Iranian Journal of Operations Research
Vol. 9, No. 2, 2018, pp. 31-48

Intelligent Mapping

A. Fathollahzadeh[1]

This paper is directed to the question of how to model and design an efficient tool for the intelligent mapping which is based on both dynamic and efficient storage of data and soft computing. The former is performed by our method that learns how to store, search and delete the data. After pointing out the limitation of the crisp evaluation of the distance between two points, we argue in favor of soft computing which is based on the extension of metric space to interval one and then to the fuzzy metric. A-Star algorithm is used to illustrate our model along with the injection of competitive data structures.

Keywords: *Automata, Classification, Possibility and necessity measures, A-star.*

Manuscript was received on 13/09/2018, revised on 21/11/2018 and accepted for publication on 28/11/2018.

1. Introduction

Classical map matching determines which road a vehicle is on, based on inaccurate measured locations, such as GPS points, where 10 nearest road segments in a radius of 200 meters are searched. This approach uses the observation probability.

Among the drawbacks of this approach, we can mention twofold limitations. The first one is the static storage of data, where there is no way to insert a new data; *e.g.*, it is not rare to observe a new building in a location just in a period of two months, or the destruction of an existing house, etc. The second one is the poor performance of the hard computing which often make guiding search impossible.

The contribution of this paper is an aid to the realization of the intelligent mapping along with the following properties:

- Acceptable time's query.
- Flexible software for being injected in the final engine using Galileo.

Here, flexible means how efficient the collection of information associated with locations can be done. In particular, how these data, *i.e.*, both locations and information associated with locations, called hereafter, for short, keys and key-values, respectively, can efficiently and dynamically be stored along with the data transmitted by Galileo which is the global navigation satellite system is being created by the European Union through the European Space Agency.

The rest of this paper is organized as follows. Section 2 describes how multiple key-values (for short, values) of the same key can be collected efficiently. Then how to learn for storing, searching and deleting dynamically all the data is outlined in Section 3. Using soft computing along with the extension of metric space to interval one and then to the fuzzy metric being advocated by a method

1 Artificial Intelligence Group, University of Tabriz, Tabriz, Iran, Email: abfzadeh@gmail.com.
Part of this work is done in the University of CentraleSupelec, France.

which provides a unique data structure is the theme of Section 4. The A-star algorithm being used to design the tool along with the injection of competitive data structures is described in Section 5. The paper ends with concluding remarks in Section 6.

2. Efficient and Dynamic Collection of Key-values

In intelligent mapping, we may have the same location with two or more geographical indications, such as two ones which are situated in Iran and France, respectively:

Gul-Tapé	$\{30, 91\}$
Saint-Colombe	$\{05, 16, 17, 21, 25, 33, 35, 40, 46, 50, 69, 76, 77, 89\}$

where each number indicates a particular geographical department in the mentioned country. In other words, we have to deal with the *homographs* (h_g), *i.e.*, a word defined over a finite alphabet associated with two or more values.

In the statistic community parlance [2, 3], building dynamic homographs of large data can be viewed as an important part of the factor selection. Factor selection is always a difficult task just as in general model selection of the machine learning, statistic, etc. For this purpose, we need a more robust method, thereby helping to reduce the model uncertainty.

In addition to the traditional approaches such as the aggregation in a linear form defined by the full truth or the true model in terms of factor [20], we have identified a **new** challenging task that improves upon the quality of factor selection with respect to the elimination of the redundancy. This is done by the injection of our homograph method [6] into the factor selection which is based on the efficient collection of homographs to reduce the redundancy. For instance, let us consider the word "*in*" in Table 1.

Table 1. Reducing the redundancy via homograph. '?' stands for unknown value. **Left**: Input with 7 entries, **Right**: Output with 4 entries including 2 homographs (h_gs) and 2 sets of homograph-values (h_vs).

In	prep
for	prep
In	
Of	prep
In	{adv, adj}
good	prep
In	{adj, adv}

$\Rightarrow$

in	{?, {**adj**, **adv**}, **prep**}
for	{adj, adv}
of	{*prep*}
good	{*prep*}

As appears from the left part of Table 1, we may except to find any permutation of any subset of a set with n elements. The maximum number of the subsets of S, plus S, is based on the following two theorems [6].

Theorem 2.1. The number of permutations of n distinct objects taken r at a time, denoted by P_r^n, where repetitions are not allowed, is given by $P_r^n = \frac{n!}{(n-r)!}$.

Table 2. $\#\pi$: The number of the permutations of a set S with n elements. The symbols ✓ and ✓ denote insignificant and impossible, respectively. The impact of $\#\pi$ on the calculation of the maximum number of the sets and the subsets of S, $\max(\#h_v, n)$ is obvious, *e.g.*, for $S = \{a, b, c\}$, $\max(\#h_v, 3) = 1 + 3 + 3! + 3! = 16$.

n	$\#\pi$	mi/sec	bi/sec	tr/sec
10	3628800	✓	✓	✓
11	39916800	seconds	✓	✓
12	479001600	minutes	✓	✓
13	6227020800	hours	seconds	✓
14	87178291200	day	minutes	✓
15	1307674368000	weeks	minutes	seconds
16	20922789888000	Month	hours	seconds
17	355687428096000	Years	days	minutes
18	6402373705728000	✓	month	hours
19	121645100408832000	✓	✓	days
20	2432902008176640000	✓	✓	month

Theorem 2.2. The maximum number of the subsets of S with n elements, including the empty set and S is denoted by $\max\big((\#pi_{hv}, n)\big)$, where repetitions are not allowed, is: $\max(\#pi_{hv}, n) = 1 + \sum_{r=1}^{n} \frac{n!}{(n-r)!}$.

Note that processing a permutation often, as in our case, costs much more than generating it. Table 2 shows the situation is even worse for the calculation of $\max(\#h_v, n)$.

We have designed a *linear-time* algorithm [6] for this task. Table 3 shows the result of our algorithm applied to the classification a geographical domain in 13 classes, where each class has the same number of alternated department-codes (e.g., $\{59, 60\}$).

2.1. Main Algorithm

The main algorithm of building dynamic homograph (*bdh*) operates in six modes: *insertion*, *deletion*, *retrieval*, *save*, *restore* and *dump* (i.e. to show the contents of the data structures).

A *string* is a sequence of zero or more symbols from an alphabet **Sigma**. The length of a string x is denoted by $|x|$. We will treat string as array in the *C* programming language. So $x[0]$ shall denote the first character of x, $x[1]$ its second character, etc.

