Cluster-Specific Heuristics for Constraint Solving

Seda Polat Erdeniz, Alexander Felfernig, Muesluem Atas,
Thi Ngoc Trang Tran, Michael Jeran, and Martin Stettinger

Institute of Software Technology, Graz University of Technology,
Inffeldgasse 16b/II, 8010 Graz, Austria
{spolater,alexander.felfernig,muesluem.atas,ttrang,mjeran,martin.stettinger}@ist.tugraz.at
http://ase.ist.tugraz.at/

Abstract. In Constraint Satisfaction Problems (CSP), variable ordering heuristics help to increase efficiency. Applying an appropriate heuristic can increase the performance of CSP solvers. On the other hand, if we apply specific heuristics for similar CSPs, CSP solver performance could be further improved. Similar CSPs can be grouped into same clusters. For each cluster, appropriate heuristics can be found by applying a local search. Thus, when a new CSP is created, the corresponding cluster can be found and the pre-calculated heuristics for the cluster can be applied. In this paper, we propose a new method for constraint solving which is called *Cluster Specific Heuristic (CSH)*. We present and evaluate our method on the basis of example CSPs.

Keywords: Configuration; Constraint Satisfaction Problems; Variable and Value Ordering Heuristics; Clustering; Performance Optimization

1 Introduction

Configuration systems [3, 7] are used to find solutions for problems which have many variables and constraints. An example of a configuration problem can be *the customization of cars* where many hardware and software modules exist and all of them should work together without any conflicts. Configuration problems can be formulated as a constraint satisfaction problem (*CSP*) [19] which is defined as a triple *(V,D,C)* where V is a set of variables, D is set of domains for each variable, and C is a set of constraints. The constraint set may also include the user requirements (R) if available.

In this paper, we propose a new method for constraint solving which is called *Cluster Specific Heuristics (CSH)*.[1] Figure 1 shows the main contribution of CSH in constraint solving. CSH creates clusters of similar instances of a CSP which

[1] The work presented in this paper has been conducted within the scope of the European Union Horizon 2020 research project AGILE (Adoptive Gateways for dIverse MuLtiple Environments – www.agile-project-iot.eu.).

can be exploited to learn and generate specific search heuristics for each cluster. Whenever a new instance of the original configuration problem occurs, CSH finds the most relevant cluster for this CSP and it applies the pre-calculated cluster-specific heuristic to solve this CSP. We show that, compared to other heuristics, CSH increases the performance of CSP solving in many scenarios.

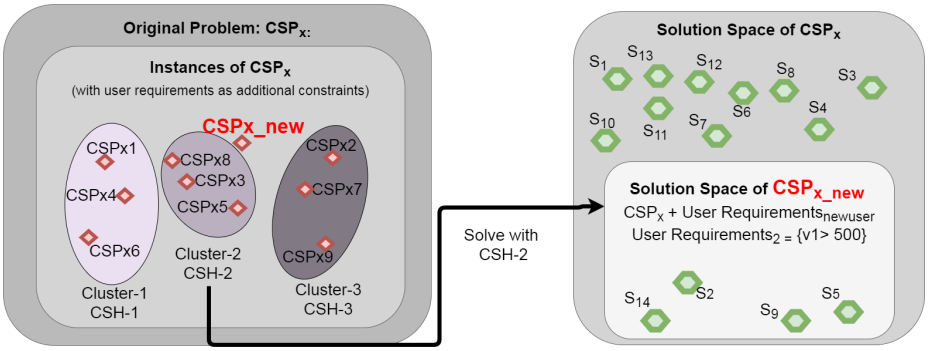

Fig. 1. Contribution of CSH in CSP solving. K-Means clustering is applied on the cluster elements $CSPx1$,$CSPx2$,..,$CSPx9$. 3 clusters are generated. Specific variable ordering heuristics are learned for each cluster as CSH-1, CSH-2 and CSH-3 where CSH-i stands for the learned variable ordering heuristic for the Cluster-i. Whenever we get a new instance of the original problem $CSPx$ as $CSPx\_new$, we can solve $CSPx\_new$ by using the calculated variable ordering heuristics. In this example, we use CSH-2, since the most similar CSPs to $CSPx\_new$ are in Cluster-2. The CSP solver finds a solution faster by using CSH compared to other global heuristics.

