ITAT 2016 Proceedings, CEUR Workshop Proceedings Vol. 1649, pp. 179–186
http://ceur-ws.org/Vol-1649, Series ISSN 1613-0073, © 2016 O. Trunda, R. Brunetto

Fitness landscape analysis of hyper-heuristic transforms for the vertex cover problem

Otakar Trunda and Robert Brunetto

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské náměstí 25, Praha, Czech Republic
otakar.trunda@mff.cuni.cz, robert@brunetto.cz

Abstract: Hyper-heuristics have recently proved efficient in several areas of combinatorial search and optimization, especially scheduling. The basic idea of hyper-heuristics is based on searching for search-strategy. Instead of traversing the solution-space, the hyper-heuristic traverses the space of algorithms to find or construct an algorithm best suited for the given problem instance. The observed efficiency of hyper-heuristics is not yet fully explained on the theoretical level. The leading hypothesis suggests that the fitness landscape of the algorithm-space is more favorable to local search techniques than the original space.

In this paper, we analyse properties of fitness landscapes of the problem of minimal vertex cover. We focus on properties that are related to efficiency of metaheuristics such as locality and fitness-distance correlation. We compare properties of the original space and the algorithm space trying to verify the hypothesis explaining hyper-heuristics performance. Our analysis shows that the hyper-heuristic-space really has some more favorable properties than the original space.

1 Introduction

Hyper-heuristics are becoming more and more popular in the field of combinatorial search and optimization. They transfer the search process from the space of *candidate solutions* to a space of *algorithms that create candidate solutions*. Such approach combines metaheuristic techniques with algorithm selection and proved to be efficient on many domains [2]. Several theoretical results about hyper-heuristics have been established. For example: given a set of low-level algorithms, the hyper-heuristic combination of them can outperform each of those individuals on *all* domains. *"The hyper-heuristic lunch is free [8]."* However, the efficiency and inefficiency of hyper-heuristics on some domains is not yet fully understood.

In this paper, we explore the hypothesis, that the efficiency of hyper-heuristics is caused by the fact that the space of algorithms is easier to explore for local search techniques than the original space. We use fitness landscape analysis to compare properties of hyper-heuristic space with the original space and to determine which one is more suitable for local search. We work with the vertex cover problem, as an example of a well-established and hard optimization problem.

In the next section, we provide some basic background on the vertex cover problem, fitness landscape analysis techniques and hyper-heuristics. The third section presents our design of a hyper-heuristic for the vertex cover. In the fourth section, we then define the spaces we analyse and the results of the analyses are given in the fifth section. The paper is concluded by a discussion about the results and possible future work.

2 Background

2.1 Vertex cover problem

An undirected graph G is a tuple $G = (V,E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, $E = \{e_1, e_2, \ldots, e_m\}$, such that $\forall e_i \in E, e_i = \{a_i, b_i\}, a_i, b_i \in V, a_i \neq b_i$ (no loops) and $\forall e_i, e_j \in E, i \neq j \implies e_i \neq e_j$ (no multiple edges). A set $S \subseteq V$ is a vertex cover in G iff $\forall e_i \in E, \exists s_j \in S$, s. t. $s_j \in e_i$.

An undirected vertex-labelled graph G is an undirected graph together with a cost function $c : V \mapsto \mathbb{R}^+$ that assigns positive costs to vertices. Given an undirected vertex-labelled graph $G = (V,E)$, the minimal vertex cover problem deals with finding a set $S \subseteq V$ minimizing $\sum_{s_i \in S} c(s_i)$ under the condition that S is a vertex cover in G. From now on, by *graph* we will always mean undirected vertex-labelled graph.

We use several other standard graph notions: for $v_i \in V$, by $\Gamma(v_i)$ we denote the neighborhood of v_i i. e. the set of all vertices that are connected to v_i by an edge. $\Gamma(v_i) = \{v_j \in V \mid \exists e_i \in E, e = (v_i, v_j)\}$. By $deg(v_i)$ we denote the degree of vertex. $deg(v_i) = |\Gamma(v_i)|$. A vertex v is called *leaf* if it's degree is 1. A graph is called *regular*, if all its vertices have the same degree.

We forbid loops in the graph as they don't present any new challenge to the problem. Vertices with loops always have to be in the vertex cover, so by a linear preprocessing (removing all loopy vertices), we can reduce the problem with loops to an equivalent problem without loops.

We forbid multiple edges between the same pair of vertices as well. We could simply leave multiple edges in the graph as they have no effect on the set of vertex covers nor on the optimality of the solutions, but it would change degrees of vertices which might lead our search algorithms astray.

Many real-life optimization problems from transportation, scheduling and operations research can be directly reduced to vertex cover. As for the complexity of the

problem, the vertex cover is a well-known NP-complete problem, a 2-approximation algorithm exists, and using parametrized complexity, the best known optimal algorithm for the unweighted version runs in time $O(kn + 1.274^k)$, where n is the number of vertices and k is the size of an optimal vertex cover [3]. In practical applications, heuristics are typically used [1].