Table 3. Construction of 1408 **new** sets of French City and Villages.

Cardinal	**Total**	**Example**	**Cardinal**	**Total**	**Example**
2	805	Abancourt	9	11	Montigny
3	279	Fours	10	5	Le Pin
4	98	Artigues	11	3	Saint-Loup
5	56	Vaux	12	2	Beaulieu
6	21	Castillon	13	2	Sainte-Marie
7	14	Mons	14	1	Sainte-Colombe
8	11	Bagneux	15	0	

For any $n \in \mathbb{N}$, let $[n] = \{0, 1, \dots, n-1\}$. The input is a *user-file* of the following (customary) form: $f = \{(k_i, v_i) \mid i \in [n]\}$, where each (k_i, v_i) represents an entry with strings k_i and v_i standing for a *key* and *key-value* (for short *value*), respectively. Hereafter, word, string and key is used interchangeably.

The preprocessing phase of the main algorithm has 2 steps. The first step outputs f_t, the temporary file $f_t = \{(k_j, v_j) \mid j \in [n'], n' \leq n\}$ which identifies possible missing values and eliminates possible empty entries of f. The second step re-sizes f_t for obtaining the output, namely the temporary dictionary noted by t_{dict}. Since the best hash table sizes are powers of two, so we compute $TSize$ (i.e., the size of t_{dict}), by invoking the function RoundUpPow2, which rounds up to the next highest power 2 of n'. For not to do slow operation of modulo a *prime*, we used 32(N)-bits based of the Jenkins hash function [9] noted by JenkinsHash to hash a variable-length key into a 32-bit value. Since t_{dict} contains at most $TSize =$ RoundUpPow2 (n') keys, to optimize yet the hash function, another function, namely, Ilog2U32 is invoked which uses *De Bruijn logarithmic index* to compute the log base 2 of an N-bit integer in $O(lg(N))$ operations with multiply and lookup.

Below, we describe the processing of the first (insert) mode. We write p_{dict} to refer to the permanent dictionary which may be empty (*i.e.*, no previous file has been submitted to *bdh*) or acquired by past or multiple uses of the above modes.

Let $k \in t_{dict}$. We write v_p and v_c to refer to the values of p_{dict} and t_{dict}, respectively. Assume p_{dict} is empty. If k is a homograph, then the function PSUB(v_c) is invoked to form v_p, the output set, and the pair (k, v_p) is stored into p_{dict}. If not, only (k, v_c) is stored into p_{dict}.

n	n′	Ticks	Seconds
1640	1571	3772	0.004
5802	5802	9701	0.010
9675	7256	13785	0.014
25273	25247	35567	0.036
32715	32052	42979	0.043
39323	32082	52105	0.052
53668	46330	72751	0.073
287872	212115	421609	0.422

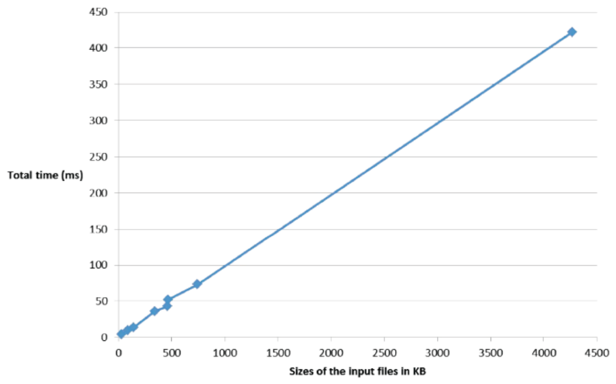


Figure 1. Time measures

Algorithm 1: Insertion algorithm.

```
    Input: key ($k$), value ($v_c$) and $p_{dict}$.
    Output: Update $pdict$
1  $h \leftarrow$ JenkinsHash ($k$), $v_g \leftarrow p_{dict}[h].value$, $P \leftarrow v_g.q$, $Q \leftarrow v_c.q$;
2  if (P=0) then
3      $pdict[h].value \leftarrow (Q > 1)?$ PSUB ($v_c$):$v_c$;
4  else
5      if ((P = 1) and (Q = 1)) then
6          $data_1 \leftarrow v_c.data[0]$, $data_2 \leftarrow v_g.data[0]$
7          if ($data_1 \neq_S data_2$) then
8              $v.data \leftarrow (data_1 <_S data_2)?\ data_1 + data_2 : data_2 + data_1$;
9              $v.q \leftarrow 2$, $pdict[h].value \leftarrow v$;
10     else
11         $v.q \leftarrow P + Q$, $v.data \leftarrow v_c.data + v_g.data$, $pdict[h].value \leftarrow$ PSUB ($v$)
12 function ConsTreap ($v$) Removes any duplicate.
13 begin
14     $half = (v.q)/2$, $data \leftarrow v.data$
15     if ($v.q = 1$) then return (CreateNode (data[0]))
16     $v_1.data \leftarrow data$, $v_1.q \leftarrow half$; i.e. Split the data in half
17     $v_2.data \leftarrow data + half$, $v_2.q \leftarrow v.q - half$;
18     return (Union (ConsTreap ($v_1$), ConsTreap ($v_2$)))
```

When p_{dict} is not empty, lines 5-11 of Algorithm 1 shows how p_{dict} can properly be maintained depending on v_p and v_c. If both v_p and v_c are simple and distinct strings, then in lines 6-10 the sorted concatenation of v_p and v_c without using treaps [1] is used. Otherwise, the function PSUB(v_c) is invoked in line 13.

Each call of PSUB uses the operation Union. Given two treaps tr_1 and tr_2, Union (tr_1, tr_2) returns a treap tr that is the union of the two of them. To maintain the heap order, the root of tr has the largest priority. Let v be the key of tr. Union *splits* the other treap by v into a less-than v and greater-than treap with values greater than v, and possibly a duplicate node with a value equal to v. Then recursively it finds the union of the left child of tr and the less-than treap and the union of the right child of tr and the greater-than treap. The results of the two unions of operations becomes the left and right subtrees of tr, respectively.

Algorithm 2: Union($tr1$, $tr2$).

```
1  if ($tr1 = NULL$) or ($tr2 = NULL$) then return ((tr1)? tr1:tr2);
2  if ($tr1.priority < tr2.priority$) then root = tr1, tr1 = tr2, tr2 = root;
3  $duplicate \leftarrow$ Split($\&left, \&right, tr2, tr1.value$);
4  $tr1.left \leftarrow$ Union($tr1.left, left$), $tr1.right \leftarrow$ Union($tr1.right, right$)
5  return (tr1);
```

The operation Split destructively splits the treap tr into two treaps: the "left treap" is a treap with key-values less than "value" and the "right treap" is a treap with values greater than "value".

