Solving the weighted Maximum Constraint Satisfaction Problem using Dynamic and Iterated Local Search

Diana Kapsa
University of British Columbia
Department of Computer Science
dkapsa@cs.ubc.ca

Jacek Kisynski
University of British Columbia
Department of Computer Science
kisynski@cs.ubc.ca

Abstract

Weighted Max-CSP is one of the NP-hard problems that has been studied for a long time due to its relevance for various research areas e.g. operations research. Nevertheless, stochastic local search methods for solving weighted Max-CSP remain largely unexplored. In this project we developed different variants of Iterated and Dynamic Local Search algorithms and empirically analyzed them. The obtained results give a solid basis for a further development of more sophisticated algorithms for weighted Max-CSP.

1 Introduction

An instance of the constraint satisfaction problem (**CSP**) is defined by a set of variables, a domain for each variable and a set of constraints. A solution is a variable assignment for all variables that satisfies all constraints. **Max-CSP** can be regarded as the generalization of CSP; the solution maximizes the number of satisfied constraints. Max-CSP is usually considered with regards to over-constrained CSP instances, in which it is often impossible to satisfy all constraints. In **weighted Max-CSP**, each constraint is associated with a positive real value as a weight. The solution maximizes the total sum of the satisfied constraints weights. Weights reflect the importance of constraints. In particular, they might be used to encode distinction between hard and soft constraints.

Solving the weighted max-CSP problem is computationally hard, as it is the generalization of the CSP problem, which is $\mathcal{NP}$-complete.

An example of a problem that can be naturally encoded into Max-CSP is *university examination timetabling* ([6]). Another practical example is *radio link frequency assignment* ([1], [2]). Such problems involve different categories of constraints according to their relevance.

1.1 Formal Definitions

The formal definitions of *CSP instance, weighted CSP instance, variable assignment* and *weighted Max-CSP problem* are as follows [9]:

Definition 1.1 (CSP instance) *A **CSP instance** is a triple $\mathbf{P} = (\mathbf{V}, \mathbf{D}, \mathbf{C})$, where $\mathbf{V} = \{x_1, x_2, \ldots, x_n\}$ is a finite set of n variables, $\mathbf{D}$ is a function that maps each variable x_i to the set $\mathbf{D}(\mathbf{x_i})$ of possible values it can take (domain of x_i) and $\mathbf{C} = \{C_1, C_2, \ldots, C_m\}$ is a finite set of constraints. Each constraint C_j is a relation over an ordered set $Var(C_j)$ of variables from $\mathbf{V}$, i.e., for $Var(C_j) = \{y_1, y_2, \ldots, y_k\}$, $C_j \subseteq \mathbf{D}(y_1) \times \mathbf{D}(y_2) \times \ldots \times \mathbf{D}(y_k)$. The elements of the set C_j are called* satisfying tuples *of C_j and k is called* the arity of the constraint C_j

In a **binary CSP instance**, the constraints are unary or binary.

Definition 1.2 (Weighted CSP instance) *A **weighted CSP instance** is a pair $(\mathbf{P}, \mathbf{w})$, where $\mathbf{P}$ is a CSP instance and $\mathbf{w} : \{C_j | j \in [1, 2, \ldots, m]\} \mapsto \mathbb{R}^+$ is a function that assigns a positive real value to each constraint C_j of $\mathbf{P}$. $\mathbf{w}(C_j)$ is called the* weight *of constraint C_j*

Definition 1.3 (Variable assignment) *Given the CSP instance $\mathbf{P} = (\mathbf{V}, \mathbf{D}, \mathbf{C})$, a **variable assignment** of $\mathbf{P}$ is a mapping $\mathbf{a} : \mathbf{V} \mapsto \bigcup\{\mathbf{D}\}$ that assigns to each variable $x_i \in \mathbf{V}$ a value from its domain $\mathbf{D}(x_i)$.* Assign($\boldsymbol{P}$) *denotes the set of all possible variable assignments for P.*

Definition 1.4 (Weighted Max-CSP) *Given a weighted CSP instance $\mathbf{P}' = (\mathbf{P}, \mathbf{w})$, let $f(\mathbf{P}', \mathbf{a})$ be the total weight of constraints of $\mathbf{P}$ satisfied under variable assignment $\mathbf{a}$:*

$$f(\mathbf{P}', \mathbf{a}) = \sum_{j=1}^{m} \{\mathbf{w}(C_j) | C_j \text{ is a constraint of } \mathbf{P} \text{ and } \mathbf{a} \text{ satisfies } C_j\}.$$

*The **weighted Max-CSP** problem is to find a variable assignment $\mathbf{a}^*$ that maximizes the total weight of the satisfied constraints in $\mathbf{P}$:*

$$\mathbf{a}^* \in \text{argmin}\{f(\mathbf{P}', \mathbf{a}) | \mathbf{a} \in \text{Assign}(\boldsymbol{P})\}.$$

As

$$\begin{aligned} &\text{argmin}\{f(\mathbf{P}', \mathbf{a}) | \mathbf{a} \in \text{Assign}(\mathbf{P})\} = \\ &= \text{argmax}\left\{\textstyle\sum \mathbf{w}(C_j) - f(\mathbf{P}', \mathbf{a}) | C_j \text{ is a constraint of } \mathbf{P} \ \wedge \ \mathbf{a} \in \text{Assign}(\mathbf{P})\right\}, \end{aligned}$$

weighted Max-CSP might be considered as a minimization problem, where the objective is to find a variable assignment which minimizes the total weight of the unsatisfied constraints. In our paper we use the minimization approach.

In the rest of this paper, we focus on the binary weighted Max-CSP with positive integer weights[1] and $D(x_1) = D(x_2) = \ldots = D(x_n) \subset \mathbb{N}^+$.

[1] This does not result in the loss of generality, as n-ary relations can be described using binary relations and real weights can be scaled into integers.

2 Related Works

Among the previously applied methods for solving Max-CSP, complete algorithms like branch-and-bound and backtracking based techniques [18] play an important role. Exponential growth of the complexity with growing instance size is their main disadvantage. Besides, for practice related problems like scheduling as well as for large-scale instances, achieving a guaranteed optimal solution might be extremely time consuming.

As both the best solution quality and the required run time (CPU-time or iterations number) are important criteria for measuring the performance of a Max-CSP solver, incomplete techniques get more interesting. Even though not much work has been done in this area, several stochastic local search algorithms have been developed that had a major impact on all later contributions.

2.1 The Min-Conflict-Heuristic

One of the first major efforts for solving the CSP using SLS has been made by Minton et al. [13]. Although their algorithm addresses the CSP rather than the Max-CSP, it inspired several of the later native Max-CSP solvers. As it is often used as a comparison criterion for the performance of the different algorithms, we will also consider this algorithm in our experiments.

The *Min-Conflicts heuristic* is driven by the idea of "repairing a complete but inconsistent assignment by reducing inconsistencies". The original version of the algorithm starts with an initial random assignment $\mathbf{a} : \mathbf{V} \mapsto \bigcup\{\mathbf{D}\}$ and a value $f(\mathbf{a})$ of the objective function. In each local search step first a variable x_i is chosen uniformly at random from the conflict set $\mathcal{C}(\mathbf{a})$. Given the assignment $\mathbf{a}$, the conflict set is defined as the set of all variables $x_i \in \mathbf{V}$ that appear in at least one currently violated constraint (see Figure 1). In a second step a value $d \in \mathbf{D}(\mathbf{x_i})$ is chosen (1-exchange neighborhood) such, that by assigning d to x_i the total number of subsequently violated constraints is minimized. Among several values that satisfy this criterion, one is chosen uniformly at random.

Based on this value-ordering heuristic, an *Iterated Improvement* variant can be applied to the Max-CSP, the **MCH**. In this case the objective function value is given by sum of weights of all violated constraints under the assignment $\mathbf{a}$. After generating an initial random assignment, in each search step the algorithm tries to minimize the total weight of the violated constraints for the randomly selected variable. After randomly choosing a variable x_i from the current conflict set, MCH computes the sum of the weights of the violated constraints related to this variable x_i for all possible domain values. From the best-scored values (there might be several candidates with equal objective function value) the algorithm then selects one uniformly at random. MCH terminates when the specified solution quality has been achieved or a fixed number of iterations has been exceeded. The step function implementation is using the neighborhood evaluation table presented in 5.3, of complexity $O(|\mathbf{D}|)$. The resulting pseudo code is listed in Figure 1.

While efficient in terms of run time, the algorithm has no possibility to escape from a local minimum and the Min-Conflicts heuristic has a major drawback: stagnation. Based on this, several other variants have been developed. In their studies on the unweighted Max-CSP, Galinier and

procedure basic_MCH(π')
  input: problem instance $\pi' \in \Pi'$
  output: solution $s \in S(\pi')$ or $\emptyset$;

  s:= init(π')
  $\hat{s}$:=s
  while not terminate(π', s) **do**
    s:= mch_step(π', s)
    if $f(\pi', s) \leq f(\pi', \hat{s})$ **then**
      $\hat{s}$:=s
    end
  end
  if $\hat{s} \in S'(\pi')$ **then**
    return $\hat{s}$
  else
    return $\emptyset$
  end
end basic_MCH

procedure mch_step(π, s)
  input: problem instance π, candidate solution s;
  output: candidate solution s';

  $\mathcal{C} = \{x_i \in \mathbf{V} | x_i$ is a variable currently in conflict$\}$
  x_i:= random_from_set($\mathcal{C}$)
  $\mathcal{I}^*(s, x_i) = \{d \in \mathbf{D}(\mathbf{x_i}) | d$ minimizes the total weight of currently violated constrains
      in which x_i appears$\}$
  d^*:= random_from_set($\mathcal{I}^*(s, x_i)$)
  s':= $s_{|x_i=d^*}$
  return s'
end mch_step

Figure 1: MCH on Max-CSP; init(π') returns a random candidate solution using a uniform distribution; random_from_set(A) returns a random element from set A using a uniform distribution; $S'(\pi')$ is the set of feasible solutions; a feasible solution is defined as a variable assignment and according to the Definition 1.3, depends on $\mathbf{V}$ and $\mathbf{D}$.

Hao [8] compared the performance of their Tabu Search variant to that of a Min-Conflict algorithm combined with a random-walk strategy, the **WMCH** [19].

After choosing a conflicting variable x_i as already mentioned, the WMCH picks randomly a value d from the x_i domain space $\mathbf{D}(x_i)$ with probability wp. With probability $1 - wp$, it performs like a basic MCH step. Considering the weight of violated constraints instead of their number, one can easily extend this algorithm to solve Max-CSP (Figure 2). This basic noise strategy leads to an improved performance, as can be concluded from our experimental results (see Figure 10).

```
procedure WMCH(π', wp)
  input: problem instance π' ∈ Π', walk probability wp;
  output: solution s ∈ S(π') or ∅;
  s:= init(π')
  ŝ:= s
  while not terminate(π', s) do
    with probability wp do
      C = {x_i ∈ V | x_i is a variable currently in conflict}
      x_i:= random_from_set(C)
      d:= random_from_set(D(x_i))
      s:= s_{|x_i=d}
    otherwise
      s:= mch_step(π', s)
    end
    if f(π', s) ≤ f(π, ŝ) then
      ŝ:=s
    end
  end
  if ŝ ∈ S'(π') then
    return ŝ
  else
    return ∅
  end
end WMCH
```

Figure 2: WMCH on Max-CSP; init(π') returns random candidate solution using uniform distribution; random_from_set(A) returns random element from set A using uniform distribution; $S'(\pi')$ is the set of feasible solutions.

2.2 Tabu Search for Max-CSP

In order to solve Max-CSP, Galinier and Hao [8] combined the Min-Conflicts heuristic with tabu search by applying the tabu tenure to each $(variable, value)$ pair. Even though extremely expensive in terms of run time, the **TSGH** successfully escapes from local minima.