2.2 Fitness landscape analysis

In the theory of local search algorithms, like evolutionary algorithms, hill-climbing, tabu-search and so on, the fitness landscape analysis is used to assess efficiency of the algorithm on a given problem. The theory of fitness landscapes can also be helpful when designing new metaheuristic algorithm for a specific problem [9, 10].

A fitness landscape of an optimization problem is a set M together with a function $f : M \mapsto \mathbb{R}$ and a distance measure $d : M \times M \to \mathbb{R}^+$. M denotes the set of all candidate solutions, f is the fitness function which tells us how good the candidate solution is, and d measures distance between candidate solutions. We work with combinatorial problems, so the set M will be discrete and finite.

In Figure 1, there is an example of a fitness landscape of a small instance of the Travelling Salesman Problem [11]. The tree in the upper part enumerates the set of all permutations which constitutes the set M. Beneath it, there is a graph of the fitness function for each point of M (the length of a tour corresponding to each permutation). The points of M are arranged linearly by their lexicographical order which induces the distance measure and structure. The marked point corresponds to the solution shown in Figure 2 which is a local optimum.

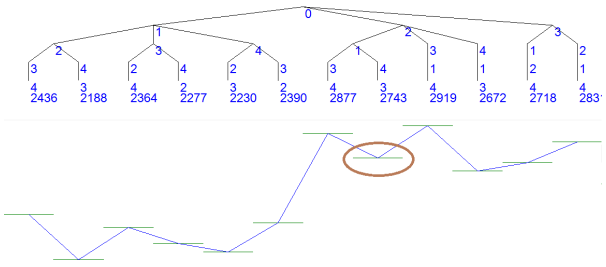

Figure 1: Example of a fitness landscape of a small TSP

The measure d is typically not given explicitly, but instead it is derived from the *search operators* that the algorithm uses. (For example a mutation operator used by genetic algorithms.) Formally: a unary search operator p is a map $p : M \mapsto 2^M$, where $p(m)$ is the set of all possible modifications of m - i.e. the set of possible results of the operator. When applying the operator p to a point $m \in M$, the algorithm replaces m by one of the points from $p(m)$. The point can be selected randomly, or by some criterion, e.g. minimum, in case of greedy operators. The algorithm repeatedly applies operators to move from one candidate solution to another.

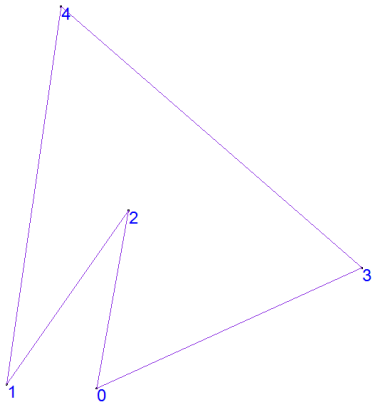

Figure 2: One of the solutions of the TSP example

The role of the distance measure is to introduce a structure to the set M. For the purposes of analysis, it is especially important to define neighborhoods of points. For $m_i \in M$, we denote the neighborhood of m_i as $N(m_i)$. It is a set of all points from M that are *close to* m_i.

If d is given explicitly, then N can be defined as $N(m_i) = \{m_j \in M \mid d(m_i, m_j) \leq \varepsilon\}$. Typically ε is taken to be 1. In the much more common case when we are given the set of search operators P instead of d, the neighborhood is defined as $N_P(m_i) = \{m_j \in M \mid \exists p \in P, m_j \in p(m_i)\}$, i.e. the set of all solutions reachable from m_i by an application of a single operator. The operators can also induce a distance measure as $d_P(m_i, m_j) =$ *minimum number of applications of operators from* P *that allow moving from* m_i *to* m_j. Formally:

1. $d_P(m_i, m_i) = 0$
2. $d_P(m_i, m_j) = (\min_{m_k \in N_P(m_i)} d(m_k, m_j)) + 1$

In the following text, we will use a few simple notions from function analysis and theory of search. Given a function $f : M \mapsto \mathbb{R}^+$, that we want to minimize, we say:

- $x^* \in M$ is an optimal solution (or global optimum) iff $\forall m \in M : f(x^*) \leq f(m)$
- $x \in M$ is a local optimum iff $\forall m \in N(x) : f(x) \leq f(m)$
- if x is not an optimal solution, then the escape distance of x is $ed(x) = \min_{\{m \in M \mid f(m) < f(x)\}} d(x, m)$ = distance to the nearest point with strictly lower fitness value

Fitness landscape analysis studies properties of fitness landscapes that are related to performance of local search algorithms. For example, smooth and globally convex function with a single local optimum is much easier to deal with than highly rugged function with many local optima. Several methods for analyzing fitness landscapes have been developed:

- Size of M: smaller $|M|$ leads to higher performance and vice versa

- Number of local optima: many local-search algorithms are attracted to local optima. The higher number of local optima therefore slows down the convergence to global optimum.

