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

$\boldsymbol{c_{old}}$ is the old color of the vertex v
$\boldsymbol{c_{new}}$is the new color of the vertex v

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

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

For cache optimization reasons, the conflicts array is implemented as one dimensional array, while accessing it by k*v + c (k - epoch, v - vertex, c - color). This guarantees a linear structure, which can be scanned, once it is loaded in the memory. Also, the structure is allocated only once, before the start of the guided search. This single allocation removes unnecessary complex memory management.

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.

Note that an important condition is not to merge colorings, which have do not have conflicts with k – 1 colors.

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* (sideways) 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
```

The 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
```

Note that this implementation uses only the current improvement, without needing to calculate the weights and indicators every time, when guidance of the next move is needed.

GLS can be thought as a process using a SQL table with CREATE statement (note that the implementation uses similar structure):

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

The table is being populated by the local search, while visiting the neighbors. After that, the following query is executed to select the next move:

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

In case that the query returns more than one result, then all are collected and a random one is chosen from them. This means that ties are broken randomly. After that the table is truncated.

TERMINATION

The guided local search can run almost infinitely. For example, if it is working with a 5-partite graph, once it starts to color the graph with 4 colors, it will try to find one, although no can be found. Therefore the criteria when the algorithm to be stopped is very important. The following strategies are implemented in the solution:

- Maximal number of iterations – keep a global counter of the guided local search. That is how many times the local search was called. Once this counter goes above given limit, the algorithm terminates. It was found as an idea that this number can be set as ten times the count of nodes, but this was not further studied.
- Maximal execution time (timeout) – for what time the guided local search can be run. A clock is set once the search starts and is checked on every weight update or on every 100^{th} iteration. Once the time form the start is overlapped, the search terminates.
- For many of the benchmarks, there is already found a lower bound or exact value for the chromatic number. Therefore, it is implemented as a configurable property. Once during the epochs, the number k of used colors equals to the value of this property, the execution terminates.

The first and the second methods are suitable for exploring an unknown graph, while the second and the third methods are used for the experimental evaluation.

HEURISTICS

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.

Dynamic λ parameter

The choosing of this parameter is problem specific task, which includes the solving of the problem with many different values of this parameter with the same instance. Therefore the following heuristic was found in [7]. In the implementation it is calculated, before the first weights update, using:

$$\boldsymbol{\lambda}' = \frac{C_{initial} - \sum_{i=0}^{first\ update} \Delta C}{steps}$$

Where $C_{initial}$ is the initial number of conflicts, ΔC is the difference between the conflicts in current improvement and the next improvement, $steps$ is the number of iterations before the first weight update. Note that penalty keeping and dynamic λ cannot be applied together.

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.

EXPERIMENTAL SETUP

For the experimentation with the implementation and figuring out which should be the best parameters of the guided local search, the following characteristics were studied:

- Initialization strategy – greedy, bipartite, random
- Epoch strategy – merge colors and start from scratch
- Epoch merge source and target color group – minimal, maximal, median. Note that the random choosing of groups was not experimented, because it is not only graph dependent. It depends also and from the seed value.
- Heuristic comparison. The combinations of the implemented heuristics were done:
    - Clean – without any strategies
    - Penalty keeping
    - Aspiration moves
    - Dynamic λ
    - Penalty keeping and aspirations
    - Aspirations and dynamic λ

      Note that the combination of dynamic λ and penalty keeping was not done, because dynamic λ measures the performance until the first penalty update and the penalty keeping starts the first epoch with penalties.
- Different values of **λ** in the range [0.1, 10] with step 0.3
- Different values of the sideways in the range [1, 2, 4, 8, 16, 32]

The experiments were done with a range of a collection of 138 benchmark graphs with known chromatic numbers or at least known lower chromatic number bounds [4]. Most of the graphs are coming from the DIMACS graph coloring competition.

At the first step a process is executed to parse the benchmark graphs into the format known from the project and upper boundaries are calculated using the described techniques. After that a coloring is generated using the best known so far parameters. This process builds the main results table, shown in the results page.

