Guided local search

Solving the graph coloring problem

Author: Hristo Georgiev, 2089138

Project for Algorithm Engineering, Karlsruhe Institute of Technology, Summer Semester 2017

Guided by: Demian Hespe, Yaroslav Akhremtsev, Peter Sanders

Date: September 2017

Keywords: Graph coloring, Chromatic Number, Guided local search

CONTENTS

PROBLEM STATEMENT

The problem, being solved is the vertex graph coloring (GCP). In it we are given an undirected graph G = (V, E) and a set of colors Γ. The finite set V is the set of vertices, while the set E ⊂ V × V is the set of edges.

A coloring is a mapping $\varphi: V \rightarrow \Gamma$ that assigns a unique color to each vertex. The set of colors is written as a set of natural numbers: $\Gamma = \{1 \dots k\}$ and, hence $|\Gamma| = k$. A conflict in a given coloring is a pair of connected vertexes (u, v) in the graph, such that φ(u) = φ(v). A coloring is said to be feasible (or legal) if there it contains no conflicts.

The decision version of the GCP, called the vertex k-coloring problem, consists in finding a feasible coloring using a defined number k of colors. It can formally be defined as
Input: An undirected graph G = (V, E) and a set of colors Γ with |Γ| = k ≤ |V |.
Question: Is there a k-coloring $\varphi: V \rightarrow \Gamma$ such that $\varphi(u) \neq \varphi(v)$ for all (u, v) ∈ E?

The chromatic number χ_G is a characteristic of the graph and corresponds to the smallest k such that a feasible k-coloring exists. The optimization version of GCP, also known as the chromatic number problem, consists in determining χ_G and can be formalised as
Input: An undirected graph G = (V, E) and a set of colours Γ with |Γ| = k ≤ |V |.
Question: Which is the smallest k such that a feasible k-coloring exists?

The chromatic number problem can be approached by solving a decreasing sequence of k-coloring problems until for some k a feasible coloring cannot be found. In this case, the best feasible coloring uses k + 1 colors and this is the chromatic number of the graph.

It is well known that the k-coloring problem for general graphs is *N P*-complete and that the chromatic number problem is *N P*-hard.

ITERATIVE IMPROVEMENT

Iterative improvement is technique that approaches a solution by progressive approximation, using the k^{th}approximate solution to find the $(k+1)^{th}$ approximate solution. Before applying this technique, four problem specific components have to be defined:

- Set of candidate solutions *S*. It is clear that for the graph k-coloring this set is the set of possible vectors with length |V| with numbers smaller than k.
- Initialization procedure, which selects from the set *S* a starting solution.
- Neighborhood structure. That is a mapping $N: S \rightarrow 2^S$. The neighborhood of a candidate solutions *s*, denoted as *N*(*s*), is the set of all candidate solutions that are neighbors of *s*. S and N define a graph, called neighborhood graph, where the elements of S are the vertices and the neighborhood relationship of N determine the edges between the vertices.
- Evaluation function $f: S \rightarrow R$, necessary to guide the search through *S*. The function serves to access candidate solutions in the neighborhood of the current solution and to gain the local information necessary to decide where to move. Commonly, but not necessarily, a global optimum for the evaluation function corresponds to an optimal solution of the problem.

A local optimum of an evaluation function *f* with respect to its neighborhood structure *N* is a candidate solution s ∈ S such that f(s) ≤ f(s'), ∀s' ∈ *N*(*s*).

In addition to these components, a further element to be devised is the search strategy. In the most general case, a search strategy is defined by the step function, that is, a pair (s, s') ∈ S × S of neighboring search positions (s' ∈ N (s)) such that the probability of the algorithm to go from s to s' is larger than zero. The execution of the step function defines a move. The most widely used search strategies are:

- Best improvement - one of the neighboring candidate solutions, that achieves a maximal improvement in the evaluation function is randomly selected. This requires an exhaustive exploration of the neighborhood at each step, that is, all neighbors are evaluated before selection.
- First improvement - instead, the first improving step encountered in the exploration of the neighborhood is selected. The exploration can be random or ordered. Search steps are often computed faster in first improvement but the improvement is typically smaller than with best improvement.

