAP.

1.4.4 Genetic Algorithms

The first genetic algorithm was applied to optimization problems by Holland [13] in 1975. Because genetic algorithms can be used for a lot of optimization problems, there is still a lot of research on these algorithms. We will see that genetic algorithms are inspired by nature. The algorithm makes use of evolution mechanisms. We look at some feasible solutions as individuals and they form together a population. In the population pairs of individuals get children, new solutions, and these new solutions are applied to the population.

The genetic algorithm starts with a set of initial solutions, we will call this set the initial population. We consider a solution in the current population as a member or an individual. Like in nature, we select a pair of individuals from the current population and produce with this pair of solutions a new solution. We can see now the pair of individuals as the parents and the new solution as the child. For producing the new solution we use cross-over-rules. Bad solutions or weak individuals are eliminated and so we get a new current population with the best children and the best parents. This process is repeated until the stop criterion

is satisfied. The stop criterion is often a time limit, limit of number of iterations or if the population consist only of equal solutions or solutions with a small difference.

During the process a mutation or immigration is applied periodically. Because of this the algorithm gets better solutions faster. Mutation means that some individuals are modified and immigration is that a few new random individuals enter the current population.

1.4.5 Greedy Randomized Search

The greedy randomised adaptive search procedure, also called GRASP, was proposed by Feo and Resende [7]. The GRASP is a heuristic for hard combinatorial optimization problems and was in 1994 applied to the QAP by Li, Pardalos and Resende [15]. Some implementations of GRASP yield best known solutions for most problems of QAPLIB [3] and even improves the best known solution for a few problems.

GRASP is a two phase heuristic. The first phase is a construction phase. In this phase we construct good solutions and in the second phase, the local improvement phase, we search for better solutions in the neighbourhood of these solutions.

In the construction phase we assign two facilities i_0, j_0 to two locations k_0, l_0. We choose this pair of assignments with the help of a greedy component. This greedy component chooses these pairs in such a way that

$$c_{i_0 j_0} d_{k_0 l_0} = \min\{c_{ij} d_{kl} : i, j, k, l \in \{1, 2, ..., n\}, i \neq j, k \neq l\},$$

where matrix C is the cost matrix, matrix D the distance matrix and n the size of the problem. But in this way we do not have freedom in the search procedure and we can easily get trapped in a locally optimal solution with poor quality. Therefore we choose the pair of facilities i_0, j_0 and the locations k_0, l_0 a little different. Besides the greedy component we make use of random elements. Therefore we define $r = \lfloor \beta(n^2 - n) \rfloor$, where $0 < \beta < 1$ is a control parameter and n the size of the problem. Then we look at the non-diagonal entries of the cost matrix C and sort the r smallest entries non-decreasingly. So we get:

$$c_{i_1 j_1} \leq c_{i_2 j_2} \leq ... \leq c_{i_r j_r}$$

in the same way, we take the non-diagonal entries of the distance matrix D and sorted the r largest entries non-increasingly. Here we get:

$$d_{i_1 j_1} \geq d_{i_2 j_2} \geq ... \geq d_{i_r j_r}$$

We define $r' = \lfloor \alpha\beta(n^2 - n) \rfloor$, where $0 < \alpha < 1$ is again a control parameter. Next we sort the costs of the possible pairs of assignments, $c_{i_1 j_1} d_{k_1 l_1}$, $c_{i_2 j_2} d_{k_2 l_2}$, ..., $c_{i_r j_r} d_{k_r l_r}$, non-decreasingly. Now we only take the smallest r' costs of the possible pairs of assignment in account and choose out of these pairs randomly the facilities i_0, j_0 and the locations k_0, l_0.

In the next steps of the construction phase we assign the remaining facilities to the remaining locations, one facility is assigned to one location at a time. We make use of the intermediate costs $a_{jl} = \sum_{(i,k)\in\Gamma} (c_{ij} d_{kl} + c_{ji} d_{lk})$ for choosing a facility j and a location l, where Γ is the set of already assigned pairs of assignments. If there are m pairs (i, j) of unassigned facilities and locations, we select one pair randomly from the $\lfloor \alpha m \rfloor$ smallest intermediate costs a_{jl}. This is repeated until all facilities are assigned to a location.

The local improvement phase searches in the neighbourhood of the solution constructed in phase one for possible improvements. This phase consists of a standard local search algorithm. These two phases are repeated a certain number of times.

1.4.6 Ant Systems

The last heuristic we discuss is ant systems. This heuristic is recently developed and has produced good results for well known problems like the travelling salesman problem and the QAP.

An ant system is, like the name says, based on the behaviour of an ant colony in search for food. The ants of the ant colony first randomly search for food in the neighbourhood of the nest. If an ant finds a source of food, she takes a bit of it and takes it back to the ant nest. The way back to the nest she leaves a trail of pheromones. With this trail it is possible to find the source again during a future search. The intensity of pheromones on the trail is related to the number of ants who visited the food source and took food from it. So the intensity of pheromones is also proportionally related to the amount of food in the food source.

Now we will imitate the behaviour of the ants. First we call the set of feasible solutions the area searched by the ants. Further the amount of food in a food source is the value of the objective function and the pheromone trail is a component of adaptive memory.

To illustrate the idea of ant systems, applied to a QAP, we make use of the algorithm of Gambardella, Taillard and Dorigo [9]. The algorithm is iterative and in each iteration we compute m solutions. Here m is the number of ants searching for food and so m is a fixed control parameter. In the beginning the ants search randomly for food, so the first m solutions are randomly generated. If the ants found food they leave a pheromone trail. We translate this pheromone trail in the matrix $T = \tau_{ij}$. When ants generated solutions we can see τ_{ij} as the measure for the desirability of locating facility i at location j. Matrix T is in the beginning a constant matrix. This constant is proportional to the inverse of the best found solution so far.

During a next search for food the ants are influenced by the pheromone trails. So in the next iterations we compute m solutions, but this time we make first use of the matrix T and then we apply an improvement method. Further we update our matrix T after each iteration, because if many ants have visited a food source the intensity of the pheromone trail is increased. Therefore we define $\phi*$ as the best found solution so far and $f(\phi*)$ as its objective value. In the iterations the entries $\tau_{i\phi*(i)}$ of T are increased by a value which is proportional to $f(\phi*)$.

If there is no improvement of the best known solution after a lot of iterations, the whole algorithm starts again with m different random solutions.

Chapter 2

A New Heuristic for the Quadratic Assignment Problem

This chapter is based on the article 'A new heuristc for the quadratic assignment problem' of Zvi Drezner [5]. In his article, Zvi Drezner introduces a new heuristic for solving the quadratic assignment problem. In the first section of this chapter we describe the heuristic and in the second section we look at the short cuts in this algorithm. The results present in the article we discuss in the last section of this chapter.

2.1 The Algorithm

This heuristic starts with a start solution, we call this solution the center solution. Before we can discuss the algorithm of this heuristic we first have to define Δp as the distance between the center solution and the solution p. Here Δp is the number of facilities in p that are not in their center solution site or, equivalently, Δp is the number of components of p that differs from the components of the center solution. We call p a permutation of the center solution.

Δp has the following properties:

1. For all single pair exchanges of the center solution, that is if two facilities switch there locations, $\Delta p = 2$.

2. There are no permutations p with distance $\Delta p = 1$. Because there can not be just one facility out of place, there has to be at least a second facility out of place.

3. Let n be the length of a solution. Then $\Delta p \leq n$, because no more than n facilities can be out of place.

4. We apply an additional pair exchange on a permutation p, call this new permutation $p+$. Then $\Delta p+ = \Delta p + \Delta\Delta p$, where $\Delta\Delta p$ is the difference between Δp and $\Delta p+$. $\Delta\Delta p$ is between -2 and $+2$ and only the additional pair of exchanged facilities affect the value of $\Delta\Delta p$.

Description of One Iteration

The algorithm of this new heuristic is iterative. Therefore we describe first a whole iteration.

An iteration starts with a centersolution. We first search in the neighbourhood of this centersolution and if we did not find a better solution we look to solutions with a larger distance Δp from our centersolution. We increases this search-distance, δp, till the distance Δp is d, where $d \leq n$ is a parameter and we call d the depth of the search. If a better solution than the centersolution is found during the search we replace the center solution by this new found solution and the iteration starts from the beginning. If no better solution is found during the search, the iteration is complete.