The remainder of this paper is organized as follows. We first provide an overview of the used algorithms in Section 2. We then state the problem definition and our solution in Sections 3 and 4. Finally, we present our experimental results and comparisons with state-of-the-art heuristics in Sections 5 and 6. Finally, in Section 7, we discuss issues for future work and conclude the paper.

2 Background

CSP Solvers solve a CSP defined as *(V,D,C)* by generating search trees among different instances of variables and search for satisfactory solutions based on the constraints of the problem. There exist different open source libraries for solving CSPs, for example, Choco Solver [17]. To improve the performance of such solvers, search is guided by so-called variable and value ordering heuristics. Variable and value ordering heuristics are generally used in problems such as *configuration*, *job shop scheduling*, and *integrated circuit design* [18].

Our CSH approach uses *K-Means clustering* [10] to cluster different instances of the same CSP. K-Means clustering is applied to the user requirements included

in the CSPs. For example, if two CSPs include similar values for their variables, then these two CSPs can be grouped in the same cluster. K-Means clustering is popular for cluster analysis in data mining. It creates k clusters by minimizing the sum of squares of distances between cluster elements.

Formula 1 shows the minimization function of k-Means clustering where k is the number of target clusters, S is a cluster set, μ_{i} is the average value of cluster elements in the S_{i} and x is a cluster element in S_{i}.

$$min \sum_{i=1}^{k} \sum_{x \in S_{\mathrm{i}}} \|x - \mu_{\mathrm{i}}\|^2 \tag{1}$$

If there is more than one variable in the cluster elements that will be clustered, then the difference between x and μ_{i} can be calculated based on the *Euclidean n-distance* as shown in Formula 2 where x_{j} is the j^{th} variable in the cluster element x and μ_{ij} is the average value of j^{th} variables of the cluster elements in the i^{th} cluster.

$$x - \mu_{\mathrm{i}} = \sqrt{\sum_{j=1}^{n} (x_{\mathrm{j}} - \mu_{\mathrm{ij}})^2} \tag{2}$$

As an example, we can consider clustering 6 elements with 2 variables into 2 clusters (k=2) after 4 iterations (i=4). Before clustering starts, two clusters can be created randomly. In every iteration, we calculate a cluster element's total distance to both clusters. In the first iteration, we randomly take a cluster element, $x = (x_1{=}100, x_2{=}200)$ and compare it with the clusters S_1 with a mean value $\mu_1 = (\mu_{11}{=}300, \mu_{12}{=}400)$ and S_2 with a mean value $\mu_2 = (\mu_{21}{=}0, \mu_{22}{=}400)$. The distance between x and the clusters is calculated based on Formula 2 which gives 282.84 for S_1 and 223.60 S_2. Based on Formula 1, putting x into the cluster S_2 minimizes the total distance between cluster elements. In follow-up iterations, we take another cluster element and apply the same distance calculations and put it into a more similar cluster to minimize total distances. After 4 iterations, we obtain 2 clusters with a low distance between the elements in the same cluster. If we increase the number of iterations, it is often possible to determine even lower total distances.

CSH applies a *genetic algorithm* [14] to generate cluster-specific variable ordering heuristics.

3 Cluster-Specific Heuristics

In this work, we aim to find an answer to the question: "If we cluster instances of the a CSP and apply a cluster-specific variable ordering heuristic, could we increase the overall performance of a CSP solver for solving a new instance of the same CSP?".

We implemented Algorithm 1 and Algorithm 2, using the Java libraries Choco Solver [17] (for CSP solving) and Java Machine Learning [10] (for K-Means

clustering). Besides, we implemented the "Simple Genetic Algorithm"[2]. The genetic algorithm applies mutations and crossover operations on variable order arrays (individuals). *Fitness value* is the execution time of the CSP solver with this order. Consequently, a smaller fitness value indicates a better variable order.

