10th International Conference of the Practice and Theory of Automated Timetabling
PATAT 2014, 26-29 August 2014, York, United Kingdom

FlexMatch - A Matching Algorithm with linear Time and Space Complexity

Nina Nöth · Peter Wilke

Abstract FlexMatch, a new matching algorithm with linear time and space complexity is introduced.

FlexMatch is based on a self organizing cell structure yielding next neighbour candidates for optimized matching.Results obtained when applying the FlexMatch algorithm on a real world problem are presented.

Keywords Matching Algorithm · Linear Time and Space Complexity · Self Organizing Cell Structure

1 Introduction

Our research group also acts as competence centre at our university regarding multi-criteria optimization problems. Recently we have been approached to implement a matching portal for students looking for hands-on training outside our faculty labs. The duration and extent depend on the degree and subject. As the metropolitan region of Nuremberg and especially the City of Erlangen is a national centre for medical engineering and technology numerous companies, ranging from very small business to major players, are offering this type of off-campus training.

The matching problem consists of a facts based part and a political component. Of course all companies would prefer to hire the best students, and of course most students would prefer a major company. But from a regional

Nina Noeth
E-mail: Nina.Noeth@Studium.Informatik.Uni-Erlangen.DE

Peter Wilke
University of Erlangen-Nuernberg
Computer Science Department
Multi Criteria Optimisation Group
Snail-mail: Martensstrasse 3, 91058 Erlangen, Germany
E-mail: Peter.Wilke@FAU.DE

developer's view students should get familiar with small but highly specialized companies even though they might not be widely known. And from an university's point of view all these training opportunities should be offered to all students.

2 The Problem

The problem to be solved consists of matching students with companies' training offers, where their constraints reflect the student's resp. companies' requirements.

It should be obvious that this optimization problems has contradictory constraints.

The initial situation consists of supply (companies' offers) and demand (students' requirements) which are both entered in a form with several categories. The values entered are either single numerical values, ranges or binary/boolean values.

In the context of this paper the concrete nature of the constraints or data is of no interest. It is sufficient to regard the data as a multidimensional vector and to presume the existence of a cost function to evaluate the current solution.

The solution should reflect the policy that students should be introduced to all kinds of companies and that companies should be offered students of all performance levels. But of course the individual requirements on both sides are the most significant matching criteria.

The solution consist of one company for each student and one student for each training opportunity offered. If this matching doesn't lead to a acceptance each party can request additional suggestions.

3 The FlexMap- and FlexMatch-Algorithms

On the top level abstraction layer solving the problem is divided into the following steps:

1. Building two separate ring structures: one connecting requirements and the other connecting requirements and offers having a preferable small distance to each other,
2. Matching the requirements and offer by using the resulting neighbourhood relation of the ring structures.
3. Improving the single matches by looking for better partners in a deeper neighbourhood.

3.1 FlexMap

FlexMap [Fritzke and Wilke(1991)] is a self-organizing neural network, which is linear in its time and space complexity. Problems similar to the Travelling

Salesman Problem can be solved using the growing ring structure yielding the round-trip we're looking for. I.e. the FlexMap algorithm connects each city with it's next – with respect to the length of the Hamiltonian cycle – neighbours, inducing some kind of a topology order on the nodes.

The basic idea is a growing cell structure. The initial structure consists of three cells. Repeated insertion and distribution steps extends the structure until all cells can be matched with its corresponding node, e.g. the city (Fig. 2). A node (e.g. a city) is chosen randomly and it next neighbour edge is calculated, a new cell is inserted in the middle of that edge and the cells are moved towards the chosen node (Fig. 1).

Table 1 shows the $O(n)$-version of the FlexMap algorithms in detail, followed by it's structogram.