When we increase the search-distance δp we do not want to look at all solutions with $\Delta p = \delta p$, because if we do that we will look at all solutions and that takes far to much time if the problem becomes large. Therefore we make use of three lists and a control parameter K. In these lists we put only the K best found solutions with the distances $\Delta p = \delta p$, $\Delta p = \delta p + 1$ and $\Delta p = \delta p + 2$ respectively and in the next step, when our search depth is δp, we only look at the pairexchanges of the solutions in the first list. If we do so, we look at solutions with distance Δp between $\delta p - 2$ and $\delta p + 2$. Here we look only at the solutions with Δp is $\delta p + 1$ or $\delta p + 2$ and not at all solutions.

Now we will describe this iteration step by step and in more detail.

1. We select a center solution.

2. Start with $\delta p = 0$. We make three lists with solutions, $list_0$, $list_1$ and $list_2$. The solutions in $list_0$ all have distance $\Delta p = \delta p$. The members of $list_1$ and $list_2$ have respectively distances $\Delta p = \delta p + 1$ and $\Delta p = \delta p + 2$. Because $\delta p = 0$, $list_0$ has just one member, the center solution. The two other lists are empty, because we did not find solutions with distances $\delta p + 1$ and $\delta p + 2$ yet. The best found solution is the new centersolution.

3. Go over all the solutions in $list_0$. For each solution of $list_0$ evaluate all pair exchanges for that solution. Note that if $list_0$ is empty, we can ignore the rest of Step 3 till Step 6 because there are no solutions in $list_0$ and so we can immediately go to Step 6.

   If the exchanged solution is better than the best found solution, that is when the objective value of the exchanged solution is less than the objective value of the best found solution, then this exchanged solution becomes our best found solution. We proceed to evaluate the rest of the exchanges, because there can be another permutation that is even better than this new best found solution and in that case we will again update the best found solution.

4. If in Step 3 a better best found solution is found by scanning all the exchanges of the solutions from $list_0$, set the center solution to the new best found solution and go to Step 2.

5. We go again over all the solutions in $list_0$. For each solution of $list_0$ we evaluate all pair exchanges for that solution and if the distance Δp is δp or lower, ignore the permutation and proceed the scan of the pair-exchanges of the solution in $list_0$.

   If its distance Δp is $\delta p + 1$ or $\delta p + 2$, we will decide whether this solution has to be in $list_1$, $list_2$ respectively. This is performed as follows:

   - We first check whether the list is shorter than K or the solution is better than the worst list member. If none of them is the case we just can ignore this permutation

and we can proceed with the next pair exchange of the solution from $list_0$. If at least one from above conditions is satisfies go to the next point.

- Then we look whether the solution is not identical to a list member. We do this by comparing its objective function value to those of list members, and only if it is the same as that of a list member the whole permutation is compared.
- If an identical list member is found, ignore the permutation and proceed with scanning the exchanges of permutation p.
- Otherwise,

  A list consists of maximally K solutions, where K is a parameter of the algorithm. So if the list is shorter than K the permutation is added to the list.

  If the list is of length K, the permutation replaces the worst member of the list.
- A new worst list member is identified.

6. Once we look at all pairexchanges of the solutions in $list_0$, we move $list_1$ to $list_0$, $list_2$ to $list_1$ and empty $list_2$.
7. Set $\delta p = \delta p + 1$.
8. If $\delta p = d + 1$, where d is the depth of the search, we stop with the iteration.
9. If $\delta p < d + 1$, our search is not depth enough yet, so we will return to Step 3.

We will apply short cuts to this iteration to make it faster. Later in this section we discuss these shortcuts.

Description of the Algorithm

This algorithm repeats the iteration several times, here we describe step by step how to do this.

1. We generate a start solution randomly. This solution is our center solution and it is also the best found solution.
2. Set a counter $c = 1$.
3. The depth $d \leq n$ of an iteration is recommend to be $d = n$ or very close to it, because then we have a depth search. Therefore we select d randomly in $[n - 4, n - 2]$. Now we perform an iteration on the center solution.
4. If the iteration improved the best found solution go to Step 2.
5. Otherwise, we increase the counter, so $c = c + 1$, and
   - If $c = 1, 3$ use the best found solution in the list with $\delta p = d$ as the new center solution. In the last iteration the list with $\delta p = d$ is the old $list_0$. Go to Step 3.
   - If $c = 2, 4$ use the best solution found throughout the last iteration, which is unequal to the old center solution and the best found solution so far, as the new center solution. Go to Step 3.
   - If $c = 5$ the whole algorithm is complete.

Background of the Heuristic

This heuristic applies concepts from tabu search and genetic algorithms. In tabu search we disallow backward tracking and so we force the search away from previous solutions. Tabu search makes therefore use of a tabu list. Zvi Drezner does not use such a list, but he uses in his heuristic the distance Δp as tabu mechanism. We do not look at solutions with distance $\Delta p \leq \delta p$. In this way we proceed farther and farther away from the center solution because Δp increases throughout the iteration. Like in tabu search we can only go back if a better solution is found.

The lists in the heuristc consist of a maximum of K members. We can consider the members of the list as the individuals of a population like in genetic algorithms. In genetic algorithms two individuals get children, with two solutions is another solution found, and the weak individuals or bad solutions are thrown out the population. In the new heuristic it goes a little different. Here a new solution comes just from one solution in the list instead of from two solutions, but like in genetic algorithms the bad solutions in the list are replaced by better ones.

2.2 Short Cuts

In the beginning of this chapter we described a few properties of Δp. The last property was that if we apply an additional pair exchange on a permutation p, call this new permutation $p+$, then $\Delta p+ = \Delta p + \Delta\Delta p$ and $\Delta\Delta p$ is between -2 and $+2$. Only the additional pair of exchanged facilities affect the value of $\Delta\Delta p$. Therefore if we do a pair exchange on permutation p, we can compute $\Delta p+$ in an easy way. If we calculate $\Delta p+$ with the definition of Δp we need a running time of $O(n)$, but if we use the above mentioned property we can do it in a runningtime of $O(1)$. If we apply this smart method for calculating $\Delta p+$ in Step 5 the algorithm becomes faster.

Most QAP's are symmetric, because usually the distance between location i and location j is the same as the distance between location j and location i. Also the costs between facilities i and j are the most of the time the same as the costs between facilities j and i. Furthermore the diagonal of the matrices D and C are mostly equal to zero, because the distance between identical locations is zero and the cost between a facility itself is also zero. Therefore we introduce here two shortcuts for symmetric QAP's with zero diagonal. Zvi Drezner uses this short cuts in his heuristic. These short cuts are also explained in [20]. Note that if this short cuts are implemented in the algorithm, the algorithm can only be used for symmetric problems with zero diagonal.

Define Δf_{rs} as the change in the value of the objective function f by exchanging the sites of facilities r and s. In his article Zvi Drezner writes that we can calculate Δf_{rs} in this way:

$$\begin{aligned}
\Delta f_{rs} &= 2\sum_{i=1}^{n} \{c_{ir}[d_{p(i)p(s)} - d_{p(i)p(r)}] + c_{is}[d_{p(i)p(r)} - d_{p(i)p(s)}]\} \\
&= 2\sum_{i=1}^{n} [c_{ir} - c_{is}][d_{p(i)p(s)} - d_{p(i)p(r)}]
\end{aligned}$$

But that is not correct, it has to be:

$$\begin{aligned}\Delta f_{rs} &= 2\sum_{i=1,i\neq r,s}^{n}\{c_{ir}[d_{p(i)p(s)}-d_{p(i)p(r)}]+c_{is}[d_{p(i)p(r)}-d_{p(i)p(s)}]\} \\ &= 2\sum_{i=1,i\neq r,s}^{n}[c_{ir}-c_{is}][d_{p(i)p(s)}-d_{p(i)p(r)}]\end{aligned} \tag{2.1}$$

We can verify this with equation (1.1). Say we have a solution p and we exchanged the pair rs in p and so we get p'. Now we want to calculate the difference between the objective value of p and the objective value of p'. We denote the objective value of p as f_p. $\Delta f_{rs} = f_{p'} - f_p$ and so we get with equation (1.1):

$$\Delta f_{rs} = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}d_{p'(i)p'(j)} - \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}d_{p(i)p(j)} \tag{2.2}$$

We know that $c_{ij} = c_{ji}$ and that $d_{ij} = d_{ij}$, because the problem is symmetric. Further we know that $c_{ii} = 0$ and also $d_{ii} = 0$. The solution p is mostly equal to permutation p', namely $p(i) = p'(i)$ for all $i \neq r, s$ and further $p(r) = p'(s)$ and $p(s) = p'(r)$. If we combine all these properties with equation (2.2) we get equation (2.1).