- Average size of neighborhood: large neighborhood decreases the number of local optima, but dramatically increases time that the algorithm needs to traverse the space, and operators with large range of values are close to random search. It is therefore important to keep the neighborhoods small.

  For example, if all points of M are neighbors, then there are no local optima except the global ones. On the other hand, if neighborhoods are empty, i.e., no pair of points are neighbors, then every point is a local optimum. There is therefore a trade-off between size of neighborhoods and number of local optima.

- Fitness-distance correlation (FDC): even with only one local optimum, the algorithm might not be able to find it if it is surrounded by points with high fitness values. It is therefore important that the optimal solutions are surrounded by low-fitness valued points and are far from high-valued points. FDC measures how the distance between points corresponds to difference between their fitness values. Ideally, there should be a strong positive correlation – i.e. the further from the nearest global optimum the point is, the higher fitness value it should have.

  Formally, the FDC is computed as: $FDC = \frac{c_{fd}}{\rho(f)\rho(d_{opt})}$, where c_{fd} is a covariance of f and d_{opt}, $c_{fd} = \frac{1}{|M|}\sum_{i=1}^{|M|}(f(m_i)-\overline{f})(d_{m_i,opt}-\overline{d_{opt}})$, $\overline{f}$ is average fitness value over the whole M, $d_{m_i,opt}$ is a distance between m_i and the nearest optimal solution, and $\overline{d_{opt}}$ is the average of $d_{m_i,opt}$ over the whole M. $\rho(f)$ and $\rho(d_{opt})$ are standard deviations of f and d_{opt} respectively.

  FDC is in range $[-1,1]$. In the ideal case, where $f(m_i) = d_{m_i,opt}$, the FDC = 1, for a random function, the FDC is close to 0 and FDC = -1 means that the optimal solution is "hidden" among high-valued points.

- Ruggedness: rugged function is opposite to smooth - it is erratic, with large differences in fitness values between nearby points. Ruggedness is computed as $R = \frac{\overline{f(x)f(x)}_{d(x,y)=1}-(\overline{f})^2}{\overline{f^2}-(\overline{f})^2}$. Where *overline* denotes average over all values [7]. R is always in $[-1,1]$, value of 1 indicates constant function, 0 means that values of neighbouring points are independent and -1 indicates that the neighbouring points have opposite values (i.e. every point is a local extreme). Note that higher ruggedness coefficient actually denotes more smooth function which is favorable for local search.

- Average escape distance: low escape distance allows the algorithm to quickly find a point with better fitness value. Ideally the $ed(m)$ should be constant for each $m \in M$. We only compute average of $ed(m)$ over local optima since for points which aren't locally optimal, the escape distance is always 1.

The landscape-analysis techniques have to traverse the whole search-space of the problem multiple times. As such search-spaces are typically very large, the analysis can only be done on small instances. There are techniques that allow to estimate some properties even in large spaces by sampling, but we won't be using those here.

Note that our main purpose here is not to actually *solve* large instances of vertex cover, but rather to verify the hypothesis, that hyper-heuristic spaces might have very different properties and might be much more favorable for local search techniques than the original space.

2.3 Hyper-heuristics

Hyper-heuristics represent a new approach to search and optimization which combines metaheuristics, automated parameter tuning, algorithm selection and genetic programming. Nowadays, the algorithm selection approaches are becoming more and more popular in combinatorial optimization, as it is clear that no single algorithm can outperform every other on all domains, and therefore, the most suitable algorithm has to be selected for the problem at hand [4, 5].

Hyper-heuristics try to build an algorithm suited for given task by combining so called *low-level algorithms* during the search. A pool of simple algorithms is given and the hyper-heuristic combines them into more complex units. The combination procedure is often based on some kind of evolutionary computation and the quality of resulting units is measured by how well they can solve the original task. The approach was especially successful in the area of scheduling and it is now applied to many other kinds of problems [2].

Instead of searching the *solution space* directly, hyper-heuristics search the *algorithm space*. The approach is based on an assumption that *similar algorithms will find solutions of similar quality (not necessarily close to each other)* which implies that the algorithm space has some favorable properties: high locality, i.e., elements close to each other have similar evaluation, and low number of local extrema. See figure 3. On spaces with those properties, optimization metaheuristics can find good solutions quickly. Of course, such improvements come for a price: evaluating an element from the algorithm space takes much more time because it involves searching for a solution in the solution space.