The next steps are run on a selection of graphs with known chromatic numbers.

Epochs strategy

The second step is comparison of all possible epoch strategies. Guided local search is run on every selected benchmark graph with different values for initialization, epoch strategy, merge sources and destinations. For this experiment the values of $\lambda = 1$ (as suggested in [2]) and *sideways* = 2 (as suggested in [5]). The single experiment of a values combination terminates once the lower boundary is reached or timeout of 10 seconds exceeds. After obtaining the results for all the graphs, aggregations, which are discussed later, are being made.

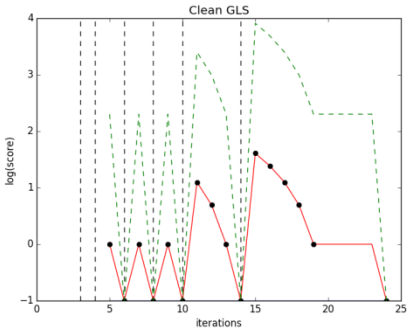
Initial coloring	Source	Destination	Iterations	Time	Improvements
Greedy	Minimal	Maximum	12570	12	1348
Greedy	Maximum	Median	26968	17	7972
Greedy	Median	Maximum	25034	18	7982
Greedy	Minimal	Median	30420	28	2222
Greedy	Median	Minimal	30420	28	2222
Greedy	Minimal	Minimal	34749	34	2198
Random	Minimal	Minimal	66009	39	40037
Bipartite	Median	Median	63265	40	27334
Bipartite	Minimal	Minimal	67062	51	24563
Bipartite	Maximum	Median	67032	51	29395
Bipartite	Minimal	Maximum	74382	52	29226

To*p 11 rows of strategy selection on cti graph with timeout 60 seconsds.*
The graph has 16840 vertices, 48232 edges and chromatic number 3

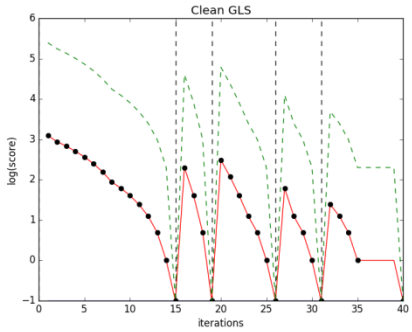
Initialization comparison

After the best epoch strategy is obtained, an experiment is being made to see graphically how the epochs look like for the different initialization strategies. (See the Legends section).

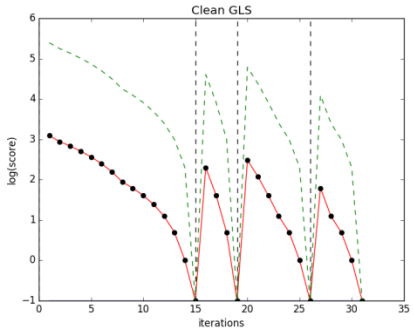
Here is shown the plot of the execution of clean GLS on the graph 3-FullIns_3 form the CAR source. These plots show a simple execution of the GLS without penalties. It was observed, that the pattern keeps for the other graphs.



Greedy



Bipartite



Random

From this graph the following conclusion can be done:

- The greedy strategy starts with no conflicts and it makes small steps to find the colorings in the next epochs.
- The bipartite strategy begins with a big number of initial conflicts and requires a longer first epoch to resolve those conflicts.
- The random strategy has shorter first epoch than the bipartite. For more complex graphs this is of big importance.