Besides introducing a tabu tenure, in each search step algorithm considers all $(variable, value)$ combinations for a potential flip which intensifies the search. The underlying idea for choosing the next flip is again the Min-Conflict Heuristic. After computing the performance of each $(variable, value)$ pair as the sum of the weights of all violated constraints that one violated when assigning the value d to a variable x_i, TSGH chooses the pair with the best performance. The termination criteria are similar to the MCH. Figure 3 shows the pseudo-code of the TSGH applied to the Max-CSP.

The implementation of the tabu search variant requires the careful consideration of the underlying data structures. In each search step we consider all $|\mathbf{V}| \times |\mathbf{D}|$ $(variable, value)$ combinations

procedure TSGH(π', f, tl)
input: problem instance $\pi' \in \Pi'$, objective function $f(\Pi')$, tabu tenure tl;
output: solution $s \in S(\pi')$ or $\emptyset$;

s:= init(π')
init_tabu_list(tl)
$\hat{s}$:=s
while not terminate(π', s) **do**
$\mathcal{I}(s) = \{(x_i, d), x_i \in \mathbf{V}, d \in \mathbf{D}(x_i) | (x_i, d)$ not tabu or $f(\pi', s_{|x_i=d}) \leq f(\pi', \hat{s})\}$
$\mathcal{I}^*(s) = \{(x_i, d) \in \mathcal{I}(s) |$ assigning d to x_i minimizes the total weight of conflicts for $x_i\}$
(x_i^*, d^*):= random_from_set($\mathcal{I}^*(s)$)
update_tabu_list($(x_i, d^*), tl$)
s:= $s_{|x_i=d^*}$
if $f(\pi', s) \leq f(\pi', \hat{s})$ **then**
$\hat{s}$:=s
end
end
if $\hat{s} \in S'(\pi')$ **then**
return $\hat{s}$
else
return $\emptyset$
end
end TSGH

Figure 3: TSGH on Max-CSP; init(π') returns candidate solution chosen randomly using a uniform distribution; init_tabu_list(tl) randomly initializes the tabu list using a uniform distribution; random_from_set(A) returns a random element from set A using a uniform distribution; update_tabu_list($(x_i, d), tl$) adds the pair (x_i, d) to the tabu list and removes the oldest pair if the length of the tabu list is greater than tl; $S'(\pi')$ is a set of feasible solutions.

instead of all k possible values for one randomly chosen variable. As already mentioned the computation of the evaluation function value for each of the $|\mathbf{V}| \times |\mathbf{D}|$ pairs requires time $O(|\mathbf{D}|)$. Consequently, the implementation of the step function of the TSGH has time complexity $O(|\mathbf{V}| \times |\mathbf{D}|)$.

Since the choice of the next step is much greedier, we expect a better performance in terms of solution quality, but a worse performance in terms of run time. In order to compensate for the higher complexity of each search step, we can use special data structures to find the neighbor with the best evaluation function in the current neighborhood (see Section 5).

Figure 10 compares the performance of all algorithms on randomly generated instances from the Uniform Binary Random Model (see section 6.1) from the same instance classes as used by Galinier and Hao [8], but additionally using constraint weights, each random uniformly chosen from a domain $[0, \ldots, 99]$.

In Section 5 we present major implementation issues more in detail, as well as their impact on the

performance of the so far described algorithms and on the instance size used for the experimental work.

2.3 Randomized Rounding with MCH

The latest major contribution by Lau [11] combines a new approximation method based on randomized rounding and semidefinite programming with the already described MCH. His experimental results show this new algorithm to perform better than both the MCH and WMCH on solvable, random instances. Unfortunately, he does not compare the performance of his new algorithm to that of the TSGH. The satisfiable randomly generated instances used by Lau are also available in an unweighted version on the web site of van Beek [4], whose main research concentrates on solving binary CSPs using backtracking methods.

3 Iterated Local Search

As one of the rather straight-forward but powerful SLS methods, **ILS** (Iterated Local Search) tries to achieve a good tradeoff between intensification via local search methods and diversification by using a perturbation procedure after each encountered local minimum. As a further important element of this framework, an acceptance criterion is used to control the balance between perturbation and local search.

Clearly the ILS depends highly on the quality of the underlying stochastic local search procedure. The greedier this procedure is, the more effective the perturbation is required to be. The combination of the already mentioned key elements of an Iterated Local Search algorithm is ideally chosen in such a way that the results achieved are better than just sequentially performing several stochastic local search steps.

Based on the mentioned in Section 2 local search strategies we considered several variants of ILS: **ILS-MCH**, **ILS-WMCH** and **ILS-TSGH**. Due to the different run-time performance and achieved solution quality it is not obvious that the greediest local search strategy, TSGH, would lead to better results in this more general framework.

For perturbation we choose between different approaches. Besides random picking, we can use a fixed number of random walk steps. Using a random noise strategy shows considerable improvement in the case of the WMCH. Consequently, a perturbation procedure based on a fixed number of random walk steps should likewise help escaping from local minima. We also considered a third perturbation strategy - a random flip of a non-conflicting variable - but tests showed that it did not improve solution quality.

We denote algorithms with random picking as perturbation with the suffix "**-RP**", those with random walk steps with the suffix "**-RS**".

The acceptance criterion compares the current solution to that one of the previous iteration and chooses the better with respect to the objective function $f(\pi, s)$ with a certain acceptance proba-

```
procedure ILS-WMCH(π′, wp, bp, ws)
  input: problem instance π′ ∈ Π′, objective function f(Π′), walk probability wp,
         acceptance probability bp, number of random walk steps ws;
  output: solution s ∈ S(π′) or ∅;
  s:= init(π′)
  s:= ls-WMCH(π′, f, wp, s)
  ŝ:= s
  while not terminate(π′, s) do
    s′:= perturb(π′, s, n)
    s″:= ls-WMCH(π′, s′)
    if f(π′, s″) ≤ f(π, ŝ) then
      ŝ:=s″
    end
    s:= accept(π′, s, s″, bp)
  end
  if ŝ ∈ S′(π′) then
    return ŝ
  else
    return ∅
  end
end ILS-WMCH
```

Figure 4: ILS-WMCH on Max-CSP; init(π') returns a random candidate solution using a uniform distribution; ls-WMCH is equivalent to WMCH presented in Figure 2, except it does not have the initialization procedure; $S'(\pi')$ is a set of feasible solutions.

bility bp. The higher the acceptance probability the greedier the algorithm, and the more likely is that we will return to the local minimum encountered during the previous iteration. In this case a stronger perturbation is required, e.g. a higher number of random walk steps.

The additional complexity due to the ILS framework combined with the distinct stagnation behavior of the subsidiary local search procedures, except TSGH, motivated an appropriate control mechanism. When the evaluation function value does not change over a fixed number of iterations, the underlying local search is terminated. This fixed number is subject to carefully tuning and varies considerably among the different algorithms. A more detailed description of the tuning process is given in 7.1.1.

For each of the developed ILS variants the stagnation control mechanism was motivated as follows:

ILS-MCH: Stagnation is the main drawback of the basic MCH. By including MCH into the ILS scheme we expect to diminish this problem. Using appropriate perturbation and acceptance strategies, we try to escape from the local optimum achieved in the local search procedure. Even a very naive perturbation (e.g. random picking) combined with a greedy acceptance procedure will surely achieve a better solution quality than the initial MCH.

On the other side this approach increases time complexity from $O(|\mathbf{V}| \times |\mathbf{D}|)$ for the basic

MCH to $O(max\_ils\_iterations \times |\mathbf{V}| \times |\mathbf{D}|)$, where max_ils_iterations is the maximum number of performed ILS iterations. Consequently, we expect a considerable longer run-time and worse performance in comparison to other non-ILS approaches (e.g. setting the max_ils_iterations to $|\mathbf{V}| \times |\mathbf{D}|$ leads to the same complexity in case of TSGH).

ILS-WMCH: Although less greedy than the TSGH, WMCH is, due to it's lower complexity, better with respect to the run-time. The resulting algorithm is presented in Figure 4.

ILS-TSGH: Based on the already presented perturbation and acceptance procedures and in combination with a TSGH as local search method, we developed the ILS-TSGH. The interesting question arising in this context is about the efficiency of ILS-TSGH compared to that one of the ILS-WMCH. TSGH considers in each move the entire 1-exchange-neighborhood and tries to find the move that leads to the best performance, meaning to minimize the weight of violated constraints for the respective variable. WMCH considers only $\frac{1}{|\mathbf{V}|}$ of this neighborhood. An appropriate choice of perturbation and acceptance methods could compensate for the additional quality achieved by the TSGH. As in case of the TSGH, by keeping the maximum number of performed ILS iterations (max_ils_iterations) below $|\mathbf{V}| \times |\mathbf{D}|$, we expect especially in case of very large instances to have a comparably good performance. For the TSGH, experimental work is required in order to conclude on the gain of using the ILS framework.

4 Dynamic Local Search

Applying Dynamic Local Search (**DLS**) to more prominent NP-hard problems (particularly to SAT) leads to algorithms that outperform other approaches. Based on this - and additionally motivated by the similarities between MAX-SAT and MAX-CSP - we implemented a DLS algorithm for weighted MAX-CSP.

Unlike all other approaches presented so far, DLS uses a dynamic adjustment of the evaluation function in order to escape from local minima. When the subsidiary local search procedure encounters an optimum, the algorithms changes the evaluation of the found solution such that further improvement can be done. Penalizing the affected solution components is one common way of implementing this.

Our DLS algorithm is based on the random walk variant of the min-conflicts based iterative improvement, WMCH (see Section 2.1), as subsidiary local search procedure. Using the 1-exchange-neighborhood, the WMCH is a local conflict driven best improvement algorithm; each move is chosen in a two-step decision process that has only $O(|\mathbf{D}|)$ complexity compared to the usual $O(|\mathbf{V}| \times |\mathbf{D}|)$ required for scanning the entire neighborhood. The best DLS for SAT uses the standard best improvement technique as underlying local search strategy. However, in case of MAX-CSP, the size of the neighborhood can be considerably larger, due to the nature of the problem. Therefore, a complete scan of the neighborhood is less likely to bring significant improvement.

When penalizing the encountered local minimum, we have to consider the interaction with the original weights associated with each of the constraints. Therefore we choose to penalize each

procedure DLS-WMCH-PP(π')
  input: problem instance $\pi' \in \Pi'$;
  output: solution $s \in S(\pi')$ or $\emptyset$;

  s:= init(π')
  $\hat{s}$:=s
  init_penalties(π')
  while not terminate(π', s) **do**
    $g' = g + \sum\{penalties[i][s[i]]|i = 1, \ldots, |\mathbf{V}|\}$
    s:= ls_WMCH(π', g', s)
    $penalties$:= update_penalties($\pi', s, penalties$)
    if $f(\pi', s) \leq f(\pi', \hat{s})$ **then**
      $\hat{s}$:=s
    end
  end
  if $\hat{s} \in S'(\pi')$ **then**
    return $\hat{s}$
  else
    return $\emptyset$
  end
end DLS-WMCH-PP

procedure update_penalties(π, s)
  input: problem instance π', candidate solution s;
  output: penalties table $penalties$;

  $\mathcal{C} = \{x_i \in \mathbf{V}|x_i$ is a variable currently in conflict$\}$
  for each $x_i \in \mathcal{C}$ **do**
    $penalties[i][s[i]]$:= $penalties[i][s[i]] + p$
  end
  return $penalties$
end MCH_step

Figure 5: DLS-WMCH-PP on Max-CSP; init(π') returns a random candidate solution using a uniform distribution; $penalties$ is a $|\mathbf{V}| \times |\mathbf{D}|$ matrix in which each element (i, j) indicates the penalty for considering (x_i, d_j) as a solution component; $penalties$ matrix is initialized with zeros by procedure init_penalties(π'); g' guides the local search of the underlying WMCH based local search procedure.S is the n-dimensional solution matrix $S'(\pi')$ is a set of feasible solutions.

solution component, in our case each (variable, value) pair of the solution and derive the following evaluation function:

$$g'(\pi', s) = \begin{array}{l} \sum(w(C_j)|C_j \text{ is an unsatisfied constraint for s})+ \\ \sum(penalties(x_i, d_j)|(x_i, d_j) \text{ is a solution component}) \end{array}$$

Penalizing each $(variable, value)$ pair leads to a similar effect as using a tabu tenure (see Section

5.2) but has three major differences:

- Reduced complexity when choosing the next flip compared to TSGH
- Using incremental penalties instead of a fixed tabu tenure leads to algorithms that avoid a certain solution component but not forbid it for a fixed number of iterations.
- The tabu status of a $(variable, value)$ pair is reset after a fixed number of iterations, while the penalties, even in the case of smoothing, tend to remain or even increase.