Consider the following example: we want to color a graph with the fewest colors possible. Using a genetic algorithm, we could come up with a metaheuristic that will work on the set of all possible assignments of color to vertices a the fitness function will penalize violations (connected vertices having the same color) and high number

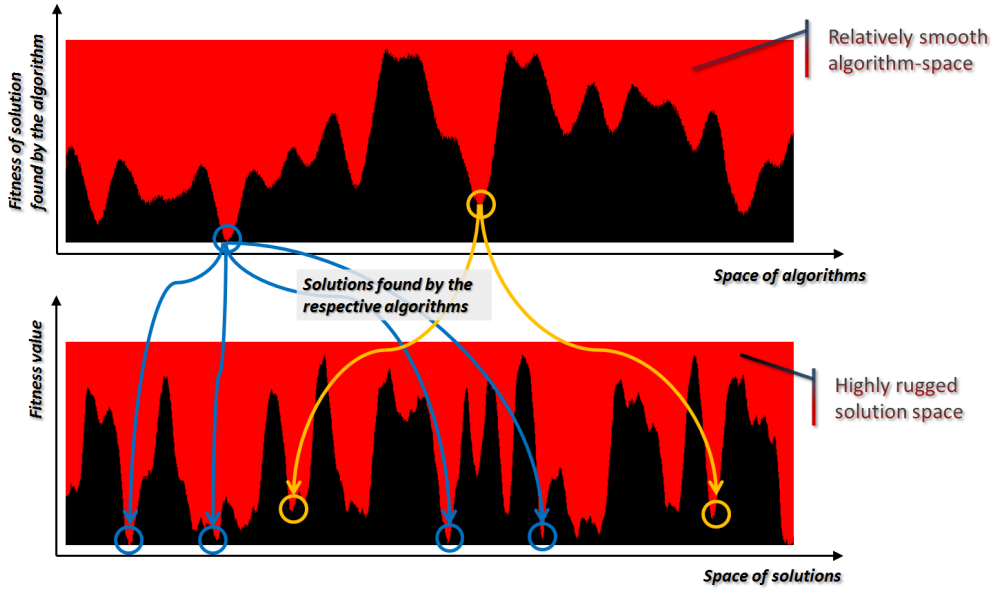

Figure 3: The hypothesized schema of a hyper-heuristic transformation

of color used. Such algorithm might converge to globally optimal solution, but it would take a long time.

We could also use a greedy algorithm which works as follows: picks a vertex from a graph according to some criterion and colors it by the lowest color possible, according to the colors of its neighbours. Then it picks another vertex and so on until the graph is colored. The greedy algorithm is very fast, but in most cases it wont converge to global optimum.

There are several vertex-picking criteria, for example:

- Largest degree (L) - picks a vertex with the largest number of neighbours
- Saturation degree (S) - picks a vertex who's neighbours are colored with the largest number of different colors
- Incidence degree (I) - picks a vertex with the largest number of colored neighbours

Based on these three low-level greedy algorithms, a hyper-heuristic would search a space of all combinations of them (the algorithm space). On a graph with 5 vertices, the algorithm space would consist of all sequences of the length 4 containing symbols L, S and I. To evaluate the point from the algorithm space, the hyper-heuristic would use the algorithm to find a solution, then evaluates the solution by the original fitness function and uses the value to evaluate the algorithm. For example a point *ILS* (from the algorithm space) is evaluated as follows:

1. Select a vertex based on the I criterion and color it with the lowest possible color
2. Select a vertex based on the L criterion and color it
3. Select a vertex based on the S criterion and color it
4. Color the last vertex
5. Compute the total number of colors used and use it as the fitness value of the point (ILS)

The hyper-heuristic might also be based on an evolutionary algorithm, which would in this case work on the algorithm space. This is popularly summarized as *Heuristics select moves, hyper-heuristics select heuristics*. The hyper-heuristic might be able to find an optimal solution, and more importantly, it might be able to find high-quality solutions much faster than the former approach.

3 Hyper-heuristic for minimal vertex cover

We have designed a hyper-heuristic for the vertex cover in the similar manner as in the example mentioned earlier. We used a pool of simple greedy algorithms and the hyper-heuristic combines them in order to solve the problem. The low-level algorithms work sequentially as described in Algorithm 1.

Algorithm 1 Greedy vertex cover

```
1: S ← ∅
2: while G contains some edge do
3:     v ← select vertex according to some criterion
4:     S ← S ∪ {v}
5:     remove v from the graph (with incident edges)
6: return S
```

As the vertex-selection criteria, we use the following:

- Leightest (W) : selects the vertex with minimal weight (cost)

- Deg (G) : selects the vertex with maximal degree
- Sub (S) : selects the vertex v with maximal [(sum of costs of neighbors) - cost of v] i.e. selected = $argmax_{v\in V}(\sum_{\{w\in\Gamma(v)\}} c(w) - c(v))$
- WDeg (G) : selects the vertex with minimal *weight / degree*. Selected = $argmin_{v\in V}(c(v)/deg(v))$
- Div (D) : selects the vertex v with minimal ratio of cost of v and sum of costs of neighbors. Formally selected = $argmin_{v\in V}(c(v)/\sum_{\{w\in\Gamma(v)\}} c(w))$
- Leaf (L) : selects the vertex v with minimal difference between cost of v and (sum of costs of neighbors of v that are leaves), e.i. selected = $argmin_{v\in V}(c(v) - \sum_{\{w\in\Gamma(v)\ |w\ is\ a\ leaf\}} c(w))$
- Neig (N) : selects the vertex with maximal sum of costs of neighbors
- Next (X) : selects the vertex which is the next in order after the vertex selected by (W) (we used this as an example of non-greedy, chaotic algorithm)

We break ties simply by the ordinal numbers of vertices, i.e., if more than one vertex achieve the optimal value of the criterion, we select among then the one with the lowest number.