Split searches down the treap in key-value order to find the root of the subtreap that contains key-values that are too large (small) to be in the left (right) subtreap (then changes direction of the search). Split makes this subtreap the other branch of right (left) subtreap. This subtreap in turn has nodes that are too small (large) and need to be moved back to where the subtreap was taken, and so on until it reaches a leaf or a node with the same key as "values". Split returns either the leaf (NULL) or this node. The expected time to split two treaps into two treaps of size n and m is $O(\lg n + \lg m)$. Figure 1 reports the time measures done on 8 input files of large data, where there were neither the set of key-values, missing key-values nor the commentary texts.

3. Learning How to Store

For the application of large-scale dictionaries two major problems have to be solved: *fast lookup speed* and *compact representation*. Using *automata* we can achieve fast lookup by determinization and compact representation by minimization. For providing information for the recognized words one can use the *transducers* (i.e., automata with outputs) [15, 19].

Finite-state transducers can be used to map a language onto a set of values. We have introduced an alternate representation [7] for such a mapping, consisting of associating a finite-state automaton accepting the input language with a decision tree representing the output values. The advantages of this approach are that it leads to more compact representations than transducers, and that decision trees can easily be synthesized by machine learning techniques. We have proposed a competitor to the transducers [7, 5] which combines *automata* and *machine learning* theories with the following desired properties:

1. The number of the states (and hence the transitions) representing the input language of our method is less as compared to the transducers.

2. In constructing transducers, we have to represent every transition by a data structure of at least two fields: one for the symbol representing the transition, another for the label-value (for short, label) associated with the symbol. So, in order to properly calculate the outputs, the label set needs to have the algebraic structure e.g., semiring in the case of weighted automata [15, 19]. In our approach the transitions are not labeled with outputs; the cost of exploring the automata is low.

3. In most applications (e.g., those of using part of speech tagging) there may be (many) identical output values. When you use the transducers there is no guarantee to save the amount of space for the identical information, whereas in our approach such economy is allowed.

In order to explain intuitively the benefits of our method, we give a very simple example as following. Let $V = \{Asia, Europa\}$ be the output values of three following countries: $K = \{Iran, Iraq, Ireland\}$. In order to determine the output values of any element of K one can *learn* the *decision tree* based on the mutual information; if the key (of K) ends with 'n/q' then retrieve '$Asia$'} else '$Europa$'.

An acyclic finite-state automaton: a graph of the form $g = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, q_0 is the start state, $F \subseteq Q$ is the accepting states. δ is a partial mapping $\delta: Q \times \Sigma \to Q$

denoting *transition*. For $a \in \Sigma$, the notation $\delta(q, a) = \perp$ is used to mean that $\delta(q, a)$ is undefined. The extension of the partial δ mapping with $x \in \Sigma^\star$ is a function $\delta^\star: Q \times \Sigma^\star \to Q$ and defined as follows:

$$\delta^\star(q, \varepsilon) = \quad q$$
$$\delta^\star(q, ax) = \begin{cases} \delta^\star(\delta(q, a), x), & \text{If } \delta(q, a) \neq \perp \\ \perp, & \text{otherwise.} \end{cases}$$

The property $\delta^\star$ allows fast retrieval for variable-length strings and quick unsuccessful search determination. The pessimistic time complexity of $\delta^\star$ is $\mathcal{O}(|x|)$ with respect to a string x. A finite automaton is said to be (n_s, n_t) −automaton if $|Q| = n_s$ and $|E| = n_t$, where E denotes the set of the edges (transitions) of g.

Table 4. Left: The profit trend of 10 restaurants where CP, HB and CK stand for competition, Hamburger and Chelo-Kabab (Iranian food), respectively. **Right:** Its compressed form. the strings of three first columns of each row of the left Table is compressed in one string using the first character of each string (e.g., “old” is transformed into ‘o’).

Age	CP	Type	Profit
old	no	CK	▼
midlife	yes	CK	▼
midlife	no	HB	▲
old	no	HB	▼
new	no	HB	▲
new	no	CK	▲
midlife	no	CK	▲
new	yes	CK	▲
midlife	no	HB	▼
old	yes	CK	▼

String	Value
onC	▼
myC	▼
mnH	▲
onH	▼
nnH	▲
nnC	▲
mnC	▲
nyC	▲
mnH	▼
oyC	▼

3.1. More Illustrations

The framework for learning the output language of an input language is described in [5, 7]. In this subsection, we only illustrate that method using another example. A decision tree (dt) is a direct acyclic graph of nodes and arcs. At each node, a simple test is made; at the leaves a decision is made with respect to the class labels (values associated with a word in our case).

Example: The left part of Table 4 shows the data for 10 restaurants using four attributes. One can find out the attribute age is the best to be selected at first; this indicates that it is most likely that a decision can be made quickly if one first asks for the age of a restaurant. If the answer to this question is ‘new’ or ‘old’, then the profit can be predicted by ‘up’ or ‘down’, respectively. If the answer is `midlife', then another question must be posed, about the presence of competition. After this answer is known, the profit trend can be determined.

Figure 4 shows the data structure of a node of m-ary decision. The first field contains a nonnegative integer, say i for $0 \leq i \leq \ell$, where ℓ denotes the length of the longest word(s) of the input language. If $i = 0$, this means that the node is a leaf one, otherwise the node is an internal one (including the root node). The second filed represents either a best string or the output value. The third filed is m-pointers to other nodes, each indicating which node has to be followed in the tree when searching the output value.

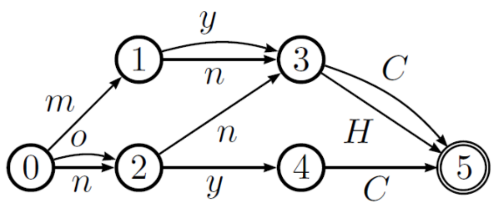


Figure 2. A (6,10) automaton for recognizing ten keys of the right part of Table4.

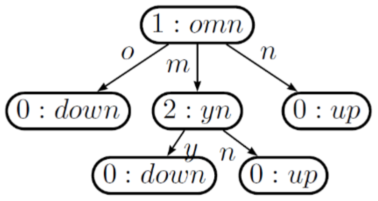


Figure 3. Learned m-ary decision tree of the right part of Table 4.