Pseudo-code for Iterative Improvement

```
Function Iterative_Improvement (problem instance I)
  Generate an initial solution s ∈ S
  while (s is not local optimum in N(s)) do
      select a solution s' from N(s) such that f(s') < f(s)
      s = s'
  end
  return a solution s that is local optimum in N(s)
```

INITIAL COLORINGS

For the purpose of the project the following strategies for creating an initial coloring of the input graph were used:

- **Random coloring** – Given a number of the coloring k, this method builds a coloring, by assigning random numbers in the range [0, k) to each vertex. The resulting coloring will contain conflicts with very high probability.
- **Bipartite coloring** – The algorithm tries to build a bipartite coloring of the graph by using only 2 colors. It looks like a modification of DFS, but instead of coloring the visited vertexes in only one color, it uses alternating two colors (for example 1 for even depths, 2 for odd depths). This algorithm is guaranteed to answer the question is the graph bipartite (does it have 2-coloring). At average, the resulting coloring contains less conflicts than the random coloring.
- **Greedy coloring** – This is an adaptation of simple greedy scheme [6]. It is guaranteed to find a legal coloring with at most (the biggest number of edges connected to a vertex) + 1 colors.

The initial coloring is then given to the solver for further improvements.

EVALUATION FUNCTION

The implementation uses as an evaluation function the number of edges, which connects conflicting vertices:

$$f(C) = |\{(u,v): (u,v) \in \mathrm{E} \cap \varphi(u) = \varphi(v)\}|$$

The goal is to find a solution C^*such that $f(C^*) = 0$. The naïve implementation of counting those conflicting edges is to go through all vertices of the graph and to check all their neighbors if they have the same color. It is clear that this solution has time complexity $O(|\mathrm{V}|^2)$. This is time consuming and therefore a faster strategy was searched. It was found in [2].

For this purpose dynamic programming was applied by using an array with size k * |V|. The initialization of this array has time complexity $O(|\mathrm{V}|^2)$ and can be expressed using the following pseudo-code:

Pseudo-code for conflicts initialization

```
Function Initalize_conflicts(G, φ)
  conflicts =0
  for each v in V do
    for each u connected to v do
      conflicts(v, φ(u)) = conflicts(v, φ(u)) + 1
    end
  end
  return conflicts
```

The evaluation function then can be expressed as the sum of *conflicts*(*v*, φ(*v*)) for every vertex v in G.

NEIGHBORHOOD

The neighborhood *N*(*C*) that was implemented in the project is called restricted one-exchange neighborhood. It can be defined as the set of colorings *C'* obtained from *C* by changing the color of exactly one vertex that is involved in a conflict.

The evaluation function of the coloring obtained after changing the color of vertex *v* from c_{old} to c_{new} can be expressed by:

$$f(C') = f(C) + conflicts(v, c_{new}) - \ conflicts(v, c_{old})$$

This gives us time complexity of *O*(1) for calculating the function of a solutions in the neighborhood.

After changing the color of a vertex, the conflicts structure has to be updated. This is done by applying the following pseudo-code:

Pseudo-code for conflicts update

c_{old} is the old color of the vertex v
c_{new} is the new color of the vertex v

for each u connected to v **do**
    conflicts(u, c_{old}) = conflicts(u, c_{old}) - 1
    conflicts(u, c_{new}) = conflicts(u, c_{new}) + 1
end

The time complexity of this procedure is amortized *O*(|V|) – this occurs very rare in the real graphs

SEQUENTIAL GRAPH K-COLORING

The search for graph coloring consists of the following steps:

1. Test the graph if it is bipartite. If yes – return the bipartite coloring.
2. Find an uppe bound of the chromatic number of G. Set k = this upper bound.
3. Use an iterative improvement to find a k-coloring of the graph.
    1.1. If such is found:
        - If k is equal to the lower chromatic number bound return the coloring
        - Store the coloring as result
        - Set k = k – 1
        - Reduce the colors in the coloring by one
    1.2. Else:
        - return the found coloring with minimal colors

In step 1.1 it is not clear how exactly to reduce the number of the colors of a given coloring with one. For this there are two general strategies:

- **Scratch -** create a random coloring with k-1 colors.
- **Merge** - select a color group and add it to another color group.

For the merging of groups the following strategies were implemented:

- **Random** - select a random color group.
- **Minimal** - select the color group with minimal count of vertices.
- **Maximal -** select the color group with maximal count of vertices.
- **Median -** select the color group which is the median of the count of vertices.

BOUNDS OF THE CHROMATIC NUMBER

For the upper bound of the chromatic number there exists many theoretical results how to evaluate it. The most classical result is the Brooks' theorem, which gives as an upper bound the maximal number of outgoing edges from a vertex in the graph plus one. There were found two interesting theorems in [3], which are referenced just as Theorem 2 and Theorem 3.

For the lower bound, there is the check if the graph is bipartite or not. For many of the benchmark graphs the exact chromatic number is known and for some there is only a lower bound. Therefore, so that the search can terminate earlier, the lower bound can be set in the configuration.