Initially all weights are set to 0 as we assume that randomly generated assignment is not a local optimum. The algorithm penalizes (after each local search phase) the encountered solution by incrementing the respective components by a constant penalty factor. Due to the fact that WMCH has no special escape mechanism, and due to the increased number of total iterations (inner and outer loop), we use - as in case of the ILS - a stagnation based termination (see Section 3). The resulting pseudo-code is presented in the Figure 5.

Depending on which of the solution components are penalized at the end of the local search phase we distinguished, implemented and experimented the three following different variants:

DLS-WMCH-TP: The total penalization variant increases weights for all $(variable, value)$ pairs involved in the currently encountered local optimum. The idea is to simply change the evaluation function in such a way that it avoids already analyzed positions in the search space.

DLS-WMCH-PP: Here the penalization is restricted to $(variable, value)$ pairs that involve variables from the current conflict set. The motivation behind this variant is to keep good solution components and avoid the rest.

DLS-WMCH-NP: The non-conflicting penalization variant increases weights only of those variables that are currently not violating any constraints. The resulting moves might lead new solutions that could not be encountered else due to the accepted deterioration of the evaluation function.

5 Implementation

In the following we describe interesting implementation aspects that we encountered during the project.

5.1 Data Structures

During our project, we used a data structure that allows the representation of sufficiently large unary and binary instances. In a $|\mathbf{V}| \times |\mathbf{V}|$ matrix, where $\mathbf{V}$ is the set of variables, we consider all possible unary and binary constraints. Each of the elements of this matrix consists of a $|\mathbf{D}| \times |\mathbf{D}|$

matrix, where $\mathbf{D}$ is domain of variables. The $|\mathbf{D}| \times |\mathbf{D}|$ matrix at position (i, j) in the $|\mathbf{V}| \times |\mathbf{V}|$ matrix encodes one constraint between two variables x_i and x_j. Position (a, b) in the constraint matrix at position (i, j) specifies the penalty for assigning the values d_a and d_b to x_i respectively x_j. If the value of the element (d_a, d_b) is different from 0, a constraint is broken when the mentioned assignment occurs. The element (d_a, d_b) corresponds to the weight of this constraint. If the value of (d_a, d_b) is equal to 0, no constraint is broken or its weight is 0. For representing large problem instances, it turned out to be extremely important to only allocate memory for existing and not for all possible constraints, since the $|\mathbf{V}| \times |\mathbf{V}|$ matrix tends to be sparse.

5.2 Tabu List Implementation

For the implementation of the tabu list we considered two data structures: a linked list of the tabu set elements and a $|\mathbf{V}| \times |\mathbf{D}|$ array for the tabu status of each of the $(variable, value)$ pairs. Whereas the notion of tabu list suggests rather the first alternative, the latter one is the more efficient option. The time needed for retrieving an element out of a list is linear in the length of the list, which corresponds to the tabu tenure tl. Therefore, it is comparatively expensive to find out if an element is set tabu. The larger the instance, the higher the optimal tabu tenure is likely to be and consequently the retrieval time. As opposed to this, retrieval from an array requires constant time. Given the fact that time performance in this context is more important than used memory, we decided in favor of the second alternative.

5.3 Neighborhood Evaluation

Especially in the case of the TSGH, in which we consider the entire 1-exchange-neighborhood in each local search step, the implementation of the neighborhood evaluation is an important issue. Based on the technique described in [8] we used a $|\mathbf{V}| \times |\mathbf{D}|$ array for storing the evaluation function values for each move. The element (i, j) in the $|\mathbf{V}| \times |\mathbf{D}|$ matrix specifies the resulting weight of violated conflicts for variable x_i that occur when assigning the value d_j to the variable x_i. After the random initialization at the beginning of the local search procedure, we compute all matrix values. Hence we only have to update the $|\mathbf{V}|$ affected values in the $|\mathbf{V}| \times |\mathbf{D}|$ matrix after each move. Consequently the complexity of each move is $O(|\mathbf{V}|)$.

5.4 Random Number Generation

As for every stochastic local search the random number generator is extremely important. In our case we use a linear congruential generator implementation provided by AT&T (urand.c). The random number generator is initialized by using the current calendar time (function time_t time(time_t *tp) from the time.h C library). Alternatively the user can specify a random seed for initialization in order to make results deterministic.

6 Instances

6.1 Random Instances - Uniform Random Binary CSP Model

In order to measure and compare the performance of our algorithms to both complete and incomplete approaches, we conducted experiments on randomly generated binary instances. Our instances are currently generated according to the *Uniform Random Binary CSP model* ([16]). Each instance is characterized by the number of variables n, the domain size d as well as the density p and the tightness q of constraints. The density p describes the probability of a constraint occurring between two CSP variables, while q specifies the conditional probability that a value pair is allowed given that there is a constraint between the two variables.

The higher p and lower q is, the less likely the instance is to be satisfiable. In the rest of our paper we call such instances hard. The lower p and higher q is, the more likely the instance is to be satisfiable. In the rest of our paper we call such instances easy. In our experiments we generated instances from the same classes as used by Galinier and Hao [8] up to a number of 100 variables and 15 values in each of the respective domains. As also used by [11], we generated instances with 20 variables in order to make performance results more comparable. Values for p and q were chosen so that we cover both easier and harder instances. For our tests we used eighteen test-sets, with ten instances in each test-set.

For all instances generated based on the Uniformed Random Binary CSP model we use one weight for all constraints between each two variables. Full profit of our data structure as described in subsection 5.1 is only taken by the crafted data. Constraint weights are generated uniformly at random in a range from $[0, \ldots, 99]$.

6.2 Crafted Instances

We planned to use instances of the International Timetabling Competition [12]. The data consists of 20 instances defining scheduling problems with up to 440 events in $10 - 11$ rooms and 45 time slots (5 days, 9 hours each day). Additional constraints (hard and soft with different weights) are motivated by feature characterization of events and rooms (up to 10 different features in one instance), room sizes and student preferences (up to 350 students). All instances have a perfect solution with no broken constraints.

After in-depth analysis of the problem of encoding such instances into weighted Max-CSP instances, we decided not to use this data. The reason for this is that timetabling instances involve a lot of k-ary constraints for $k > 2$. Such constraints might be encoded as multiple binary constraints (as we only consider unary and binary constraints), but this is fairly difficult, results in growth of number of constraints, requires vast amount of memory space for data structures (or requires functional encoding of constraints) and makes the description of the problem not natural.

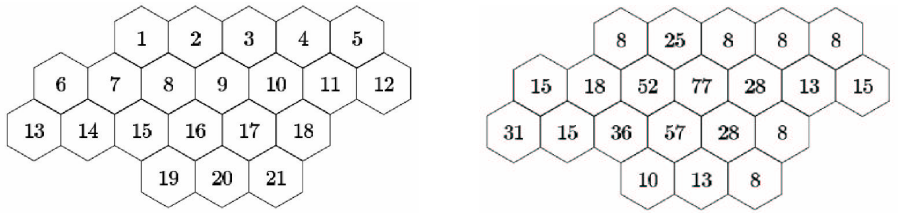

Figure 6: Network structure of the Philadelphia problems (left); demand **D1** (right) (source: [7]).

6.3 Real World Instances

We considered using either *sports tournament scheduling* instances (e.g. in [20]) or *frequency assignment problem* (**FAP**) instances. We decided to use the latter ones, as different formulations of the FAP problem are conceptually easier, and instances as well as experiment results are widely available ([7], [5], [15]).

6.3.1 Frequency Assignment Problem

The frequency assignment problem (sometimes also called channel assignment problem) arises in the area of wireless communication (e.g. GSM networks). One can find many different models of the FAP problem (due to many different applications), but they all have two common properties:

i frequencies must be assigned to a set of wireless connections so that communication is possible for each connection

ii interference between two frequencies (and what follows, quality loss of signal) might occur in some circumstances which depend on:

(a) how close the frequencies are on the electromagnetic band

(b) how close connections are to each other geographically.

There are also many objectives, which define the quality of an assignment - the goal is to obtain the highest possible quality. Survey [2] gives an extensive overview on different models, problem classifications, applied methods and results. We decided to use the so called *Philadelphia instances*, which are one of the most widely studied so far in the FAP area.

6.3.2 Philadelphia Instances

Philadelphia instances were introduced in paper [3] in 1973. They describe a cellural phone network around Philadelphia (Figure 6, left). The cells of a network are modeled as hexagons[2],

[2]Nowadays this simplified approach is no longer used. Nevertheless, Philadelphia instances are still being explored - the most recent results are available at [7].

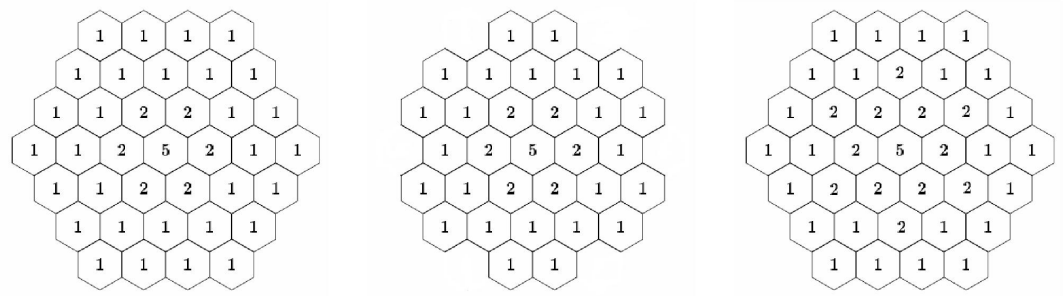

Figure 7: Reuse distances **R1** (left); **R2** (center); **R3** (right) (source: [7]).

Instance	Demand vector	Total demand	Reuse distance	Minimal span
P1	**D1**	481	**R1**	426
P2	**D1**	481	**R2**	426
P3	**D2**	470	**R1**	257
P4	**D2**	470	**R2**	252
P5	**D3**	420	**R1**	239
P6	**D3**	420	**R2**	179
P7	**D4**	962	**R1**	855
P8	**D1**	481	**R3**	524
P9	**D5**	1924	**R1**	1713

Figure 8: Philadelphia instances (source [7])

each cell requires some number of frequencies (Figure 6, right). Tuples (cell number, number of frequencies required) form a *demand vector*. Considered demand vectors include:

$\mathbf{D1} = (8, 25, 8, 8, 8, 15, 18, 52, 77, 28, 13, 15, 31, 15, 36, 57, 28, 8, 10, 13, 8)$,
$\mathbf{D2} = (5, 5, 5, 8, 12, 25, 30, 25, 30, 40, 40, 45, 20, 30, 25, 15, 15, 30, 20, 20, 25)$,
$\mathbf{D3} = (20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20)$,
$\mathbf{D4} = (16, 50, 16, 16, 16, 30, 36, 104, 154, 56, 26, 30, 62, 30, 72, 114, 56, 16, 20, 26, 16)$,
$\mathbf{D5} = (32, 100, 32, 32, 32, 60, 72, 208, 308, 112, 52, 60, 124, 60, 144, 228, 112, 32, 40, 52, 32)$.

