Intermediate Consistencies by Delaying Expensive Propagators

Andreï Legtchenko, Arnaud Lallouet, AbdelAli Ed-Dbali
Université d’Orléans – LIFO
BP 6759 – F-45067 Orléans – France

Abstract
What makes a good consistency? Depending on the constraint, it may be a good pruning power or a low computational cost. By “weakening” arc-consistency, we propose to define new automatically generated solvers which form a sequence of consistencies weaker than arc-consistency. The method presented in this paper exploits a form of regularity in the cloud of constraint solutions: the density of solution orthogonal to a projection. This approach is illustrated on the sparse constraints “to be a n-letters english word” and crossword CSPs, where interesting speed-ups are obtained.

Introduction
Since their introduction (Mackworth 1977), CSP consistencies have been recognized as one of the most powerful tools to strengthen search mechanisms. Since then, their considerable pruning power has motivated a lot of efforts to find new consistencies and to improve the algorithms to compute them.

Consistencies can be partially ordered according to their pruning power. However, this pruning power should be put into balance with the complexity of enforcing them. For example, path-consistency is often not worth it: its pruning power is great, but the price to pay is high. Maintaining path-consistency during search is thus often beaten in practice by weaker consistencies. Similarly, on many useful CSPs, bound-consistency is faster than arc-consistency even if it does not remove values inside the variables’ domains: this is left to the search mechanism ensuring the completeness of constraint solving. Moreover, the overall performance of a solver is also determined by the interaction between the consistency and the search strategy. Powerful consistencies are not always the best choice. It is sometimes more interesting to find the optimal ratio between the advantage of pruning and the computational cost needed to enforce it. The absence of prediction of the potential power of a consistency comes from two sources: consistencies exploit subtle properties which are often incomparable and they do not always use the same data-structures.

Recently, there has been a great interest in the automatic building of consistency operators and in finding ad-hoc efficient constraint representations. First (Apt & Monfroy 2001) and (Abdennadher & Rigotti 2003) use CHRs (Fruhwirth 1998) as a target language to build a solver. Powerful propagators may be discovered in this framework but at the price of a high complexity which limits the method to constraints of modest arities and rather small domains. In (Dao et al. 2002) has been introduced an approximation framework we also use in this paper to approximate bound-consistency. This work has been extended in (Lallouet et al. 2003) for arc-consistency: a constraint is approximated by a set of covering blocks obtained by a clustering algorithm and only these blocks are used in the reduction process. But unfortunately, not all constraints are suitable for clustering. It is needed that the constraint has locally dense areas which can be agglomerated meaningfully. Finding new representations to compute a consistency has also been tackled in (Barták & Mecl 2003) and (Cheng, Lee, & Stuckey 2003): they use clustering techniques to find an exact representation of the constraint. Hence the resulting consistency is the full arc-consistency and speed-ups are difficult to obtain while competing with highly optimized implementations. In (Monfroy 1999), the idea of using weaker functions has been proposed as a preprocessing to speed-up the computation of a classical consistency (in this case, box-consistency). From the search mechanism point of view, this is an internal recipe to compute the consistency. In contrast, our operators define on purpose a weaker consistency which is used during search.

In this paper, we contribute to this line of work by providing a range of custom consistencies intermediate between bound- and arc-consistency for sparse constraints which have no dense areas. First, these constraints exist: we take as example the \( n \)-ary constraint “to be a \( n \)-letter english word”. There is absolutely no block regularity and the words are sparsely distributed in the solution space. For example, by using the 4176 five-letters words of /usr/dict/words, we get a density of 0.035%. A clustering algorithm would either consider a large number of clusters, limiting the interest of the representation, or filter a too few number of non-solution out of the search space, yielding a very weak consistency.

Our method is as follows. For a given constraint, we express the arc-consistency as a set of particular elementary reduction functions. Each elementary function is only con-
cerned in the withdrawal of one value in a variable’s domain. Depending on the number of supports for the target value, these functions do not have the same computational cost. Our technique consists in applying the computationally cheap operators as a consistency during the reduction phase and delaying the expensive ones until instantiation of all variables. For this we split the set of functions in two: functions smaller than a given threshold are iterated and longer ones are delayed. The closure of all the operators defines a new consistency for a given constraint, weaker and sometimes quicker than arc-consistency.