Algorithm 1: Learn Cluster-Specific Heuristics

Input: $OriginalCSP$, $sampleInstancesOfOriginalCSP$
Output: *clusters*, *heuristics*
1 Cluster $sampleInstancesOfOriginalCSP$ based on K-Means and Euclidean n-distance algorithms
2 Learn *heuristics* based on Simple Genetic Algorithm
3 Return *clusters* and *heuristics*

Algorithm 1 takes $OriginalCSP$ and $sampleInstancesOfOriginalCSP$ as input and returns *clusters* and the corresponding specific variable ordering *heuristics* as output. Algorithm 2 takes the output of Algorithm 1 and the new instance of the original CSP CSP_{new} as input.

Algorithm 2: Solve the CSP with learned CSH

Input: CSP_{new}, *clusters*, *heuristics*
Output: *solution*
1 Find *mostSimilarCluster* for CSP_{new}
2 Solve the CSP_{new} with related learned *heuristic*
3 Return *solution*

To explain our approach, we introduce a CSP with 5 variables as shown in Table 1. In the subsections below, we provide more details of the main steps of CSH which are "Clustering", "Learning Heuristics", and "Solving a new CSP".

3.1 Clustering

If we do not have instances of an original CSP in terms of historical data (e.g. log files), we need to generate some sample instances to create clusters and learn specific heuristics. For our example we generated 10 different sample instances of the original CSP as depicted in Table 2. We combine CSP_{cars} with different user constraints to generate new instances of the same problem which are denoted as $\text{CSP}_{\text{carsInstance1}}$ - $\text{CSP}_{\text{carsInstance10}}$.

The CSP solver tries to find satisfactory values for the non-initialized variables (all variable domains are represented as integers which have positive values). Since we are trying to find a variable ordering heuristic for similar CSPs,

[2] www.theprojectspot.com.

CSP_{cars}	Definitions
Variables	v_1, v_2, v_3, v_4, v_5
Domains	$DomainRange_{forAllVariables}$=[0..100000]
Constraints	C_1: ($v_1 \leq 200$) $\Longrightarrow$ ($v_5 \geq 600$) C_2: ($v_4 = v_5$) $\Longrightarrow$ ($v_2 \geq 1000$) C_3: ($v_3 \geq 20000$) $\Longrightarrow$ ($v_4 = 80$)

Table 1. A simple CSP definition (CSP_{cars} as an abstract representation of a car customization problem).

Instances of the Problem CSP_{cars}	Different User Requirements (R)
$CSP_{carsInstance1} = CSP_{cars} \wedge R_1$	R_1: ($v_2 = 2000 \wedge v_5 = 500$)
$CSP_{carsInstance2} = CSP_{cars} \wedge R_2$	R_2: ($v_2 = 7000 \wedge v_5 = 700$)
$CSP_{carsInstance3} = CSP_{cars} \wedge R_3$	R_3: ($v_2 = 1000 \wedge v_5 = 300$)
$CSP_{carsInstance4} = CSP_{cars} \wedge R_4$	R_4: ($v_4 = 80$)
$CSP_{carsInstance5} = CSP_{cars} \wedge R_5$	R_5: ($v_4 = 70$)
$CSP_{carsInstance6} = CSP_{cars} \wedge R_6$	R_6: ($v_4 = 50$)
$CSP_{carsInstance7} = CSP_{cars} \wedge R_7$	R_7: ($v_1 = 100 \wedge v_2 = 500$)
$CSP_{carsInstance8} = CSP_{cars} \wedge R_8$	R_8: ($v_1 = 800 \wedge v_2 = 250$)
$CSP_{carsInstance9} = CSP_{cars} \wedge R_9$	R_9: ($v_3 = 10$)
$CSP_{carsInstance10} = CSP_{cars} \wedge R_{10}$	R_{10}: ($v_3 = 80000$)

Table 2. Sample instances of the car customization problem CSP_{cars}

it is reasonable to group the same non-initialized variables into the same cluster. Therefore, before clustering, non-initialized variables in the CSP problems are assigned to "-1000", since all initialized variables should have positive values due to the domain definitions ($DomainRange_{forAllVariables}$=[0..100000]) where the minimum value is 0. This assignment decreases the similarity between the CSPs which have different initialized variables. The similarity between two CSPs (using only the user requirements inside the CSPs) is calculated based on the Formula 2. JavaML's K-Means calculates the distance between x and μ based on *Euclidean n-space distance*. We used JavaML K-Means with its default constructor which applies 100 iterations (i=100) during the K-Means clustering and creates 4 clusters (k=4) in our case.