GUIDED LOCAL SEARCH

Guided local search (GLS) is a Stochastic local search method that modifies the evaluation function in order to escape from local optima. In this algorithm, GLS uses an augmented evaluation function g defined as

$$h(C) = f(C) + \lambda . \sum_{i=0}^{|E|} w_i . I_i(C)$$

, where *f*(*C*) is the usual evaluation function, λ is a parameter that determines the influence of the penalties on the augmented cost function, w_i is the penalty weight associated to edge *i*, and $I_i(C)$ an indicator function, which that takes the value *1* if the end points of edge *i* are in conflict in *C* and *0* otherwise. The penalties are initialized to *0* and are updated each time an iterative improvement algorithm reaches a local optimum of *h*. The modification of the penalty weights is done by first computing a utility u_i for each violated edge, $u_i = \frac{I_i(s)}{1+w_i}$, and then incrementing the penalties of all edges with maximal utility by one. The underlying local search is a best-improvement algorithm in the restricted 1-exchange neighborhood. Once a local optimum is reached, the search continues for a maximum number of *sw* plateau moves before the evaluation function *h* is updated.

Pseudo-code for Guided local search

G – graph
x – Initial coloring of G
N – neighborhood
λ – augemantation parameter

```
function Guided_Local_Search(G, x)
   for each edge e do
      w[e] = 0
   end
   s = x
   do
      h = augemented function of f
      x = Local_Search(G, x)
      calculate the indicators I
      calculate the utilities utils
      for each edge e do
         if e is argmax(utils) dp
            w[e] = w[e] + 1
         end
      end
   while (not termination condition)
   return s
```

Pseudo-code for Local search

```
function Local_Search(G, x)
  do
    y = solution in N(x) such that h(x) is minimized, breaking ties randomly
    Δh = h(y) – h(x)
    if Δh == 0 then
      sideways = sideways +1
    else
      sideways = 0
    end
    if Δh <= 0 then
      x = y
    end
    if f(x) < f(s) then
      s = x
    end
  while (Δh <= 0) and (sideways < sw)
  return x
```

A tricky part in the implementation is the passing of the moves. For this purpose a structure which contains the following fields is used:

- **node** – the color of which vertex is updated
- **to** – the new color
- **total** – augumented score of the move
- **delta_conflicts** – what is the difference of the conflicts by changing the color
- **delta_guidance** - what is the difference of the guidance by changing the color

This structure guarantees that the move can be applied correctly and identified, while keeping the heuristic to break the ties randomly.

This situation is similar to a database table, which has the following CREATE statement:

```
CREATE TABLE neighbors (
  conflicts BIGINT UNSIGNED,
  guidance BIGINT UNSIGNED,
  vertex BIGINT UNSIGNED,
  color BIGINT UNSIGNED
);
```

Then to select the vertex coloring with minimal cost is similar to the SQL query:

```
SELECT vertex, color, conflicts, guidance
FROM neighbors
WHERE conflict + guidance = (
   SELECT MIN(conflicts + guidance)
   FROM neighbors
);
```

The field guidance of the neighbors is calculated using the following pseudo-code:

Pseudo-code for neighbors guidance

v – the looked vertex
c – the new color
guidance – the guidance of *v*

```
result = guidance
for each neighbor u of v do
    if φ(u) == φ(v) then
        result = result - weights[(v, u)]
    else if φ(u) == c then
        result = result + weights[(v, u)]
    end
end
return result
```

IMPROVEMENTS

Keep the penalties

When using Merge strategy for the coloring between the epochs, it seems reasonable to keep the penalty weights, because GLS will start with guidance from the previous epoch. In this case, after merging the color sets, the new indicators are updated and the guidance score is calculated again.

Aspiration moves

An aspiration move is defined to be a move such that a new best found solution is generated by that move, and that move would not have otherwise been chosen by the local search using the augmented objective function.

A pseudo-code, which defines the selection of an aspiration move:

Pseudo-code for Aspiration move

```
function Aspiration_move(G, x)
    z = solution in N(x) such that g(x) is minimized, breaking ties randomly
    if (f(z) < f(s)) and ((h(z) – h(x)) > 0) then
       return z
    else
        return nil
    end
```

This function is called before the selection of y in Local Search and sets the result, if it is not nil, to x. Then the cycle is continued.

Fast Guided Local Search

This is a heuristic, which forbids the backward moves. During the solving, a 2-D array with size O(k * N) is kept. It is initialized with zeros. If the responding cell to a node with current color is one, then the move is skipped. When a move is being made the cell, which responds to the old color and the vertex, is marked with one and all neighbors' cells are marked with zeros.

This is considered as a speed-up heuristic, which does not allow backward moves to be looked.