Performance with different heuristics

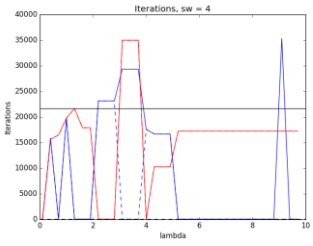
The next experiments have to find how the heuristics work. For this purpose the guided local search is run on the selected benchmark graphs and the following performance characteristics were tracked:

- Iterations made
- Resolved conflicts – sum of the initial conflicts in the colorings during the epochs
- Improvements made
- Resolve ratio – what part of the iteration were improving
- Minimums – how many times a local minimum was met – no movement was available without updating the weights
- Updates – how many times the weights were updated
- Aspirations – how many aspiration moves were done
- Aspiration ratio – what part of the improvements were aspiration moves
- Time – execution time of the GLS in sec
- Colors – how many colors were obtained in the final solution

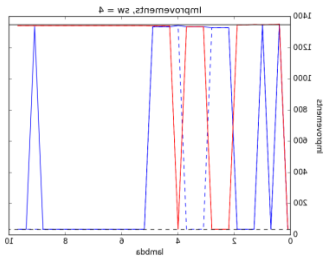
These experiments were executed with the following characteristics:

- sideways – in the range {1,2,4,8,16,32}
- λ – in the range [0.1;10.0] with step 0.3

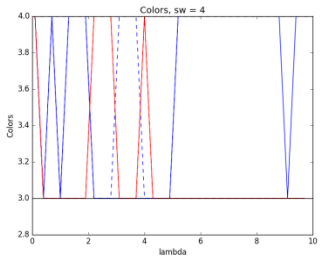
The experiment has two outputs – a .CSV file containing the performance characteristics and generated images, showing a plot of a single characteristic for a given values of the sideways and λ comparing the different heuristics



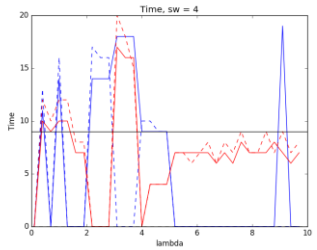
Iterations



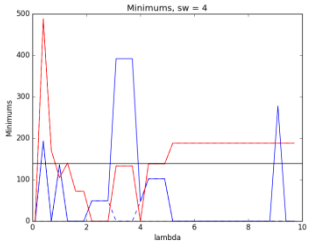
Improvements



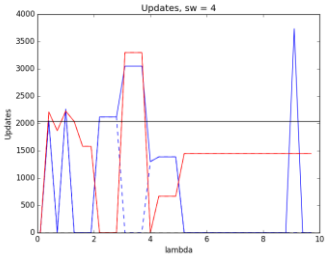
Found colors



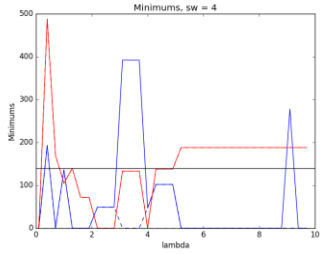
Time



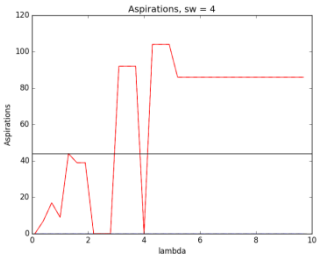
Minimums



Updates



Minimums



Aspirations

Example plots of the performance of the experiment with the cti graph.

RESULTS FROM THE EXPERIMENT

Epochs strategy

The results are definitely showing that starting from Scratch takes much more iteration to find the final coloring compared to iterative merge of the colors. Therefore it is important to select a good merge strategy.

To select the most optimal combination for a given group, first the results without conflicts are taken. Then the following aggregation fields are needed:

- Average normalized iterations – the iterations for one combination for one graph are divided by the maximum iterations found for this graph (normalization). After that the average for all normalized iterations is taken.
- Average normalized time – similar procedure for the time
- Average color difference – similar procedure for the difference between the colors in the found solution and the known lower boundary for the graph.
- Hits – how many times the combination could make an improvement of the initial coloring. That is it could manage to pass the first epoch in the timeout of 10 seconds. Note that if the timeout is set to a bigger value, this field will have a bigger value, because more execution time is given for the search.

After taking those average results for every source group, we can create one more aggregation, which will contain the normalized average values for the different strategies.