K-Means clustering is applied to the instances of the original CSP given in Table 2. After running JavaML K-Means clustering with its default constructor as mentioned in [1], we obtained 4 clusters as can be seen in Table 3. For example, $CL_{carsInstance1}$=($CSP_{carsInstance1}$, $CSP_{carsInstance2}$, $CSP_{carsInstance3}$), where

all CSPs have the same initialized variables which are v_2 and v_5. This means, for all these CSPs, the CSP solver will find solutions by assigning satisfactory values for the same non-initialized variables v_1, v_3 and v_4. For the CSPs in CL_1, the CSP solver will search in the same solution set. Therefore, it is applicable to use similar heuristics for these CSPs.

3.2 Learning Heuristics

After finding clusters, using the *Simple Genetic Algorithm* for each cluster, variable ordering heuristics are calculated as shown in Table 3.

Clusters	CSPs	Variable Ordering
CL_1	$CSP_{carsInstance1}$, $CSP_{carsInstance2}$, $CSP_{carsInstance3}$	v_1, v_2, v_5, v_4, v_3
CL_2	$CSP_{carsInstance4}$, $CSP_{carsInstance5}$, $CSP_{carsInstance6}$	v_4, v_2, v_5, v_3, v_1
CL_3	$CSP_{carsInstance7}$, $CSP_{carsInstance8}$	v_1, v_5, v_2, v_3, v_4
CL_4	$CSP_{carsInstance9}$, $CSP_{carsInstance10}$	v_1, v_5, v_4, v_2, v_3

Table 3. Clusters and corresponding heuristics. Based on the similar instances of the same problem in the clusters, CSH finds the variable ordering that will be used for solving the new instances of the same problem.

In line 2 of Algorithm 1, the *Simple Genetic Algorithm* is used to learn the heuristics based on the clustered sample CSPs. We applied some adaptations during the implementation of this algorithm. In our implementation, *individuals* are variable orders (an array of variable names such as: $[v_4,v_5,v_1,v_3,v_2]$) rather than the binary arrays as in the original implementation.

When an individual is created by the algorithm, the corresponding *fitness* values are also calculated. The fitness of an individual is described by the execution time of the CSP solver using the corresponding variable ordering heuristic. After all, the genetic algorithm selects the best individual according to the execution time. Shorter execution time implies a better individual. In our genetic algorithm, we set the target of fitness to 10000 milliseconds. This means the population evolution loop can be executed until the target fitness value is reached or 10 seconds are exceeded.

3.3 Solving a new CSP

A new user activates the configuration system and defines his/her own requirements as R_{new}: $(v_1 = 500 \wedge v_5 = 100)$. R_{new} is combined with the original CSP which is CSP_{cars} and a new instance is created as CSP_{new}. First, the most similar cluster to CSP_{new} is retrieved by CSH. The most similar cluster is CL_1 because the CSPs in the CL_1 and CSP_{new} have the same variables in the user

requirements(v_2 and v_5). The CSP Solver applies the variable ordering heuristic of CL_1 (v_1, v_2, v_5, v_4, v_3) (this heuristic is given as an input to the constructor of the Choco Solver) to find a solution as seen in Table 4.

Solution of CSP_{new}				
v_1	v_2	v_3	v_4	v_5
500	7000	30000	80	100

Table 4. The CSP solver finds a solution for this CSP where the solution is consistent with all constraints and the domain values defined in the CSP.

4 Experimental Results

We tested and measured the performance of CSH and compared it with the heuristics available in the Choco Solver.[3] Choco Solver [11] comes with built-in variable and value ordering heuristics.[4] To solve a constraint satisfaction problem, a built-in heuristic can be chosen for a specific search problem. In this context, we generated our specific variable ordering that can be given to the Choco solver as an input during the solver initialization.

We tested 13 built-in variable ordering heuristics of Choco Solver and compared these with CSH. We ran our tests over 4 CSPs as shown in Table 5. Each CSP has a different number of variables and domain ranges (domains are the same for each variable in a CSP). For example, in CSP_1 there are 100 variables with domain D:[0..10000]. Sample instances of the original CSP are generated by selecting the variables randomly and assigning random values with respect to their domain definitions.

