Bachelor Thesis

Boosting Local Search for the Maximum Independent Set Problem

Jakob Dahlum

Abgabedatum: 06.11.2015

Betreuer: Prof. Dr. rer. nat. Peter Sanders
Dr. rer. nat. Christian Schulz
Dr. Darren Strash

Institut für Theoretische Informatik, Algorithmik
Fakultät für Informatik
Karlsruher Institut für Technologie

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Abstract

An independent set of a graph $G = (V, E)$ with vertices V and edges E is a subset $S \subseteq V$, such that the subgraph induced by S does not contain any edges. The goal of the *maximum independent set problem* (*MIS problem*) is to find an independent set of maximum size. It is equivalent to the well-known *vertex cover problem* (*VC problem*) and *maximum clique problem.*

This thesis consists of two main parts. In the first one we compare the currently best algorithms for finding near-optimal independent sets and vertex covers in large, sparse graphs. They are Iterated Local Search (ILS) by Andrade et al. [2], a heuristic that uses local search for the MIS problem and NuMVC by Cai et al. [6], a local search algorithm for the VC problem. As of now, there are no methods to solve these large instances exactly in any reasonable time. Therefore these heuristic algorithms are the best option.

In the second part we analyze a series of techniques, some of which lead to a significant speed up of the ILS algorithm. This is done by removing specific vertices and structures within the graph, which have a low probability of being part of the MIS. These are vertices of high degree, vertices within dense subgraphs or the like. In addition, we analyze the behavior of the ILS algorithm when inserting the cut vertices back into the graph. We also perform exact reductions which reduce the graph's complexity without destroying information about the MIS. We can revert them after running ILS on the reduced graph.

We present the experimental results of the comparison of ILS and NuMVC, as well as the improvement of ILS by removing vertices and the like. The used instances are known graphs from literature like road maps, social networks, meshes and web graphs.

Zusammenfassung

Eine unabhängige Menge eines Graphen $G = (V, E)$ mit Knotenmenge V und Kantenmenge E ist eine Teilmenge $S \subseteq V$, so dass der durch S induzierte Teilgraph keine Kanten enthält. Das Ziel des *Maximum Independent Set Problems* (*MIS Problem*) besteht darin, unter allen unabhängigen Mengen die Menge mit maximaler Mächtigkeit zu finden. Es ist äquivalent zu den bekannten Problemen *Vertex Cover* (*VC*) und *Maximum Clique*.

Diese Arbeit besteht aus zwei Hauptteilen. Im ersten Teil vergleichen wir die zurzeit besten heuristischen Algorithmen zum Finden von nahezu optimalen unabhängigen Mengen und Vertex Covers in großen, wenig dichten Graphen. Diese sind Iterated Local Search (ILS) von Andrade et al. [2], eine Heuristik, die per lokaler Suche das MIS Problem angeht, sowie NuMVC von Cai et al. [6], ein lokaler Suchalgorithmus für das VC Problem. Derzeitige Methoden ermöglichen es noch nicht, diese großen Instanzen exakt zu lösen, weshalb wir auf Näherungslösungen zurückgreifen müssen.

Im zweiten Hauptteil analysieren wir eine Reihe von Techniken unter denen einige eine signifikante Beschleunigung des ILS-Algorithmus erzielen. Dies geschieht durch das Entfernen bestimmter Knoten und Strukturen, welche eine geringe Wahrscheinlichkeit haben, in der maximalen unabhängigen Menge (MIS) zu liegen. Hierbei handelt es sich um Knoten mit hohem Grad, Knoten innerhalb von dichten Teilgraphen oder Ähnlichem. Ebenso untersuchen wir das Verhalten des ILS-Algorithmus beim Einfügen der entfernten Knoten zurück in den Graph. Desweiteren führen wir exakte Reduktionen durch, welche den Graph vereinfachen ohne Informationen über das MIS zu zerstören. Diese Reduktionen lassen sich nach der Durchführung von ILS rückgängig machen.

Wir stellen die experimentellen Ergebnisse des Vergleiches von ILS und NuMVC, sowie der Verbesserung von ILS durch Knotenentfernungen und dergleichen vor. Die verwendeten Instanzen sind bekannte Graphen aus der Literatur wie Straßennetze, soziale Netzwerke, Meshes oder Webgraphen.

Acknowledgements

I would like to thank Prof. Dr. Peter Sanders, Dr. Christian Schulz and Dr. Darren Strash for allowing me to work on this very interesting subject. Dr. Christian Schulz and Dr. Darren Strash continuously pushed me forward and gave good advice. I especially owe thanks to Dr. Darren Strash for answering countless mails and questions that I had. And finally, thanks to my girlfriend for the stress relief when I needed it.