Timeout

The execution of the program can be restricted in time, thus allowing better setting for an early stop criterion.

EXPERIMENTS

All the DIMACS challenge instances were tested with the GLS implementation and very promising results were obtained. For more details, please see the additional excel file.

For the further experiments we chose the benchmark graph cti. This is a graph with 16840 vertices and 48232 edges. The chromatic number of this graph is 3, but the estimation of the upper bound always gives 7.

Deciding the merge strategy

An execution of all discussed merging strategies was made. The random selection of the color groups was skipped, because it does not give guaranteed results for the execution and is not dependent only on the graph (depending on the seed it could give different results). The results are shown in the table below.

Initial coloring	Source	Destination	Iterations	Time	Improvements
Greedy	Minimal	Maximal	11754	4	1316
Greedy	Median	Median	22710	9	4483
Greedy	Maximal	Minimal	26214	14	1322
Greedy	Minimal	Minimal	34206	16	2161
Greedy	Minimal	Median	34944	18	2169
Greedy	Median	Minimal	34944	16	2169
Bipartite	Minimal	Maximal	38760	17	8894
Greedy	Maximal	Median	38815	20	4659
Bipartite	Median	Median	44354	24	8589
Bipartite	Minimal	Median	59002	29	7135
Bipartite	Median	Maximal	61530	35	8934

Top 11 rows of strategy selection for the GLS

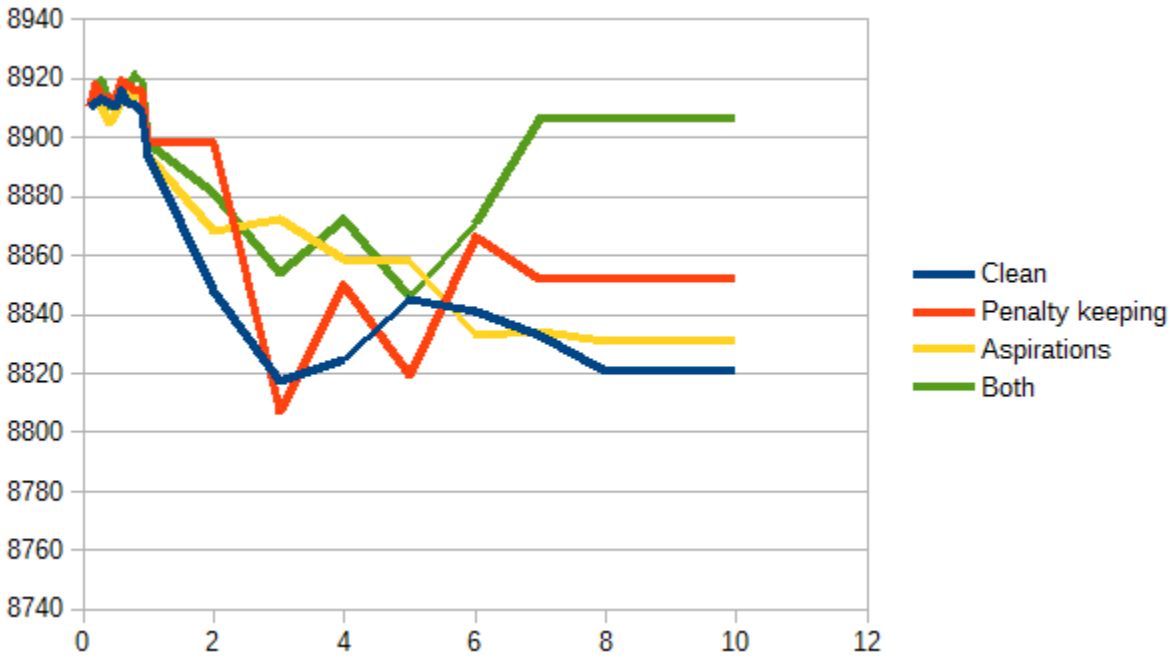
From the above shown table there are only *Greedy* and *Bipartite* initial colorings. Also the best results in the both cases are given by *Minimal* source and *Maximal* destination.