Intuitively, the cost of an elementary reduction is directly related to the number of supports present in the hyperplane orthogonal to the target value. For example, in figure 1 representing cuts in the solution space of a constraint, the value \( a \) for the variable \( X \) is supported by a large number of tuples: in order to eliminate \( a \) from \( X \)’s domain, we have to check all these tuples. On the other side, the value \( b \) in figure 2 is only supported by a few tuples. But since \( a \) is well supported, it is more likely part of a solution than \( b \). Hence we spend a large work trying to eliminate a value which is likely part of a solution. We propose to postpone the evaluation of \( a \)’s function until instantiation of all variables (i.e. on satisfiability test) while actively trying to eliminate \( b \) by consistency. It could be argued that algorithms like AC6 which compute lazily the set of supports could be faster, but since the resulting consistency is weaker than AC, the technique could apply to any AC algorithm. The main difficulty rely in determining the best threshold.

The paper is structured as follows:

- A framework to express consistencies. Consistencies are usually built as global CSP properties. But it is now rather common to express them modularly by the greatest fix-point of a set of operators associated to the constraints, which is computed using a chaotic iteration (Apt 1999). We present a framework which allows, starting from arbitrary operators, to progressively add properties in order to build a consistency.

- A consistency construction method. For a given constraint, we express the arc-consistency as a set of particular reduction operators. These operators do not have the same computational cost. Expensive ones are delayed until instantiation of all variables. The closure of all the operators defines a new consistency for a given constraint, weaker but quicker than arc-consistency.

- An example. The interest of these consistencies is shown by the ”word” constraints and the crossword CPSs.

### An approximation framework

In this section, we present a general approximation scheme for consistencies. Let \( V \) be a set of variables and \( D = (D_X)_{X \in V} \) their (finite) domains. For \( W \subseteq V \), we denote by \( D_W \) the set of tuples on \( W \), namely \( \Pi_{X \in W} D_X \). Projection of a tuple or a set of tuples on a variable or a set of variables is denoted by |, natural join of two sets of tuples is denoted by \( \times \). If \( A \) is a set, then \( \mathcal{P}(A) \) denotes its powerset and \( |A| \) its cardinal.

**Definition 1 (Constraint)** A constraint \( c \) is a pair \((W,T)\) where:

- \( W \subseteq V \) is the arity of the constraint \( c \) and is denoted by \( \text{var}(c) \).
- \( T \subseteq D_W \) is the set of solutions of \( c \) and is denoted by \( \text{sol}(c) \).

The join of two constraints is defined as a natural extension of the join of tuples: \( c \times c' = (\text{var}(c) \cup \text{var}(c'), \text{sol}(c) \times \text{sol}(c')) \).

A CSP is a set of constraints. Join is naturally extended to CSPs and the solutions of a CSP \( C \) are \( \text{sol}(\times C) \). A direct computation of this join is too expensive to be tractable, especially when considering that it needs to represent tuples of the CSP’s arity. This is why a framework based on approximations is preferred, the most successful of them being the domain reduction scheme where variable domains are the only reduced constraints. So, for \( W \subseteq V \), a search state consists in a set of yet possible values for each variable: \( s_W = (s_X)_{X \in W} \) such that \( s_X \subseteq D_X \). The search space is \( S_W = \Pi_{X \in W} \mathcal{P}(D_X) \). The set \( S_W \), ordered by pointwise inclusion \( \subseteq \), is a complete lattice. Likewise, union and intersection on search states are defined pointwise. The whole search space \( S_V \) is simply denoted by \( S \).