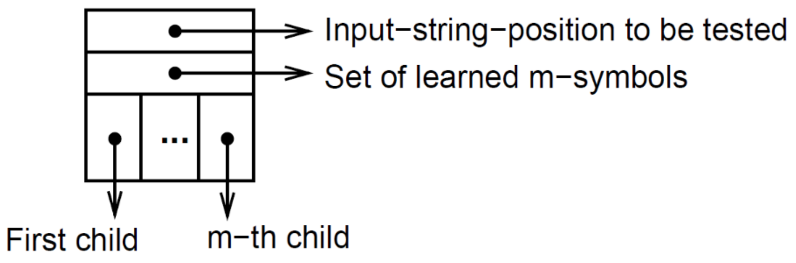


Figure 4. Node of m-ary decision tree.

A decision tree (dt) is a direct acyclic graph of nodes and arcs. At each $node$, a simple test is made; at the $leaves$ a decision is made with respect to the class labels (values in our case). The dt is introduced in the machine learning (ML) community [13].

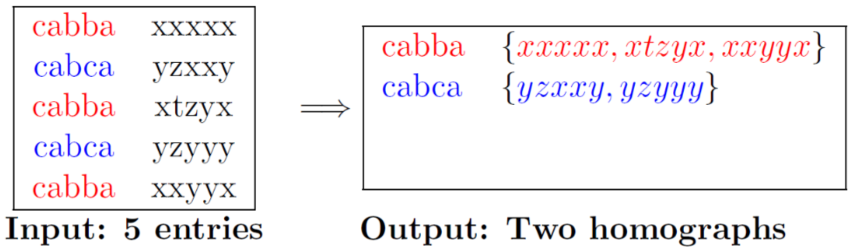
At the top of this tree expressed by Figure 4, by way of three rules situated in the right one can see the attribute age; this indicates that it is most likely that a decision can be made quickly if one first asks for the age of a restaurant. If the answer to this question is ‘new’ or ‘old’, then the profit can be predicted by ‘down’ or `up', respectively. If the answer is ‘midlife’, then another question must be posed, about the presence of the competition. After this answer is known, the profit trend can be determined.

3.2. Compact Representation of Homographs

As we have mentioned the advantage of our method compared to the transducers for storing the keys and their values are saving the space and gaining time. In this subsection, first the formal definition of the transducer [15] is given, then an example is used to illustrate the advantages of our method.

Formally, a finite transducer T is a 6-tuple $(Q, \Sigma, \Gamma, I, F, \delta)$ [14] such that Q is a finite set, the set of states, Σ is a finite set, called the input alphabet, Γ is a finite set, called the output alphabet, I is a subset of Q, the set of initial states, F is a subset of Q, the set of final states; and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ (where ϵ is the empty string) is the transition relation. We can view (Q, δ) as a labeled directed graph, known as the transition graph of T: the set of vertices is Q, and $(q, a, b, r) \in \delta$ means that there is a labeled edge going from vertex q to vertex r. We also say that a is the input label and b the output label of that edge.

The extended transition relation δ^* is defined as the smallest set such that $\delta \subseteq \delta^*$, $(q, \epsilon, \epsilon, q) \in \delta^*$ for all $q \in Q$, and whenever $(q, x, y, r) \in \delta^*$ and $(r, a, b, s) \in \delta$ then $(q, xa, yb, s) \in \delta^*$. Figure 5 shows the transducer of the following input which is first compacted by bdh in a compact file for being represented by a graph of (13,16).



cabba	xxxxx
cabca	yzxxy
cabba	xtzyx
cabca	yzyyy
cabba	xxyyx

Input: 5 entries

$\Longrightarrow$

cabba	$\{xxxxx, xtzyx, xxyyx\}$
cabca	$\{yzxxy, yzyyy\}$

Output: Two homographs

Figure 6 shows that our solution is competitive along with an easy usage. That is to say, given x (i.e. user-string), if it can be spelled out by the unlabeled automaton of the input language (K), then use the decision tree to output its value, else return *nil* witness for failure i.e., $x \notin K$.

Examples: Consider $x =$ onC along with g the unlabeled automaton of Figure 2. Invoking $\delta^*(0, x)$ confirms that $x \in K$ where K is the keys of Table 3. So, its decision tree shown in Figure 4 is used to output the value of x. As in case of the homograph, let $x =$ cabba. Similar to the previous example, via Figure 6, the value $\{xxxxx, xtzyx, xxyyx\}$ is returned.

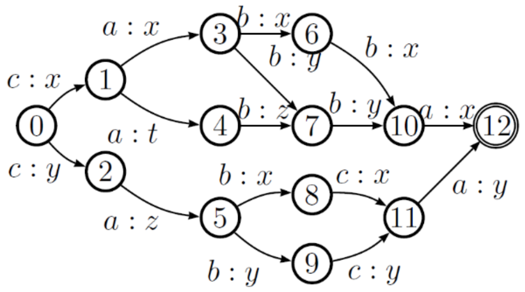


Figure 5. Transducer: g = (13,16). Source: A. Kempe, Xerox Research Center Europe [11].

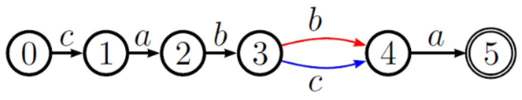


Figure 6. Our alternative: A (6,6)-automaton along with one decision rule. based on colored-transitions i.e., if $b_2 =$'b' then $\{xxxxx, xxyyx, xtzyx\}$, else $\{yzxxy, yzyyy\}$, where b_2 denotes the second character from right to left of any key (of input language) which is already recognized by this automaton.

4. A-Star

A-star algorithm [17, 18] is used in path finding and graph traversal, which is the process of finding an efficiently directed path between multiple points, called "nodes". It enjoys a widespread use due to its performance and accuracy.

A-star has been studied extensively by researchers along with several variants [10] formulated in terms of exact weighted graphs (e.g., Figure 7 (a): starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node. At each iteration of its main loop, A-star needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A-star selects the path that minimizes $f(n) = g(n) + h(n)$, where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ gives an heuristic estimating the cost of the cheapest path from n to the goal. The heuristic is problem-specific. The traveling salesman problem (TSP) is the problem of finding a minimal cost closed a tour that visits each location once. Below, using Figure 7 (a), a description of how g and h of TSP can be calculated is provided.

4.1. Computing g and h

For the algorithm to find the actual shortest tour, the heuristic function must be admissible, meaning that it never over estimate the actual cost to get to the nearest goal node.