Selection of the λ parameter (1)

To estimate a good value of the λ parameter, GLS was executed with Bipartite initial coloring of the cti graph, using Minimal source and Maximal destination for the values of λ ={0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 3, ,4, 5, 6, 7, 8, 9, 10}. The reason why the Bipartite initial coloring was chosen, is because more initial conflicts were wanted. It also responds to pseudo-randomized coloring with two colors. The following execution characteristics were then recorded:

- improvements made
- number of initial conflicting edges
- execution time
- number of weight updates
- number of met local minimums
- initial guidance
- number of aspirations moves

Required improvements



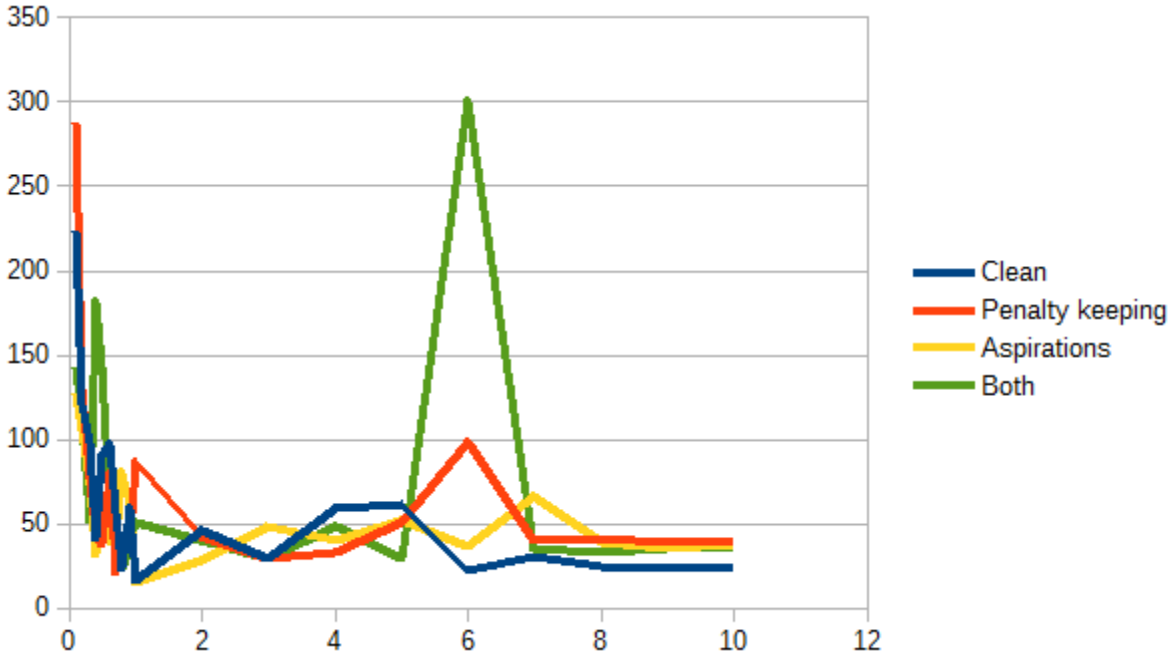
How the number of the done improvements depends on λ

The number of done improvements does not change much – it has maximal value of 8920 and minimal value of 8810. For small values of the λ parameter, the required number of improvements is almost equal for the different heuristics. It is interesting that combination of aspiration moves and penalty keeping gives the biggest number of improvements. This means that they are working against each other.

Number of initial conflicting edges

It turns out that the sum of the number of initial conflicting edges does not depend on the λ parameter and the applied heuristics. It is more dependent on the initial coloring and the updating strategies.