Δf_{rs} is equal to the objective value of f_{rs} minus objective value f. For calculating the objective value of f_{rs} we can use equation (1.1). Calculating f_{rs} by using this equation requires $O(n^2)$ time. But calculating Δf_{rs} in a smarter way with (2.1) the calculation only requires a calculating time of $O(n)$.

But Taillard describes in his article [20] an even faster way for calculating Δf_{rs}. He makes therefore use of the pair that is exchanged before the pair uv, call this pair rs. Define $\Delta_{uv} f_{rs}$ the change in the value of the objective function between the exchanged permutation by rs, and an additional exchanged pair uv. This change in the value of the objective function can be calculated in $O(1)$ if the pairs rs and uv are mutually exclusive. Two pairs are mutually exclusive if they cannot occur at the same time. For example two pairexchange of kl and km are mutually exclusive, because they both change the location of facility k and that can not happen at the same time.

We know from (2.1) that

$$\Delta f_{uv} = 2\sum_{i=1}^{n}[c_{iu}-c_{iv}][d_{p(i)p(v)}-d_{p(i)p(u)}]$$

Then, by checking which terms change if we exchange uv in the solution f_{rs}:

$$\begin{aligned}\Delta_{uv} f_{rs} = \Delta f_{uv} &+ 2[c_{su}-c_{sv}-(c_{ru}-c_{rv})][d_{p(r)p(v)}-d_{p(r)p(u)}] \\ &+ 2[c_{ru}-c_{rv}-(c_{su}-c_{sv})][d_{p(s)p(v)}-d_{p(s)p(u)}]\end{aligned}$$

We can write this as:

$$\begin{aligned}\Delta_{uv} f_{rs} = \Delta f_{uv} + \quad & 2[c_{su}-c_{sv}-(c_{ru}-c_{rv})] \\ & [d_{p(s)p(u)}+d_{p(r)p(v)}-d_{p(s)p(v)}-d_{p(r)p(u)}]\end{aligned} \tag{2.3}$$

We can use (2.1) for calculating Δf_{uv} in the equation $\Delta_{uv} f_{rs}$. This way of calculating $\Delta_{uv} f_{rs}$ is faster than first calculating Δf_{rs} and then calculating Δf_{uv}. Now we have to calculate only Δf_{uv} and then we can compute very easily $\Delta_{uv} f_{rs}$.

2.3 Results

Zvi Drezner tested his algorithm for all symmetric problems in QAPLIB [3], the library for QAP's. He ran his algorithm 120 times for each problem and each value of K. Here we present a part of his results. In table 2.1 is given how often the optimal solution is reached out of the 120 times and the runningtime of one run in seconds, a run consists of $\frac{120}{K}$ individual runs. In the second table 2.2 is the percentage of average solution over the best solution, this is the same as the difference between the average solution and the optimal solution divided by the optimal solution.

Table 2.1: Results 1 from the article for population sizes (K = 1,2,4,10)

Problem	Best	$K=1$		$K=2$		$K=4$		$K=10$	
	Known	n	t	n	t	n	t	n	t
Kra30a	88900	62	3.4	70	3.2	53	3.0	33	2.8
Kra30b	91420	37	3.4	20	3.1	16	3.0	12	2.8
Nug30	6124	62	3.7	62	3.3	60	3.0	17	2.8
Tho30	149936	76	3.7	81	3.3	54	3.0	49	2.7
Esc32a	130	112	3.9	116	3.7	109	3.7	102	3.8
Esc32b	168	120	3.3	120	3.0	120	2.9	120	2.9
Esc32h	438	120	2.6	120	2.5	119	2.4	108	2.4
Tho40	240516	4	10.1	1	9.2	2	8.7	0	8.2
Esc64a	116	120	18.5	120	17.5	120	17.2	120	18.4

n Number of times out of 120 that the best known solution obtained
t Time in seconds per run (Each run consists of $120/K$ individual runs)

Table 2.2: Results 2 from the article for population sizes (K = 1,2,4,10)

Problem	Best	$K=1$	$K=2$	$K=4$	$K=10$
	Known	p	p	p	p
Kra30a	88900	0.63	0.58	0.81	1.14
Kra30b	91420	0.08	0.12	0.15	0.26
Nug30	6124	0.04	0.03	0.05	0.13
Tho30	149936	0.09	0.09	0.15	0.20
Esc32a	130	0.10	0.05	0.17	0.28
Esc32b	168	0	0	0	0
Esc32h	438	0	0	0.00	0.05
Tho40	240516	0.19	0.23	0.23	0.30
Esc64a	116	0	0	0	0

p Percentage of average solution over the best known solution

Only for $Tho40$ with $K = 10$ the optimal solution is never obtained. The minimal solution that was obtained is 240542 and this solution was obtained 12 times. The percentage of the minimum solution over the best solution here is 0.01.

The results of this new heuristic are compared with the heuristics reported in [20]. The quality of the best solution found with these heuristics is of the same quality, but the run

times of the new heuristic are much faster.

The algorithm of these results was coded in Microsoft PowerStation Fortran 4.0 and ran on a Portege Toshiba 600MHz Pentium III lap-top.

Chapter 3

Implementation of the New Heuristic

In this chapter we will test the heuristic of Zvi Drezner for a few problems of QAPLIB [3]. In section 3.1 we explain how the algorithm is implemented and in section 3.2 the results of this implementation on the problems from QAPLIB are presented. In section 3.3 we compare these results with the results of the article and we discuss the differences between them.

3.1 Implementation of the Algorithm

For the implementation of the algorithm in MATLAB we make use of a number of functions. All these functions are in appendix A. Below we explain in detail for each function how we implemented it.

Δp and $\Delta\Delta p$

The computation of Δp is implemented in two different ways. The first way follows from the definition of Δp. We look at all the entries of the solution and compare them with the entries of the centersolution, then Δp is the number of entries that the solution differs of the center solution. The function which computes Δp in this way we call `deltap`.

The other way for computing Δp is with help of the short cut explained in the previous chapter. If we do a pairexchange rk on a solution p, call this pair exchange $p+$, we can calculate the $\Delta p+$ from Δp and $\Delta\Delta p$, because $\Delta p+ = \Delta p + \Delta\Delta p$. Sometimes we know already Δp, so this is then a better and faster way for calculating $\Delta p+$. We call the function which computes $\Delta\Delta p$ `deltadeltap`. In this function we start with $\Delta\Delta p = 0$. Then we look to the $r-th$ entry of the centersolution and compare it with the $r-th$ and the $k-th$ entry of the solution p. If $p(r)$ is the same as the $r-th$ entry of the centersolution the value of deltadeltap increases with one, this is because then the $r-th$ facility is in its centersolution site and now we change the location of this facility and so there is one facility fewer in its center side position (Δp increases with one). Further if $p(k)$ is the same as the $r-th$ entry of the centersolution, the value of deltadeltap decreases with one, because facility r was not on its center solution location in solution p and after the pairexchange it is, therefore Δp decreases with one. In the same way we compare the $k-th$ entry of the centersolution with the $r-th$ and the $k-th$ entry of the solution p.

The Objective Value and the Difference in the Objective Value

Like Δp we can calculate the objective value of a solution p in two different ways. We can use (1.1) or (2.1). If we use (1.1), we can rewrite this equation for symmetric problems with zero diagonals for a faster program. Therefore we rewrite (1.1) as:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij} d_{p(i)p(j)} \tag{3.1}$$

The function that is the implementation of (3.1) we call `objectivevalue`.

The other way of calculating the objective value makes use of the short cut presented in (2.1). We calculate the change in objective value with this short cut, we call this function `deltaobv`. The objective value of a pair exchange of solution p is now equal to the objective value of p plus the change in the objective value.