Contents

1. Introduction **7**

1.1. Motivation . . . 7
1.2. Our Results . . . 7
1.3. Related Work . . . 8
1.4. Organization of this Thesis . . . 8

2. Preliminaries **9**

3. ILS and NuMVC **11**

3.1. Iterated Local Search (ILS) . . . 11
3.2. NuMVC . . . 14

4. Cutting Algorithms **17**

4.1. Cutting Vertices and Structures . . . 17
4.2. Inserting Cut Vertices . . . 27
4.3. Exact Reductions . . . 29

5. Experiments **35**

5.1. Graph Instances . . . 35
5.2. Comparison of ILS and NuMVC . . . 36
5.3. Cutting Vertices and Structures . . . 38
5.4. Inserting Cut Vertices . . . 52
5.5. Exact Reductions . . . 55

6. Conclusion and Future Work **61**

6.1. Conclusion . . . 61
6.2. Future Work . . . 61

A. Data Structures for Cutting Algorithms **62**

1. Introduction

1.1. Motivation

The *maximum independent set* (*MIS*) problem or the closely related problems can be very helpful to solve graph-based problems. Improvements in one problem type directly lead to improvements in the related types. One important graph-related task is resource scheduling. Imagine a graph where vertices are machines that require maintenance to function properly. Due to limited space and personnel in the maintenance hall the goal is to tightly pack as many machines as possible in the building such that the repairmen can quickly check them. However, there is the restriction that certain machines cannot be checked at the same time since they require the same replacement parts or cause two much heat together making the work impossible. These restrictions are represented by edges. Moreover, the company can currently only afford one such maintenance session so it must be as beneficial as possible. The goal of finding the largest set of vertices without any edges connecting them is reached by finding the MIS. The MIS problem and related problems have many applications spanning disciplines such as information retrieval, signal transmission or aligning DNA and protein sequences [14]. Another application is to label maps efficiently [12]. Since these problems are $\mathcal{NP}$-hard [10], it is unlikely that a fast polynomial algorithm will be found. Thus, only small instances can be solved exactly in a reasonable time frame, usually via a branch-and-bound algorithm like MCS by Tomita et al. [25], the bit parallel algorithms by San Segundo et al. [22], [23] or the branch-and-reduce algorithm by Akiba et al. [1]. Still, in many cases it is "good enough" to find an approximate solution if we can do so fast. If the instances reach massive sizes there usually is no other option than heuristic methods. Therefore, the focus should be on increasing the speed of heuristic algorithms for this problem which we are aiming for in this thesis.

The current state-of-the-art algorithms for the MIS problem and the closely related VC problem are the local search algorithms ILS by Andrade et al. [2] and NuMVC by Cai et al. [6] respectively. They both are heuristic algorithms that are not expected to give exact results but achieve near-optimal results. ILS repeatedly attempts to increase its solution by taking out one vertex of it and inserting two vertices instead. Similarly, NuMVC, aiming for the smallest solution size, takes out a vertex which most likely results in an invalid vertex cover. It then proceeds to swap a solution vertex with a non-solution vertex until it reaches a valid vertex cover. In many instances both algorithms deliver near-optimal solutions for their respective problems. However, there is room for improvement as very large sparse graphs cause difficulties. These lead to slowdowns and potentially worse solution sizes and have not been taken into account yet.

1.2. Our Results

Despite the strong connection of the MIS and VC problem, no comparison between ILS and NuMVC has been done yet. We take care of that in this thesis. Additionally, our experiments show that in very large networks few specific vertices slow down local search algorithms. Our algorithms tackle this difficulty to achieve improved computation times for huge graphs. As real-world graphs are continually getting more massive, this optimization is of great significance. We experiment with different strategies to boost local search. One method is to remove the unlikely candidates before running the local search algorithm. We try removing vertices of high degree, vertices within a dense neighborhood or of high centrality. But also a multitude of other factors are taken into consideration and examined in this thesis. We found the removal of vertices with a high degree, which have a low probability of being in the MIS, to be the most promising. Another strategy is to insert these cut vertices back into the graph step by step as soon as ILS reaches a tolerable result on the cut graph. This prevents ILS from completely ignoring vertices that have a low probability of being in the MIS. This method allows us to cut a larger portion of the graph which leads to a better speed up of the algorithm while still achieving the near-optimal solutions we get from smaller removals. Our final strategy is to run exact reductions that minimize a graph to its kernel and then run ILS. The reductions reduce the workload that ILS has to do so that it can focus on the critical areas of the graph.