The distance between different cell centers is 1. Frequencies are denoted as positive integer numbers. Interference of the cells is characterized by a reuse distance vector $(d_0, d_1, d_2, d_3, d_4, d_5)$: d_k denotes the smallest distance between centers of two cells, which can use frequencies that differ by at least k without interference. Figure 7 shows graphical representation of reuse distance vectors $\mathbf{R1} = (\sqrt{12}, \sqrt{3}, 1, 1, 1, 0)$, $\mathbf{R2} = (\sqrt{7}, \sqrt{3}, 1, 1, 1, 0)$ and $\mathbf{R3} = (\sqrt{12}, 2, 1, 1, 1, 0)$.

The objective is to find frequency assignments which result in no interference and minimize the span of frequencies used, i.e., the difference between maximum and minimum frequency used.

Figure 8 defines the Philadelphia instances explored in the literature (named **P1**-**P9** in conformity with [17]) and provides the value of the optimal solution (minimal span). The instance **P1** is the original instance motivated by the above mentioned cellural phone network.

6.3.3 Encoding into Weighted Max-CSP

We encoded Philadelphia instances into the weighted Max-CSP problem structure in the following way:

variables: each frequency requirement is represented as a single variable, e.g. for demand vector **D1** there are 481 variables, 8 of them correspond to the demand of cell 1, 25 correspond to the demand of cell 2, etc.

domains: each variable has the same integer domain $[0, 1, \ldots, l + \lceil 1 \times \text{WORST\_OPT} \rceil]$, where l is the minimal upper bound known for the particular instance[3] and WORST_OPT ≥ 1.0 is a real constant.

constraints: constraints are divided into two groups:

hard binary constraints: correspond to the requiment of no occuring interference. A hard constraint between two locations is set based on the network structure and the reuse distance vector for particular instances. Weights of hard constraints are equal to some integer number greater than the maximum sum of the weights of the broken soft constraints.

soft unary constraints: correspond to the minimization of the used frequencies span. Each possible variable value is "penalized" by a unary soft constraint with weight: *value*$\times$FORCE_MIN, where FORCE_MIN≥ 1.0 is a real constant and $value$ is a value of the variable.

Setting value of two constants WORST_OPT and FORCE_MIN involves following tradeoffs:

- the larger WORST_OPT (which implies bigger domain size), the easier the algorithm may satisfy hard constraints, but also the bigger the search space will be,
- the larger FORCE_MIN (which implies larger weights for soft constraints) the more likely the algorithm minimizes span, but the more is the algorithm attracted to the smallest frequency values (which might occur to be the main weakness of such encoding).

Storing such weighted Max-CSP instances in data structures described in Section 5.1 is not realistic as it would require approximately $12GB - 24TB$ of memory. Functional encoding of constraints would result in loss of efficiency. A solution of this problem is based on the observation, that in each instance there are only five types of binary hard constraints ($x_i - x_j \geq 1, x_i - x_j \geq 2, \ldots, x_i - x_j \geq 5$) and (*size of domain*) different types of the unary soft constraints.

Constraints are stored in six two-dimensional arrays (five for each hard constraint and one for all soft constraints), indexed with domain values. Numbers in the arrays describe the total sum of weights of broken constraints for particular value assignment (0 - no constraint broken). Finally, a two-dimensional array - which is indexed with variable names - contains for each (var_i, var_j) pair

[3] For Philadelphia instances, the best lower bound known is actually the optimal solution, but one may start with $6\times$*number of variables* as an obvious upper bound and change it later using results from performed experiments.

Algorithm	REPETITIONS of LS			STAGNATION FRACTION		
	Number of variables			Number of variables		
	20	50	100	20	50	100
ILS-MCH	200	100	40	10	25	50
ILS-WMCH	100	20	10	20	25	50
ILS-TSGH	200	100	–	10	25	–
DLS-WMCH-TP	20	10	10	20	25	50
DLS-WMCH-PP	20	10	10	20	25	25
DLS-WMCH-NP	20	10	10	20	25	25

Figure 9: Local search repetitions and stagnation parameter settings for random instances.

a pointer to one of the six arrays, or a NULL pointer if there is no constraint between the variables. Particularly, for all pairs (var_i, var_i) a pointer points to the array storing the soft constraints. All arrays are allocated dynamically, which gives freedom in changing the WORST_OPT parameter.

7 Experiments

All experiments were performed using the LSF load distribution system running on Linux machines. Tuning tests were run on dual 1GHz Intel Pentium III, 256KB cache, 4GB/2GB RAM computers, final tests were run on dual 2GHz Intel Xeon, 512KB cache 4GB/2GB RAM computers. The algorithms were implemented in C and compiled with gcc 2.92.2.

7.1 Random Instances

As the instance generator does not guarantee that instances satisfiable and because of the size of the instances (exhaustive search would be intractible) we use the *absolute ratio* (sum of weights of satisfied constraints divided by sum of weights of all constraints) to measure and compare the achieved solution quality. In the following, we use the absolute ratio as indicator for the solution quality. All tuning tests (except stagnation tuning) were quality driven, and time was taken into account if a decision was not possible based on quality performance.

7.1.1 Testing Protocol

All algorithms were tested according to the following rules:

- The number of iterations for MCH, WMCH and TSGH was set to $10,000$.
- The number of repetitions of ILS and DLS local search and the stagnation limit (determining termination - see section 3) of ILS and DLS local search were determined through some pretuning tests (in case of ILS with random picking as perturbation and probability of accepting

a better solution set to 1.0, in case of DLS with penalty p set to 1). The stagnation limit was tuned in order to obtain the highest $moves/iterations$ ratio and minimum iterations number without significant loss of solution quality. The number of repetitions was tuned to achieve an overall number of iterations of local search of approximately $10,000 - 20,000$ (but no less repetitions then 10). Number of iterations of local search had cut-off 10000 (providing the stagnation criterion did not make it terminate earlier) in order to ensure reasonable experiments time. Figure 9 shows the final setting of the parameters.

The stagnation limit showed to be fairly difficult to tune. We found out that some reactive mechanism changing this parameter during search progress might be a better solution.

- ILS and DLS local search algorithms were tuned due to the results obained while testing them as stand-alone procedures.

- **Parameters tuning:** consisted of two stages:

  Range estimating: 10 runs were performed on each instance in each test-set in order to bound range of parameter values. For each instance and each parameter value, the mean absolute ratio was calculated, and then for each test-set the mean over mean absolute ratios for test-set members was calculated and used to compare performance.

  Parameters setting: Based on the results from the previous point, up to 10 different parameter values were chosen and test runs performed (10 runs on each instance). As previously, mean over mean absolute ratios for the test-set members was used to determine the optimal parameter setting for each test-set of instancess.

- **Experiments:** Using an optimal parameter setting for each test-set of instances obtained during tuning (optimal in sense of obtained solution quality), two kinds of tests were performed:

  General: 100 runs were performed on each instance in each test-set. For each instance were calculated: the mean absolute ratio, minimum and maximum absolute ratio, the mean number of iterations, the mean number of moves (iterations in which some variable was flipped, it does not include random steps in WMCH and perturbation steps in ILS), the mean moves per iterations fraction, the mean time to perform $100,000$ iterations, and the mean time to perform $100,000$ moves. Then for each test-set were calculated: the mean over mean absolute ratios for test-set members, minimum and maximum mean absolute ratios (Figures 10, 11, and 12), the mean over mean iterations, over mean moves, over mean moves per iterations fraction (Figures 13, 14, 14 and 16), the mean over mean time to perform $100,000$ iterations, and over the mean time to perform $100,000$ moves (Figures 17, 18 and 19).

  Specific: 1000 runs were performed on one instance from each test-set. The obtained data was used to produce plots.

- ILS and DLS local search algorithms were tuned due to the results obained while testing them as stand-alone procedures.

7.1.2 Tuning

WMCH: Tests with walk probability wp set to 0.00, 0.20, 0.40, 0.60, 0.80 suggested a range of $[0.00, 0.40]$. Additional tests with wp set to 0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35 were performed. The value 0.05 resulted in the best performance for all test-sets except $20.10.50.65$. Interestingly, 0.20 was the second best value for many instances, and the best value for instances in the test-set $20.10.50.65$ (which happend to consist of solvable instances). Most likely, more detailed tests would result in setting wp to a value smaller then 0.05 for many test-sets.

TSGH: Tests with tabu length tl set to 0, 50, 100, 150, 200 (except test-sets with 20 variables - it would put all possible $(variable, value)$ pairs on the tabu tenure) suggested range $[30, 150]$ for 20 variables, $[70, 200]$ for 50 variables and $[140, 230]$ for 100 variables. Additional tests with tl changing by 10 in every range for corresponding test-sets of instances were done.

For test-sets with 20 variables, values between 40 and 100 resulted in the best performance for all test-sets. Interestingly, for tl equal to 150, mean differed from one for optimal setting on the 3rd decimal place while testing test-sets with 20 variables and size of domain 10. This suggests, that implementation of the algorithm which uses techniques prefering strongly least recently flipped variables might be worth considering.

For test-sets with 50 variables, values between 70 and 150 variables resulted in the best performance. For 100 variables tl set to 140, 150 for test-sets consisting of easy instances and set to 210, 230 for test-sets consisting of hard instances was optimal.

Very long running time for test-sets with 100 variables, did not allow us to test ILS algorithms with TSGH as local search on those test-stes.

All tests for TSGH showed that, the harder the instance test-sets as well as the higher the number of variables, the longer the tabu tenure is needed in order to achieve optimal performance.

ILS-MCH-RP: Tests with probability of accepting the best so far solution bp set to 0.25, 0.5 and 0.75 suggested no change in solution quality for different values of bp. Additional tests with bp set to 0.1, 0.2, 0.35, 0.45, 0.5, 0.65 and 0.8 confirmed this observation. It is quite natural, that the tests had rather debugging motivation, as using random picking as the perturbation method renders the acceptance criterion irrelevant. The value 0.8 was chosen arbitrarily (as time also did not vary).

ILS-MCH-RS: Tests with probability of accepting the best so far solution bp set to 0.25, 0.5 and 0.75 and number of random walk steps ws set to 10, 20 and 50 (all possible combinations of those two parameters) showed that bp equal to 0.75 is the best among three tested settings. The number of random steps ws set to half of the number of variables usually resulted in the best performance.

Additional tests with bp set to 0.65, 0.75, 0.85 and ws set to the previously obtained optimal value for a particular test-set, to value 5 larger and to value 5 smaller (all possible combinations of bp and ws) were run to determine the parameters with a higher precision.

ILS-WMCH-RP: Tests with probability of accepting the best so far solution bp set to 0.25, 0.5 and 0.75 suggested no change in solution quality for different values of bp. Additional tests

with bp set to 0.15, 0.35, 0.45, 0.55, 0.65, 0.7 and 0.8 confirmed this observation. Again the tests had rather debugging motivation. The value 0.8 was chosen arbitrarily (as time also did not vary).

ILS-WMCH-RS: Tests with probability of accepting the best so far solution bp set to 0.25, 0.5 and 0.75 and number of random walk steps ws set to 10, 20 and 50 (all possible combinations of bp and ws) showed that bp equal to 0.75 is the best among three tested settings. The number of random steps ws equal to 10 and 20 usually resulted in the best performance.

Additional tests with bp set 0.65, 0.75, 0.85 and ws set to the previously obtained optimal value for a particular test-set, to value 5 larger and to value 5 smaller (all possible combinations of bp and ws) were run to determine the parameters with a higher precision.