In the implementation we did not use the even faster shortcut (2.3) of Taillard, since it was not clear from the article where to use it.

New Best Found Solution

During an iteration we check all the pair exchanges of all solutions in $list_0$. For this we have the function `newbfs`. In this function we check all these pairexchanges and change the best found solution if a better solution is found among the pair exchanges. We also indicate during the run of this function the second best found solution, we need that later in the algorithm.

With the help of three for loops we go over all pairexchanges of all solutions in $list_0$. If during this search a better solution is found we update the best found feasible solution and we clear all lists and put this new best found solution in $list_0$. Now we go again over all pairexchanges of $list_0$, but this time $list_0$ has only one member, the new solution. We repeat this whole process until no better solution is found among all the pairexchanges.

A Solution in $list_1$ or in $list_2$

In Step 5 of the iteration described in section 2.1 we go over all the solutions in $list_0$. For each solution of $list_0$ we evaluate all pair exchanges for that solution and if the distance Δp is δp or lower, we ignore the permutation and if its distance Δp is $\delta p + 1$ or $\delta p + 2$, we will decide of this solution has to be in $list_1$, $list_2$ respectively. We make this decision with the functions `lists` and `inlist`.

The function `lists` looks at all pairexchanges of all solutions of $list_0$ and checks whether the pairexchanges have a distance of $\delta p + 1$ or $\delta p + 2$. If a solution has distance $\delta p + 1$ it puts this solution together with $list_1$ in the function `inlist` and if a solution has distance $\delta p + 2$ it puts this solution together with $list_2$ in function `inlist`.

The function `inlist` checks now whether the solution has to be in the concerning list. If the list is empty, that is when the list is set to 0, we put the solution in the list and define the worst member of the list as this solution. Otherwise the function checks first whether the concerning list is shorter than K or whether the solution is better than the worst member in the list. If that is the case the solution is compared with all the solutions in the list. If it is not the same as a member from the list we apply the solution to the list if the list was shorter than K, otherwise we replace the worst list member with this solution. There after we define a new worst member of the list.

A Whole Iteration

The function `QAPiter` performs a whole iteration on a start solution. This function does the same steps as explained in section 2.1 and uses for this the functions explained before. In this function we define a list as a matrix, where the columns are the solutions in that concerning list. If for example the list has two members, the list is a $n \times 2$ matrix and the two columns of this matrix are the two solutions of the list. If a list is empty, the list is set to 0.

Best Solution of the List Memory

In Step 5 of the algorithm we use the best solution in the list with $\delta p = d$ as the new center solution if $c = 1$ or 3. The function `QAPiter` gives as output this list as the list memory. With the function `bestmemory` we can determine which solution from this list is the best one.

The Algorithm

The function `QAP` has as input the matrices C, D and the parameter K. The function tries to find, in a way as described in section 2.1, the best feasible solution. We do one step different than described in the article. In Step 4 of the algorithm description in section 2.1, Zvi Drezner writes that we have to go back to Step 2, if we found a new better solution. But we think that he meant something else, because it is not meaningfull to repeat Step 3. This is because if we found during an iteration a better solution than the current center solution, we replace the current solution with this better solution and we do all steps, except Step 1, of the iteration again. So we have already perform a whole iteration on this new best found solution. It is therefore not necessary to do the whole iteration again with this better solution as center solution, because we will not find better or different results. So we changed Step 4 such that we do again Step 2 but we skip Step 3. Step 4 is then: Set the counter c to zero and go further with Step 5. We implemented this step in this way and not as described in the article.

3.2 Results of this implementation

We coded the algorithm in MATLAB 7.6.0 (R2008a) and ran experiments on an Intel (R) Core (TM) 2 Duo @ 2.33 GHz computer. In the tables 3.1, 3.2 and 3.3 our results are presented. Table 3.1 shows how often the optimal solution is reached out of the 120 times and the runningtime of one run in seconds, each run consists of $\frac{120}{K}$ individual runs. In table 3.2 the average solution and the percentage of the average solution over the best solution are shown. Table 3.3 presents for the problems where the optimal solution was never found, the minimum solution found and the percentage of the minimum solution over the best solution.

Table 3.1: Results 1 of this implementation for population sizes (K = 1,2,4,10)

Problem	Best	$K=1$		$K=2$		$K=4$		$K=10$	
	Known	n	t	n	t	n	t	n	t
Kra30a	88900	1	472.1	1	575.3	2	444.3	1	390.0
Kra30b	91420	0	465.3	0	575.6	1	447.5	2	413.1
Nug30	6124	0	482.5	1	570.7	3	428.8	1	374.0
Tho30	149936	1	437.5	2	506.8	0	404.3	4	343.3
Esc32a	130	6	580.00	5	797.8	11	594.4	12	603.5
Esc32b	168	55	434.6	46	480.0	54	426.5	63	430.0
Esc32h	438	14	339.5	14	425.6	10	353.6	19	331.0
Tho40	240516	0	1336.9	0	1652.2	0	1224.0	0	1136.1
Esc64a	116	119	2674.7	120	3397.0	120	2791	120	2693.5

n Number of times out of 120 that the best known solution obtained
t Time in seconds per run (Each run consists of $120/K$ individual runs)

Table 3.2: Results 2 of this implementation for population sizes (K = 1,2,4,10)

Problem	Best	$K=1$		$K=2$		$K=4$		$K=10$	
	Known	a	p	a	p	a	p	a	p
Kra30a	88900	93086	4.71	92801	4.39	92647	4.21	92545	4.10
Kra30b	91420	93759	2.56	9379	2.60	93637	2.43	93574	2.36
Nug30	6124	6202	1.28	6190	1.08	6194	1.14	6194	1.14
Tho30	149936	153120	2.12	153153	2.15	153083	2.10	153040	2.07
Esc32a	130	137	5.74	137	5.04	136	4.37	137	5.08
Esc32b	168	179	6.76	180	7.18	178	6.19	177	5.56
Esc32h	438	452	3.18	449	2.51	451	2.95	447	2.10
Tho40	240516	244590	1.70	244271	1.56	244203	1.53	243600	1.28
Esc64a	116	116	0.03	116	0	116	0	116	0

a Average solution found
p Percentage of average solution over the best known solution

Table 3.3: Results 3 of this implementation for population sizes (K = 1,2,4,10)

Problem	Best	$K = 1$		$K = 2$		$K = 4$		$K = 10$	
	Known	min	p_{min}	min	p_{min}	min	p_{min}	min	p_{min}
Kra30a	88900	-	-	-	-	-	-	-	-
Kra30b	91420	91490	0.08	91580	0.1750	-	-	-	-
Nug30	6124	6128	0.07	-	-	-	-	-	-
Tho30	149936	-	-	-	-	150280	0.23	-	-
Esc32a	130	-	-	-	-	-	-	-	-
Esc32b	168	-	-	-	-	-	-	-	-
Esc32h	438	-	-	-	-	-	-	-	-
Tho40	240516	241054	0.22	240542	0.01	240620	0.04	240716	0.08
Esc64a	116	-	-	-	-	-	-	-	-

min Minimum solution found

p_{min} Percentage of minimum solution over the best known solution

3.3 Discussion of the Results

When we compare the results of the article, presented in section 2.3, and the results of our own implementation of the algorithm, presented in the previous section, it is clear that our own results are much worse than the results of the article. For instance for the problem Esc32b Zvi Drezner obtained 120 times the best known solution for $K = 1, 2, 4, 10$, but we found respectively 55, 46, 54 and 63 times the best known solution. For the problems Kra30a, Kra30b, Nug30 and Tho30 we obtained at most 3 times the best solution for $K = 1, 2, 4, 10$ in contrast with Zvi Drezner who obtained at least 12 times the best known solution for these problems and $K = 1, 2, 4, 10$. For the problems Esc32a, Esc32b and Esc32h we found at most 63 times the best known solution and Zvi Drezner obtained for most of these problems 120 times the best known solution. The problem Tho40 gives bad results for both codes, but Zvi Drezner obtained a little better results because we never obtained the best known solution for this problem. Like Zvi Drezner we got good results for the problem Esc64a for $K = 2, 4, 10$.