1.3. Related Work

Due to their importance in both theory and practice, the MIS, VC and MC problems have been widely studied [5, 7, 18, 19, 8]. The MIS problem is $\mathcal{NP}$-hard [15] and therefore may require an exponential amount of time for finding an optimal solution. One of the most successful techniques in recent years is local search. Local search algorithms are not exact but can reach near-optimal solutions quickly. They use simple operations such as swapping candidate vertices in and out of the current solution to reach neighboring solutions of the search space. Like with other meta-heuristics, from time to time the solution is perturbed which usually leads to a worse temporary solution but may open the path out of a local maximum. We focus on the local search algorithms of Andrade et al. [2] and Cai et al. [6] which are the current state-of-the-art algorithms for the MIS problem and VC problem, respectively. The former algorithm uses diversification techniques by Grosso et al. [13] as a base for departing from local maxima. There are also other works using the ILS algorithm of Andrade et al. [2] as basis for further improvements. Lamm et al. [16, 17] use evolutionary algorithms and graph partitioning. The evolutionary algorithms deal with holding more than one single current solution at once which produce offspring results with small changes. Then the profits of this new generation of results are examined to select members among them. The selected results then iteratively produce the following generations. These methods are combined with graph partitioning that allows to quickly exchange larger blocks of the independent sets. Lamm et al. [17] also use the reductions of Akiba et al. [1] to reduce the graph to its kernel without a loss of information considering the MIS. We describe these reductions in-depth in Section 4.3 where we use them ourselves.

1.4. Organization of this Thesis

This thesis is structured as follows: In Section 2 we introduce the definitions and terminology we use. The main two state-of-the-art algorithms for our topic, ILS and NuMVC, are described in detail in Section 3. The majority of this thesis is devoted to the cutting algorithms in Section 4, followed by our experiments in Section 5. They compare ILS and NuMVC and attempt to improve their performance on large sparse graphs with the cutting techniques. Lastly, concluding remarks are presented in Section 6.

2. Preliminaries

An *undirected graph* $G = (V, E)$ consists of a *vertex set* (*node set*) V and an *edge set* E, where each *undirected edge* $e = \{u, v\} \in E$, $u, v \in V$ connects vertex u with v and vice versa. The vertices u and v are called *endpoints* of e. If two vertices are connected by an edge they are called *adjacent*. We denote the number of vertices by $n = |V|$ and the number of edges by $m = |E|$. The *set of neighbors* (*neighborhood*) of a vertex $v \in V$ is defined as $N(v) = \{u \in V \mid \{u, v\} \in E\}$. Next, the neighborhood of a set S of vertices is defined as $N(S) = \{u \in V \mid u \in N(x), x \in S\} \setminus S$. When we want to include S in the neighborhood we use the definition $N[S] = N(S) \cup S$. The *degree* of a vertex $v \in V$ is defined as $deg(v) = |N(v)|$. We denote the maximum degree of a graph by Δ. An undirected graph is called *simple* if there exist no self loops or parallel edges. In addition, the *complement* of a graph G is defined as $\bar{G} = (V, \bar{E})$ where $\bar{E}$ is the set of edges not present in G. A graph $G' = (V', E')$ is called a *subgraph* of G if $V' \subseteq V$, $E' \subseteq E$ and every edge in E' connects two vertices $u, v \in V'$. The subgraph G' of G *induced by* $S \subseteq V$ is the graph $G' = (S, E')$ where $E' = \{\{u, v\} \in E \mid u, v \in S\}$. The *k-core* of an undirected graph G is the largest subset $V' \subseteq V$, such that $\deg(v) \geq k$ for all vertices v in the graph induced by V'. The *core number* of a vertex v is the highest number x such that v lies in the x-core.

Given a simple undirected graph $G = (V, E)$, an *independent set* is a subset $S \subseteq V$, such that there are no two adjacent vertices in S. An independent set is called *maximal* if it is not a subset of a larger independent set of G. The *maximum independent set* (*MIS*) *problem* is to find the independent set S in G with the highest cardinality $|S|$. The MIS problem is closely related to the *minimum vertex cover* (*VC*) *problem*, which is to find a subset $\tilde{S} \subseteq V$ with minimal cardinality, such that every edge $e \in E$ has at least on endpoint in $\tilde{S}$. If at least one of an edge's endpoints lies withing $\tilde{S}$, then the edge is called *covered*. Otherwise it is called *uncovered*. Given an optimal or approximate solution $\tilde{S}$ for the minimum VC problem, $S = V \setminus \tilde{S}$ is an optimal or approximate solution for the MIS problem and vice versa. It is easy to see that the minimum VC problem is $\mathcal{NP}$-hard as well. As a side note, another related problem is the *maximum clique problem* which is to find the largest complete subgraph of a graph (a subgraph of size k where each vertex has degree k-1). Given an MIS, the same set of vertices is a maximum clique in the complementary graph. While this has many applications we only focus on the MIS and VC problem in this thesis.

3. ILS and NuMVC

In this section we explain the main two state-of-the-art algorithms ILS [2] and NuMVC [6] which can be used to find large independent sets in practice and typically find exact solutions in smaller data sets.