Execution time



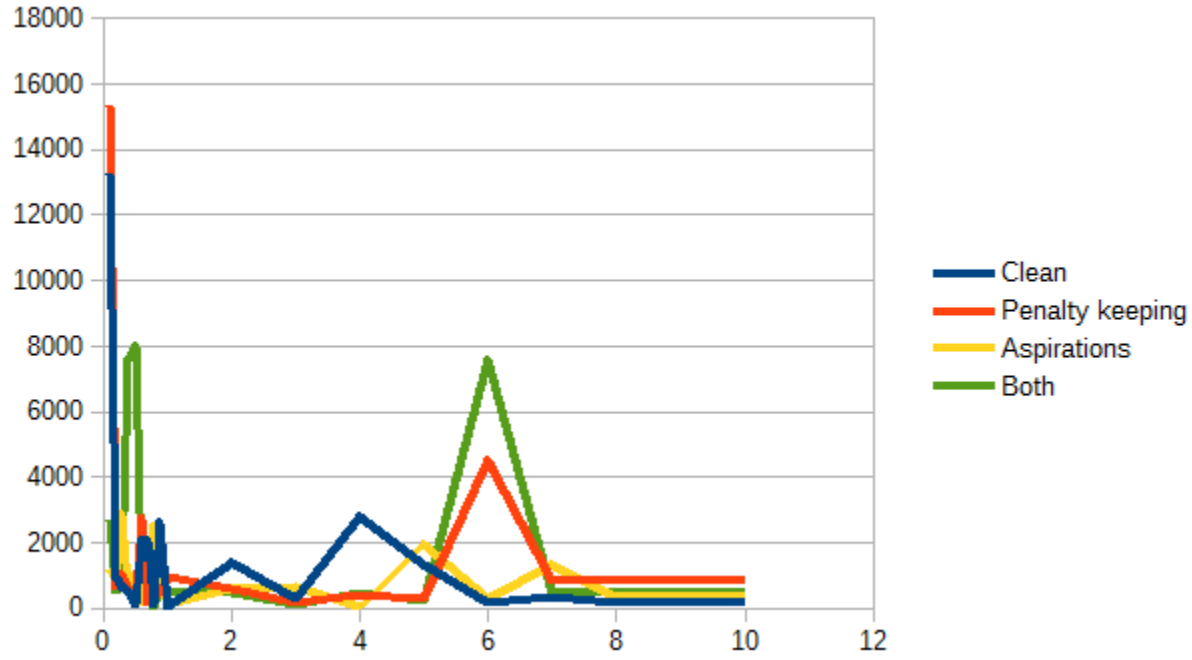
How the execution time depends on λ

For small values of the λ parameter the execution times are very similar for the combinations of heuristics. The combination of the both heuristics gave the worst execution time at the most of the cases. The penalty keeping is at third place. The clean GLS happens to be better for bigger values of the λ parameter, while the aspiration moves make it faster for smaller values.

Weight updates

It turns out that the number of the weights updates is proportional to the execution time. This means that the weight updates graphic looks like the graphic of the execution time but in different ratio. The constant for this experiment is 250.

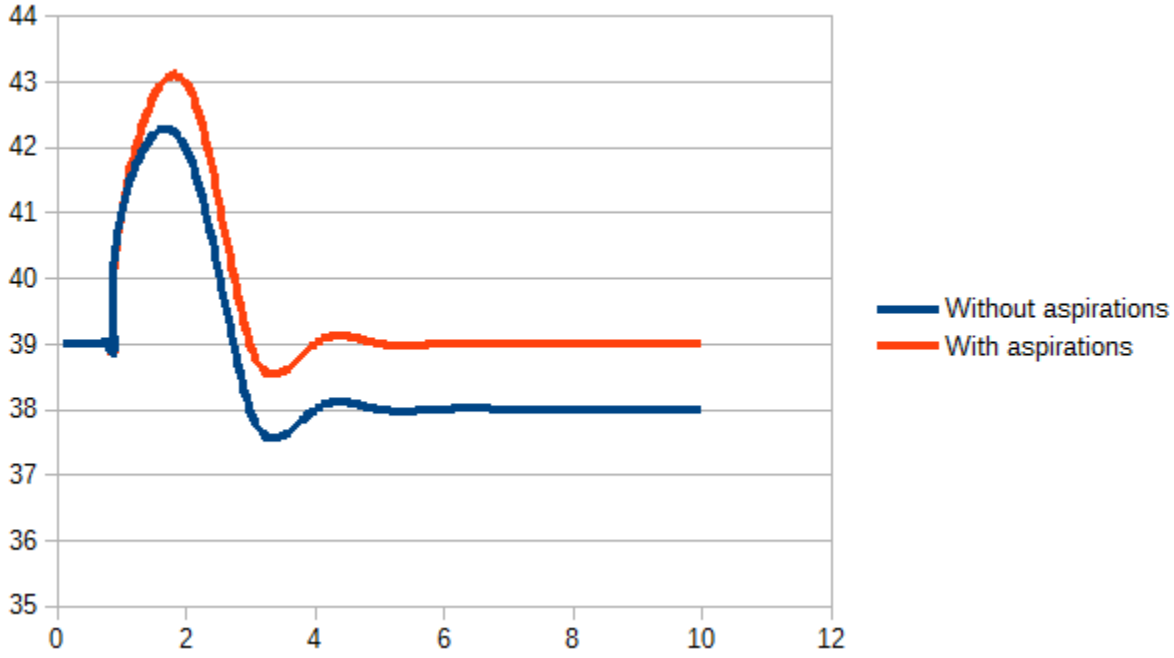
Local minimums



How the met local minimums depends on λ

For $\lambda < 1$ the combination of the two heuristics seems to give the best results in the most of the cases. For λ = [1,5] the aspiration moves make the best result for minimums. After that the clean GLS finds less minimums.