To be able to find the clusters and heuristics, we created 100 instances (to create clusters) of these 4 CSPs. During the instance creation, we selected some variables of the CSP instance randomly and each selected variable is assigned with a value randomly in the defined domain. These 4 CSPs and their generated instances are similar to the working example in Table 1 and Table 2.

Table 5 shows the performance comparison between Choco's built-in heuristics and CSH. We made the comparison with respect to runtime performance (average of the runtime of 40 new instances for each CSP). It can be observed that CSH is the fastest heuristic for solving a new instance of a known CSP.

[3] Our experiments have been conducted on an Intel Core i5-5200U PC, 2.20 GHz processor, 8 GB RAM, and 64 bit Windows 7 Operating System and Java Run-time Environment 1.8.0.

[4] choco-solver.org.

TEST RESULTS				
	4 Different Original CSPs			
Heuristics	CSP_1 100 variables D:[0..100]	CSP_2 100 variables D:[0..1000]	CSP_3 1000 variables D:[0..100]	CSP_4 1000 variables D:[0..1000]
Largest	276.302	311.875	2316.203	2192.573
Smallest	164.918	212.154	9573.073	9481.634
ActivityBased	66.480	71.612	265,688	226.500
FirstFail	48.869	32.657	125.613	109.634
AntiFirstFail	40.821	95.055	115,699	125.146
Cyclic	36509	28.691	88.640	79.543
Random	45.486	32.307	80.010	77.560
MaxRegret	58.082	53.417	172.266	171.682
Occurance	41.871	39.422	99.137	100.070
Input Order	32.307	32.657	76.277	86.074
DomOverDweg	25.076	44.203	63.448	75.136
ImpactBased	63.448	73.362	247.027	283.766
GeneralizedMinDomain	39.188	32.657	75.927	76.011
CSH	**24.959**	**24.026**	**23.443**	**74.995**

Table 5. Runtime performance (given in seconds) of the Choco solver with Choco's built-in heuristics compared to CSH. CSH is the fastest heuristic for solving a new instance of a known CSP.

5 Related Work

Jannach [9] proposes a learning solution for domain specific heuristics. In this work, it is mentioned that solving complex configuration problems often requires the usage of domain-specific search heuristics which have to be explicitly modeled by domain experts and knowledge engineers. This work significantly differs from ours since no cluster-specific heuristics are taken into account.

Li et al. [12] apply a clustering approach to divide a search problem into sub-problems. The authors apply a variable and value ordering heuristic over these clusters. In our approach, we cluster similar problems to be able to determine cluster-specific search heuristics. We do not divide a problem into sub-problems.

O'Sullivan et al.[16] use a constraint solver and decision tree learning to solve a CSP query of a user. The overall goal of this work is also to improve the overall efficiency of search.

Balduccini et al. [2] implemented domain specific heuristics for Answer Set Programming Solvers. Heuristics are learned by the proposed platform. This work is similar to ours but does not take into account clustering mechanisms. It

learns heuristics for domains as *Domain-Specific Heuristics* but does not cluster the underlying problems.

Epstein et al. [5] postulate several types of crucial sub-problems and show how local search can be harnessed to solve them before global search is triggered. A variety of heuristics and metrics are then used to guide (global) solution search.

Liu et al. [13] generate a variable ordering strategy for solving Disjunctive Temporal Problems (DTPs) which are an essential aspect for building systems that reason about time and actions. DTP model events and their relationships (as distances between events) and provide the means to specify the temporal elements of an episode with a temporal extent.

Ciccio et al. [4] introduce techniques which guarantee the consistency of the discovered models and keep the most interesting constraints in the pruned set. The level of interestingness is dictated by user-specified prioritization criteria. Merhej et al. [15] introduce an approach to assign weights to rules of thumb by sampling in a particular way from a pool of possible repairs.