3.1. Iterated Local Search (ILS)

The ILS algorithm by Andrade et al. [2] quickly finds near-optimal independent sets in large, sparse graphs. This local search algorithm uses (1,2)-*swaps* or 2-*improvements* to gradually increase the size of the current solution. This is done by taking one vertex out of the solution and inserting two vertices into the solution. In general, a (j, k)-swap takes out j vertices and inserts k vertices. A vertex is called k-*tight*, if exactly k of its neighbors lie in the solution. A 2-improvement can be applied if the inserted vertices x, y are 1-*tight* vertices with common neighbor v, which is to be removed from the solution. A 0-*tight* vertex is called *free* and can be added to the solution without any restrictions. A pseudocode representation of ILS and the 2-improvement can be seen in Algorithm 1. ILS stores the tightness for each vertex and updates it after each swap. The local search iterates over all vertices $v \in V$ and computes the set of 1-tight neighbors which, by definition, only have v as their neighbor in the solution. If there are at least two candidates which are not adjacent then the 2-improvement can be applied. Afterwards, the tightness of v's neighbors is decreased by one and the tightness of the inserted vertices's neighbors are increased by one.

The data structure used for maintaining the current solution is an array partitioned into three sections, see Figure 1. The first section holds the solution vertices followed by free vertices in the second section and non-free vertices in the last section. A non-free vertex has a tightness greater or equal to 1. The array is used as a permutation of the vertices therefore an additional array that holds every vertex's position within the permutation is needed. Furthermore, the sizes of the first and second of the permutation's sections are explicitly stored. This way, moving a vertex from one section to another can be performed in linear time by swapping it with a specific vertex and adjusting the size variables. The time for insertion or removal of a vertex into or out of the solution is proportional to its degree. The reason being that the algorithm needs to check every neighbor for adjusting its tightness and potentially moving it from the second to third section or vice versa.

Andrade et al. [2] proved that, given a maximal solution, which means that all free vertices have been inserted, one can find a 2-improvement or prove that none exists in $\mathcal{O}(m)$ time. This is due to the fact that every vertex is only looked at $\mathcal{O}(1)$ times and therefore the same holds for the edges. An implementation proposed by Andrade et al. [2] that we also use in this thesis is an incremental approach where they maintain a set of candidates for removal. They extend or reduce this set whenever the solution changes. Also, when it is clear that a vertex cannot be part of a 2-improvement, they discard it from the set so that they do not look at it again until the solution changes. This implementation also processes the available candidates at random. When the set of candidates becomes empty, no more 2-improvements can be made and the local search stops. In this case a local maximum is reached and the solution is called 2-*maximal*. Another potential improvement shown were (2,3)-swaps or 3-improvements. However, maintaining the necessary data structures for an incremental version of the 3-improvement would be more expensive and thus slower. Andrade et al. [2] therefore focus their implementation on the simpler 2-improvements.

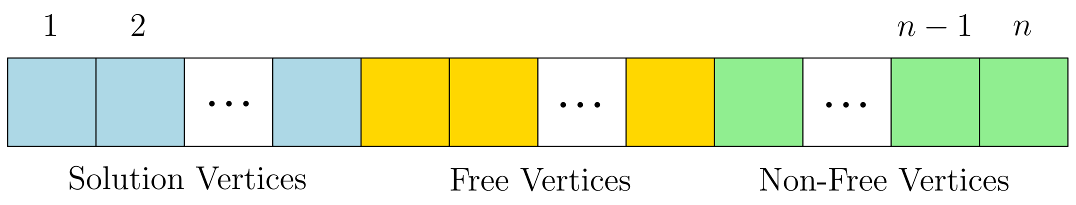

Figure 1: Vertex partition used by Iterated Local Search (ILS) algorithm. The first subset contains the current solution's vertices. The second subset contains vertices with a tightness of 0. The third subset contains vertices with a tightness greater or equal to 1.

The *Iterated Local Search* (ILS) algorithm computes a random solution and then runs in a loop until a stopping criterion is met. The loop consists of the steps perturbation, local search and potential acceptance of the new solution. In the first step, the algorithm perturbs the solution S in hope of escaping local maxima. This is done by the *force* routine which inserts k non-free vertices into the solution and removes their neighbors if they are in the solution. In most cases k is set to 1 but in rare cases (probability $1/(2|S|)$) the algorithm forces in more vertices. For diversification, in case there are more candidates for insertion the tie break favors the vertex that has not been in S for the longest time. They call this length of time the *age* of the vertex. Also, when the algorithm only forces one single vertex into S, it does not allow its removal until all other solution vertices have been tried unsuccessfully. At the end of this first step, the algorithm adds all free vertices to S if there are any. The second step is the iterative 2-improvement local search we already described. The final step is deciding whether the new solution S' computed in the previous two steps of this loop iteration becomes the new solution or is discarded. If $|S'| > |S|$ then ILS obviously takes the new better result. The algorithm is only allowed to go to a worse solution if the last $|S|$ iterations were unsuccessful and then does so with a probability $1/(1 + \delta \cdot \delta^*)$. δ is the difference between the two solution sizes and δ^* is the difference between $|S'|$ and the best solution size so far. This is to avoid straying from the current best solution too fast. Since it is possible to go to a worse solution, the best solution needs to be stored for an eventual output at the end. The permutation array may not hold the best solution when the algorithm stops. The stopping criterion for the loop used in most of the experiments in this thesis are either reaching a time limit or repeating the main loop a number of times.