Some search states we call singletonic play a special role in our framework. A singletonic search state comprises a single value for each variable, and hence represents a single

![Figure 1: Dense hyperplane in solution space](image)

![Figure 2: Sparse hyperplane in solution space](image)
A tuple is promoted to a singleton search state by the operator $\text{Sing}_c$. This notation is extended to a set of tuples: for $E \subseteq D^W$, let $[E] = \{(t \in E) | t \in E\} \subseteq S_W$. Conversely, a state is converted into the set of tuples it represents by taking its cartesian product $\Pi$: for $s \in S_W$, $\Pi s = \Pi X \subseteq D^W$. We denote by $\text{Sing}_F$ the set $[D^W]$ of singleton search states. By definition, $[D^W] \subseteq S_W$.

A consistency is generally described by a property $\text{Cons} \subseteq S$ which holds for certain search states and is classically modeled by the common greatest xpoint of a set of operators associated to the constraints. By extension, in this paper, we call consistency for a candidate $c$ an operator on $S_W$ having some properties which are introduced in the rest of this section. Let $f$ be an operator on $S_W$. We denote by $\text{Fix}(f)$ the set of fixpoints of $f$ which define the set of consistent states according to $f$.

For $W \subseteq W' \subseteq V$, an operator $f$ on $S_W$ can be extended to $f'$ on $S_{W'}$ by taking: $\forall s \in S_W, f'(s) = s'$ with $\forall X \subseteq W', s'_X = s_X$ and $\forall X \subseteq W$, $s'_X = f(s|W)_X$. Then $s \in \text{Fix}(f') \iff s|W \in \text{Fix}(f)$. This extension is useful for the operator to be combined with others at the level of a CSP.

In order for an operator to be related to a constraint, we need to ensure that it is contracting and that no solution tuple could be rejected anywhere in the search space. An operator having such property is called a preconsistency:

**Definition 2 (Preconsistency)** An operator $f : S_W \rightarrow S_W$ is a preconsistency for $c = (W, T)$ if:

- $f$ is monotonic, i.e. $\forall s, s' \in S_W, s \subseteq s' \Rightarrow f(s) \subseteq f(s')$.
- $f$ is contracting, i.e. $\forall \forall s \in S_W, f(s) \subseteq s$.
- $f$ is correct, i.e. $\forall s \in S_W, f(s) \subseteq \text{sol}(c)$.

In the last property, the second inclusion is also called correctness of the operator with respect to the constraint; it means that if a state contains a solution tuple, this one will not be eliminated by consistency. Since a preconsistency is also contracting, this inclusion is actually also an equality.

An operator on $S_W$ is associated to a constraint $c = (W, T)$ if its singleton fixpoints represent the constraint’s solution tuples $T$:

**Definition 3 (Associated Operator)** An operator $f : S_W \rightarrow S_W$ is associated to a constraint $c$ if $\text{Fix}(f) \cap \text{Sing}_W = [\text{sol}(c)]$

However, nothing is said about its behavior on non-singleton states. This property is also called singleton completeness. Note that a preconsistency is not automatically associated to its constraint since the set of its singleton fixpoints may be larger. When it coincides, we call such an operator a consistency:

**Definition 4 (Consistency)** An operator $f$ is a consistency for $c$ if it is associated to $c$ and it is a preconsistency for $c$.

Note that a consistency can be viewed as an extension to $S_W$ of the satisfiability test made on singletonic states. Consistency operators can be easily scheduled by a chaotic iteration algorithm (Apt 1999). By the singleton completeness property, the consistency check for a candidate tuple can be done by the propagation mechanism itself. Let $C = \{c_1, \ldots, c_n\}$ be a CSP and $F = \{f_1, \ldots, f_n\}$ be a set of consistencies on $S$ associated respectively to $\{c_1, \ldots, c_n\}$. If all constraints are not defined on the same set of variables, it is always possible to use the extension of the operators on the union of all variables which appear in the constraints. The common closure of the operators of $F$ can be computed by a chaotic iteration (Apt 1999). It follows from the main confluence theorem of chaotic iterations that a consistency can be constituted by combining the mutual strengths of several operators. Since we have $\text{Fix}(F) = \cap_{f \in F} \text{Fix}(f)$ and since each consistency does preserve the tuples of its associated constraint, the computed closure of all operators associated to the CSP $C$ does not reject a tuple of $c_1 \times \ldots \times c_n$ for any search state $s \in S$ because of an operator of $F$.