Note that we are looking for the combination, which has the smallest values for time, iterations and distance from the lower bound of the chromatic number, while finding at least one feasible coloring in the given timeout. Therefore a SQL query, which we want to execute on the resulting table looks like:

```
SELECT initialization, source, destination
FROM strategy_results
ORDER BY hits DESC,
        color_distnace ASC, time ASC, iterations ASC;
```

Initialization	Source	Destination	Hits	Iterations	Time	Colors
Greedy	Minimal	Minimal	1	0.327	0.494	0.192
Greedy	Maximum	Minimal	1	0.296	0.494	0.193
Greedy	**Minimal**	**Maximum**	**0.973**	**0.310**	**0.466**	**0.149**
Random	Maximum	Minimal	0.938	0.420	0.529	0.502
Random	Median	Maximum	0.938	0.411	0.542	0.547
Scratch	-	-	0.922	0.889	0.832	0.778
Bipartite	Maximum	Minimal	0.902	0.351	0.532	0.571
Bipartite	Minimal	Median	0.897	0.371	0.467	0.590

The best two combinations for each initialization strategy

It turns out that the best initialization strategy is the greedy. That is because it starts with initially legal coloring and the guided local search is oscillating around this initialization, trying to optimize it.

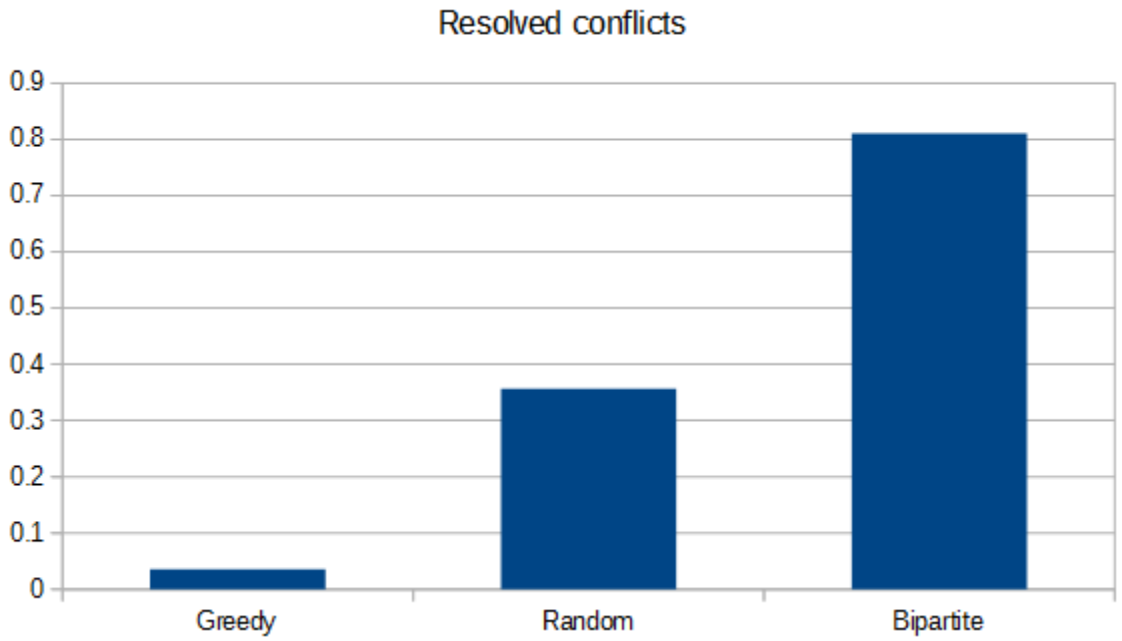
Another conclusion is that in general the random initialization is better than the bipartite initialization. The reason is, because the bipartite initialization needs more time to finish the first epoch and to start the next one, because it contains more initial conflicts than the random initialization.

The best strategy for the merges for the three initializations turns out to be Maximum source and Minimal destination, which even always has optimization of the initial coloring in the greedy initialization.