Algorithm 1: Iterated Local Search (ILS)

```
Input: Graph $G = (V, E)$
Output: Independent Set $S$ of $G$
1 $S \leftarrow \emptyset$
2 $freeVertices \leftarrow V$
3 $nonFreeVertices \leftarrow \emptyset$
4 while stopping criterion is not met do
5     $S' \leftarrow$ Perturb($S$)
6     $S' \leftarrow$ TwoImprovements($S'$)
7     $S \leftarrow$ CheckNewSolution($S'$)
8 return $S$
```

Function Perturb(Solution S)

```
α ← random number within the interval $[1, 2 \cdot |S|]$
if $\alpha \neq 1$ then $k \leftarrow 1$ else $k \leftarrow i + 1$ with probability $1/2^i$, $(i \geq 1)$
$S' \leftarrow S$
if $k = 1$ then
    insert random vertex $x \in nonFreeVertices$ into $S'$          // force routine
    foreach $v \in N(x)$ do
        if $v \in S'$ then remove $v$ from $S'$
else
    select $\kappa$ random non-free vertices          // $\kappa$ is a global constant (here: 4)
    $x \leftarrow$ vertex out of the $\kappa$ random vertices that has not been in the solution for the
    longest time
    $N_2(x) \leftarrow$ non-solution 2-neighborhood of $x$          // vertices two steps away from $x$
    insert $k$ vertices from $N_2(x) \cup \{x\}$ into $S'$ // for details, see paper of Andrade et al. [2]
    remove all $v \in S'$ from $S'$ that are adjacent to the inserted vertices
insert free vertices into $S'$ until no free vertices remain
```

Function TwoImprovements(Solution S')

```
$candidates \leftarrow S'$
while $candidates \neq \emptyset$ do
    select $v \in candidates$ at random
    $L(v) \leftarrow$ set of 1-tight neighbors of $v$, sorted by ID
    if $|L(v)| < 2$ then $candidates \leftarrow candidates \setminus \{v\}$
    else
        find two non-adjacent vertices $x, y \in L(v)$
        if such a pair does not exist then $candidates \leftarrow candidates \setminus \{v\}$
        else
            $S' \leftarrow (S' \setminus \{v\}) \cup \{x, y\}$          // 2-improvement
            $candidates \leftarrow (candidates \setminus \{v\}) \cup \{x, y\}$
            foreach $u \in N(v)$ do
                if $u$ is 1-tight then
                    $z \leftarrow$ unique neighbor of $u$ in $S'$          // new candidate
                    $candidates \leftarrow candidates \cup \{z\}$
return $S'$
```

Function CheckNewSolution(Solution S')

```
if $|S'| > |S|$ then
    return $S'$
else
    return $S'$ with probability $1/(1 + \delta \cdot \delta^*)$ and $S$ otherwise
```

3.2. NuMVC

For the previously mentioned minimum vertex cover (VC) problem, the NuMVC algorithm by Cai et al. [6] has shown near-optimal heuristic results for large instances so far. For the pseudocode see Algorithm 2. We need the following additional definitions for the NuMVC algorithm. The *cost* of a candidate solution X of graph G with given weight function w is defined as cost(G,X) $= \sum w(\tilde{e})$ where $\tilde{e} \in E$ is an uncovered edge. Also, for each vertex $v \in V$, we denote the cover contribution of v as *dscore*(v) = cost(G,C) - cost(G,C') where C is the current candidate solution and $C' = C \setminus \{v\}$ if $v \in C$, or $C' = C \cup \{v\}$ otherwise. In other words, the value dscore(v) displays a guessed significance to the candidate solution if NuMVC changes the solution status of v.

NuMVC iteratively solves the k-vertex cover problem which aims at finding a vertex cover of size k. After greedily computing an initial solution, the goal is to decrease the solution size by one and exchange vertices until the solution is a proper vertex cover. For this, the paper proposes two main strategies by the names of *two-stage exchange* and *edge weighting with forgetting*. The two-stage exchange splits up the exchange process into the two separate stages *removing stage* and *adding stage*. First, in the removing stage a vertex from the candidate solution is removed in linear time. Then, an uncovered edge's endpoint is inserted into the solution in the adding stage, which also happens in linear time. Previous algorithms, as mentioned by Luo et al. [6], that tried doing both stages at once faced a quadratic time for finding a proper pair of vertices to exchange. Edge weighting is used to determine which vertex should be removed (by calculating the cover contribution). After each exchange every uncovered edge's weight increases by one. However, early events should not have a great impact much later on and mislead the search. Therefore, once the average of all edge weights reaches a threshold the algorithm reduces all the weights with a constant factor between 0 and 1.