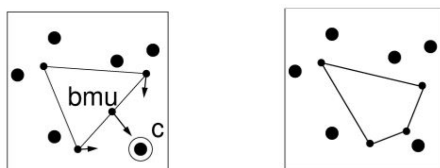


Fig. 1: Example of a distribution step: inserting a *bmu* in the neighbouring edge and moving three cells towards the node C. [Fritzke and Wilke(1991)]

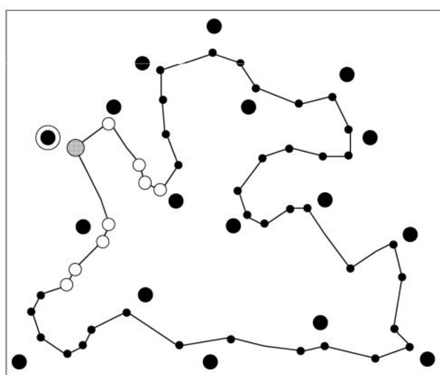


Fig. 2: Local search for the best matching unit: the previous *bmu* is shown as shaded circle and the local neighbours (here up to degree 4) as white circles. [Fritzke and Wilke(1991)]

Some remarks regarding it's complexity:

Step 1 $k_{neighbour}$ is a constant, so the neighbourhood search can be done in constant time $O(1)$.

Step 2 A cell becomes a member of the set of high error cells when is often becomes the bmu and therefore its error variable is increased quite often. As

10th International Conference of the Practice and Theory of Automated Timetabling
PATAT 2014, 26-29 August 2014, York, United Kingdom

there are only $n_{distribution}$ steps, so the search can be performed in constant time $O(n)$. Obviously we can't guarantee to find the global maximum but most likely a cell with a high error value.

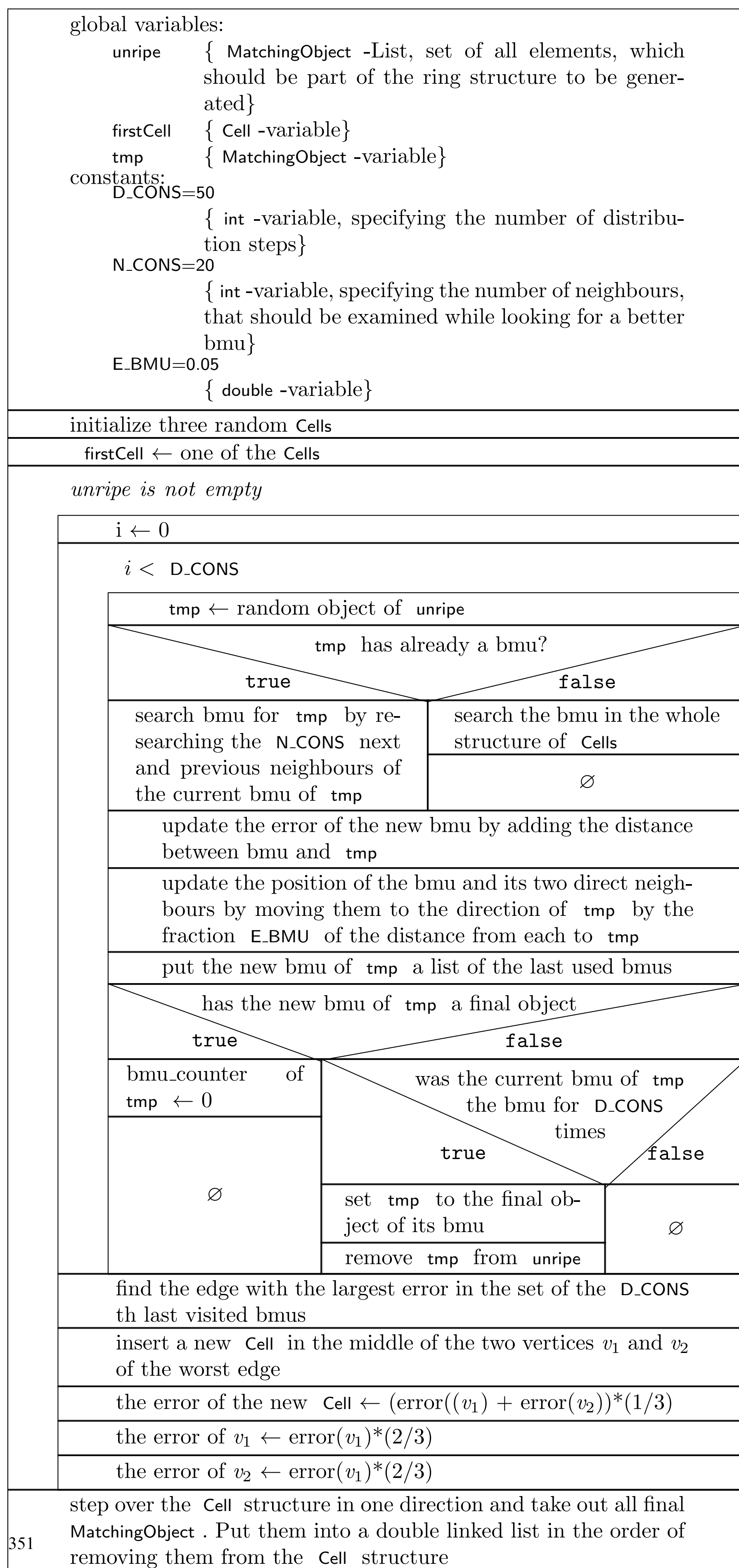
Step 3 All bookkeeping and pinning operations can be performed in constant time $O(1)$.