However the combination, which was able to give the best distance to the lower bound in this experiment, was greedy initialization with Minimal source and Maximum destination,

The results for the strategy experiments are not surprising, because they are related to the general Minimax strategy for problem solving, which was well studied in the theory of games [8].

For the initialization strategy it is important also how many conflicts it could resolve in total during the epochs. This corresponds to how much long path could the guided local search needed to go to find a solution of the graph. For this purpose similar calculation for the average normalized resolved conflicts was done.



The greedy initialization provides the lowest number of resolved conflicts, which means that the path in this situation is the shortest.

Tuning

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 (if visiting the neighbors is an expensive operation), because it cancels some of the neighbors being visited.

Performance experiments

The performance experiments were run on all selected benchmark graphs. The experiments run with different values of λ and *sideways*, running the following heuristics:

- Clean GLS
- Penalty keeping
- Aspiration moves
- Penalty keeping and aspiration moves
- Dynamic λ
- Dynamic λ and aspiration moves

The termination of a single experiment is when a coloring, which matches the known lower bound for the chromatic number of the graph, is met or a timeout of 4 seconds exceeds.

Selection of sideways

The first observation, which we want to make form those experiments, is to select a good value for the sideways. Graphs, which can be colored by the greedy algorithm with colors, which number equals to the lower bound of the chromatic number of the graph, are not of interest.

From the results for each graph are selected the number of times, when given sideways parameter was in the top 6 rows in the sorted results by the number of used colors, time and iteration. After that the count of those sideways are taken. To measure the performance of the GLS, we want at least one weight update to take place. An example SQL for this selection is:

```
SELECT sideways FROM graph_result
ORDER BY used_colors ASC, time ASC, iterations ASC
WHERE updates > 0 AND aspirations =0 AND keep_penalty = 0
LIMIT 6
```

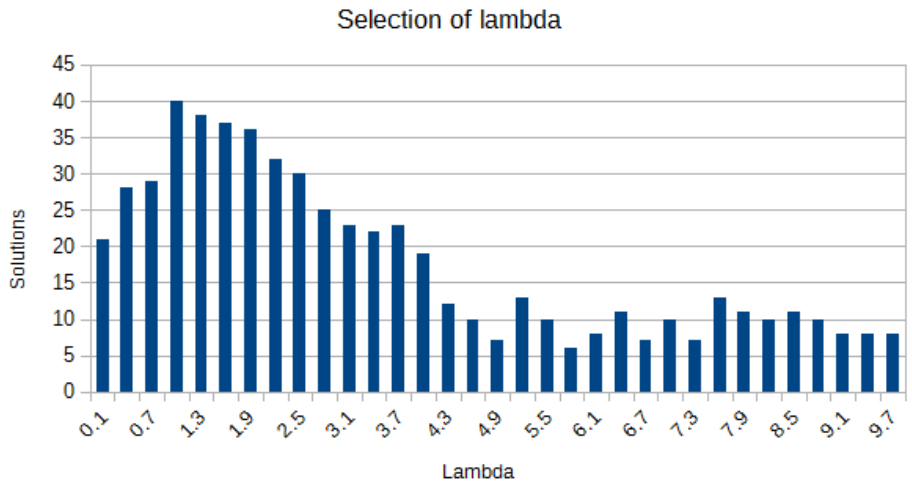
The results for the sideways can be expressed in the following bar-plot:

It turns out that the optimal values for the sideways parameter are 1 and 2. However we would like to make a little bit longer plateau moves. Therefore the value **2** is more recommended. This result responds with the recommendation from [5].