Like the number of times we obtained the best known solution, our average value of the found solutions is much worse than the average value of the solutions of Zvi Drezner. He got at most a percentage of 1.14 of the average solution over the best found solution of 1.14 and we got at most a percentage of 7.18.

For the problems and K-value's where the optimal solution was never found the percentage of the minimal found solution over the best known solution is at most 0.23.

We do not know why the results are worse than the results in the article. A possibility is the random choice of the start solution of the algorithm. Zvi Drezner does not describe how the random start solution is chosen. We implement it completely random, but it is also possible to use, for instance, a construction method. When we start with a relatively good solution the probability for finding the best solution known is larger.

The running time of our code was much longer than the runningtimes presented in the article. There are four things which can clarify these results, the programming language, the memory, an error in the implementation and the description of the shortcuts in the article. Matlab is an interpreted language and Fortran, the Language Zvi Drezner has used, is a compiled language. Codes written in a compiled language are mostly faster. At the same

time for large problems the computer needs a lot of memory and this can slow the run. That the computer needs so much memory is due to the programm language and the way we implemented it. Further Zvi Drezner described in his article which short cuts he used, but he did not write where he applied these shortcuts and therefore there is a change that we did not apply the shortcuts everywhere were it was possible. Also it is possible that we implemented some things not in a fast way.

Chapter 4

Tabu search

We have already introduced tabu search in chapter 1. In this chapter we will implement a tabu search algorithm and compare the results of this algorithm with the results of the implementation of the previous chapter. In the first and second sections of this chapter we described the used algorithm and the implementation of this algorithm. In section 3.4 we present the results and in the last section we compare these results with the results of our own implementation of the new heuristic of Zvi Drezner.

4.1 The Algorithm

Tabu search has two control parameters K and $imax$, where K is the maximum length of the tabu list and $imax$ the maximum number of iterations.

Tabu search makes use of a tabu list. In this algorithm the tabu list is a set of $2K$ or less numbers, where the numbers are different integers from the interval [1,n]. If a number i is on the tabu list, the pair exchanges ij for all j are forbidden moves.

Description of the Algorithm

1. We randomly select a start solution. Call this the center solution. This solution is the best found solution. The tabu list is empty and i, the number of iterations, is set to one.

2. Go over all the pairexchanges of the center solution. If the objective value of a pairexchange is better than the objective value of the best found solution so far, replace the best found solution with this solution. Go further with checking the rest of the pair exchanges.

3. If a better solution is found during Step 2, replace the center solution with this solution. Clear the tabu list and set i again to one. Go to Step 2.

4. If no better solution is found, replace the center solution with the best permitted pairexchange kl of the center solution. A pairexchange ij is permitted if neither i nor j are in the tabu list.

5. We will now update the tabu list. Say kl was the best permitted pairexchange of the old center solution, then:

- If the tabulist already has $2K$ numbers, we use the first-in-first-out rule. So we replace the numbers which are longest in the list with k and l.
- If the tabu list is not full yet, we add the numbers k and l to it.

6. If i is smaller than $(imax + 1)$, go to Step 2. Otherwise stop the algorithm.

4.2 The Implementation

For the implementation of the algorithm in MATLAB we make use of two functions. These functions are in appendix B. First we will explain how we implemented the tabu list and after that we will discuss the two functions in more detail.

Tabu list

Tabu search makes use of a tabu list. We implement this list with two matrices. One matrix is an n-vector, call this vector tL. The entries of the vector tL satisfy the following condition.

$$tL_i = \begin{cases} 1 & \text{there is a } j \text{ for which we have done a pairexchange } ij \text{ in the last } K \text{ iterations} \\ 0 & \text{otherwise} \end{cases}$$

where K is the length of the tabu list. The second matrix is an $(imax, 2)$-matrix. We call this matrix tL2. The $i - th$ row of this matrix consists of the numbers of the entries of the solution that are exchanged in the $i - th$ iteration.

An Iteration

In the function `iteration ts` all objective values of the pairexchanges of the center solution are compared with the objective value of the best found solution so far. After that we search for the best pair exchange of the center solution that is not on the tabu list. A pairexchange ij is not on the tabu list if tL_i and tL_j are both equal to zero.

At the end of this function we update the tabu list. Therefore we change tL_i and tL_j into 1, if ij was the best permitted pairexchange of the old centersolution. The next K iterations i and j can not be exchanged. Further we set the $i - th$ row of the matrix $tL2$ equal to the entries i and j. The length of the tabu list may not be larger than K. Therefore we make use of the matrix $tL2$. If i, the number of iterations we have done, is bigger than K we look at the $(i - K) - th$ row of the matrix $tL2$. The numbers of this row we call v and w. Now we set the $v - th$ and $w - th$ entries of the vector tL equal to zero, such that v and w can be exchanged again.

The Algorithm

The function `tabusearch` performs tabu search on a random solution. The input arguments of this function are the matrices C, D, the tabu list length K and the maximum number of iterations $imax$.

4.3 The Results of Tabu Search

The tabu search algorithm makes use of two parameters, the length of the tabu list and the maximum number of iterations. We tried for each parameter two different values, 100 and 500 maximum iterations and 5 and 10 as maximum length of the tabu list. The results of the tabu search-implementation are present in three tabels. In the tables 4.1 and 4.2 are shown, respectively for 100 and 500 iterations, how often the optimal solution is reached out of the 120 times, the runningtime of 120 individual runs, the average solution and the percentage of the average solution over the best solution. Table 4.3 presents for the problems where the optimal solution was never found, the minimum solution found and the percentage of the minimum solution over the best solution. Like the heuristic of Zvi Drezner, we coded the algorithm in MATLAB 7.6.0 (R2008a) and ran it on the same computer.

Table 4.1: Results 1 for tabu search for (K = 5,10) and 100 iterations

Problem	Best	$K = 5$				$K = 10$			
	Known	n	t	a	p	n	t	a	p
Kra30a	88900	0	28.6	94588	6.40	0	32.1	94169	5.93
Kra30b	91420	0	28.7	95640	4.62	0	32.6	95018	3.94
Nug30	6124	0	35.4	6284	2.61	0	35.2	9275	2.46
Tho30	149936	0	34.7	155320	3.59	0	32.8	155150	3.48
Esc32a	130	0	33.6	148	13.65	0	33.62	146	12.65
Esc32b	168	15	33.1	191	13.79	9	35.3	192	14.09
Esc32h	438	6	30.0	451	3.02	4	30.9	454	3.76
Tho40	240516	0	83.9	248050	3.13	0	89.9	247490	2.90
Esc64a	116	73	182.1	117	0.92	96	174.0	116	0.39

n Number of times out of 120 that the best known solution obtained
t Time in seconds per run (Each run consists of 120 individual runs)
a Average solution found
p Percentage of average solution over the best known solution

Table 4.2: Results 2 for tabu search for (K = 5,10) and 500 iterations

Problem	Best	$K = 5$				$K = 10$			
	Known	n	t	a	p	n	t	a	p
Kra30a	88900	0	115.5	94410	6.20	0	122.3	94165	5.92
Kra30b	91420	0	115.4	95479	4.44	0	120.1	95294	4.20
Nug30	6124	1	130.1	6274	2.45	0	132.0	6260	2.22
Tho30	149936	0	119.7	155080	3.43	0	122.3	154900	3.31
Esc32a	130	0	134.1	148	13.81	0	151.0	146	11.96
Esc32b	168	13	134.3	193	14.94	18	147.4	189	12.64
Esc32h	438	8	131.1	454	3.71	8	131.3	453	3.52
Tho40	240516	0	255.8	248420	3.29	0	321.1	246540	2.50
Esc64a	116	85	756.5	117	0.73	106	779.1	116.3	0.23

n Number of times out of 120 that the best known solution obtained
t Time in seconds per run (Each run consists of 120 individual runs)
a Average solution found
p Percentage of average solution over the best known solution