Steps 1 -3 The loop is performed n-times, all steps require only constant time, therefore the time complexity is linear to n, i.e. $O(n)$. The required space consists of memory space for the nodes and the cells. The maximum number of cells is equal to the number of nodes, i.e. $O(2*n) = O(n)$, which is linear in n too.

Step	Action
Init	Start with a ring structure consisting of three cells at randomly chosen positions. Initialize all error variables with 0.0.
Step 1	Perform a constant number $n_{distribution}$ of distribution steps. Update the error variable of the *bmu* (best matching unit) after every step. Keep a reference from each cell to the cell, which was most recently its bmu. When the cell is chosen again the search of its bmu is restricted to its last bmu and its neighbourhood up to $k_{neighbour}$ in each direction. $k_{neighbour}$ is a constant
Step 2	Find the edge e_{worst} in the set of high error cells connecting two cells v_1 and v_2, such that the *edgeerror* $$err(e_{worst}) := err(v_1) + err(v_2)$$ is a maximum for all edges in the ring structure. Insert a new cell c_{new} in the centre of e_{worst}. Initialize the error variable of the new cell with $$err(c_{new}) := \frac{1}{3} err(v_1) \frac{1}{3} err(v_2)$$ Adjust the error variables of v_1 and v_2 such that the total error of v_1, v_2 and c_{new} stays constant: $$err(v_1) := \frac{2}{3} err(v_1) + \frac{2}{3} err(v_2)$$
Step 3	If a cell has been bmu to a node *threshold* times the cell is pinned to that node and removed from the set of free cells. When every node has an associated cell, a solution is found. Otherwise continue with Step 1

Table 1: The $O(n)$ version of Flexmap [Fritzke and Wilke(1991)]

3.2 FlexMap Description



After the FlexMap algorithm is applied to the original matching problem we have two circular structures. One *req* linking the requirements and the other *all* linking all requirements and offers in a ring.

3.3 FlexMatch

The next step is to calculate the matching. Again we apply a growing cell structure algorithm, this time it's called FlexMatch[Nth(2014)].

Table 2 shows the $O(n)$-version of the FlexMatch algorithms in detail, followed by it's structogram. The structogram does not include step 5.

Step	Action
Step 1	Step over the circular structure *all* linking all objects. Take out all requirements having an offer as direct neighbour in the structure. and insert the pair of requirement and offer into a linked list *requested*. If one requirement has two direct neighboured offers take the one with the smaller distance.
Step 2	Step through the *requested* list and research if there are better offers for requirements. Take a pair of the list and check if the current offer of the requirement has an offer with smaller distance to the requirement in its direct neighbourhood. If this is the case swap the current offer with the better one.
Step 3	Extract the requirement of the first pair of the *requested* list as left restricting object and the requirement of the next pair in the list as right restricting object. Use only linking requirements to navigate through the list. Take the next requirement of the requirement list of the left restricting object.
Step 4	Test whether the euclidean distance to the offer of the left or of the right restricting object is smaller to the current requirement. Put the pair of the researched requirement and the smaller offer in a linked list of pairs *matchingList*. Take the next element of the current requirement of the requirement list. If this element is the right restricting object set the left restricting object to the right one and for the left one take the next element of the *requested* list. Continue with step 3. Otherwise continue with step 4. Go to step 5 when all elements of the requirement list have be processed.
Step 5	Take the structure *req* and choose one requirement *tmp* (is linked in both ring structures) . Research if the n_{better}[1] neighbours of *tmp* in both direction include a offer having a smaller distance to *tmp* then the current offer of *tmp*. If there is a better one exchange it with the current one of *tmp*. Take the next requirement of the structure *req* and repeat step 5 until you get to the first choosen requirement of step 5.