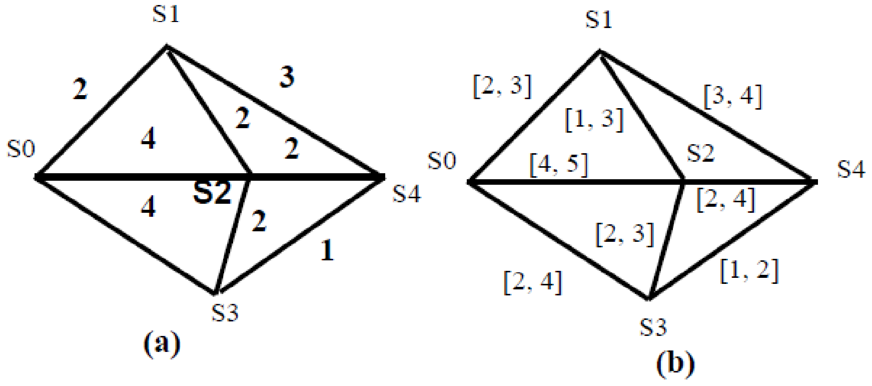


Figure 7. Exact and inexact costs.

Typical implementations of A-Star use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

Given the graph $g = (S, A)$, where S and A denote the set of vertices and edges, respectively, first, we compute the *minimal costs* arriving at each vertex of g, along with vertices, denoted by *TMC*. Table 5 (a) shows the calculation using the costs of Figure 7 (a).

Table 5. Minimal costs arriving at each vertex of Figure 7.

s_0	s_1	s_2	s_3	s_4
2	2	2	1	1

(a)

s_0	s_1	s_2	s_3	s_4
[2, 3]	[1, 3]	[1, 3]	[1, 2]	[1, 2]

(b)

Table 6. Evaluations of two optimal circuits of Figure 7.

Path	g	h	f
s_0s_1	2	6	8
$s_0s_1s_2$	4	4	8
$s_0s_1s_2s_4$	6	3	9
$s_0s_1s_2s_4s_3$	7	2	9
$s_0s_1s_2s_4s_3$	11	0	11

Path	g	h	f
s_0s_1	[2,3]	[5,10]	[7,13]
$s_0s_1s_2$	[3,6]	[4,7]	[7,13]
$s_0s_1s_2s_4$	[5,10]	[3,5]	[8,15]
$s_0s_1s_2s_4s_3$	[6,12]	[2,3]	[8,15]
$s_0s_1s_2s_4s_3s_0$	[8,16]	0	[8,16]

(a) (b)

Let $s_x \in S$. We write $assoc(s_x, TMC)$ which returns c_x the minimal cost arriving at c_x. The minimal cost for traversing from the node $n = (s_0, s_i, \cdots, s_j)$ to $n' = (s_0, s_i, \cdots, s_j, s_k)$ where $(s_j, s_k) \in A$ and $s_k \notin (s_i, \ldots, s_j)$ is $f(n') = g(n') + h(n')$ calculated as follows: $g(n') = c_{0,k} + \cdots + c_{j,k}$, $\sum_{\forall s_x \notin \{s_i, \cdots, s_j, s_k\}} c_x$, where $c_x = assoc(s_x, TMC)$.

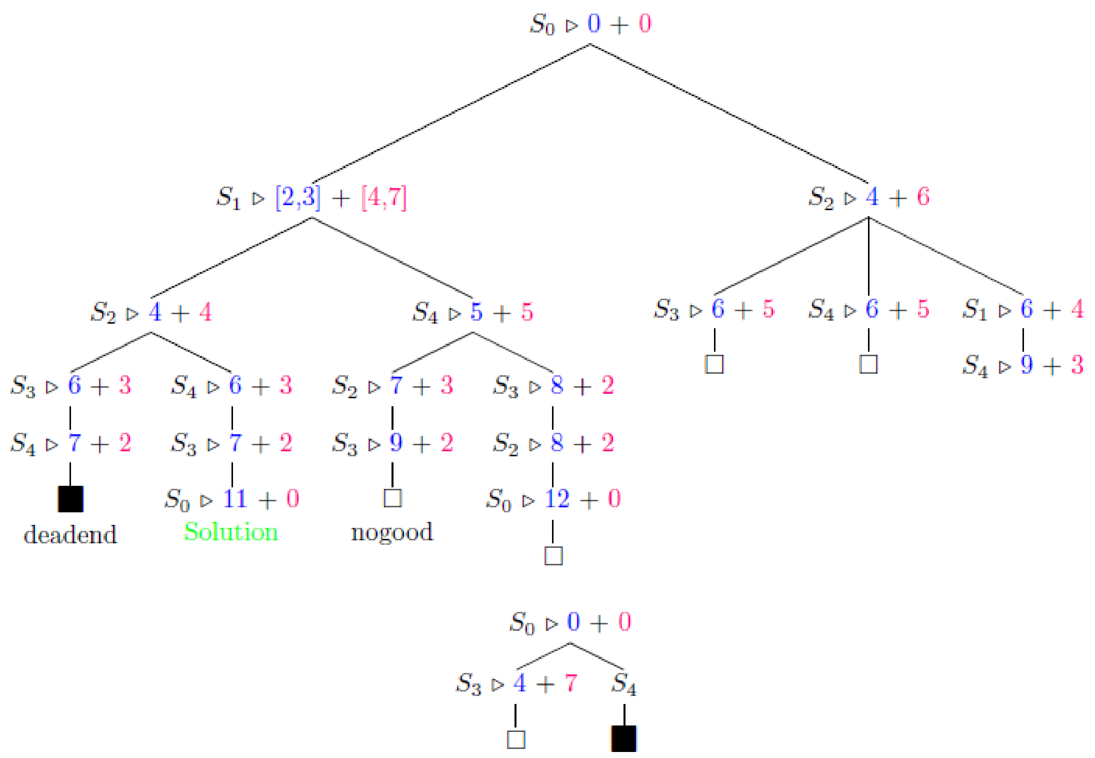

$S_0 \triangleright 0 + 0$
$S_1 \triangleright [2,3] + [4,7]$
$S_2 \triangleright 4 + 6$
$S_2 \triangleright 4 + 4$
$S_4 \triangleright 5 + 5$
$S_3 \triangleright 6 + 5$
$S_4 \triangleright 6 + 5$
$S_1 \triangleright 6 + 4$
$S_4 \triangleright 9 + 3$
$S_3 \triangleright 6 + 3$
$S_4 \triangleright 6 + 3$
$S_2 \triangleright 7 + 3$
$S_3 \triangleright 8 + 2$
$S_4 \triangleright 7 + 2$
$S_3 \triangleright 7 + 2$
$S_3 \triangleright 9 + 2$
$S_2 \triangleright 8 + 2$
$S_0 \triangleright 11 + 0$
$S_0 \triangleright 12 + 0$
deadend
Solution
nogood
$S_0 \triangleright 0 + 0$
$S_3 \triangleright 4 + 7$
S_4

Figure 8. Search tree for graph of Figure 7 (a). The departure vertex is S_0.