6 Future Work and Conclusion

In this paper, we proposed *CSH* (Cluster Specific Heuristics) which is an intelligent method that can be used to support the inclusion of search heuristics in CSP solving. With this method, the performance of the CSP solver can be improved compared to other built-in heuristics. *CSH* can be useful for CSP scenarios such as *product configuration* where different CSP instances reoccur. As *future work* we plan to apply specific clustering methods for different CSP types rather than using K-Means clustering for every CSP type. With this, we expect to increase the similarities within clusters and increase the efficiency of CSP search. We will also work on generating cluster-specific value ordering heuristics and cluster-specific heuristics for diagnosis tasks [8]. We also plan to use optimization functions to decide on the number of clusters. Thus, the number of clusters can vary depending on the underlying CSPs. We will also apply *CSH* in our ongoing European Union Horizon 2020 project AGILE which focuses a.o. on the development of efficient recommendation [6] and configuration technologies [20, 7] for different types of Internet of Things scenarios.

References

1. Abeel, T., de Peer, Y., Saeys, Y.: Java-ML: A Machine Learning Library. Journal of Machine Learning Research 10, 931–934 (2009)
2. Balduccini, M.: Learning and Using Domain-specific Heuristics in ASP Solvers. AI Commun. 24(2), 147–164 (2011)
3. Benavides, D., Felfernig, A., Galindo, J., Reinfrank, F.: Automated Analysis in Feature Modeling and Product Configuration. In: 13th International Conference on Software Reuse. vol. LNCS 7925, pp. 160–175. Pisa, Italy (2013)
4. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Resolving Inconsistencies and Redundancies in Declarative Process Models. Inf. Sys. 64, 425–446 (2017)

5. Epstein, S.L., Wallace, R.J.: Finding Crucial Subproblems to Focus Global Search. In: 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06). pp. 151–162 (2006)
6. Falkner, A., Felfernig, A., Haag, A.: Recommendation Technologies for Configurable Products. AI Magazine 32(3), 99–108 (2011)
7. Felfernig, A., Hotz, L., Bagley, C., Tiihonen, J.: Knowledge-based Configuration: From Research to Business Cases. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edn. (2014)
8. Felfernig, A., Schubert, M., Zehentner, C.: An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets. Artificial Intelligence for Engineering Design, Analysis, and Manufacturing (AIEDAM) 26(1), 53–62 (2012)
9. Jannach, D.: Toward Automatically Learned Search Heuristics for CSP-encoded Configuration Problems - Results from an Initial Experimental Analysis. In: Proceedings of the 15th International Configuration Workshop, Vienna, Austria, August 29-30, 2013. pp. 9–13 (2013)
10. Jin, X., Han, J.: K-means clustering. In: Sammut, C., Webb, G.I. (eds.) Encyclopedia of Machine Learning, pp. 563–564. Springer, Boston, MA, USA (2010)
11. Jussien, N., Rochart, G., Lorca, X.: Choco: an Open Source Java Constraint Programming Library. In: CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08). pp. 1–10. Paris, France, France (2008)
12. Li, X., Epstein, S.L.: Learning Cluster-based Structure to Solve Constraint Satisfaction Problems. Annals of Mathematics and AI 60(1–2), 91–117 (2010)
13. Liu, Y., Jiang, Y., Qian, H.: Topology-based Variable Ordering Strategy for Solving Disjunctive Temporal Problems. In: 15th International Symposium on Temporal Representation and Reasoning. pp. 129–136. IEEE (2008)
14. Man, K.F., Tang, K.S., Kwong, S.: Genetic algorithms: concepts and applications. IEEE Transactions on Industrial Electronics 43(5), 519–534 (1996)
15. Merhej, E., Schockaert, S., De Cock, M.: Repairing inconsistent answer set programs using rules of thumb: A gene regulatory networks case study. International Journal of Approximate Reasoning 83, 243–264 (2017)
16. O'Sullivan, B., Ferguson, A., Freuder, E.C.: Boosting constraint satisfaction using decision trees. In: 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004). pp. 646–651 (2004)
17. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Solver Documentation (2017)
18. Sadeh, N., Fox, M.S.: Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. AI Journal 86(1), 1–41 (1996)
19. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press (1993)
20. Walter, R., Felfernig, A., Küchlin, W.: Constraint-Based and SAT-Based Diagnosis of Automotive Configuration Problems. Journal of Intelligent Information Systems (JIIS) pp. 1–32 (2016)