Table 2: The FlexMatch

[1] In the current implementation n_{better} hast the value 20.

3.4 FlexMatch Description

local variables:
requirement { MatchingObject -List}
all { MatchingObject -List}
matchingList { MatchingPair -List}
requested { MatchingObject -List}
tmp { MatchingObject }

i ← 0

i < sizeof(all)

- i ← i + 1
- tmp is a requirement element?
  - true:
    - tmp has a offer element as direct neighbour?
      - true:
        - set the offer neighbour with the smaller distance to tmp to the offer of tmp
        - put tmp into requested
      - false: ∅
  - false: ∅
- tmp ← i -th element of all

i ← 0

i < sizeof(requested)

- tmp ← i -th element of requested
- has previous or next element of tmp a offer with a smaller distance to tmp
  - true: put the better offer to the offer of tmp and put them into the matchingList
  - false: put tmp and its offer into the matchingList
- i ← i + 1

i ← 0

tmp ← the first element of requirement

requirement is not empty

- a ← i -th element of requested
- b ← i+1 -th element of requested
- the offer of a has a smaller distance to tmp
  - true:
    - put the offer of a to the offer of tmp and put them into the matchingList
    - ∅
  - false: put the offer of b to the offer of tmp and put them into the matchingList
- remove tmp from requirement
- tmp ← the next element of tmp in requirement
- i ← i + 1

4 Difference to other matching algorithms

The most significant differences between the FlexMatch-Algorithm and non approximative matching algorithms is the cardinality of mapping and the runtime behaviour. The Gale-Shapley-Algorithm(GSA) [Gusfield and Irving(1989)] allows only a 1:1-mapping. This means every element of the entry set gets matched to one other element. The FlexMatch allows a 1:n-mapping. One element of the offering set can be proposed to one or more elements of the requesting set. The aim of algorithms like the GSA or the Hungarian Algorithm (HA) is the common weal. The FlexMatch however tries to find a solution having the priority of the individual satisfation for every element of the requesting set. The runtime complexity of the GSA is in $O(n^2)$ [Gusfield and Irving(1989), page 8] and of the HA it is $O(n^3)$ [James Munkres(1957)]. Some approximative matching algorithms[Vinkemeier and Hougardy(2005)] [Duan and Pettie(2010)] have a linear or nearly linear runtime, but they also do not permit 1:n mappings in their solutions.

5 Results

System

- CPU: Intel(R) Core(TM) i3-2350M CPU @ 2.30GHz
- Memory: 4GiB
- Compiler: Java version 1.7.0_25
- Runtime environment: OpenJDK Runtime Environment (IcedTea 2.3.10) (7u25-2.3.10-1ubuntu0.12.10.2)
- VM: OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)

The test data sets were generated randomly. The single elements of a test set belong to two subsets having the same size. One is the set of requirements and one is the set of offers. The test sets have three different characteristics:

Randomness All elements of both sets are randomly distributed in a defined area

Independency The sets of the requirements and the offers have no overlapping area.

Overlapping The elements of the requirements lie inside the area of elements of the offers.

Fig. 3 shows how the characteristics of the sets would look like in the two dimensional space. The varying problem sizes were 100, 300, 500, 700, 1000, 3000, 4000, 5000, 6000, 8000 and 10000. For each size and characteristic 25 sets were generated. So there are 10*25*3 = 750 test sets. The FlexMatch was run five times for every set.

Fig. 4 shows the different runtime behaviour of the FlexMatch and of the brute force algorithm testing a element of a subset with all elements of the other subset. The computation of the runtime was done by taking the average