Table 6 (a) shows the costs of some paths of the search for solving TSP of Figure 7 (a) shown in Figure 8, where the traversal order is depth-first. The minimal cost is 11. Notice the benefit of using the *TMC* in A-Star leading to not explore the entire tree which may be dramatically large. This is the reason why the classical map matching based on hard computing (e.g. Dijkstra's algorithm [4]) has serious limitations as mentioned earlier in the first paragraph of the introductory Section.

5. Soft Computing: Extension of Fuzzy Costs

In the case when the costs are fuzzy intervals, we must answer the three following questions:

- What will be the strategy of choosing a node (state) to develop?
- What will be the criteria for stopping the procedure?
- What will be the result at termination?

The choice of a state can be reduced to the problem of comparing fuzzy intervals $n_j = \left[\underline{f}(n_j), \overline{f}(n_j)\right]$, for $j = 1, 2, \dots, p$, where $n_1, \dots, n_p$ are the candidate states in development at the current moment (node). We are looking for the state having the smallest evaluation. When $p = 2$, we have to compare the relative position of the two intervals. So we naturally get four possible ranking criteria that indicate if n_1 is the state to be developed instead of n_2:

$$C_1{:}\, \underline{f}(n_2) \geq \overline{f}(n_1) \quad C_2{:}\, \underline{f}(n_2) \geq \overline{f}(n_1)$$
$$C_3{:}\, \overline{f}(n_2) \geq \overline{f}(n_1) \quad C_4{:}\, \overline{f}(n_2) \geq \underline{f}(n_1)$$

If C_1 is checked, despite the inaccuracy, we are sure that the evaluation of n_1 is better than that of n_2. On the other hand, if criterion C_4 is verified, we only know there is a possibility that n_1 has the smallest evaluation. When the criteria C_2 and C_3 are checked simultaneously, we can write

$$C_{23} = \widetilde{\min}\left(\left[\underline{f}(n_1), \overline{f}(n_1)\right], \left[\underline{f}(n_2), \overline{f}(n_2)\right]\right) = \left[\underline{f}(n_1), \overline{f}(n_1)\right],$$

where $\widetilde{\min}$ is the minimum operation applied at the intervals. The min operation allows one to define selection criterion C_{23} intuitively satisfying and being less strong than C_1. The selection criteria are naturally ordered according to their strengths [12]:

$$C_1 \;\Rightarrow\; C_{23} \begin{cases} \Rightarrow & C_2 \Rightarrow \; C_4 \\ \Rightarrow & C_3 \Rightarrow \; C_4 \end{cases}$$

In practice, it is suggested to select the node by applying the criteria in the order indicated by the implications above. Only C_2 and C_3 are naturally not ordered; we will choose priority C_2 if we think that the smallest values of $f(n_1)$ and $f(n_2)$ are more plausible than larger values. In the general case, where there are $p > 2$ states, we will be brought back to compare the evaluation of each n_j with $\widetilde{\min}[\underline{f}(n_k)), (\overline{f}(n_k)]$, for $k \neq j$, that is to say, with the smallest of the other assessments, using the five criteria in order suggested given above.

5.1. Classical Intervals

We consider the same problem of TSP, as given previously, but considering the costs being imprecise, that is to say that, for example, the duration of the journey connecting the cities is only imperfectly known. The function $g(n)$ is simply obtained by summing respectively the lower and upper bounds of the imprecise costs $\left[\underline{c}_{ij}, \overline{c}_{ij}\right]$ along the way corresponding to the state n. The function $h(n)$ is a sum of pre-calculated intervals for each vertex of the city graph (S, A) For the top i, we have

$$\left[\underline{h}_i, \overline{h}_i\right] = \widetilde{\min}\left[\underline{c}_{ij}, \overline{c}_{ij}\right], \qquad \text{for } (i, j) \in A,$$

and for $n = s_0 s_1 \dots s_j s_k$ we have

$$\underline{h}(n) = \sum_{l \notin \{s_0, s_1, \dots, s_k\}} \underline{h}_l,$$

$$\overline{h}(n) = \sum_{l \notin \{s_0, s_1, \ldots, s_k\}} \overline{h}_l.$$

The ranges $[\underline{h}(n), \overline{h}(n)]$ of our example are provided by Table 5 (b). Table 6 (b) shows the states developed using the five criteria $C_1, C_{23}, C_2, C_3, C_4$.

The complete trace of A-star algorithm applied to the data of Figure 7 (b) is given in Table 8, where the column number represents the order of development of the different states. The development of $(s_0 s_1 s_2 s_2 s_4)$ was considered in the 7th stage. But in this case, no successor state has been produced, whereas, the state 9 $(s_0 s_1 s_4 s_3 s_2)$ produces one that leads to the optimal circuit in the sense of C_{23} which is chosen the stop criterion. Note that the algorithm provides the Hamiltonian path and the cost of terminal state is [8,16], which is the result of the $\widetilde{\min}$ operation on the ten candidate states, precisely the son of state 9 (see Table 8). Note that if we had been content with the C_2 criterion to stop, the research would have developed only 9 summits.

Table 7. Evaluation of an optimal circuit of the right graph of Figure 7 according to C_{23} criterion.

Path	g	h	f
$s_0 s_1$	[2,3]	[5,10]	[7,13]
$s_0 s_1 s_2$	[3,6]	[4,7]	[7,13]
$s_0 s_1 s_2 s_4$	[5,10]	[3,5]	[8,15]
$s_0 s_1 s_2 s_4 s_3$	[6,12]	[2,3]	[8,15]
$s_0 s_1 s_2 s_4 s_3 s_0$	[8,16]	0	[8,16]

Table 8. Trace of the computation of f corresponding to the data of Figure 7 (b).