ILS-TSGH-RP: Tests with probability of accepting the best so far solution bp set to 0.25, 0.5 and 0.75 suggested no change in solution quality for different values of bp. Additional tests with bp set to 0.15, 0.35, 0.45, 0.55, 0.65, 0.7 and 0.8 confirmed this observation. Again the tests had rather debugging motivation. The value 0.8 was chosen arbitrarily (as time also did not vary).

ILS-TSGH-RS: Tests with probability of accepting the best so far solution bp set to 0.25, 0.5 and 0.75 and number of random walk steps ws set to 10, 20 and 50 (all possible combinations of those two parameters) showed that bp equal to 0.75 is the usually best among three tested settings. The number of random steps ws set to 10 usually resulted in the best performance.

Additional tests with bp set to previously obtained optimal value for particular test-set, to value 0.1 larger and to value 0.1 smaller and ws set to previously obtained optimal value for a particular test-set, to value 5 larger and to value 5 smaller (all possible combinations of bp and ws) were run to determine the parameters with a higher precision.

DLS-WMCH-TP: Tests with penalty p set to 1, 5 and 10 were performed. p setting 1 resulted in the best performance. Additional tests with p set to 2, 3 and 4 showed that values 1 and 2 are optimal settings for all test-sets.

DLS-WMCH-PP: Tests with penalty p set to 1, 5 and 10 were performed. p setting 1 resulted in the best performance. Additional tests with p set to 2, 3 and 4 showed that values 1 and 2 (in case of test-set $100.15.10.50$ - 4) are optimal settings for all test-sets.

DLS-WMCH-NP: Tests with penalty p set to 1, 5 and 10 were performed. They showed great variety in optimal settings. Additional tests with p set to 2, 3, 4, 6, 7, 8, 9, 11 and 12 confirmed this observation. Values 1, 2, 3, 5, 68, 10 and 11 resulted in optimal performance for different test-sets.

7.1.3 Results

MCH, WMCH and TSGH Based on the absolute ratios (Figure 10) we can conclude that among the three algorithms WMCH achieves on the average the higher absolute ratios and therefore the best quality for the same number of iterations. On solvable instances, TSGH always succeeds to solve the instance, but for 100 variables it performs significantly worse in terms of solution quality than WMCH and even sometimes than MCH.

Test set $n.d.p.q$	ABSOLUTE RATIOS														
	MCH			**WMCH**				**TSGH**				**ILS-MCH-RP**			
	Min	Mean	Max	wp	Min	Mean	Max	tl	Min	Mean	Max	bp	Min	Mean	Max
20.10.10.60	0.879	0.886	0.892	0.05	0.916	0.926	0.935	40	0.914	0.923	0.929	0.80	0.904	0.913	0.917
20.10.10.70	0.947	0.951	0.956	0.05	0.973	0.979	0.987	40	0.970	0.977	0.984	0.80	0.966	0.971	0.975
20.10.30.15	0.439	0.465	0.479	0.05	0.476	0.517	0.532	60	0.475	0.517	0.533	0.80	0.466	0.503	0.517
20.10.30.30	0.651	0.663	0.674	0.05	0.704	0.722	0.749	70	0.700	0.721	0.751	0.80	0.690	0.704	0.721
20.10.30.35	0.709	0.726	0.750	0.05	0.766	0.787	0.816	70	0.765	0.784	0.812	0.80	0.750	0.768	0.793
20.10.50.20	0.604	0.617	0.646	0.05	0.672	0.689	0.711	90	0.670	0.688	0.710	0.80	0.655	0.671	0.695
20.10.50.50	0.893	0.907	0.916	0.05	0.942	0.962	0.974	50	0.938	0.960	0.973	0.80	0.929	0.948	0.963
20.10.50.65	0.975	0.979	0.985	**0.20**	0.999	1.000	1.000	50	1.000	**1.000**	1.000	0.80	0.998	0.999	1.000
50.10.10.60	0.798	0.800	0.805	0.05	0.816	**0.819**	0.828	70	0.806	0.810	0.819	0.80	0.807	0.810	0.816
50.10.10.70	0.875	0.879	0.883	0.05	0.891	**0.895**	0.900	70	0.881	0.886	0.890	0.80	0.884	0.887	0.891
50.10.30.30	0.528	0.532	0.538	0.05	0.552	0.558	0.564	110	0.542	0.547	0.555	0.80	0.542	0.546	0.553
50.10.30.35	0.580	0.589	0.594	0.05	0.607	0.615	0.622	110	0.598	0.605	0.612	0.80	0.596	0.603	0.609
50.10.50.20	0.446	0.455	0.460	0.05	0.473	0.484	0.491	150	0.461	0.474	0.481	0.80	0.461	0.471	0.476
100.15.10.40	0.565	0.566	0.567	0.05	0.573	0.574	0.575	150	0.561	0.563	0.565	0.80	0.569	0.570	0.572
100.15.10.45	0.615	0.617	0.619	0.05	0.623	0.625	0.627	150	0.614	0.616	0.618	0.80	0.619	0.621	0.623
100.15.10.50	0.665	0.666	0.668	0.05	0.672	0.673	0.675	140	0.665	0.667	0.668	0.80	0.668	0.670	0.671
100.15.30.15	0.298	0.300	0.302	0.05	0.307	0.309	0.311	210	0.253	0.262	0.271	0.80	0.303	0.304	0.307
100.15.30.20	0.363	0.364	0.366	0.05	0.371	0.373	0.376	230	0.329	0.335	0.339	0.80	0.367	0.369	0.371

Figure 10: Experiments results - quality. Test-set of satisfiable instances, the best mean absolute ratios for each test-set, as well as other interesting values described in the text are typed in the boldface.

Test set	ABSOLUTE RATIOS																	
	ILS-MCH-RS					**ILS-WMCH-RP**				**ILS-WMCH-RS**					**ILS-TSGH-RP**			
$n.d.p.q$	bp	ws	Min	Mean	Max	bp	Min	Mean	Max	bp	ws	Min	Mean	Max	bp	Min	Mean	Max
20.10.10.60	0.65	10	0.910	0.920	0.923	0.80	0.916	0.926	0.932	0.65	15	0.917	0.928	0.937	0.80	0.908	0.916	0.975
20.10.10.70	0.85	10	0.969	0.976	0.983	0.80	0.973	0.980	0.986	0.85	15	0.974	**0.981**	0.988	0.80	0.969	0.975	0.980
20.10.30.15	0.65	20	0.470	0.511	0.527	0.80	0.477	0.518	0.537	0.60	25	0.481	**0.520**	0.537	0.80	0.452	0.484	0.514
20.10.30.30	0.75	15	0.697	0.716	0.739	0.80	0.704	0.723	0.752	0.60	20	0.704	**0.724**	0.753	0.80	0.667	0.692	0.726
20.10.30.35	0.65	10	0.760	0.778	0.805	0.80	0.765	0.788	0.819	0.85	25	0.766	**0.791**	0.820	0.80	0.745	0.769	0.795
20.10.50.20	0.75	15	0.662	0.682	0.707	0.80	0.673	0.689	0.710	0.75	20	0.674	**0.691**	0.711	0.80	0.629	0.655	0.686
20.10.50.50	0.75	10	0.935	0.957	0.971	0.80	0.945	0.962	0.973	0.60	15	0.947	**0.964**	0.975	0.80	0.921	0.946	0.964
20.10.50.65	0.75	10	0.999	1.000	1.000	0.80	1.000	**1.000**	1.000	0.50	20	1.000	**1.000**	1.000	0.80	0.999	1.000	1.000
50.10.10.60	0.75	25	0.815	0.818	0.826	0.80	0.812	0.816	0.823	0.65	20	0.815	0.818	0.826	0.80	0.777	0.780	0.788
50.10.10.70	0.65	15	0.890	0.894	0.897	0.80	0.887	0.892	0.895	0.75	10	0.890	**0.895**	0.898	0.80	0.861	0.866	0.868
50.10.30.30	0.85	20	0.551	0.556	0.563	0.80	0.548	0.552	0.559	0.85	20	0.551	0.556	0.564	0.80	0.483	0.494	0.505
50.10.30.35	0.75	25	0.606	0.613	0.620	0.80	0.603	0.609	0.616	0.50	15	0.605	0.613	0.619	0.80	0.544	0.554	0.565
50.10.50.20	0.75	25	0.471	0.483	0.489	0.80	0.466	0.478	0.483	0.85	25	0.471	0.483	0.489	0.80	0.381	0.402	0.415
100.15.10.40	0.75	45	0.576	0.577	0.578	0.80	0.577	0.579	0.580	0.85	20	0.581	**0.582**	0.583	–	–	–	–
100.15.10.45	0.85	25	0.627	0.629	0.632	0.80	0.628	0.630	0.633	0.50	15	0.631	**0.632**	0.635	–	–	–	–
100.15.10.50	0.75	25	0.676	0.677	0.678	0.80	0.677	0.679	0.680	0.85	20	0.680	**0.681**	0.683	–	–	–	–
100.15.30.15	0.85	45	0.310	0.312	0.314	0.80	0.311	0.314	0.317	0.75	5	0.314	**0.316**	0.319	–	–	–	–
100.15.30.20	0.85	45	0.374	0.377	0.379	0.80	0.376	0.379	0.381	0.85	5	0.379	**0.381**	0.384	–	–	–	–

Figure 11: Experiments results - quality. Test-set of satisfiable instances, best mean absolute ratios for each test-set, as well as other interesting values described in the text are typed in the boldface.

Test set $n.d.p.q$	ABSOLUTE RATIOS																
	ILS-TSGH-RS					**DLS-WMCH-TP**				**DLS-WMCH-PP**				**DLS-WMCH-NP**			
	bp	ws	Min	Mean	Max	p	Min	Mean	Max	p	Min	Mean	Max	p	Min	Mean	Max
20.10.10.60	0.75	15	0.913	0.923	0.929	1	0.916	**0.927**	0.935	1	0.916	**0.927**	0.934	10	0.916	0.926	0.934
20.10.10.70	0.60	15	0.973	0.979	0.986	1	0.973	0.980	0.987	1	0.973	0.980	0.988	1	0.972	0.980	0.987
20.10.30.15	0.75	10	0.462	0.499	0.525	1	0.478	0.518	0.535	1	0.477	0.518	0.533	8	0.478	0.518	0.534
20.10.30.30	0.65	10	0.690	0.711	0.738	2	0.704	0.723	0.753	1	0.703	0.723	0.754	5	0.703	0.723	0.752
20.10.30.35	0.65	15	0.758	0.780	0.809	1	0.766	0.788	0.817	2	0.765	0.788	0.817	1	0.766	0.788	0.815
20.10.50.20	0.85	15	0.647	0.674	0.704	1	0.673	0.689	0.709	1	0.673	0.690	0.709	2	0.672	0.689	0.710
20.10.50.50	0.85	10	0.936	0.959	0.973	2	0.942	0.961	0.972	1	0.944	0.963	0.974	1	0.944	0.963	0.974
20.10.50.65	0.50	20	1.000	**1.000**	1.000	1	0.999	1.000	1.000	2	0.999	1.000	1.000	1	0.999	1.000	1.000
50.10.10.60	0.85	5	0.802	0.805	0.812	1	0.816	**0.819**	0.829	1	0.816	**0.819**	0.829	3	0.816	**0.819**	0.829
50.10.10.70	0.75	10	0.882	0.886	0.889	1	0.889	0.894	0.900	1	0.889	0.894	0.900	10	0.889	**0.895**	0.901
50.10.30.30	0.85	10	0.509	0.520	0.528	1	0.552	**0.559**	0.565	1	0.553	**0.559**	0.566	1	0.552	**0.559**	0.565
50.10.30.35	0.85	10	0.573	0.581	0.589	2	0.609	0.616	0.623	1	0.609	0.616	0.623	2	0.609	**0.617**	0.624
50.10.50.20	0.85	15	0.407	0.423	0.434	1	0.473	**0.486**	0.494	2	0.472	**0.486**	0.493	11	0.473	**0.486**	0.494
100.15.10.40	–	–	–	–	–	1	0.572	0.573	0.575	1	0.572	0.573	0.574	6	0.571	0.572	0.574
100.15.10.45	–	–	–	–	–	2	0.621	0.624	0.627	1	0.621	0.623	0.627	2	0.621	0.624	0.627
100.15.10.50	–	–	–	–	–	2	0.671	0.672	0.675	4	0.670	0.671	0.674	4	0.671	0.672	0.673
100.15.30.15	–	–	–	–	–	2	0.306	0.309	0.313	1	0.306	0.308	0.312	1	0.306	0.308	0.312
100.15.30.20	–	–	–	–	–	1	0.371	0.374	0.376	2	0.370	0.372	0.375	8	0.370	0.373	0.375

Figure 12: Experiments results - quality. Test-set of satisfiable instances, best mean absolute ratios for each test-set, as well as other interesting values described in the text are typed in the boldface.