**Proposition 5 (Composition)** The composition of two preconsistencies for a constraint $c$ via chaotic iteration is still a preconsistency for $c$.

The proof is straightforward by (Apt 1999). We call $\circ$ the composition of two operators via chaotic iteration. The same property holds for consistencies instead of preconsistencies.
The goal of an automatic construction procedure is to build for a constraint a set of operators which can ensure a desired level of consistency when included in a chaotic iteration. We can compare two operators $f_1$ and $f_2$ by their reductions over the search space:

**Definition 7** An operator $f_1$ is stronger than $f_2$, denoted by $f_1 \subseteq f_2$ if $\forall s \in S, f_1(s) \subseteq f_2(s)$.

For example, it is well-known that arc-consistency is stronger than bound-consistency which in turn is stronger than $id_c$. The notion of preconsistency is interesting in the context of CSP resolution because preconsistency operators have the desired properties to be included in a chaotic iteration. Since they are weaker than consistencies, they are easier to construct automatically. But since they may accept some non-solution singleton states, it is needed to find a technique to ensure completeness. The framework we propose is to build a fast but incomplete preconsistency and to add reparation operators which are delayed until the satisfiability test.

### Delaying “long” operators

In order to build operators for an intermediate consistency, we start by giving the expression of arc-consistency with a set of special function we call elementary reduction functions. Then we weaken the arc-consistency by delaying a set of functions which are supposed to be too expensive until instantiation of all variables. We use as a measure of expensive-ness the length of the syntactic expression of the function. Hence the reduction power decreases, but the computation becomes quicker. When the choice of the length threshold varies, it defines a sequence of comparable consistencies.

**Definition 8 (Support)** Let $c = (W,T)$ be a constraint, $X \in W$ and $a \in D_X$. We call support of “$X = a$” a tuple $t \in \text{sol}(c)$ such that $t_X = a$.

**Example 9** Let $c(X,Y,Z)$ be a constraint. $D_X = D_Y = \{0,1\}$ and $D_Z = \{0,1,2\}$.

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

The sets of supports for $Z$ are:

$T_{Z=0} = \{(0,0,0), (0,1,0), (1,0,0)\}$

$T_{Z=1} = \{(1,1,1)\}$

$T_{Z=2} = \{(1,1,2), (0,1,2)\}$

We call $T_{X=a} \subseteq \text{sol}(c)$ the set of all supports of $X = a$. A value $a$ has to be maintained in the current domain of $X$ only if we have at least one $t \in T_{X=a}$ such that all projections on $Y \in W \setminus \{X\}$ are included in $s_Y$.

**Definition 10 (Supp property)** Let $c = (W,T)$ be a constraint, $X \in W$ and $a \in D_X$. We call $\text{Supp}_{X=a}(s)$ the following property:

$$
\bigvee_{t \in T_{X=a}} \left( \bigwedge_{Y \in W \setminus \{X\}} t_Y \in s_Y \right)
$$

The value $X = a$ is supported if $\text{Supp}_{X=a}(s) = \text{true}$.

**Example 11 (Example 9 running)** For all $s \in S_{(X,Y,Z)}$, we have:

$\text{Supp}_{X=0}(s) = \{(0 \in s_X \land 0 \in s_Y) \lor (0 \in s_X \land 1 \in s_Y) \lor (1 \in s_X \land 0 \in s_Y)\}$

$\text{Supp}_{X=1}(s) = \{(0 \in s_X \land 1 \in s_Y)\}$

$\text{Supp}_{X=2}(s) = \{(1 \in s_X \land 1 \in s_Y) \lor (0 \in s_X \land 1 \in s_Y)\}$

If $X = a$ is not supported, then $a$ does not participate to any solution of the CSP and therefore can be eliminated. With this notion, we define a function called elementary reduction function which concerns only one value in the variable’s domain. Each value in the initial domain of each variable has its own elementary reduction function. If a value must be eliminated, its function returns this value as a singleton, and the empty set otherwise.