$n°$	Path	g	h	f
1	s_0			
2	$s_0 s_1$	[2,3]	[5,10]	[7,13]
3	$s_0 s_1 s_2$	[3,6]	[4,7]	[7,13]
4	$s_0 s_3$	[2,4]	[5,11]	[7,15]
5	$s_0 s_3 s_4$	[3,6]	[4,9]	[7,15]
	$s_0 s_3 s_4 s_2$	[5,10]	[3,6]	[8,16]
	$s_0 s_3 s_4 s_1$	[6,10]	[3,6]	[9,16]
6	$s_0 s_1 s_3$	[5,9]	[3,5]	[8,14]
7	$s_0 s_1 s_2 s_3 s_4$	[6,11]	[2,3]	[8,14]
8	$s_0 s_1 s_2 s_4$	[5,10]	[3,5]	[8,15]
9				

$n°$	Path	g	h	f
9	$s_0 s_1 s_2 s_4 s_3$	[6,12]	[2,3]	[8,15]
	$s_0 s_1 s_2 s_4 s_3 s_0$	[8,16]	0	[8,16]
10	$s_0 s_3 s_2$	[4,7]	[4,8]	[8,15]
11	$s_0 s_3 s_2 s_1$	[5,10]	[3,5]	[8,15]
	$s_0 s_3 s_2 s_1 s_4$	[8,14]	[2,3]	[10,17]
12	$s_0 s_1 s_4$	[5,7]	[4,8]	[9,13]
13	$s_0 s_1 s_4 s_3$	[6,9]	[3,6]	[9,15]
14	$s_0 s_2$	[4,5]	[5,10]	[9,15]
15	$s_0 s_2 s_1$	[5,8]	[4,7]	[9,15]
	$s_0 s_2 s_1 s_4$	[8,12]	[3,5]	[11,17]
16	$s_0 s_1 s_4 s_3 s_2$	[8,12]	[2,3]	[10,15]

In the case of the commercial traveler, the selection on the criterion C_2 (resp. C_3) clearly returns to lead the algorithm only on the coefficients $\underline{c}_{ij}$ (resp. $\overline{c}_{ij}$), that is, we get back to the case of accurate data. In particular, if the optimal solution obtained by each of these selection criteria correspond to a Hamiltonian path, then this Hamiltonian path is optimal in the sense of criterion C_{23}, and it is provided by A-star extended to imprecise data, provided that we adopt C_{23} as stop criterion. This is what happens in the example. Note that in this example, an optimal solution in the sense of the criterion C_1 does not exist; the children states are 16 and 9, respectively. $(s_0 s_1 s_2 s_2 s_4)$ and $(s_0 s_1 s_4 s_3 s_2)$ have incomparable terminal states in the sense of C_1.

5.2. Fuzzy Costs

According to Zadeh [21], a fuzzy interval is a fuzzy convex quantity, i.e., the membership function is `quasi-convex`:

$$\forall u, v, \qquad \forall w \in [u, v], \qquad \mu_Q(w) \geq \min\left(\mu_Q(u), \mu_Q(v)\right),$$

where μ_Q is a fuzzy amount, i.e., $\mu_Q: \Re \rightarrow [0,1]$. At the current moment of the development of the search graph, we have p candidate states $n_1, n_2, \dots, n_p$, each n_i $(i = 1, \dots, p)$ being associated with a fuzzy interval $\tilde{f}(n_i)$ which restricts the possible values of the function $f(n_i)$. A suggestion to select the state to develop is to choose n_i as

$$\tilde{f}(n_i) = \widetilde{\min}\, \tilde{f}\left(n_j\right), \quad \text{for} \quad j = 1, p.$$

It is clear that such a state does not exist and in this case any state can be selected. In addition, the above formula is relatively `strong`, since it amounts to applying the criterion C_{23} to all couples α-cuts $\tilde{f}\left(n_j\right),\ j = 1, p$.

Another approach is to measure how much an evaluation is smaller than another according to the criteria $C_1 - C_4$. The $\widetilde{\min}$ operation offers no way to carry out such a quantification. This will be obtained using the four comparison indices of the fuzzy intervals.

In the case of usual intervals, each of these criteria may be verified or not. For the fuzzy assessments, these criteria will be more or less verified (see 5.3 below):

- C_1 will be evaluated by Nec($\underline{f}(n_2) > \overline{f}(n_1)$)
- C_2 will be evaluated by Nec($\underline{f}(n_2) > \underline{f}(n_1)$)
- C_3 will be evaluated by Pos($\overline{f}(n_2) > \overline{f}(n_1)$)
- C_4 will be evaluated by Pos($\overline{f}(n_2) > \underline{f}(n_1)$)

The criteria C_1 and C_3 are stricter: $>$ instead of $\geq$ for consistency with the ratings is used. Note that if the membership functions are continuous, this change has no effect. In the case of p developmental states, we compute for each state n_j the four indices

- PSE($\tilde{f}(n_i)$), Possibility of over classing,
- PS($\tilde{f}(n_i)$), Possibility of strict over-classification,
- NSE($\tilde{f}(n_i)$), Need for over-classing,
- NS($\tilde{f}(n_i)$), Need for strict over-ranking,

expressing how much $\tilde{f}(n_i)$ is smaller than the other evaluations, in the sense of C_1, C_2, C_3 and C_4. If there are more than one, we search among these states for one that maximizes NSE $\tilde{f}(n_i)$, and so on, by checking

$$NS\left(\tilde{f}(n_i)\right) \quad \leq NSE\left(\tilde{f}(n_i)\right) \quad \leq PSE\left(\tilde{f}(n_i)\right)$$
$$NS\left(\tilde{f}(n_i)\right) \quad \leq PS\left(\tilde{f}(n_i)\right) \quad \leq PSE\left(\tilde{f}(n_i)\right).$$

The criterion for stopping the procedure can be chosen in two ways: Either (1) a terminal state is selected; and (2) in addition to (1), checking a condition on one of the indices: $E \in \{NS, NSE, PS, PSE\}$ in the form $E\ \left(\tilde{f}(n_i)\right) \geq \theta$, where θ is a fixed threshold.

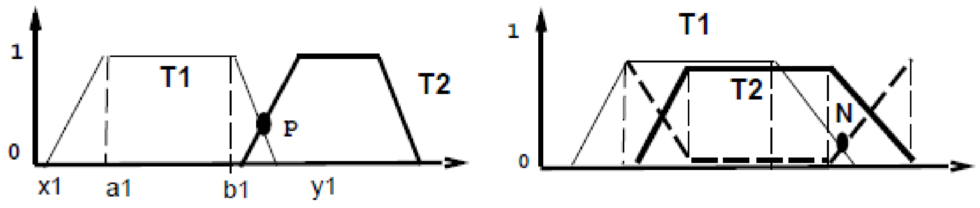


Figure 9. P and N are the possibility and necessity measures, respectively.