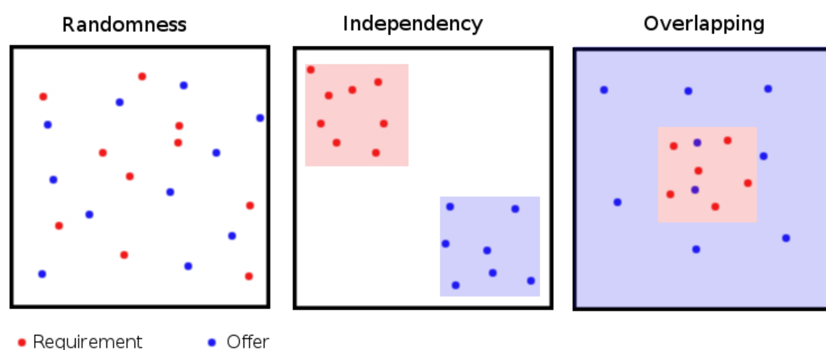


Fig. 3: Characteristics of the sets

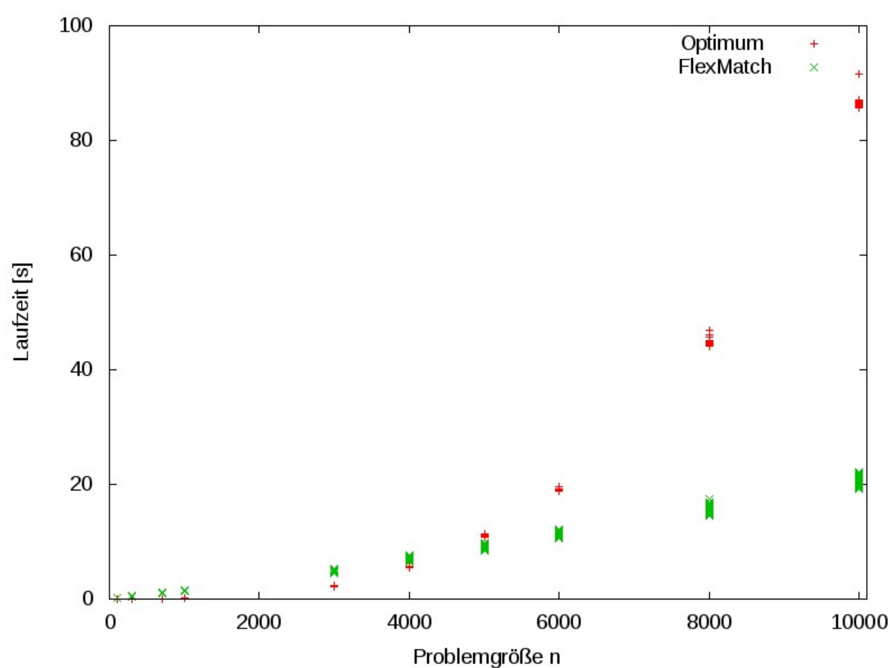


Fig. 4: Runtime behaviour

execution time per test set. The bottom line is that the FlexMatch algorithm shows a linear complexity and outperforms the brute force algorithm when the problem size exceeds the break even point at approx. 5000 elements.

Fig. 5 shows that most tests have a relative error smaller than 20%. The relative error $= (\frac{sumFM}{sumOpt} - 1) * 100\%$ was computed by summing up the distance

between all matched pairs being found by the brute force algorithm ($sumOpt$) and the FlexMatch ($sumFM$).