**Definition 12 (Elementary reduction function)** For all $X \in W$ and for all $a \in D_X$, the elementary reduction function associated to $X = a$ is $r^c_{X=a} : \mathcal{P}(D_X)$ given by

$$
\forall s \in \mathcal{P}(W), \quad r^c_{X=a}(s) = \begin{cases} \{a\}, & \text{if } \neg \text{Supp}_{X=a}(s) \\ \emptyset, & \text{otherwise} \end{cases}
$$

We call $s$-size (support size) of the elementary reduction function $r^c_{X=a}$ the cardinality of $T_{X=a}$. Arc-consistency can be defined using elementary reduction functions as follows:

$$
\forall s \in \mathcal{P}(W), \quad ac_{X=a}(s) = (s_X \cup \bigcup_{a \in D_X \setminus a} r^c_{X=a}(s))_{X \in W}.
$$

We want a consistency quicker than arc-consistency, even if it is less powerful. For this, we choose a threshold and split arc-consistency into two operators. The first one, called Short, is composed of all elementary reduction functions having their s-size less than the chosen threshold.

**Definition 13 (Operator Short)** Let $c$ be a constraint, and $n$ an integer. The operator $\text{Short}_n(c) : S_W \rightarrow \mathcal{P}(D_X)$ is given by:

$$
\forall s \in \mathcal{P}(W), \quad \text{Short}_n(c)(s) = (s_X \cup \bigcup_{a \in D_X \setminus a} r^c_{X=a}(s))_{X \in W}.
$$

The second operator, called Long, is composed of all other elementary reduction functions, which are fired only on singleton states for completeness:

**Definition 14 (Operator Long)** Let $c$ be a constraint and $n$ an integer. The operator $\text{Long}_n(c) : S_W \rightarrow \mathcal{P}(D_X)$ is given by:

$$
\forall s \in \mathcal{P}(W), \quad \text{Long}_n(c)(s) = \begin{cases} (s_X \cup \bigcup_{a \in D_X \setminus a} r^c_{X=a}(s))_{X \in W}, & \text{if } s \in \text{Sing}_W \\ s, & \text{otherwise} \end{cases}
$$

This operator is used only to reject non-solution tuples.

**Example 15 (Example 11 running)** For the variable $Z$, we can make two operators following the definition of $\text{Short}_2$ and $\text{Long}_2$. Let the threshold to 2. The operators for $Z$ are:

$\text{Short}_2(c)(Z) : sz \mapsto sz \setminus r^c_{Z=1}(s) \cup r^c_{Z=2}(s)$

$\text{Long}_2(c)(Z) : sz \mapsto sz \setminus r^c_{Z=0}(s)$

The operator $\text{Long}_2(Z)$ is delayed because its reduction power is small (only one value can be eliminated), and the effort to compute $r^c_{Z=0}(s)$ is judged too high. This operator is fired only if $s \in \text{Sing}(X,Y,Z)$. 

Proposition 16 For all integer $n$, Short$_c(n)$ is a preconsistency for $c$.

Proof First, we show that Short$_c(n)$ is monotonic. Let $s$ and $s'$ in $S_W$ such that $s \subseteq s'$. Then, $\forall X \in W, \forall a \in D_X$, $\text{Supp}_{X=a}(s) \Rightarrow \text{Supp}_{X=a}(s')$. From which it follows that $\forall X \in W, \forall a \in D_X, \neg \text{Supp}_{X=a}(s') \Rightarrow \neg \text{Supp}_{X=a}(s)$. In the case of $s'$, there are less values to eliminate than in the case of $s$. So Short$_c(n)(s) \subseteq$ Short$_c(n)(s')$. The operator Short$_c(n)$ is monotonic.

Short$_c(n)$ is contracting by construction. It it also correct by construction: it eliminates less values than arc-consistency. Hence Short$_c(n)$ is a preconsistency. □

The operator Long$_c(n)$ is also a preconsistency, the proof is similar to Short$_c(n)$. When both operators are in a chaotic iteration, we get a preconsistency:

Proposition 17 Let $c = (W, T)$ a constraint and $n$ an integer. The composition Short$_c(n) \circ$ Long$_c(n)$ is a preconsistency for $c$.