The space of algorithms consists of all sequences of letters *W,G,S,D,L* (corresponding to selection criteria) of a fixed length. The i-th symbol in the sequence determines the criterion by which the i-th vertex is selected. If a valid vertex cover is found before all symbols are used, the rest of the sequence is ignored. In much more frequent case when we have already traversed the whole sequence but there are still edges in the graph, we use two strategies: (i) start over and read symbols from the beginning of the sequence (*repeat* strategy) and (ii) use the last symbol in the sequence for as long as there are edges in the graph (*last* strategy).

4 Analysis of the fitness landscapes

The hyper-heuristic can be viewed as a transformation of the search space. We will now describe properties of the original and transformed spaces. We use the notation of fitness landscapes as defined earlier.

4.1 Original space

- M: set of all vertex covers (not necessarily minimal in inclusion)
- f: total weight of the vertex cover - sum of costs of all vertices in the cover
- d: distance between vertex covers is measured as the number of elements on which the two covers differ. Formally: $S_1, S_2 \subseteq V$: $d(S_1,S_2) = |\{S_1 \setminus S_2\} \cup \{S_2 \setminus S_1\}|$

The distance measure is related to the search operators *add vertex* and *remove vertex*. One application of such operator creates a solution within distance of 1 from the original point.

4.2 Hyper-heuristic space

- M: set of all fixed-length sequences of symbols corresponding to low-level heuristics
- f: total weight of the vertex cover found by the application of the sequence
- d: distance between sequences is measured by the *Levenshtein distance* which is the minimal number of operations *add symbol*, *remove symbol* and *replace symbol* needed to transform the first sequence into the second.

This distance measure is intuitively related to search operators *add symbol*, *remove symbol* and *replace symbol*.

We distinguish the hyper-spaces by the set of low-level algorithms that are used. We do not always use all the algorithms, partially because of the high computational cost and partially because on some kinds of graphs, the criteria degenerate. For example, on a uniformly-weighted graph, the algorithm *Neig* will work exactly the same as *Degree* so its not worth using both of them. Furthermore, we would like to be able to asses performance of single algorithms, pairs, 3-tuples, 4-tuples and so on.

We work with these types of spaces: original space (denoted *Cover*), hyper-space with a specific set of low-level algorithms L and a fixed length of sequences k (denoted $H_{L,k}$). We distinguish two alternatives as mentioned earlier: (i) a space, where during the evaluation, the sequence is applied repeatedly from the beginning until a valid cover is found (denoted $H^{repeat}_{L,k}$), and (ii) a space, where during the evaluation, after reaching the end of sequence, the last symbol is applied repeatedly until a valid cover is found (denoted $H^{last}_{L,k}$).

5 Experiments

We generated a set of graphs, constructed the corresponding spaces and computed fitness landscape metrics for those spaces. We used graphs of various sizes, densities and types. For each type of graphs, we generated about $30 \sim 40$ instances and average the results.

We used graphs of sizes $n = 16 \sim 45$ vertices (not all values from the range were used). The densities were 0.1, 0.2, 0.3, 0.4, 0.5 and 0.7. The density of c means that there were $\frac{c}{2} \cdot n \cdot (n-1)$ edges in the graph. We used two policies to add edges - uniformly randomly (select random pair of vertices and add an edge until the desired number of edges is reached) and regularly (adds edges more likely to vertices with low degrees to create a near-regular graph). Weights of vertices are integers taken uniformly from ranges $(50 \sim 50)$, $(50 \sim 60)$, $(50 \sim 100)$ and $(50 \sim 1000)$.

In total, we generated over 24 000 graphs, roughly half of them were near-regular and half of them used the uniform edge distribution. For each graph, we constructed the vertex covers-space and 20 hyper-spaces (for various combinations of parameters - $H^{last}_{L,k}$ and $H^{repeat}_{L,k}$ with different sets L). The experiments run on 11 computers with 8 cores each for 20 hours on each.

We present graphs comparing various metrics between spaces. On x-axis, there is the number of vertices in the graph, y-axis shows values of the particular metric. We plot several color-distinguished series in the same graph, each series represents one space. The space is described in the legend by an enumeration of low-level algorithms used. Exclamation mark at the end denotes the *last* strategy; no mark means that the *repeat* strategy is used. The word *cover* denotes the original space of all vertex covers.