Table 4.3: Results 3 for tabu search for (K = 5,10) and 100, 500 iterations

Problem	Best	$K = 5$				$K = 10$			
	Known	$i = 100$		$i = 500$		$i = 100$		$i = 500$	
		min	p_{min}	min	p_{min}	min	p_{min}	min	p_{min}
Kra30a	88900	90700	2.02	90090	1.34	90500	1.80	90220	1.48
Kra30b	91420	91580	0.18	92060	0.70	91580	0.18	91490	0.08
Nug30	6124	6150	0.42	-	-	6128	0.07	6136	0.20
Tho30	149936	150280	0.23	150280	0.23	150554	0.41	150216	0.19
Esc32a	130	132	1.54	136	4.62	134	3.08	132	1.54
Esc32b	168	-	-	-	-	-	-	-	-
Esc32h	438	-	-	-	-	-	-	-	-
Tho40	240516	242814	0.96	243708	1.33	241266	0.31	241290	0.32
Esc64a	116	-	-	-	-	-	-	-	-

min Minimum solution found
p_{min} Percentage of minimum solution over the best known solution

4.4 Comparison of the results

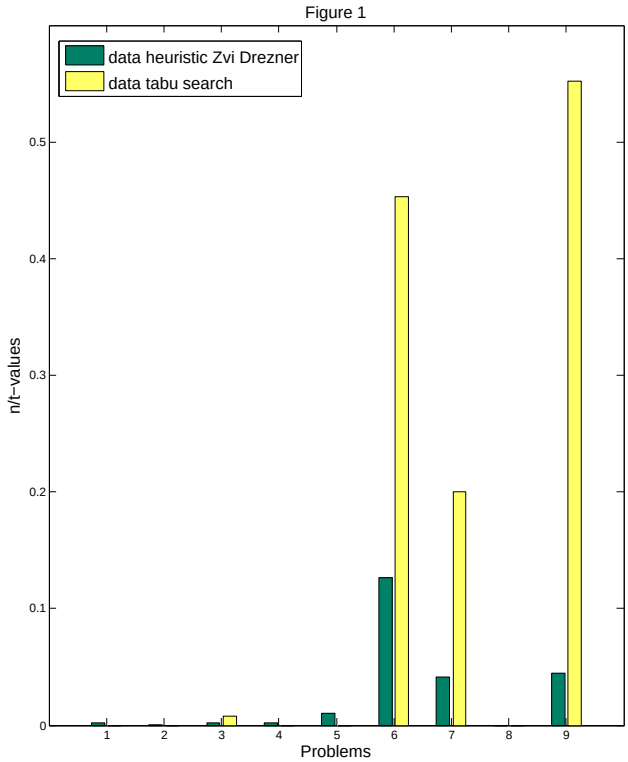
If we compare the results of our own implementation of the heuristic of Zvi Drezner and the results of tabu search, it seems that the heuristic of Zvi Drezner gets better results because out of the 120 times the optimal solution is more often reached. But if we look at the runningtimes of both heuristics, we see that tabu search is much faster than the other heuristic. To compare the results we calculate how often an optimal solution is reached per second. So we divided the number that the optimal solution is reached with the runningtime of 120 individual runs, we denote this value as n/t.

We calculate for each problem and each heuristic for which values of the parameters the

best n/t value is reached. These parameters and the corresponding n/t values are present in table 4.4. The n/t values are also present in figure 1.

Table 4.4: n/t-values for both heuristics

Problem	Zvi Drezner		Tabu search		
	K	n/t	k	i	n/t
Kra30a	1	0.0021	-	-	0
Kra30b	4	0.0006	-	-	0
Nug30	4	0.0017	5	500	0.0077
Tho30	1	0.0023	-	-	0
Esc32a	1	0.0103	-	-	0
Esc32b	1	0.1266	5	100	0.4545
Esc32h	1	0.0412	5	100	0.2000
Tho40	-	0	-	-	0
Esc64a	1	0.0445	10	100	0.5517



In the table and the figure we see that the n/t-value for the problems Kra30a, Kra30b, Tho30 and Esc32a are higher for the heuristic of Zvi Drezner than these values for tabu search.

In all these cases tabu search never finds the optimal solution and in most of the cases the heuristic of Zvi Drezner found the optimal solution just once. That is why we have to remark that the time needed for a run of the heuristic of Zvi Drezner is much more than the time needed for a run of tabu search. So in the runningtime of the heuristic of Zvi Drezner we can run tabu search a couple of times and if we run tabu search more often the probability of finding the optimal solution increase. So if we run both heuristics a certain time it is possible to find more often the optimal solution with tabu search, but this we did not study that. For the problems Nug30, Esc32b, Esc32h and Esc64h the n/t-values are much higher than the values obtained with the heuristic of Zvi Drezner. This is because tabu search is so much faster.

Until now we only compare the runningtimes and the number of times the optimal solution is reached out of 120 times for both heuristics. Now we will compare the percentage of the average solution over the best known solution and the percentage of the minimum solution found over the best known solution. The results of the percentage of the average solution are presented in the table 4.5 and figure 2. The results of the percentage of the minimum solution are presented in the table 4.6.

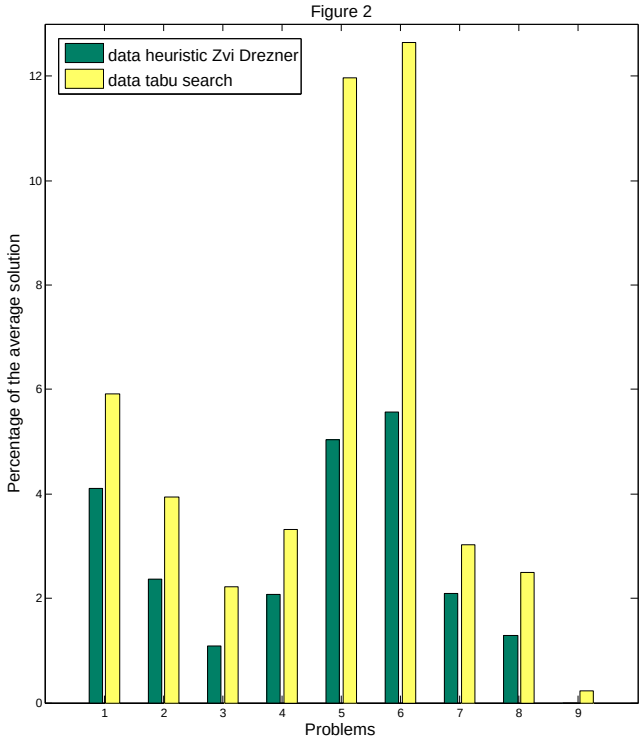
Table 4.5: percentage of the average solution for both heuristics

Problem	Zvi Drezner		Tabu search		
	K	p	k	i	p
Kra30a	10	4.10	10	500	5.92
Kra30b	10	2.36	10	100	3.94
Nug30	2	1.08	10	500	2.22
Tho30	10	2.07	10	500	3.31
Esc32a	2	5.04	10	500	11.96
Esc32b	10	5.56	10	500	12.64
Esc32h	10	2.10	5	100	3.02
Tho40	10	1.28	10	500	2.50
Esc64a	10	0	10	500	0.23

Table 4.6: percentage of the minimum solution for both heuristics

Problem	Zvi Drezner		Tabu search		
	K	p	k	i	p
Kra30a	-	0	5	500	1.34
Kra30b	4, 10	0	10	500	0.08
Nug30	2, 4, 10	0	5	500	0
Tho30	1, 2, 10	0	10	500	0.19
Esc32a	-	0	5, 10	100, 500	1.54
Esc32b	-	0	-	-	0
Esc32h	-	0	-	-	0
Tho40	2	0.01	10	100	0.31
Esc64a	-	0	-	-	0

In these two tables and the figure we can see that the heuristic of Zvi Drezner for either



the average percentage as the minimum percentage for all problems yields better results. In the figure we see that especially for the problems Esc32a and Esc32b the percentage of the average solution over the optimal solution the heuristic of Zvi Drezner has much better results. But again we have to take into account that the running time of the heuristic of Zvi Drezner is much more than the running time of tabu search.

It is difficult to compare the results of both heuristics, tabu search found for 5 problems not the optimal solution but it is much faster than the other heuristic. The heuristic of Zvi Drezner found more often the optimal solution, but due to the speed its n/t-values are very low. Further we see that the percentage of the average solution as well as the percentage of the minimum solution are better for the heuristic of Zvi Drezner.