Selection of λ

For the estimation of the λ parameter a similar way as for the sideways can be done. The SQL for the estimation of the best λ parameter, related to the results shown below, is the following:

```
SELECT lambda FROM graph_result
ORDER BY used_colors ASC, time ASC, iterations ASC
WHERE updates > 0 AND aspirations =0 AND keep_penalty = 0
LIMIT 10
```



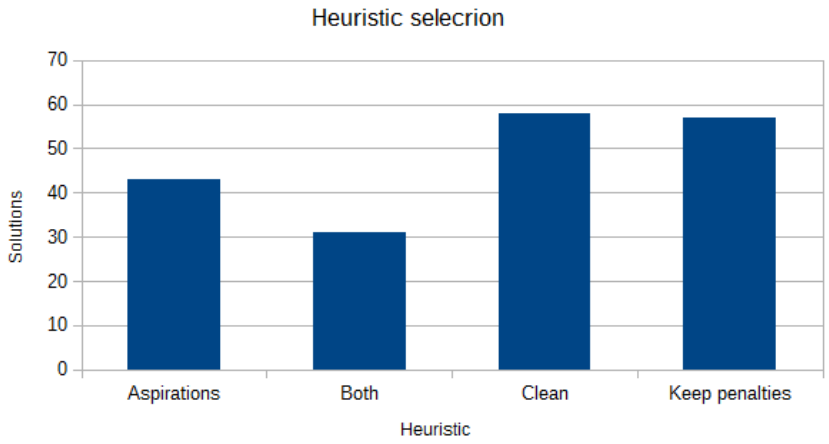
The best value for the λ parameter is 1, which refers to the recommendations from [5] and [2].

Heuristic selection

If we execute similar query for the heuristics as this:

```
SELECT with_aspirations,  keep_penalties FROM graph_result
ORDER BY used_colors ASC, time ASC, iterations ASC
WHERE updates > 0 AND lambda = 1 AND sideways = 2
LIMIT 3
```

and aggregate its result, then following bar-plot is obtained:



The following conclusions can be done from the results above the given timeout from 4 seconds:

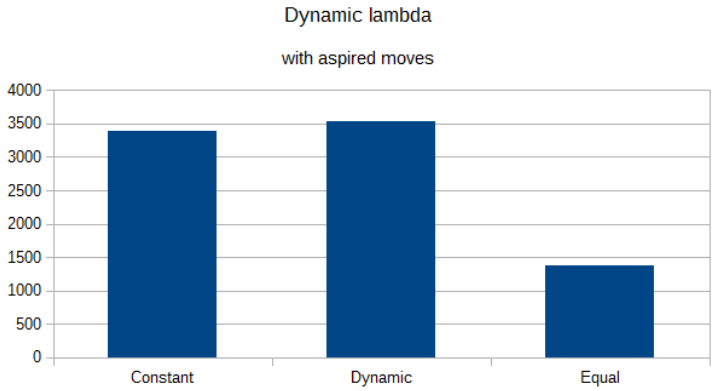
- The clean guided search works very well and fast.
- The both heuristics do not cooperate very well.
- The keep penalties heuristic has almost the same performance as the clean search. This means that it almost does not affect it.
- The aspiration moves do not work for the given timeout. That is because they are an overhead for this timeout. However, if longer timeouts are allowed, then the aspiration moves turn out to be also a good heuristic. This is because they allow a second branch of movements after the weights were established, while visiting neighboring solutions with smaller number of conflicts. This is a more detailed, self-correcting approach for studying the neighborhood.

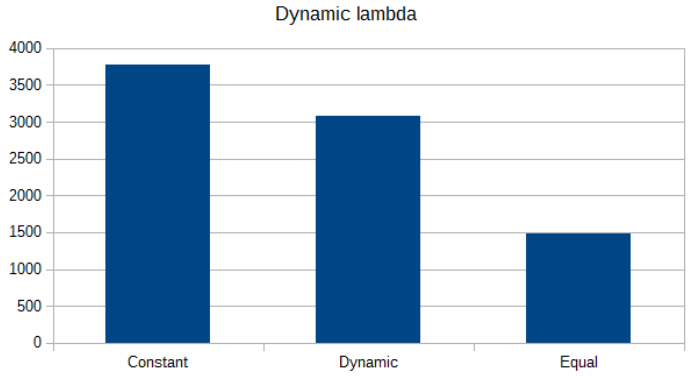
Dynamic λ

To see how the dynamic λ performs compared to the constant λ a comparison is made with the other heuristics. It is done by fetching the results from the query:

```
SELECT used_colors, time, iterations FROM graph_result
ORDER BY used_colors ASC, time ASC, iterations ASC
WHERE updates > 0 AND sideways = 2
LIMIT 3
```

After that every row is compared for equality on those fields (used_colors, time, iterations) with the result from dynamic λ with and without aspirations. The results are shown in the bar-plots below:





The main result is that the dynamic λ helps to the aspiration moves to be comparable or even better with the clean guided local search.