Test set	**MCH** 10000 iterations		**WMCH** 10000 iterations			**TSGH** 10000 iterations			**ILS-MCH-RP**			
$n.d.p.q$	Moves	Mov./Iter.	wp	Moves	Mov./Iter.	tl	Moves	Mov./Iter.	bp	Iterations	Moves	Mov./Iter.
20.10.10.60	31.012	0.003	0.05	2002.588	0.200	40	3327.842	0.333	0.80	10500.910	3736.082	0.356
20.10.10.70	40.507	0.004	0.05	2124.478	0.212	40	3462.229	0.346	0.80	10482.210	3797.591	0.362
20.10.30.15	37.504	0.004	0.05	1814.863	0.181	80	3056.575	0.306	0.80	10266.240	3574.039	0.348
20.10.30.30	32.393	0.003	0.05	1856.148	0.186	60	3131.316	0.313	0.80	10346.890	3622.315	0.350
20.10.30.35	34.746	0.003	0.05	1882.541	0.188	70	3170.242	0.317	0.80	10264.220	3594.565	0.350
20.10.50.20	90.859	0.009	0.05	1829.868	0.183	90	3056.093	0.306	0.80	9946.699	3490.420	0.351
20.10.50.50	125.877	0.013	0.05	2051.390	0.205	50	3343.827	0.334	0.80	10080.215	3690.211	0.366
20.10.50.65	282.337	0.029	0.20	1986.828	0.564	50	1020.303	0.369	0.80	6075.607	2489.684	0.412
(iterations)	(9890.578)			(3534.394)			(2789.527)					
50.10.10.60	73.569	0.007	0.05	2004.178	0.200	70	3314.140	0.331	0.80	21843.530	5776.835	0.264
50.10.10.70	72.919	0.007	0.05	2041.939	0.204	70	3369.213	0.337	0.80	21619.520	5745.309	0.266
50.10.30.30	72.284	0.007	0.05	1931.659	0.193	110	3200.357	0.320	0.80	21717.978	5707.091	0.263
50.10.30.35	72.730	0.007	0.05	1948.643	0.195	110	3222.968	0.322	0.80	21737.690	5720.194	0.263
50.10.50.20	70.163	0.007	0.05	1892.688	0.189	150	3145.070	0.315	0.80	21137.940	5529.541	0.262
100.15.10.40	166.805	0.017	0.05	2095.040	0.210	150	7199.704	0.720	0.80	26827.160	5611.462	0.209
100.15.10.45	166.054	0.017	0.05	2098.858	0.210	150	7152.123	0.715	0.80	26737.030	5606.602	0.210
100.15.10.50	167.309	0.017	0.05	2112.034	0.211	140	6711.086	0.671	0.80	26716.140	5606.627	0.210
100.15.30.15	161.146	0.016	0.05	2045.651	0.205	210	8966.861	0.897	0.80	26397.410	5515.595	0.209
100.15.30.20	163.968	0.016	0.05	2058.576	0.206	230	8938.752	0.894	0.80	26465.470	5532.712	0.209

Figure 13: Experiments results - iterations vs moves. Test-set of satisfiable instances is typed in the boldface.

Test set	ILS-MCH-RS					ILS-WMCH-RP				ILS-WMCH-RS				
$n.d.p.q$	bp	ws	Iterations	Moves	Mov./Iter.	bp	Iterations	Moves	Mov./Iter.	bp	ws	Iterations	Moves	Mov./Iter.
20.10.10.60	0.65	10	8519.747	2479.427	0.291	0.80	24474.300	6813.840	0.278	0.65	15	23228.940	6153.309	0.265
20.10.10.70	0.85	10	8689.798	2655.299	0.306	0.80	23538.070	6815.867	0.290	0.85	15	23176.110	6625.219	0.286
20.10.30.15	0.65	20	9497.180	3074.493	0.324	0.80	22358.700	5966.305	0.267	0.60	25	21676.720	5624.431	0.259
20.10.30.30	0.75	15	9087.866	2824.497	0.311	0.80	23255.710	6272.649	0.270	0.60	20	22065.290	5732.281	0.260
20.10.30.35	0.65	10	8131.222	2279.907	0.280	0.80	23116.390	6251.404	0.270	0.85	25	22541.890	5968.876	0.265
20.10.50.20	0.75	15	8712.162	2697.211	0.310	0.80	21262.700	5713.638	0.269	0.75	20	20245.320	5233.971	0.259
20.10.50.50	0.75	10	8188.884	2506.358	0.306	0.80	20826.240	6047.902	0.290	0.60	15	19357.890	5388.561	0.278
20.10.50.65	0.75	10	3340.174	1233.088	0.372	0.80	4873.338	2740.170	0.565	0.50	20	4852.199	2731.250	0.567
50.10.10.60	0.75	25	17799.800	3790.929	0.213	0.80	17889.530	4596.523	0.257	0.65	20	15196.170	3549.633	0.234
50.10.10.70	0.65	15	14967.730	2683.466	0.179	0.80	18272.400	4723.228	0.258	0.75	10	14206.110	3146.239	0.221
50.10.30.30	0.85	20	15967.467	3071.280	0.192	0.80	16771.344	4264.706	0.254	0.85	20	13798.833	3139.457	0.228
50.10.30.35	0.75	25	17366.420	3625.284	0.209	0.80	16914.170	4309.789	0.255	0.50	15	13597.030	3026.138	0.223
50.10.50.20	0.75	25	16707.820	3446.534	0.206	0.80	15899.810	4006.744	0.252	0.85	25	13491.490	3105.505	0.230
100.15.10.40	0.75	45	21183.890	3476.732	0.164	0.80	90797.050	19191.170	0.211	0.85	20	85991.780	16804.640	0.195
100.15.10.45	0.85	25	16898.360	2228.942	0.132	0.80	90792.670	19254.630	0.212	0.50	15	86379.910	16917.570	0.196
100.15.10.50	0.75	25	17022.930	2259.983	0.133	0.80	90760.970	19329.470	0.213	0.85	20	86655.530	17085.430	0.197
100.15.30.15	0.85	45	20270.250	3288.399	0.162	0.80	88965.090	18386.200	0.207	0.75	5	82022.420	15409.200	0.188
100.15.30.20	0.85	45	20514.270	3335.778	0.163	0.80	88789.130	18482.140	0.208	0.85	5	82763.260	15631.930	0.189

Figure 14: Experiments results - iterations vs moves. Test-set of satisfiable instances is typed in the boldface.

Test set	ILS-TSGH-RP				ILS-TSGH-RS				
$n.d.p.q$	bp	Iterations	Moves	Mov./Iter.	bp	ws	Iterations	Moves	Mov./Iter.
20.10.10.60	0.80	10018.690	7069.177	0.706	0.75	15	10022.610	6565.270	0.655
20.10.10.70	0.80	10098.820	7090.055	0.702	0.60	15	10135.730	6659.285	0.657
20.10.30.15	0.80	18100.060	15348.690	0.848	0.75	10	18135.640	14525.540	0.801
20.10.30.30	0.80	14084.610	11271.270	0.800	0.65	10	14111.790	10359.339	0.734
20.10.30.35	0.80	16085.780	13147.790	0.817	0.65	15	16100.680	12626.290	0.784
20.10.50.20	0.80	20099.920	17433.230	0.867	0.85	15	20124.950	16923.780	0.841
20.10.50.50	0.80	12244.120	9496.585	0.776	0.85	10	12355.240	8595.807	0.696
20.10.50.65	0.80	6274.724	4815.019	0.769	0.50	20	4357.702	3261.023	0.752
50.10.10.60	0.80	9496.920	6133.632	0.646	0.85	5	9488.541	3552.524	0.374
50.10.10.70	0.80	9493.919	6085.964	0.641	0.75	10	9496.459	3970.339	0.418
50.10.30.30	0.80	13529.967	10276.489	0.760	0.85	10	13557.611	8508.241	0.628
50.10.30.35	0.80	13525.970	10236.200	0.757	0.85	10	13552.330	8396.491	0.620
50.10.50.20	0.80	17546.390	14390.440	0.820	0.85	15	17573.380	13195.000	0.751
100.15.10.40	–	–	–	–	–	–	–	–	–
100.15.10.45	–	–	–	–	–	–	–	–	–
100.15.10.50	–	–	–	–	–	–	–	–	–
100.15.30.15	–	–	–	–	–	–	–	–	–
100.15.30.20	–	–	–	–	–	–	–	–	–

Figure 15: Experiments results - iterations vs moves. Test-set of satisfiable instances is typed in the boldface.

Test set	DLS-WMCH-TP				DLS-WMCH-PP				DLS-WMCH-NP			
$n.d.p.q$	p	Iterations	Moves	Mov./Iter.	p	Iterations	Moves	Mov./Iter.	p	Iterations	Moves	Mov./Iter.
20.10.10.60	1	16284.180	5365.587	0.329	1	16229.420	5347.568	0.329	10	15849.140	5205.607	0.328
20.10.10.70	1	15448.580	5307.075	0.344	1	15290.590	5245.526	0.343	1	15101.690	5175.647	0.343
20.10.30.15	1	12865.710	3895.664	0.303	1	12973.300	3929.676	0.303	8	12503.280	3747.024	0.300
20.10.30.30	2	14351.500	4484.110	0.312	1	13706.910	4251.147	0.310	5	13466.320	4139.549	0.307
20.10.30.35	1	14008.790	4391.325	0.313	2	14465.420	4557.994	0.315	1	13754.320	4271.540	0.311
20.10.50.20	1	12370.660	3733.519	0.302	1	12354.640	3730.672	0.302	2	11616.750	3484.532	0.300
20.10.50.50	2	12457.150	4107.728	0.330	1	11961.950	3945.192	0.330	1	11728.390	3872.339	0.330
20.10.50.65	1	2281.343	813.814	0.358	2	2169.025	787.192	0.365	1	2332.264	839.016	0.361
50.10.10.60	1	56595.100	18501.730	0.327	1	56437.520	18446.440	0.327	10	55606.820	18127.290	0.326
50.10.10.70	1	57839.380	19235.690	0.333	1	58290.440	19375.380	0.332	10	57738.160	19154.340	0.332
50.10.30.30	1	50434.333	15924.578	0.316	1	50629.222	15986.444	0.316	1	50196.911	15793.878	0.315
50.10.30.35	2	52078.330	16540.140	0.318	1	51311.460	16268.890	0.317	2	50471.820	15940.550	0.316
50.10.50.20	1	47974.470	14872.230	0.310	2	47774.240	14847.150	0.311	11	46818.600	14452.090	0.309
100.15.10.40	1	99910.310	33046.850	0.331	1	75660.070	25048.590	0.331	6	75331.040	24911.530	0.331
100.15.10.45	2	99924.050	33197.650	0.332	1	75925.270	25228.120	0.332	2	76591.400	25398.250	0.332
100.15.10.50	2	99948.060	33401.000	0.334	4	79503.610	26656.270	0.335	4	76847.710	25623.720	0.333
100.15.30.15	2	99953.440	32182.840	0.322	1	72040.220	23205.970	0.322	1	71302.940	22913.690	0.321
100.15.30.20	1	99929.760	32345.130	0.324	2	74337.230	24138.280	0.325	8	72512.970	23462.880	0.324

Figure 16: Experiments results - iterations vs moves. Test-set of satisfiable instances is typed in the boldface.