Proof According to the proposition 5, Short$_c(n) \circ$ Long$_c(n)$ is a preconsistency. But $\forall s \in \text{Sing}_W$, Short$_c(n) \circ$ Long$_c(n)(s) = ac_c(s)$, so Short$_c(n) \circ$ Long$_c(n)$ is associated to $c$. Therefore Short$_c(n) \circ$ Long$_c(n)$ is a consistency for $c$. □

Elementary reduction functions of Long$_c(n)$ are not fired on non-singletonic states. If the threshold $n$ is not too big, the set of elementary reduction functions of Long$_c(n)$ is not empty. In that case, the consistency Short$_c(n) \circ$ Long$_c(n)$ is weaker than arc-consistency, but it can be computed faster, since that we have less functions to evaluate. If $n$ is large enough, all elementary reduction functions are in Short$_c(n)$ and we get the same reduction power and computational cost as arc-consistency.

Implementation and Example

Implementation. A system generating the operators Short$_c(n)$ and Long$_c(n)$ has been implemented. The language used to express the operators is the indexical language (van Hettenryck, Saraswat, & Deville 1991) of GNU-Prolog (Diaz & Codognet 2001). An indexical operator is written “$X$ in $r$” where $X$ is the name of a variable, and $r$ is a range of possible values for $X$ and which may depend on other variables’ current domains. If we call $r$ the current domain of $X$, the indexical “$X$ in $r$” can be read declaratively as the second-order constraint $x \subseteq r$ or operationally as the operator $x \mapsto x \cap r$. For a given constraint and a threshold, our system returns two sets of $|W|$ indexical operators, i.e. one for each variable. The first set defines the Long$_c(n)$ operator and the second Short$_c(n)$. In total, we have $2 \times |W|$ indexical operators for a constraint $c = (W, T)$. The closure by chaotic iteration of all $2 \times |W|$ operators is equivalent to the consistency Short$_c(n) \circ$ Long$_c(n)$. The indexical operators for Long$_c(n)$ are delayed with a special trigger $\text{val}$ in the indexical language which delays the operator until a given variable is instantiated. This is how we get that these operators are not iterated on non-singletonic search states. This delay is obtained at no cost because GNU-Prolog uses separate queues. In all cases, the generation time is about one second.

Example. The sparse distribution described in figure 1 and 2 occurs, for example, in the case of the constraints $\text{word}_3(X_1, X_2, \ldots, X_n)$. For $n = 3$, the constraint $\text{word}_3(X, Y, Z)$ means that $XYZ$ is a 3-letters English word. We consider that domains are ordered by lexicographic ordering. In figure 4 is presented the projections of $\text{word}_3(X, Y, Z)$ on different planes. When using UNIX dictionary /usr/dict/words, this constraint has 576 solutions. The constraints $\text{word}_4(X, Y, Z, U)$ (with 2236 solutions) and $\text{word}_5(X, Y, Z, U, V)$ (with 4176 solutions) have the same regularity.

The CSP we use consists in finding a solution for crossword grids of different sizes. The CSP is composed only by a set of constraints $\text{word}_n$. The domain of all variables is $\{a, \ldots, z\}$. An example of a 7x7 grid and its model are presented table 1.

Only the first solution is computed. Some benchmarks are presented in the tables 2, 3, 4. The full reduction power of arc-consistency is obtained for the following thresholds:
Table 2: Some results for 7x7 grid

<table>
<thead>
<tr>
<th>Threshold for</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>word₂</td>
<td>word₃</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>fd_relation</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Some results for 10x10 grid

<table>
<thead>
<tr>
<th>Threshold for</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>word₂</td>
<td>word₃</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>65</td>
</tr>
<tr>
<td>fd_relation</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Some results for 15x15 grid

<table>
<thead>
<tr>
<th>Threshold for</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>word₂</td>
<td>word₃</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>130</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>130</td>
</tr>
<tr>
<td>1</td>
<td>70</td>
</tr>
<tr>
<td>fd_relation</td>
<td></td>
</tr>
</tbody>
</table>

References