The algorithm uses a vertex's cover contribution to determine whether it should be removed from the candidate solution C. Note that if vertex $v \in C$ it follows that dscore(v) ≤ 0 and dscore(v) ≥ 0 otherwise. For $v \in C$ with the lowest absolute cover contribution they consider v to be the least likely of the solution vertices to be part of the k-cover and therefore remove v. They also avoid running in circles between earlier local maxima. This is done by giving each vertex a *state*. A state of 0 represents that since the vertex's removal none of its neighbors have changed their state. A vertex with state 0 is not allowed to be inserted back into the solution. Whenever a vertex is removed or inserted, all of its neighbor's states are changed to 1.

In summary, after initializing all edge weights and states with 1 and dscore(v) = deg(v), NuMVC greedily constructs an initial vertex cover and saves it as current best solution C^*. This is followed by the main loop which the algorithm repeats until a specified time limit is reached. First, the algorithm checks whether the current candidate solution is a vertex cover. If that is the case NuMVC saves it as the new best solution C^* and then removes a vertex with the highest cover contribution from it. Afterwards it performs the two-stage exchange, breaking ties for the removal in favor of the vertex which has been in the solution for the longest time. After exchanging and updating the states the uncovered edges' weights are increased by 1. The weight forgetting may happen afterwards if the threshold is reached. After the loop NuMVC returns C^*.

Algorithm 2: NuMVC

```
Input: Graph $G = (V, E)$
Output: Vertex Cover of $C^*$ of $G$
1  weight($e$) $\leftarrow$ 1 for each $e \in E$
2  state($v$) $\leftarrow$ 1, dscore($v$) $\leftarrow$ deg($v$) for each $v \in V$
3  $C \leftarrow \emptyset$
4  insert a random uncovered edge's endpoint into $C$ until it is a vertex cover
5  $C^* \leftarrow C$
6  while *time limit not reached* do
7      if *there is no uncovered edge* then
8          $C^* \leftarrow C$
9          remove a vertex with the highest *dscore* from $C$
10         continue
11     TwoStageExchange ()
12     EdgeWeighting ()
13 return $C^*$
```

Function TwoStageExchange

```
1  select $u \in C$ with highest *dscore*                // tie break favors older vertex
2  $C \leftarrow C \setminus \{u\}$, $state(u) \leftarrow 0$
3  $state(z) \leftarrow 1$ for every $z \in N(u)$
4  select random uncovered edge $e$
5  select endpoint $v$ of $e$ with $state(v) = 1$ and highest *dscore*        // tie break with age
6  $C \leftarrow C \cup \{v\}$, $state(z) = 1$ for every $z \in N(v)$
```

Function EdgeWeighting

```
1  $weight(e) \leftarrow weight(e) + 1$ for each uncovered edge $e$
2  if *average edge weight reaches threshold* then
3      $weight(e) \leftarrow \lfloor \rho \cdot weight(e) \rfloor$ for each uncovered edge $e$, where $\rho \in$(0,1)
```

4. Cutting Algorithms

In this section we present the methods aiming at improving current algorithms for the MIS problem in both speed and result size. These are cutting techniques to ease the computation of large independent sets in difficult instances and methods to merge selected cut vertices back into the graph to obtain larger independent sets. Lastly, we perform exact reductions to simplify graphs as another method to improve computation time.

4.1. Cutting Vertices and Structures

Our intuition is that ILS could get stuck in local maxima more easily on large graphs. This could be due to vertices of high degree or other properties that we discuss in detail in this section. Another cause could be structures of multiple vertices. Slowdown on larger instances is more likely as some properties like very high vertex degrees are expected to be more frequent on these graphs. In this section we find out which specific vertices or subgraphs lead to this and remove them before we run ILS in hope of speeding up the computation and maybe even increase the solution's size.

Naive Cutting of High Degree Vertices

The first potential cause of slowdown of ILS that comes to mind are very high degree vertices. The algorithm is largely influenced by vertex degrees since the basic concept of the (1,2)-swap requires updating all neighbors' tightnesses. Moving around a high degree vertex and its neighbors in the algorithm's partitioning data structure described in Section 3.1 is expensive. Many real-world graphs follow a scale-free degree distribution. For a comparison of some of our used graphs' degree distributions, see Figure 2. This means that with an increase in vertex set cardinality the highest degrees are likely to increase as well. So, for the hard instances, despite being sparse graphs over all, these few very high degree vertices may slow down ILS significantly. This leads to our first algorithm being a naive removal of the highest degree vertices without updating degrees.

We do this by calculating the degree distribution within the graph. Since we do not update vertex degrees in this experiment, we can draw all the needed information from this distribution. This allows us to easily compute the degree λ such that all vertices $v \in V$ with $\deg(v) > \lambda$ can be safely removed and the vertices with $\deg(v) < \lambda$ can be safely kept. All that is left is cutting enough random vertices with exactly degree λ until the desired amount of cut vertices is reached.