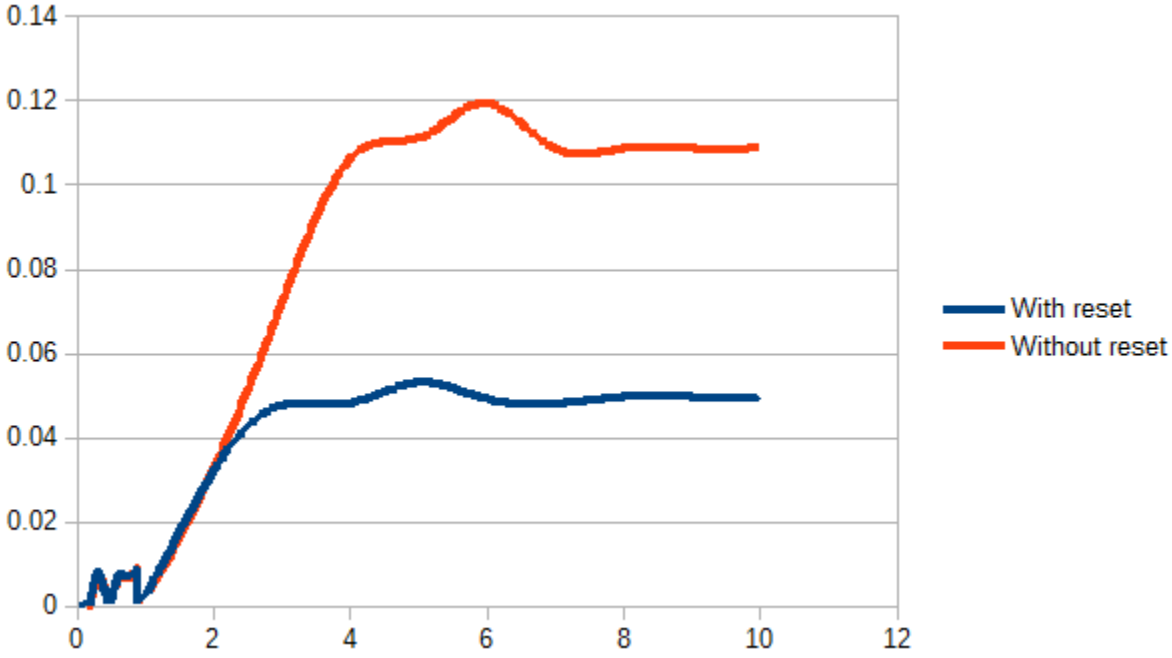
Initial guidance



How the initial guidance depends on λ

It turns out that the initial guidance depends very slightly on the λ parameter.
Also, the aspiration moves could possibly add some more initial guidance, but it is not of so big importance.

Aspirated moves



How the ratio between aspirations and improvements depends on λ

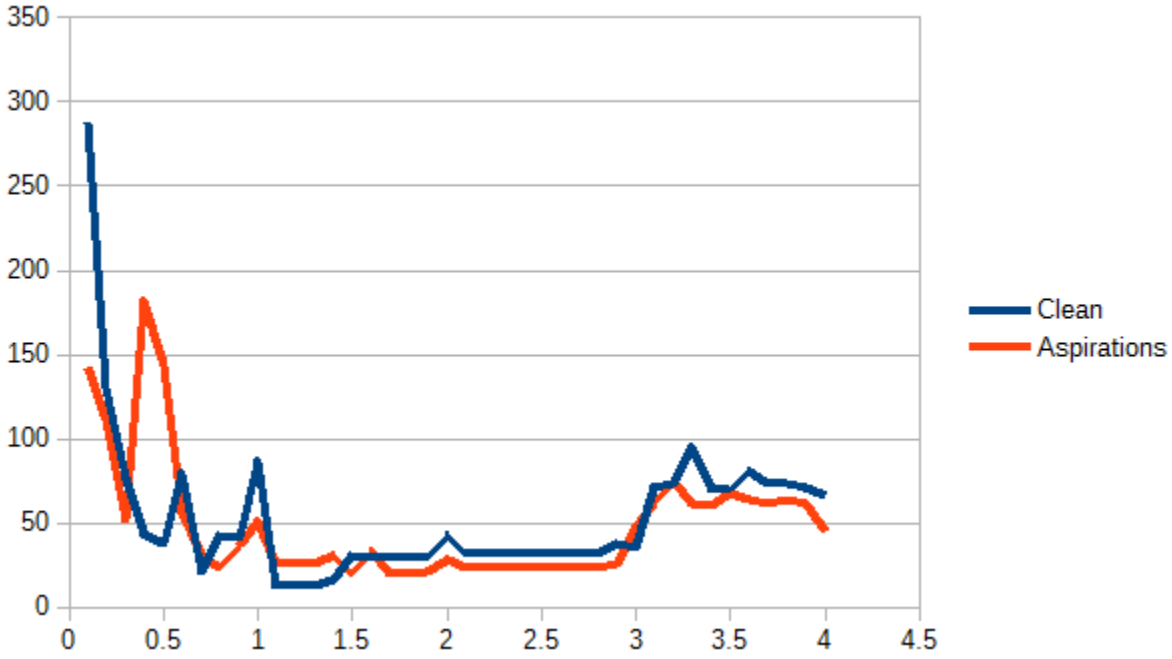
The aspirations are very rare, when the parameter λ is smaller than 1. After this point, the aspirated moves take a bigger part of all improvements. From the diagram it is easy to see that when the weights are not reset, more aspiration moves are being made. This is happens, because then the weights are forbidding moves, which did not have point in the previous epoch, but can have more meaning to be checked in the new epoch.