Chapter 5

Conclusion and Discussion

In this thesis we wanted to compare the new heuristic of Zvi Drezner with tabu search. We implemented both heuristics in MATLAB and ran the implementations on the same computer. First we compared the results of our own implementation of the heuristic of Zvi Drezner with the results of this heuristic presented in the article. There are big differences between these results. Our implementation did not often find the optimal solution while Zvi Drezner did, and our runningtime was much longer. Things which can clarify these differences are the programming language, the memory of the computer, the way of programming and the description of the algorithm in the article.

The implementation of the heuristic of Zvi Drezner and the implementation of tabu search were implemented both in MATLAB and run on the same computer. We compared the results we got with both heuristics, but we can not conclude that one heuristic is better than the other. That is because of the big differences in runningtime. Tabu search found for some problems optimal solutions in less times than the heuristic of Zvi Drezner, but for some problems the optimal solution was never obtained. While the heuristic of Zvi Drezner found for all, except one, problems the optimal solutions at least once, but the runningtime was very long and due to this its n/t-values was very low.

In his article Zvi Drezner writes that his heuristic has equal results as other heuristics for most problems, but that his running time is much faster. Our results do not verify this, but our results of the heuristic of Zvi Drezner were worse than his results so with his results it can be possible.

It would be interesting to research which heuristic finds the most times the optimal solution within a certain time. Because then we do not have the big difference between runningtimes and is it easier to compare the heuristics. Further it may be interesting to implement the tabu search in a different way than we did now. For example we can make a tabu list which permits more moves or we can implement tabu search in such way that it performs more often a single tabu search on different random solutions and than only give the best solution found. This last change in the implementation of tabu search will certainly decrease the average solution found and maybe also the minimum solution such that the optimal solution is more often reached, but it will also increase the runningtime.

Appendix A

Matlab: New Heuristic

```
function[deltap] = deltap(x, centersolution)

% Function for calculating delta p of a solution x. The distance, delta p, is
    the number of facilities in x that are not in their center solution site.

deltap = 0;

for i = 1:length(x)
    if x(i) - centersolution(i) ~= 0
        deltap = deltap + 1;
    end
end

end
```

```
function[deltadeltap] = deltadeltap( x, centersolution, r, k)

% This function computes the difference in delta p between the solution x
% and the rk-pairexchange of x.

deltadeltap = 0;

if x(r) == centersolution(r)
    deltadeltap = 1;
elseif x(k) == centersolution(r);
    deltadeltap = -1;
end

if x(k) == centersolution(k)
    deltadeltap = deltadeltap + 1;
elseif x(r) == centersolution(k);
    deltadeltap = deltadeltap - 1;
end

end
```

```
function[obv] = objectivevalue(x, C, D)

% This function calculates the objective value of a solution x. Note that
% the problem has to be symmetric with a zero diagonal.

obv = 0;
n = length(x);

for i = 1:n-1
    for j = i+1:n
        obv = obv + 2 * C(i, j) * D(x(i), x(j));
    end
end

end
```

```
function[deltaobv] = deltaobv(r, s, w, C, D)

% This function computes the difference in objective value between the
% solution w and the rs-pairexchange of w. Note that the problem has to be
% symmetric with zero diagonal.

sum = 0;

for i = 1:length(w)
    if i ~= r && i~= s
   sum = sum + (C(i, r) - C(i, s)) * (D(w(i), w(s)) - D(w(i), w(r)));
    end
end

deltaobv = 2 * sum;

end
```

```
function[bfs, obvbfs, secondbfs, secondbfsv] = newbfs(list0, obvbfs, C, D, bfs,
     secondbfs, secondbfsv)

% Evaluate all pair exchanges for the solutions in list0. If the exchanged
% solution is better than the best found solution, update the best found
% solution and proceed to evaluate the rest of the exchanges. If a new best
% found solution is found by scanning all the exchanges, set the center
% solution to the new best found solution (is list0) and do this iteration
    again.

newbfs = 1;
v = size(list0);
x = 0;

while newbfs == 1
    newbfs = 0;
    for i = 1:v(2)
        if x == 0;
            obvlist0 = objectivevalue(list0(:, i), C, D);
        end
        for j = 1:v(1)-1
            for k = j+1:v(1)

                obvw =  deltaobv(j, k, list0(:, i), C, D) + obvlist0;

                if obvw < obvbfs
                    bfs = list0(:, i);
                    h = bfs(j);
                    bfs(j) = bfs(k);
                    bfs(k) = h;
                    obvbfs = obvw;
                    newbfs = 1;
                elseif obvw < secondbfsv || secondbfsv == 0
                    secondbfs = list0(:, i);
                    h = secondbfs(j);
                    secondbfs(j) = secondbfs(k);
                    secondbfs(k) = h;
                    secondbfsv = obvw;
                end
            end
        end
    end
    if newbfs ==1
        list0 = bfs;
        obvlist0 = obvbfs;
        x = 1;
        v(2) = 1;
    end
end
end
```

```
function[list1, list2, worstmember2, obvworstmember2] = lists(list0, C, D, K,
    list1, worstmember1, obvworstmember1, list2, dp, bfs)

% This function looks at all pair-permutations and determines with the
% inlist-function of they will be applied to list1 or list2 or to none list.

v = size(list0);

worstmember2 = 0;
obvworstmember2 = 0;

for i = 1:v(2)
    obvlist0 = objectivevalue(list0(:, i), C, D);
    for j = 1:v(1)-1
        for k = j+1:v(1)

            deltapw = dp + deltadeltap(list0(:, i), bfs, j, k);

            if deltapw == dp + 1
                w = list0(:, i);
                h = w(j);
                w(j) = w(k);
                w(k) = h;
                obvw = obvlist0 + deltaobv(j, k, list0(:, i), C, D);
                [list1, worstmember1, obvworstmember1] = inlist(list1, w, obvw,
                    C, D, K, worstmember1, obvworstmember1);
            elseif deltapw == dp + 2
                w = list0(:, i);
                h = w(j);
                w(j) = w(k);
                w(k) = h;
                obvw = obvlist0 + deltaobv(j, k, list0(:, i), C, D);
                [list2, worstmember2, obvworstmember2] = inlist(list2, w, obvw,
                    C, D, K, worstmember2, obvworstmember2);
            end
        end
    end
end

end
```

```
function[list, worstmember, obvworstmember] = inlist(list, p, obvp, C, D, K,
    worstmember, obvworstmember)

% This function determines of a solution p is applied to the list.

v = size(list);

if  list(1, 1) == 0
    list = p;
    worstmember = 1;
    obvworstmember = obvp;
elseif v(2) < K || obvp < obvworstmember
    for i = 1:v(2)
        obv(i) = objectivevalue(list(:, i), C, D);
        help = 1;
        if obvp - obv(i) == 0 && deltap(p, list(:, i)) == 0
            help = 0;
        end
    end

    if help == 1
        if v(2) < K
            list = [list p];
            v(2) = v(2) + 1;
            if obvp > obvworstmember
                obvworstmember = obvp;
                worstmember = v(2) + 1;
            end
        else
            list(:, worstmember) = p;
            obv(worstmember) = obvp;

            obvworstmember = obv(1);
            worstmember = 1;
            if K ~= 1
                for i = 2:v(2)
                    if obv(i) > obvworstmember
                        obvworstmember = obv(i);
                        worstmember = i;
                    end
                end
            end
        end
    end
end

end
```

```
function[bfs, obvbfs, memory, secondbfs] = QAPiter(centersolution, K, d, C, D,
    n)

% This function is a whole iteration.

dp = 0;
list0 = centersolution;
list1 = 0;
list2 = 0;
bfs = centersolution;
obvbfs = objectivevalue(bfs, C, D);
secondbfsv = 0;
secondbfs = 0;
worstmember1 = 0;
obvworstmember1 = 0;

while dp <= d

    [bfs, obvchange, secondbfs, secondbfsv] = newbfs(list0, obvbfs, C, D, bfs,
        secondbfs, secondbfsv);

    if obvchange - obvbfs ~= 0
        obvbfs = obvchange;
        list0 = bfs;
        list1 = 0;
        list2 = 0;
        dp = 0;
        worstmember1 = 0;
        obvworstmember1 = 0;
    end

    [list1, list2, worstmember2, obvworstmember2] = lists(list0, C, D, K, list1
        , worstmember1, obvworstmember1, list2, dp, bfs);

    if list1(1, 1) == 0
        memory = list0;
        list0 = list2;
        list1 = 0;
        list2 = 0;
        dp = dp + 1;
        worstmember1 = 0;
        obvworstmember1 = 0;
    else
        memory = list0;
        list0 = list1;
        list1 = list2;
        list2 = 0;
        worstmember1 = worstmember2;
        obvworstmember1 = obvworstmember2;
    end
    dp = dp + 1;
end

end
```

```
function[centersolution] = bestmemory(memory, C, D)

% This function determines which solution is the best from the list memory.

mem = size(memory);
obj = objectivevalue(memory(:,1), C, D);
x = 1;

if mem(2) >= 2
for i = 2:mem(2)
    if obj > objectivevalue(memory(:,i), C, D);
        obj = objectivevalue(memory(:,i), C, D);
        x = i;
    end
end
end

centersolution = memory(:, x);

end
```

```
function[bfs, obvbfs] = QAP(K, C, D)

% This function is the whole algorithm. Note that in this algorithm short
% cuts for symmetric problems with zero diagonal are applied, so the input
% matrices C, D have to be symmetric with zero diagonal.

n = size(C);

% Select a random center solution.
% Randperm(n(1))' gives a vector with a permutation of n numbers.
centersolution = randperm(n(1))';

bfs = centersolution;
obvbfs = objectivevalue(bfs, C, D);

c = 0;

while 1

    % Select d randomly in [n-4, n-2].
    % Rand(1) gives a random number of the interval (0, 1), so 3*rand(1) gives
    % a random number of the interval (0, 3). Ceil round off upwards these
    % numbers, so ceil(3*rand(1)) gives the integer numbers 1, 2 and 3. So
    % ceil(3*rand(1)) + n - 5 gives then a number randomly in [n-4, n-2].
    d = ceil(3*rand(1)) + n(1) - 5;

    [bfs2, obvbfs2, memory, x] = QAPiter(centersolution, K, d, C, D, n(1));

    if obvbfs > obvbfs2
        c = 0;
        bfs = bfs2;
        obvbfs = obvbfs2;
    end

    c = c+1;
    if c == 1 || c == 3
        centersolution = bestmemory(memory, C, D);
    elseif c == 2 || c == 4
        centersolution = x;
    else
        break;
    end
end

end
```