Cutting Vertices According to Their Core Number

We now take into account that after removing a vertex from the graph, all of its neighbors' degrees decrease by 1. So instead of naively removing vertices according to their initial degree we cut them according to their core number. A k-core decomposition iteratively removes the minimum degree vertex to compute the graph's k-core. Since we want to cut the highest degree vertices instead we perform the k-core decomposition on the graph's complement. Thus, we achieve the complement of a k-core in the original graph which only contains vertices of degree less than or equal to $|V| - k$. For sparse original graphs k is close to $|V|$. This potentially decreases the upper bound of the remaining vertex degrees more than the naive cutting technique. As mentioned before, a (1,2)-swap involving a high degree vertex is expensive. Additionally, whenever ILS performs such a swap, the neighbors' likelihood

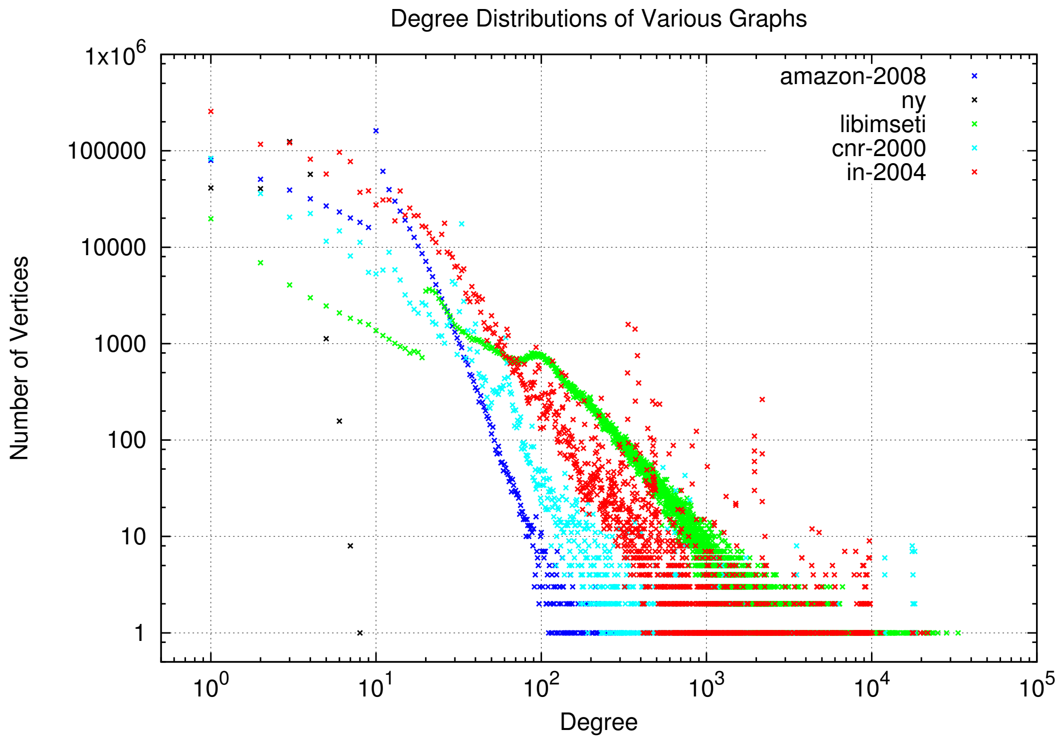

Figure 2: Degree distribution of some of the graphs we use in our experiments. Road map `ny` has no degree higher than 8 and `amazon-2008` does not get much higher than a few hundreds. We expect these two to not slow down ILS. However, we expect `libimseti`, `cnr-2000` and `in-2004` to cause slowdown as they contain several vertices with degrees well over 10,000.

Algorithm 3: Cutting High Degree Vertices

```
Input: Graph $G = (V, E)$, cutting factor $f$
Output: Cut Graph $G' = (V', E')$
unmark all $v \in V$
$k \leftarrow |V| \cdot f$                                   // amount to be cut
$\Delta \leftarrow$ maximum degree in $G$
$degDistribution : Array[1..\Delta]$ of int                  // initialize with zero
foreach $v \in V$ do degDistribution[deg($v$)]++
$\lambda \leftarrow \Delta + 1$
$sum \leftarrow 0$
for int $i$ from $\Delta$ to 0 do
    if $sum + degDistribution[i] > k$ then break
    $sum \leftarrow sum + degDistribution[i]$
    $\lambda \leftarrow i$
while more than $k$ vertices unmarked do
    mark random unmarked vertex $v \in V$ with deg($v$) $= \lambda$ as saved
return graph $G'$ induced by vertices marked as saved
```

for being in the next swap are high. The reason is that their change in tightness may allow new swaps that were not possible before. We therefore aim at minimizing the cut graphs highest degrees. To do so, we use a bucket priority queue and insert every vertex in the bucket corresponding to its degree. We then proceed to iteratively remove a highest degree vertex and put the removed vertex's neighbors in the next lower bucket. For this cutting technique we use functionality of *KaHIP*, a family of high quality partitioning programs by Sanders and Schulz [21]. It allows us to easily access various graph properties and iterate over vertices, neighbors and the like. KaHIP provides the bucket priority queue to easily cut highest degree vertices and update degrees after the removal.