5.3. Possibility and Necessity Measures

The meaning of the verification of $C_1 - C_4$, evoked above in the case of fuzzy assessments, are implemented using the *possibility and necessity measures*.

Let Θ and $P(\Theta)$ be the non-empty set involving all possible events and the power set of Θ, respectively. $\forall A \subseteq P(\Theta)$, $\exists$ non-negative number, *possibility measure*, noted by $Pos(A)$ satisfying the followings:

- $Pos(\emptyset) = 0, Pos(\Theta) = 1,$
- $\forall A, B \in P\Theta \;\; A \subseteq B \rightarrow Pos(A) \leq Pos(B),$
- $\bigcup_k Pos(A_K) = \mathrm{Sup}_k Pos(A_k).$

The counterpart of the possibility measure of A is the necessity, $Nec(A) = 1 - Pos(A^c)$, which is defined on $(\Theta, P(\Theta), Pos)$ as follows:

- $Nec(\emptyset) = 0, Nec(\Theta) = 1$
- $Pos(A) \geq Nec(A)$
- $Pos(A) = 1 \rightarrow Nec(A) = 0$; and
- $Nec(A) > 0 \rightarrow Pos(A) = 1.$

Let $q_1 = [a_1, b_1, g_1, d_1]$ and $q_2 = [a_2, b_2, g_2, d_2]$ be two trapezoidal representations of two fuzzy assessments, where, $[a, b]$ is the support, g and d denote the left and right margins, respectively. Then, $Pos(q_1, q_2)$ is calculate as follows:

$$\pi(q_1, q_2) = \begin{cases} 0, & \text{if } (\alpha_1 + \alpha_2 \geq \max\{\alpha_2, \alpha_1\}) \\ 0, & \text{if } (\alpha_2 > \alpha_1) \wedge (g_2 = d_1 = 0) \\ 0, & \text{if } (\alpha_1 > \alpha_2) \wedge (g_1 = d_2 = 0) \\ born\left(\dfrac{a_2 - b_1}{d_1 + g_2}\right), & \text{if } \alpha_2 > \alpha_1 \\ born\left(\dfrac{a_1 - b_2}{d_2 + g_1}\right), & \text{if } \alpha_1 > \alpha_2 \end{cases}$$

where $born(x) = \max(0, x - 1)$, $\alpha_1 = b_2 - a_1$ and $\alpha_2 = b_1 - a_2$. Figure 9 illustrates the values of the possibility and necessity measures.

Note that we have used the *unique* data structure, namely the trapezoidal representation for dealing the cost of fuzzy terms (e.g., $[a, b, g, d]$), the classical intervals (e.g., $[a, b]$ by $[a, b, 0, 0]$), and the numbers (e.g., a by $[a, a, 0, 0]$).

6. Conclusions Remarks

We presented two major tasks for an intelligent mapping: dealing with homographs which abound in any operational tool, and how to maintain and update large data. The implementation made in the programming language C showed that the performance of our method is not only acceptable but it can be injected into an operational system. The method can be extended to deal with the constraints (e.g. eating the fish in a good restaurant while visiting a region) formulated by user. Taking into account various kinds of data structures used in the soft computing for the purpose of comparison is useful work to be done.

Acknowledgment

Many thanks to the Editor-in-chief of the Iranian Journal of Operations Research for his valuable helps that improves the readability of this paper.

References

[1] Blelloch, G.E. and Reid-Miller, M. (1998), Fast set operations using treaps. In Proceedings of 10th SPAA.

[2] Burnham, K.P. and Anderson, D.R. (2002), Model selection and multimdoal inference: A practical information-theoretic approach, New York: Springer.

[3] Dawid, A.P. (1984), A present position and potential developments: Some personal views. StatisticaltTheory, The prequential approach (with discussion), *J.R. Statistical Soc. A.*, 147, 178-292.

[4] Dijkstra, E. W. (1959), A note on two problems in connexion with graphs, *Numerische Mathematik*, 1: 269-271.

[5] Fatholahzadeh, A. (2005), Learning the morphological features of a large set of words, *Journal of Automata, Languages and Combinatorics*, 10(5/6), 5/6, 655-669.

[6] Fatholahzadeh, A. (2016), Building incremental homographs of big data, First International Workshop on Big Data Mathematical and Statistical Tools for Life Science, May 14-20, Amirkabir University of Teheran-IPM, Tehran,

[7] Fatholahzadeh, A. (2003), Implementation of dictionaries via automata and decision trees, In: Champarnaud, J.M. and Maurel D. (Eds), Lecture Notes in Computer Science 2608, Implementation and Application of Automata, Springer-Verlag, Berlin Heidelberg, 95-105.

[8] Fredman, M. L. and Tarjan, R. E. (1987), Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the Association for Computing Machinery*, 34(3): 596--615.

[9] Jenkins, B., “Jenkins hash coding”, http://burtleburtle.net/bob/c/lookup3.c, May 2006.

[10] Hart, P.E., Nilsson, N. J. and Raphael, B. (1968), A Formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*. 4 (2), 100-107.

[11] Kempe, A. (2000), Factorizations of ambiguous finite-state transducer, *International Conference on Implementation and Application of Automata*}, 157-164.

[12] Prade, H. and Dubois D. (1985), Théorie des possibilités, applications á la représentation des connaissances en informatique, Masson, 978-2-225-80579-0.

[13] Quinlan, R. (1993), C4.5: Programs for Machine Learning, Morgan Kaufmann.

[14] Maurel, D. and Daciuk J., (2006), Les transducres á sorties variables, In: Proceedings of

TALN, Leuve, Belgium, 237-245.

[15] Maurel, D. and Guenthner F. (2005), Automata and Dictionaries, Individual author and King's College, London.

[16] Mitchel, T.M. (1993), Machine Learning, Mc Graw-Hill.

[17] Nilsson, N. J. (1980), Principles of Artificial Intelligence, Palo Alto, California: Tioga Publishing Company.

[18] Russell, S. and Norvig, P. (2009) Artificial Intelligence: A Modern Approach (3rd ed.), Prentice Hall.

[19] Yu, S. (1997), Regular languages. In: Rozenberg, G. and Salomma, A. (Eds), Handbook of formal languages, Vol. 1, Word, Language, Grammar, Springer Science & Business Media.

[20] Xu, M. and Golay, M.W. (2006), Data-guided combination by decomposition and aggregation, *J. Machine Learning*, 63(1), 43-67.

[21] Zadeh, L.A. (1978), PRUF-a meaning representation language for natural, *Int. J. Man-Machine Studies*, 10, 395-460.