Appendix B

Matlab: Tabu Search

```
function[bfs, obvbfs] = tabusearch(C, D, K, i_max)

% Here is K the length of the tabulist and i_max is the maximum number of
% iterations. K has to be smaller than n/2 - 2.

nhelp = size(C);
n = nhelp(1);
x = randperm(n(1))';
i = 1;
tL = zeros(n, 1);
tL2 = zeros(i_max, 2);
bfs = x;
obvx = objectivevalue(x, C, D);
obvbfs = obvx;

while(i <= i_max)

    [nieuwbfs, nieuwbfsv, twbfs, twbfsv, tL, tL2] = iterationts(x, obvx, C, D,
        bfs, obvbfs, tL, tL2, K, i, i_max)

    if nieuwbfsv < obvbfs
        i= 1;
        bfs = nieuwbfs;
        obvbfs = nieuwbfsv;
    end

    x = twbfs;
    obvx = twbfsv;
    i = i + 1;
end

end
```

```
function[bfs, obvbfs, twbfs, twbfsv, tL, tL2] = iterationts(x, obvx, C, D, bfs,
    obvbfs, tL, tL2, K, i, i_max)

n = length(x);
nieuwbfs = 1;

while nieuwbfs == 1
    nieuwbfs = 0;
    twbfsv = 0;

    for j = 1:n-1
        for k = j+1:n
            obvw =  deltaobv(j, k, x, C, D) + obvx;
            if obvw < obvbfs
                bfs = x;
                h = bfs(j);
                bfs(j) = bfs(k);
                bfs(k) = h;
                obvbfs = obvw;
                nieuwbfs = 1;
            end
            if obvw < twbfsv || twbfsv == 0
                if tL(j) == 0 && tL(k) == 0
                    twbfs = x;
                    h = twbfs(j);
                    twbfs(j) = twbfs(k);
                    twbfs(k) = h;
                    twbfsv = obvw;
                    a = j;
                    b = k;
                end
            end
        end
    end

    if nieuwbfs ==1
        x = bfs;
        obvx = obvbfs;
        twbfsv = 0;
        tL = zeros(n, 1);
        tL2 = zeros(i_max, 2);
        i = 1;
    end
end

tL(a) = 1;
tL(b) = 1;
tL2(i, :) = [a b];
if i > K
    aold = tL2(i - K, 1);
    bold = tL2(i - K, 2);
    tL(aold) = 0;
    tL(bold) = 0;
end

end
```

Bibliography

[1] E. S. Buffa, G. C. Armour and T. E. Vollmann, *Allocating Facilities with CRAFT*, Harvard Business Review 42, 1962, 136-158.

[2] R. E. Burkard and E. Çela, *The Quadratic Assignment problem.* In: D. Du and P. M. Pardalos (Eds.): Handbook of Combinatorial Optimaization, Volume 3, 241-337, Kluwer Academic Publishers, Dordrecht, 1998.

[3] R. E. Burkard, S. E. Karisch and F. Rendl, *QAPLIB - A Quadratic Assignment Problem Library*, Journal of Global Optimization, 10, 1997, 391-404. An on-line version is available via World Wide Web at the following URL: `http://www.opt.math.tu-graz.ac.at/~karisch/qaplib`

[4] E. Çela, *The Quadratic Assignment Problem: Theory and Algorithms.* Kluwer Academic Publishers, Dordrecht, 1998.

[5] Z. Drezner, *A New Heuristic for the Quadratic Assignment Problem.* Journal of Applied Mathematics and Decision Sciences, 6(3), 2002, 143-153.

[6] A. N. Elshafei, *Hospital layout as a Quadratic Assignment Problem.* Operations Research Quartely 28, 1977, 167-179.

[7] T. Feo and M. G. C. Resende, *Greedy Randomized Adaptive Search Procedures*, Journal of Global Optimization 6, 1995, 109-133.

[8] G. Finke, R. E. Burkard and F. Rendl *Quadratic Assignmet Problems.* In: S. Martello, G. Laprote, M. Minoux, Ć. Ribeiro (Eds.): Annals of Discrete Mathematics 31, Surveys in Combinatorial Optimization, 61-82. Elsevier science publishers B.V., Amsterdam, 1987.

[9] L. M. Gambardella, E. D. Taillard, and M. Dorigo, *Ant Colonies for the QAP*, Technical Report IDSIA-4-97, 1997, Istituto dalle Molle Di Studi sull' Intelligenza Artificiale, Lugano, Switzerland.

[10] P. C. Gilmore, *Optimal and Suboptimal Algorithms for the Quadratic Assignment Problem*, SIAM Journal on Applied Mathematics 10, 1962, 305-313.

[11] F. Glover, *Tabu search - Part I*, ORSA Journal on Computing 1, 1989, 190-206.

[12] F. Glover, *Tabu search - Part II*, ORSA Journal on Computing 2, 1989, 4-32.

[13] J. H. Holland, *Adaption in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[14] T. C. Koopmans and M. J. Beckmann, *Assignment Problems and the Location of Economic Activities*, Econometrica, 25, 1957, 53-76.

[15] Y. Li, P. M. Pardalos and M. Resende, *A Greedy Randomized Adaptive Search Procedure fot the Quadratic Assignment problem.* In: P. Pardalos and H. Wolkowicz (Eds.): Quadratic Assignment and Related Problems, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 16, 1994, 237-261.

[16] H. Müller-Merbach, *Optimale Reihenfolgen*, Springer-Verlag, Berlin, Heidelberg, New York, 1970, 158-171.

[17] M. A. Pollatschek, N. Gershoni and Y. T. Radday, *Optimization of the typewriter keyboard by computersimulation.* Angewandte Informatik 10, 1976, 438-439.

[18] K. H. Rosen, *Handbook of Discrete and Combinatorial Mathematics*, CRC press LLC, Boca Raton, Florida, 2000, 702-705.

[19] S. Sahni and T. Gonzalez, *P-compete Approximation Problems*, Journal of the ACM 23, 1976, 555-565.

[20] E. D. Taillard, *Comparision of Iterative Searches for the Quadratic Assignment Problem*, Location Science, 3, 1995, 87-105.