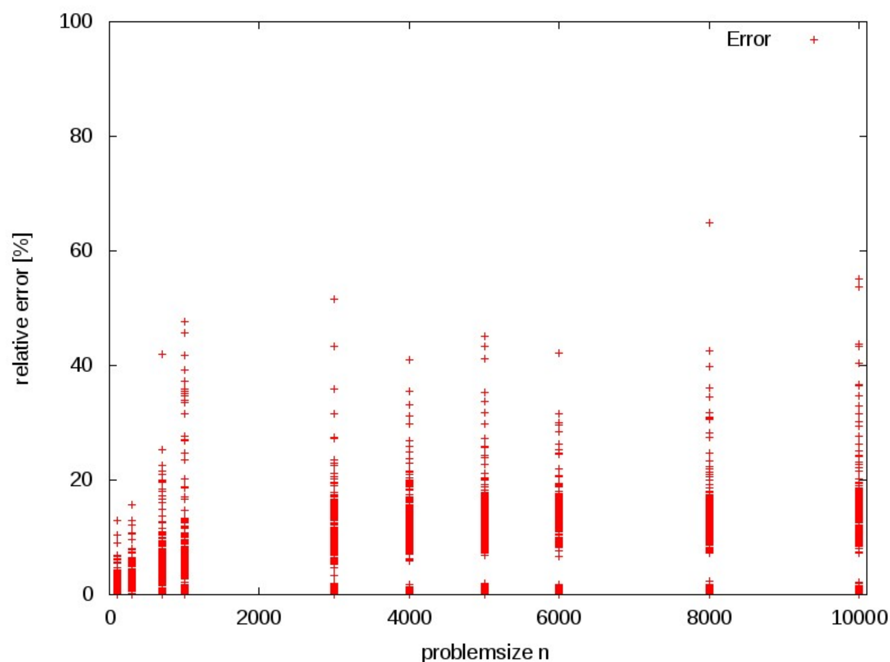


Fig. 5: Error statistics

Fig. 6 shows quintessential how the cell ring structure over all elements of the two sets generated by the FlexMap based part of the FlexMatch could behave in a two dimensional space with a problem size of 20. Fig. 7 and 8 represent the ring structures *req* linking all requirements and *all* linking all requirements and offers. Fig. 9 Shows the suggested matching pairs as a result of using the neighbourhood relation of the two ring structures.

6 Summary and outlook

In this paper we have shown how a matching algorithm with linear time and space complexity using a self organising map is designed and implemented. The matching algorithm yields results of sufficient quality but some artefacts have to be investigated as their costs are beyond the statistical expectations.

Currently we are running experiments using different data sets, initial situations and problem size and would like to improve the results and runtime behaviour further more.

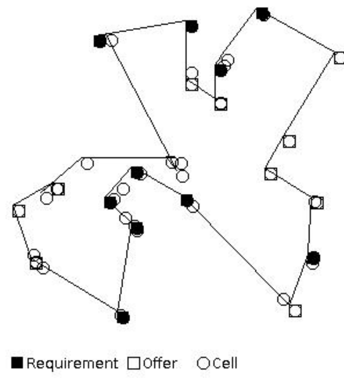


Fig. 6: Cell-Structure

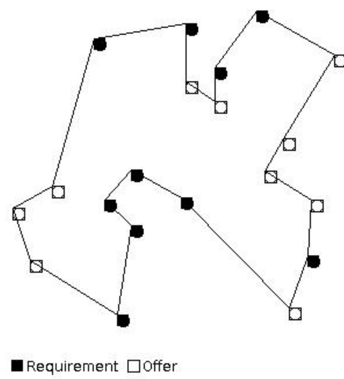


Fig. 7: Requirements and offers linked in a structure

The next steps would be the deployment of the software in the web-based matching portal and analysis of the real world problem behaviour.

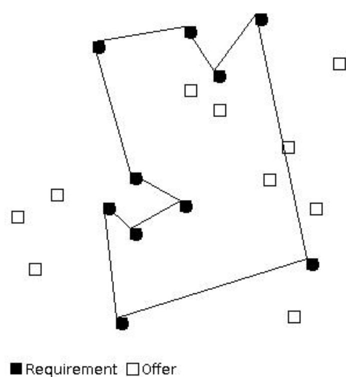


Fig. 8: Requirements linked in a structure

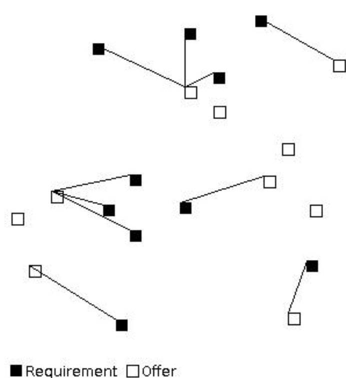


Fig. 9: Linked matchingpairs

References

[Duan and Pettie(2010)] Duan R, Pettie S (2010) Approximating maximum weight matching in near-linear time. In: Proceedings 51st IEEE Symposium on Foundations of Computer Science (FOCS), pp 673–682

[Fritzke and Wilke(1991)] Fritzke B, Wilke P (1991) FLEXMAP - A neural network with linear time and space complexity forthe travelling salesman problem. In: Proceedings IJCNN Int. Joint Conf. Neural Networks, Singapore

[Gusfield and Irving(1989)] Gusfield D, Irving RW (1989) The stable marriage problem: Structure and algorithms. Foundations of computing, MIT Press, Cambridge and Mass

[James Munkres(1957)] James Munkres (1957) Algorithms for the assignment and transportation problems. Journal of the Society for Industrial and Applied Mathematics 5(1):32–38

[Noeth(2014)] Noeth N (2014) Optimierte Vergabe von Praktikumsplaetzen

[Vinkemeier and Hougardy(2005)] Vinkemeier DED, Hougardy S (2005) A linear-time approximation algorithm for weighted matchings in graphs. ACM TRANSACTIONS ON ALGORITHMS 1(1):107–122