Test set	TIME IN CPU SECONDS TO PERFORM 100 000 ITERATIONS/MOVES														
	MCH		**WMCH**			**TSGH**			**ILS-MCH-RP**			**ILS-MCH-RS**			
$n.d.p.q$	Iterations	Moves	wp	Iterations	Moves	tl	Iterations	Moves	bp	Iterations	Moves	bp	ws	Iterations	Moves
20.10.10.60	0.440	145.144	0.05	0.445	2.221	40	0.448	1.345	0.8	0.537	1.510	0.65	10	0.522	1.794
20.10.10.70	0.406	102.912	0.05	0.411	1.935	40	0.415	1.200	0.8	0.492	1.357	0.85	10	0.478	1.564
20.10.30.15	0.351	94.561	0.05	0.360	1.986	80	0.362	1.183	0.8	0.419	1.204	0.65	20	0.422	1.303
20.10.30.30	0.410	129.019	0.05	0.413	2.224	60	0.414	1.323	0.8	0.491	1.402	0.75	15	0.489	1.573
20.10.30.35	0.419	122.847	0.05	0.421	2.234	70	0.424	1.338	0.8	0.501	1.430	0.65	10	0.494	1.762
20.10.50.20	0.312	38.425	0.05	0.322	1.759	90	0.325	1.063	0.8	0.379	1.080	0.75	15	0.381	1.230
20.10.50.50	0.342	28.298	0.05	0.349	1.702	50	0.352	1.054	0.8	0.419	1.146	0.75	10	0.414	1.352
20.10.50.65	0.306	11.213	0.20	0.325	0.576	50	0.328	0.890	0.8	0.406	0.988	0.75	10	0.399	1.074
50.10.10.60	2.188	297.982	0.05	2.034	10.147	70	2.016	6.083	0.8	2.636	9.967	0.75	25	2.173	10.204
50.10.10.70	1.951	267.750	0.05	1.931	9.456	70	1.921	5.702	0.8	2.508	9.436	0.65	15	2.047	11.416
50.10.30.30	1.646	228.229	0.05	1.634	8.459	110	1.617	5.054	0.8	2.021	7.691	0.85	20	1.735	9.022
50.10.30.35	1.670	229.698	0.05	1.659	8.512	110	1.647	5.112	0.8	2.060	7.830	0.75	25	1.773	8.492
50.10.50.20	1.024	146.214	0.05	1.042	5.505	150	1.046	3.327	0.8	1.203	4.598	0.75	25	1.099	5.329
100.15.10.40	12.721	762.867	0.05	11.480	54.796	150	> 1000	> 1300	0.8	14.926	71.360	0.75	45	11.903	72.526
100.15.10.45	12.819	772.115	0.05	11.385	54.243	150	> 1130	> 1500	0.8	14.968	71.378	0.85	25	12.013	91.073
100.15.10.50	12.995	776.800	0.05	11.215	53.101	140	> 1200	> 1700	0.8	14.911	71.054	0.75	25	11.889	89.550
100.15.30.15	10.208	633.510	0.05	8.880	43.409	210	> 1130	> 1200	0.8	11.783	56.391	0.85	45	9.374	57.786
100.15.30.20	10.485	639.463	0.05	9.101	44.212	230	> 900	> 1000	0.8	11.961	57.215	0.85	45	9.605	59.071

Figure 17: Experiments results - time. Test-set of satisfiable instances is typed in the boldface.

Test set	TIME IN CPU SECONDS TO PERFORM 100 000 ITERATIONS/MOVES													
	ILS-WMCH-RP			**ILS-WMCH-RS**				**ILS-TSGH-RP**			**ILS-TSGH-RS**			
$n.d.p.q$	bp	Iterations	Moves	bp	ws	Iterations	Moves	bp	Iterations	Moves	bp	ws	Iterations	Moves
20.10.10.60	0.80	0.463	1.664	0.65	15	0.468	1.766	0.80	7.244	10.265	0.75	15	6.838	10.439
20.10.10.70	0.80	0.427	1.475	0.85	15	0.436	1.526	0.80	6.911	9.843	0.60	15	6.333	9.639
20.10.30.15	0.80	0.371	1.392	0.60	25	0.384	1.480	0.80	5.570	6.570	0.75	10	5.519	6.893
20.10.30.30	0.80	0.433	1.605	0.60	20	0.436	1.677	0.80	6.594	8.241	0.65	10	6.556	8.934
20.10.30.35	0.80	0.435	1.610	0.85	25	0.443	1.674	0.80	6.816	8.339	0.65	15	6.776	8.642
20.10.50.20	0.80	0.334	1.242	0.75	20	0.339	1.313	0.80	4.977	5.738	0.85	15	5.065	6.024
20.10.50.50	0.80	0.365	1.257	0.60	15	0.370	1.330	0.80	5.841	7.532	0.85	10	5.824	8.373
20.10.50.65	0.80	0.325	0.574	0.50	20	0.329	0.580	0.80	5.623	7.315	0.50	20	5.584	7.432
50.10.10.60	0.80	2.020	7.861	0.65	20	2.120	9.075	0.80	119.823	185.537	0.85	10	146.235	390.631
50.10.10.70	0.80	1.910	7.391	0.75	10	2.009	9.071	0.80	116.636	181.952	0.75	10	147.852	353.726
50.10.30.30	0.80	1.617	6.360	0.85	20	1.724	7.577	0.80	100.981	132.959	0.85	10	123.917	197.590
50.10.30.35	0.80	1.636	6.420	0.50	15	1.751	7.869	0.80	102.225	135.090	0.85	10	124.065	200.349
50.10.50.20	0.80	1.022	4.057	0.85	25	1.140	4.951	0.80	64.856	79.084	0.85	15	69.530	92.601
100.15.10.40	0.80	13.467	63.712	0.85	20	11.287	57.760	–	–	–	–	–	–	–
100.15.10.45	0.80	13.386	63.122	0.50	15	11.339	57.895	–	–	–	–	–	–	–
100.15.10.50	0.80	13.466	63.230	0.85	20	11.321	57.422	–	–	–	–	–	–	–
100.15.30.15	0.80	9.310	45.046	0.75	5	8.919	47.474	–	–	–	–	–	–	–
100.15.30.20	0.80	8.916	42.832	0.85	5	9.082	48.086	–	–	–	–	–	–	–

Figure 18: Experiments results - time. Test-set of satisfiable instances is typed in the boldface.

Test set $n.d.p.q$	TIME IN CPU SECONDS TO PERFORM 100 000 ITERATIONS/MOVES								
	DLS-WMCH-TP			**DLS-WMCH-PP**			**DLS-WMCH-NP**		
	p	Iterations	Moves	p	Iterations	Moves	p	Iterations	Moves
20.10.10.60	1	0.487	1.477	1	0.491	1.492	10	0.508	1.547
20.10.10.70	1	0.458	1.333	1	0.461	1.345	1	0.458	1.337
20.10.30.15	1	0.396	1.308	1	0.402	1.328	8	0.411	1.372
20.10.30.30	2	0.458	1.465	1	0.456	1.471	5	0.455	1.483
20.10.30.35	1	0.460	1.466	2	0.471	1.494	1	0.464	1.494
20.10.50.20	1	0.353	1.168	1	0.367	1.215	2	0.355	1.185
20.10.50.50	2	0.387	1.174	1	0.387	1.172	1	0.392	1.187
20.10.50.65	1	0.365	1.018	2	0.386	1.057	1	0.381	1.055
50.10.10.60	1	2.216	6.781	1	2.390	7.312	3	2.273	6.974
50.10.10.70	1	2.117	6.365	1	2.241	6.742	10	2.260	6.814
50.10.30.30	1	1.781	5.641	1	1.818	5.757	1	1.860	5.912
50.10.30.35	2	1.803	5.677	1	1.825	5.757	2	1.878	5.949
50.10.50.20	1	1.104	3.560	2	1.069	3.441	11	1.145	3.709
100.15.10.40	1	14.335	43.339	1	14.488	43.763	6	14.509	43.876
100.15.10.45	2	14.377	43.275	1	14.612	43.978	2	14.614	44.072
100.15.10.50	2	14.288	42.754	4	14.413	42.988	4	14.445	43.322
100.15.30.15	2	11.067	34.371	1	10.796	33.516	1	10.818	33.665
100.15.30.20	1	11.041	34.110	2	10.826	33.340	8	10.916	33.738

Figure 19: Experiments results - time. Test set of satisfiable instances is typed in the boldface.

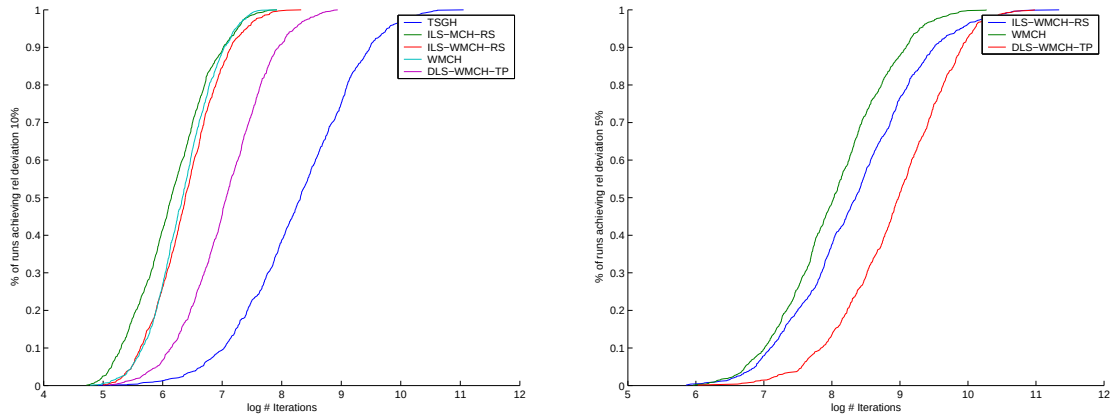

Figure 20: QRLD for instance $50.10.30.30.1$. Relative deviation of solution quality 10% (left) and 5% (right) from the best solution encountered during our experiments (semilog plot).

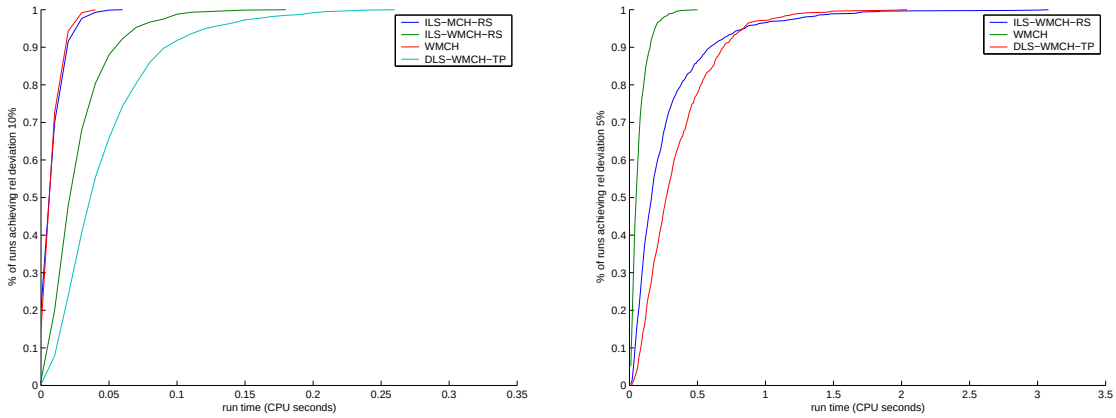

Figure 21: QRTD for instance $50.10.30.30.1$. Relative deviation of solution quality 10% (left) and 5% (right) from the best solution encountered during our experiments.