Conclusion from the second experiment

From the execution time section, it seems that λ in [1, 4] should be reasonable.
The combination of the two heuristics seems not to be such a good idea, after comparing the results from the experiment, because the big peak points.
Also the keeping of the penalties seems not to be such a good idea, because it has a bigger execution time. It is caused, because it adds much more movements without improvements, causing more weight updates to take place.
The aspiration moves seem to be a good heuristic, just because they add a second branch of moves, which the GLS can make, while slightly modifying the local search. It also meets less minimums in the range [1, 4]

Selection of the λ parameter (2)

The first experiment for selecting the λ parameter was repeated with the same setup for the values of the parameter λ in the range [1,4] with step of 0.1, but without keeping the penalties.



How the execution time depends on λ

It is not hard to see that in the given interval, the aspiration movements help for a faster finding of the solution. It was also observed that the number of times when a local minimum is reached is almost the same for the both heuristics in the interval [2,3]. Also the aspiration moves require less weight updates and make more, but smaller improvements of the solution, which actually helps the final solution to be found faster.

For the value of λ we can then choose 2.5 and to apply the aspiration moves heuristic.

Fast local search

To see how the fast search heuristic works for the restricted one exchange neighborhood, fast GLS was run with all possible epoch strategies. It turned out that for the implementation of the neighborhood this heuristic is actually an overhead and finds the solution much slower. So it is not suggested as a good improvement. But if for the neighborhood structure there was not available such good optimization, then this heuristic makes more sense (visiting the neighbors is an expensive operation), because it cancels some of the neighbors being visited.

FUTURE WORK

The guided local search can be extended using the following strategies:

- A third optional parameter will be added to the GLS, which will be a list with the size of the node set. The values in it will have the following meaning:
    - **ALLOWED** = 0, the vertex is free for coloring
    - **DO NOT CHANGE** = 1, the initial coloring of the vertex cannot be changed, but skipped.
    - **MARKED** = 2, set once the algorithm sets a color of the vertex. Then all of the neighbors of the changed vertex are marked as allowed. Once a vertex is set as MARKED, it is skipped.
- **Sudoku Solver –** A Sudoku could be thought as a 9-coloring of a graph, which nodes are the cells and the edges are the neighbors of the cells and the other cells in the bigger cell. The colors respond to the digits from 1 to 9.
- **Fast Local search –** with only one array for keeping the node status. The status MARKED responds to this heuristic. The idea is that backward moves are not allowed. This is considered as a speed-up heuristic.
- **Random moves** - the motivation behind random moves is to try to prevent GLS getting "stuck" in one part of the search space (for example, when λ is too low) and force it to move into other areas of the search space that it might not otherwise visit.

BIBLIOGRAPHY

1. M. Chiarandini, I. Dumitrescu, and T. Stutzle. “Stochastic Local Search Algorithms for the Graph Colouring Problem”, FG Intellektik, FB Informatik, TU Darmstadt, 2005. [https://pdfs.semanticscholar.org/7ccf/1713b645ccda4093981e2847d987e1e18f1f.pdf]
2. M. Chiarandini. “Stochastic Local Search Methods for Highly Constrained Combinatorial Optimization Problems”. PhD thesis, FG Intellektik, FB Informatik, TU Darmstadt, 2005. [http://tuprints.ulb.tu-darmstadt.de/595/1/ChiarandiniPhD.pdf]
3. M. Soto, A. Rossi, M. Sevaux. “Three new bounds on the chromatic number”. Universite de Bretagne-Sud. Centre de Recherche, Lorient Cedex, France, 2010. [http://www.sciencedirect.com/science/article/pii/S0166218X11003039]
4. Graph Coloring Benchmarks [https://sites.google.com/site/graphcoloring/vertex-coloring]
5. P. H. Mills , Extensions to Guided Local Search, PhD thesis, Department of Computer Science University of Essex 2002. [http://www.bracil.net/csp/papers/Mills-GLS-PhD2002.pdf]
6. Graph coloring | Set 2 (Greedy algorithm) [http://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/]