The next note is that only dynamic λ is not as good as the clean variant. The solution is to look for a coefficient, with which to multiply the λ coefficient. This could make the both variants of dynamic λ even better than the constant value of the parameter.

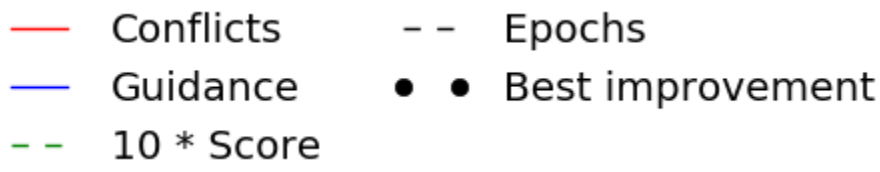
FUTURE WORK

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

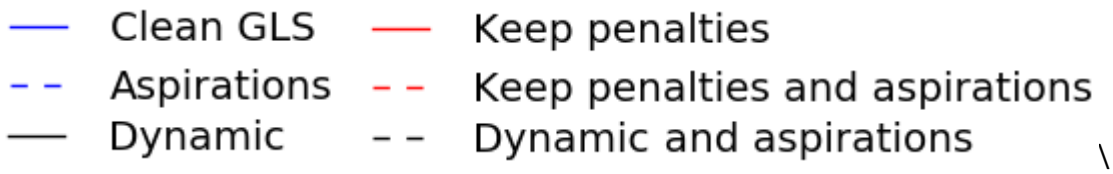
- **Bigger timeout**- an execution of the second experiment with bigger timeout (e.g.20, 60 seconds) would give even more details on the parameters, with which the guided local search should be tuned. However this is a long execution, because a full test of one graph with the setup above takes around 810 times the set timeout.
- **Extended bipartite** – apply the DFS altering scheme as in the bipartite coloring, but instead of 2 color classes, initially use the upper bound, which is calculated for the given graph. This strategy looks promising, because it is possess structure, which is similar to the random coloring, but should contains less conflicts.
- **Better greedy strategy** - only a simple greedy strategy was tested. There exists a lot of greedy heuristics, which could give a better initial coloring than this from the greedy implementation. This should definitely lower the count of iterations for finding a solution.
- **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. For further discussion, see [5].
- **Restricted graph coloring** – This is a variation of the graph coloring, where in the beginning some of the nodes are given colors, which should not be changed during the search. For this purpose, 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.
- **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
- **Minesweeper –** It can be shown that the game Minesweeper has solving algorithm, which belongs to NP-complete. Since k-graph coloring is also NP-complete, a reduction from it to the Minesweeper game should exist
- **Hill climbing epochs strategy** – it is known that the number of colorings as a function of the different colors, with which they are made is not a continuous function. For example it could happen that the solved graph has 2 colorings with 16 colors, **none** colorings with 15 colors and 1 coloring with 14 colors. The sequential epoch strategy will probably hang up on the 15 colors, because the guided local search will give weights to all 15 colors and the chance to give no weight to one color seems negligible. Therefore a strategy, which does not sequentially process the number of used colors should be looked for. An example strategy is the hill climbing.

LEGENDS

For the initialization comparison, use this legend:



For the performance comparison, use this legend:



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/]
7. Guided local search [https://en.wikipedia.org/wiki/Guided_Local_Search]
8. Minimax [https://en.wikipedia.org/wiki/Minimax]