We use a normalization so that different columns are of a similar magnitude and can be depicted in the same graph. Instead of plotting *criterion*, we plot *criterion / number of vertices - number of vertices*. With this normalization, values related to different number of vertices are not directly comparable, but values related to the same number of vertices are, which is good enough for our purposes. With this normalization, we plot *weight of the cover divided among each vertex*.

We plot averages over all generated graphs grouped into categories by number of vertices. We tried to distinguish the results further by edge-generation policy (regular graphs vs. random graphs), edge-density and width distribution. In most cases, such further distinguishing didn't provide any new information - the results were very similar in each of the smaller categories.

***Last* vs. *repeat* strategy** First, we have tried to compare the two ways of evaluating the sequences - repeat the whole sequence from the beginning and repeating only the last symbol, i.e. the spaces $H^{repeat}_{L,k}$ and $H^{last}_{L,k}$. We believe that different algorithms should be used in the beginning of the search and different one at the end, so the space $H^{last}_{L,k}$ should contain better solutions. We measure the value of the global optimum in each space. The graph is shown in Figure 4. $H^{last}_{L,k}$ is slightly better than $H^{repeat}_{L,k}$ for all L.

Solution quality in spaces Hyper-spaces doesn't contain all the vertex covers, they only contain some of them. For example, the set of *all* vertices is a valid vertex cover, but it is never generated by any algorithm sequence. On the other hand, some vertex covers might be generated by many different sequences of algorithms. To ease the search, the hyper-space should contain an algorithm that can generate an optimal solution, there should be a large number of different sequences that generate high-quality solutions and very few or none sequences that generate low-quality solutions.

To assess the quality of solutions generated by points in the hyper-space, we created a histogram of quality of all solutions from original and hyper-space. It is depicted

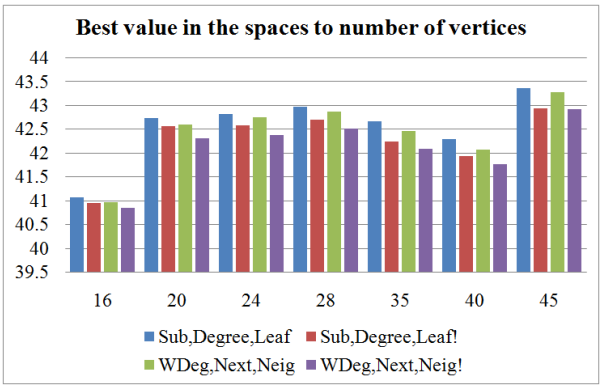

Figure 4: Best solutions in $H^{repeat}_{L,k}$ and $H^{last}_{L,k}$

in Figure 5. Red columns come from the original space and yellow ones come from one of the hyper-spaces. The hyper-space is smaller in size, so the total volume of yellow is smaller. Solutions in the original space seem to follow normal distribution, while in the hyper-space, most of the points generate near-to-optimal solutions or near-to-average solutions. The behavior can be explained as follows: high-quality solutions are generated due to greediness of the selection criteria and average-quality solutions are generated since their number in the original space is very high.

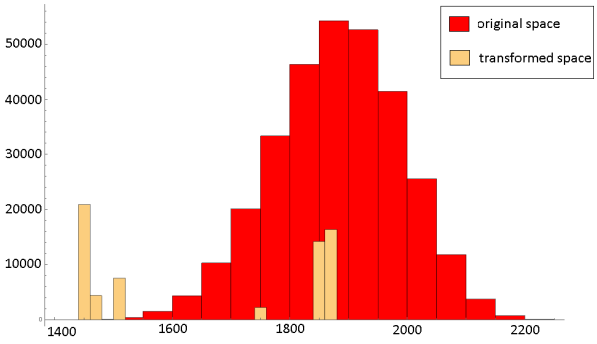

Figure 5: Histogram of distribution of solutions based on their quality in the two spaces.

We also monitor the average value of solutions in each space. Results are shown in Figure 6. All three hyper-spaces show significantly lower average value than the original space. (I.e. even a blind random search should provide better solutions in the hyper-space then in the original space.)

Combination of algorithms We test the hypothesis, that the combination of low-level algorithms can out-perform each of the individual algorithms. In Figure 7 there is the quality of an optimal solution in several hyper-spaces. *Min* denotes the best solution that can by found by repeated application of just one low-level algorithm. Other colors correspond to combinations. For all graph size, qual-

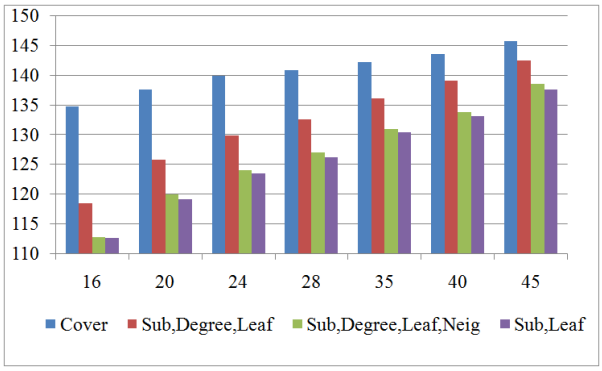

Figure 6: Average quality of solutions in spaces

ity of optimal solution is better for most combinations of low-level algorithms. This result supports the hypothesis and suggests that the hyper-heuristic might be worth using even with its overhead (evaluating points in hyper-space is costly). TODO in most cases, however cover je lepsi nez H-H takze transformation vede ke ztrate optimalniho reseni.