From Figure 13 we can conclude that TSGH stagnates less than the other two, which is also expected due to the use of the tabu tenure.

In terms of run-time all three algorithms achieve similar performance on instances with up to 50 variables and 10 domain size values (see Figure 17). On much larger instances, TSGH behaves considerably worse.

We may conclude that WMCH is the better algorithm in case not satisfiable instances.

ILS-MCH-RP and ILS-MCH-RS In terms of quality (Figures 10 and 11) ILS-MCH-RS always achieved better results.

ILS-MCH-RP performed more iterations than ILS-MCH-RS (Figures 13 and 14) and had higher

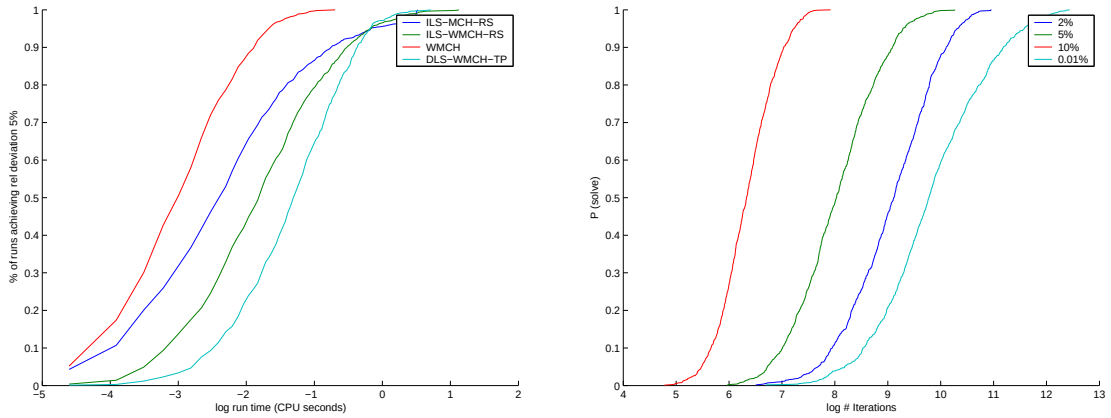

Figure 22: QRTD for instance $50.10.30.30.1$; relative deviation of solution quality 5% from the best solution encountered during our experiments (left). Set of QRLD for WMCH on instance $50.10.30.30.1$; relative deviation of solution quality of 10%, 5%, 2% and 0.01% from the best solution encountered during our experiments (right).

$moves/iterations$ ratio (which implies that it stagnated less). Nevertheless that did not help it obtain better solution quality.

ILS-MCH-RP was slightly slower than ILS-MCH-RS (Figure 17) in terms of run time.

To summarize, ILS-MCH with random steps is, as expected, a better algorithm than ILS-MCH with random picking.

Compared to WMCH, which is the best from the first group, the ILS-MCH-RS performs less good in terms of absolute ratios for instances with 20 and respectively 50 variables. For 100 variables ILS-MCH-RS shows better performance than WMCH. Nevertheless we have to consider the iterations performed by the ILS-MCH-RS in order to achieve this result: for 50 and 100 variables it did approximately up to twice as many iterations as WMCH (Figure 14). From Figure 17 we conclude that the required run time to perform a certain number of iterations is the approximately the same for both algorithms which implies that ILS-MCH-RS spent twice as much time to achieve the above mentioned quality.

ILS-WMCH-RP and ILS-WMCH-RS With regard to the absolute ratios, ILS-WMCH-RP behaves in case of 20 variables similar to the best so far analyzed algorithm, WMCH. For 50 variables ILS-WMCH-RP is worse than both WMCH and ILS-MCH-RS. For 100 variables ILS-WMCH-RP shows the best so far performance (Figure 11).

Based on the same criterion, ILS-WMCH-RS outperforms all other algorithms considered in this paper.

When considering the above mentioned results, we have to point out that for 20 and 50 variables both ILS-WMCH-RP and ILS-WMCH-RS performed about twice as many iterations as WMCH (and respectively twice as many and comparable as many as ILS-MCH-RS). For 100 variables

it performed nine times more iterations that WMCH (five times as many as ILS-MCH-RP) - see Figure 14. The run time performance (for the same number of carried iterations) is similar to that in case of the ILS algorithms based on MCH (Figure 18).

As in case of the ILS algorithms based on MCH as a subsidiary local search procedure, again ILS-WMCH-RS outperforms ILS-WMCH-RP in terms of the achieved solution quality.

ILS-TSGH-RP and ILS-TSGH-RS Both ILS-TSGH-RP and ILS-TSGH-RS showed very week performance in terms of quality (Figure 11) - for 50 variables it is worse than MCH and even its subsidiary local search procedure TSGH. For 20 and 50 variables the performed up to twice as many iterations as WMCH (results for 100 variables are not available).

In terms of run time ILS-TSGH-RP is from five-teen times (for 20 variables) up to fifty times slower (for 50 variables) than other ILS algorithms (for the same number of carried iterations) - see Figure 18. The same applies to ILS-TSGH-RS.

DLS-WMCH-TP, DLS-WMCH-PP and DLS-WMCH-NP Among the three DLS variants, no significant difference in terms of achieved absolute ratios can be observed. As different weightening procedures are used, we can conclude that that both conflicting and non-conflicting variables have an impact on the stagnation behavior.Consequently a more sophisticated penalization procedure needs to be developed.

Together with the ILS-WMCH-RS, the DLS variants achieved the best absolute ratios among all twelve algorithms. Even though the required iterations number and respectively CPU run-time are by far higher than compared to the WMCH. This conclusion is also clearly showed in the Figures 20, 21 and 22. For lower quality bounds WMCH reaches considerably faster the required solution quality.

7.2 Real World Instances

Preliminary tests on real world instances were quite disappointing. We have applied various algorithms and all of them had problems with satisfying hard constraints and did not minimize span.

8 Conclusions and Future Work

From our experimental work we can conclude that improving the performance of algorithms for the Max-CSP with respect to both solution quality and run-time is not an easy task. Even though our algorithms sometimes achieve better approximation ratios, the required time is higher than in case of the WMCH, which proved to be the best among the previously proposed methods.

Future work comprises both experimental as well as implementation aspects.

First it would be interesting to complete the picture of the already initiated tests. In order to make results more comparable to previous work, we could consider testing on the same randomly generated instances as used by Lau in his experiments. Furthermore tests could be carried on instances with non-uniform distribution, as those are closer to real-world related problems. Longer tests with the same total cut-off time for all algorithms and with subsequent comparison of the achieved solution quality, could also lead to more significant conclusions.

New implementation issues include the dynamical adjustment of the number of random walk steps while searching the state space. Similar the reactive stagnation could bring further improvement. With respect to our Dynamic Local Search approach, we could investigate further aspects of adjusting penalties. Unlike other DLS algorithms our current implementation does not penalize broken constraints but solution components. Constraint based penalization should also therefore be considered.

Acknowledgments. We gratefully acknowledge help from Thomas Stützle (early versions of the C code for testing framework) and Holger Hoos (tips on different issues), Roland Wenzel (who spent many hours helping us debugging the code and proofreading the draft version of this paper), James Slack (bash script tricks and linguistic help) and Mathias Hamel (who proofread the draft version of the paper).

The implementation of MCH, WMCH and TSGH algorithms was part of Diana's research assistantship for Professor Holger Hoos.

During debugging we used Kalev Kask's deterministic[4] Weighted Max-CSP solver [10].

References

[1] K.I. Aardal, C.A.J. Hurkens, J.K. Lenstra and S. Tiourine, *Algorithms for the radio link frequency assignment problem*, Technical Report UU-CS-1999-**36**, Universiteit Utrecht, Utrecht, Netherlands, November 1999.

[2] K.I. Aardal, S.P.M. van Hoesel, A.M.C.A. Koster, C. Mannino and A. Sassano, *Models and solution techniques for frequency assignment problems*, Technical Report ZIB-Report 01-**40**, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, Germany, December 2001.

[3] L.G. Anderson, *A simulation study of some dynamic channel assignment algorithm in a high capacity mobile telecommunication system*, IEEE Transactions on Communications, **COM-21**:1294-1301, 1973.

[4] P. van Beek, *A C library of routines for solving binary constraint satisfaction problems*, URL: http://ai.uwaterloo.ca/∼vanbeek/software/software.html.

[4] Beside the two deterministic algorithms, it has implemented a "stochastic local search" algorithm. There is no more details provided, source is not available and there is no sign of the publication describing an SLS method for Weighted Max-CSP by Kask or other members of his research group.

[5] A. Caminada, *CNET France Telecom frequency assignment benchmark*, URL: http://www.cs.cf.ac.uk/User/Steve.Hurley/f_bench.htm.

[6] M. Carter, G. Laporte and S. Lee, *Examination timetabling: Algorithms strategies and applications*, Journal of Operations Research Society, **74**:373-383, 1996.

[7] A. Eisenblätter and A.M.C.A. Koster, *FAP web - A web site devoted to frequency assignment*, URL: http://fap.zib.de, 2003.

[8] P. Galinier and J.K. Hao, *Tabu search for maximal constraint satisfaction problems*, in *Proceedings of the Third International Conference Principles and Practice of Constraint Programming (CP)*, Lecture Notes in Computer Science, **1330**:196-208, Springer Verlag, Berlin, 1997.

[9] H.H. Hoos and T. Stützle, Stochastic Local Search - Foundations and Applications, Morgan Kaufmann Publishers, *to appear 2003*.

[10] K. Kask, *CSP solver*, URL: http://www1.ics.uci.edu/∼kkask/csp.htm

[11] H.C. Lau, *A new approach for weighted constraint satisfaction*, Constraints, **7(2)**:151-165, 2002.

[12] Metaheuristics Network, *International Timetabling Competition*, URL: http://www.idsia.ch/Files/ttcomp2002/index.html, 2002.

[13] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird, *Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems*, Artificial Intelligence, **52**:161-205, 1992.

[14] D. Poole, A. Mackworth and R.Goebel, Computational Intelligence: a logical approach, Oxford University Press, New York, USA, 1998.

[15] French Society of Operations Research and Decision Analysis, *ROADEF Challenge 2001*, URL: http://www.prism.uvsq.fr/ ∼vdc/ ROADEF/ CHALLENGES/2001/, 2001.

[16] B.M. Smith, *Locating the Phase Transition in Binary Constraint Satisfaction Problems*, School of Computing Research Report **94.16**, University of Leeds, May 1994.

[17] C. Valenzuela, S. Hurley and D.H. Smith, *A permutation based genetic algorithm for minimum span frequency assignment*, Lecture Notes in Computer Science **1948**:907-916, 1998.

[18] R.J. Wallace, *Enhancements of branch and bound methods for the maximal constraint satisfaction problem*, in *Proceedings of the AAAI National Conference on Artificial Intelligence*, vol textbf1, 188-195, AAAI Press/The MIT Press, Menlo Park, CA USA, 1996.

[19] R.J. Wallace, *Analysis of heuristics methods for partial constraint satisfaction problems*, in E. Freuder (editor) *Principles and Practice of Constraint Programming - CP'96*, Lecture Notes in Computer Science **1118**, 482-496, Springer-Verlag, Berlin, Germany, 1996.

[20] H. Zhang, *Generating College Conference Basketball Schedules by a SAT Solver*, in *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*, 281-291, 2002.