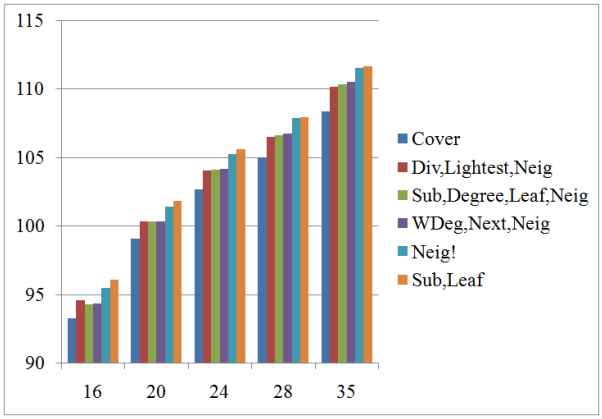

Figure 7: Best solutions of single algorithms and combinations

Fitness-distance correlation Figure 8 shows the FDC for several spaces. In the original space, FDC decreases with the number of vertices, while in the original space it stays high regardless of the size of the graph. (Note that higher FDC is better for local search algorithms.) This graph is shown without normalization, as the FDC is naturally in $[-1,1]$. The rest of graphs are also without normalization since it's not needed.

Ruggedness In Figure 9 there is the ruggedness of our spaces. Most of hyper-spaces have higher ruggedness that the original space which is again better for local search algorithms. The ruggedness seem to be independent on the size of the graph.

Escape distance Figure 10 shows the average escape distance (ED) of some of our spaces. The ED is systematically lower in all hyper-spaces. The difference might not

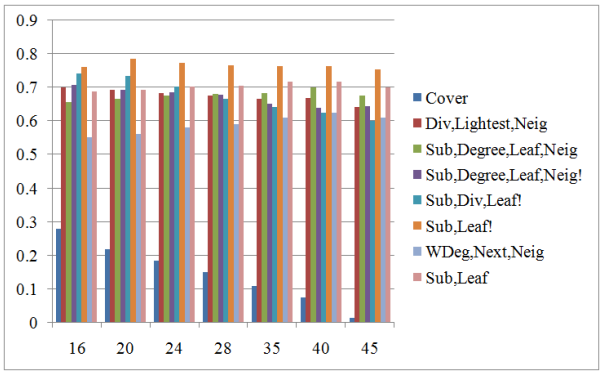

Figure 8: FDC of spaces

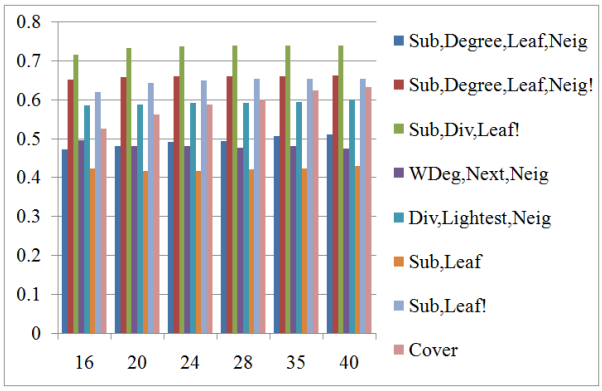

Figure 9: Ruggedness of spaces

be large, but note that the amount of work needed to escape the local minimum is $(number\ of\ neighbors)^{ED}$, so any savings in the exponent are significant.

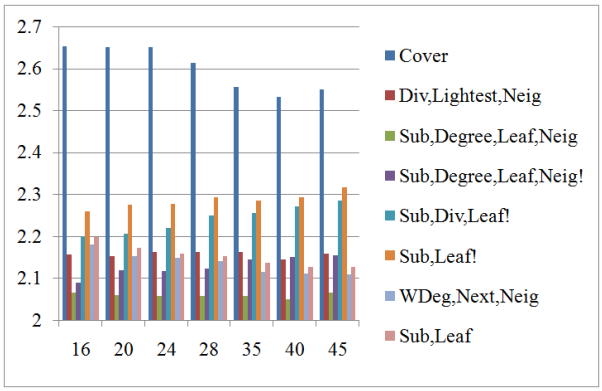

Figure 10: Escape distance from local optima

Number of local optima The graph 11 shows frequency of local minima in our spaces (i.e. the number of local minima over all nodes). The result is much better in the original space than in the hyper-spaces. This is partially caused by the fact that hyper-spaces contain many points that have the same fitness value and we define local minima as non-strict - i.e. points on plateaus are all considered local minima.

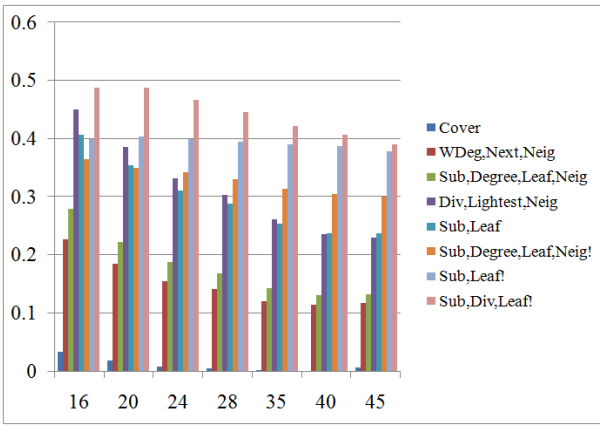

Figure 11: Frequency of local optima

6 Conclusions and future work

We presented a hyper-heuristic transformation for the vertex cover problem and constructed several variants of hyper-heuristic spaces using a set of greedy algorithms. We measured several important features of the original space and the hyper-spaces and by comparing them we have experimentally verified the hyper-heuristic-hypothesis that tries to explain the efficiency of hyper-heuristics in practise by properties of hyper-space. Our results can be summarized as follows:

1. Combination of greedy algorithms is in most cases much better that using single algorithm all the time
2. Only small part of the original space can be generated from the hyper-space. Most of high-quality solutions can be generated, but sometimes we loose the optimal solution by the transformation
3. The transformation improves fitness-distance correlation, escape distance and average solution quality. It has some positive effect on ruggedness as well, but it increases the frequency of local optima in the space.
4. There were no significant differences between various types of graphs, e.g. regular vs. random. We believe that this is caused by small size of our test graphs.

As future work, we would like to continue the analysis using more landscape analysis techniques, such as *neutrality* [6], *decomposability* etc. Also, we would like to add more low-level algorithms into the pool, especially less-greedy ones. Greediness of the algorithms causes that only a very small portion of vertex covers are reachable from the hyper-space. In some cases, more than 50% of all algorithms in the space generated the same vertex cover. Such small range of unique fitness values then creates large plateaus, which contribute to an illusion of high locality and fine ruggedness. It is also responsible for the large number of local extrema that we observed in the hyper-spaces.

It would also be beneficial to actually run some search algorithm on the hyper-space (for some large instances) and prove that the hyper-heuristic really works in practice and is at least comparable to metaheuristics working directly in the solution space.

Acknowledgement

The research is supported by the Grant Agency of Charles University under contract no. 390214 and it is also supported by SVV project number 260 104. We would like to thank the anonymous reviewers for their useful comments and suggestions.

References

[1] Eric Angel, Romain Campigotto, and Christian Laforest. Implementation and Comparison of Heuristics for the Vertex Cover Problem on Huge Graphs. In *11th International Symposium, SEA 2012*, volume 7276 of *Lecture Notes in Computer Science*, pages 39–50, Bordeaux, France, June 2012.

[2] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and R. Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, Dec 2013.

[3] Jianer Chen, Iyad A. Kanj, and Ge Xia. *Improved Parameterized Upper Bounds for Vertex Cover*, pages 238–249. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[4] Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. *Autonomous search*. Springer-Verlag, 2012.

[5] R. Kumar, S. K. Singh, and V. Kumar. A heuristic approach for search engine selection in meta-search engine. In *Computing, Communication Automation (ICCCA), 2015 International Conference on*, pages 865–869, May 2015.

[6] Marie-Eléonore Marmion, Clarisse Dhaenens, Laetitia Jourdan, Arnaud Liefooghe, and Sébastien Vérel. On the neutrality of flowshop scheduling fitness landscapes. *CoRR*, abs/1207.4629, 2012.

[7] Peter Merz and Bernd Freisleben. Fitness landscapes, memetic algorithms, and greedy operators for graph bipartitioning. *Evol. Comput.*, 8(1):61–91, March 2000.

[8] Riccardo Poli and Mario Graff. *Genetic Programming: 12th European Conference, EuroGP 2009 Tübingen, Germany, April 15-17, 2009 Proceedings*, chapter There Is a Free Lunch for Hyper-Heuristics, Genetic Programming and Computer Scientists, pages 195–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[9] Franz Rothlauf. *Representations for genetic and evolutionary algorithms (2. ed.)*. Springer, 2006.

[10] Franz Rothlauf. *Design of Modern Heuristics*. Natural Computing Series. Springer, 2011.

[11] Mohammad-H Tayarani-N. and Adam Prügel-Bennett. An analysis of the fitness landscape of travelling salesman problem. *Evolutionary Computation*, 24(2):347–384, Jun 2015.