A variation of this technique focuses on the tiebreak for removing vertices with the same degree. The version from above relies on the internal implementation of the queue's method for removing the highest degree vertex which performs a simple pop back at the highest bucket that is not empty. The bucket queue also uses push back whenever a vertex is inserted so by implementation the default tie break is in favor of the latest vertex inserted into the highest bucket. We introduce a more thought out tie break that relies on a dense neighborhood instead of the implementation. We already noted that a swapped vertex's neighbors are potentially the next candidates for a (1,2)-swap. In order to reduce the follow-up computation time, neighborhoods need to be less dense. If there are several vertices with the same highest degree, we compute the sum of their neighbors' degrees and break the ties in favor of the vertex with the highest sum. To save computation time we calculate these sums for all vertices first and only update them when removing a vertex.

Algorithm 4: Cutting High Degree Vertices with a Bucket Priority Queue

```
Input: Graph $G = (V, E)$, cutting factor $f$
Output: Cut Graph $G' = (V', E')$
1 $k \leftarrow |V| \cdot f$                                    // amount to be cut
2 $\Delta \leftarrow$ maximum degree in $G$
3 $bucketQueue \leftarrow$ new bucket_pq($\Delta$)              // call constructor
4 insert each $v \in V$ into the bucket corresponding to deg($v$)
5 while less than $k$ vertices removed from bucketQueue do
6     $v \leftarrow bucketQueue$.deleteMax()
7     foreach $w \in N(v)$ do
8         put $w$ in the next lower bucket
9 return graph $G'$ induced by remaining vertices in bucketQueue
```

Algorithm 5: Cutting High Degree Vertices with a Bucket Priority Queue (Tiebreak with Dense Neighborhood)

```
Input: Graph $G = (V, E)$, cutting factor $f$
Output: Cut Graph $G' = (V', E')$
$k \leftarrow |V| \cdot f$                                          // amount to be cut
$\Delta \leftarrow$ maximum degree in $G$
$neighborDegSums$ : $Array[1..|V|]$ of int                         // initialized with zeros
$bucketQueue \leftarrow$ new bucket_pq($\Delta$)                    // call constructor
foreach $v \in V$ do
    insert $v$ into the bucket corresponding to deg($v$)
    foreach $w \in N(v)$ do
        $neighborDegSums[w] \leftarrow neighborDegSums[w] + \deg(v)$
while less than $k$ vertices removed from bucketQueue do
    $v \leftarrow$ bucketQueue.DeleteDenseMax()
    foreach $w \in N(v)$ do
        put $w$ in the next lower bucket
        $neighborDegSums[w] \leftarrow neighborDegSums[w]$ - deg($v$)
return graph $G'$ induced by remaining vertices in bucketQueue
```

Function DeleteDenseMax

```
$size \leftarrow$ size of highest non-empty bucket
if $size = 1$ then
    $v \leftarrow bucketQueue$.deleteMax()
else
    $v \leftarrow u \in$ highest bucket with highest $neighborDegSums[u]$
    remove $v$ from $bucketQueue$
return $v$
```

Low Degree Reductions and Approximate Reductions

The next idea is based on the papers by Asgeirsson and Stein [3, 4] in which they present several techniques for the kernelization of graphs. Some of these techniques find vertices that are definitely in an MIS or definitely in a minimum VC. These vertices then require no further processing and are removed from the graph entirely thus making the remaining graph smaller. These methods are called exact reductions and remove easily evaluable vertices (of low degree) such that ILS does not waste computation time on them. If we only use exact reductions we can reconstruct an optimal vertex cover for the whole graph after computing an optimal vertex cover for the reduced graph. The same way, these reductions can be used for approximate results for the reduced graph which are optimally enhanced when reversing the reductions. We simply invert which cut vertices we remember for the final solution since we are aiming for an independent set and not a vertex cover. The exact reductions only take care of comparably unproblematic vertices, so additional reductions are necessary to really speed up ILS. Asgeirsson and Stein [3, 4] introduced approximate reductions which may lead to a worse result but further reduce the complexity of the graph and the total running time. The approximate reductions remove structures of multiple vertices and guarantee a worst case approximation ratio of 3/2. Asgeirsson and Stein [3] showed that the removal of structures like triangles or larger cycles leaves many low degree vertices in the remainder of the graph. The idea is to then repeat the exact reductions on the low degree vertices. Iterative use of as many exact reductions as possible followed by few approximate reductions hopefully results in the speed up of ILS we aim for. Following are the descriptions of the exact and the approximate reductions.

The exact reductions are only dependent on a vertex's degree. If the degree of a vertex $v \in V$ is 0, it is certainly part of an MIS. If $\deg(v) = 1$ then v is part of our MIS while its sole neighbor is cut completely. And lastly, if $\deg(v) = 2$ and its neighbors x,y are adjacent then there exists an MIS that contains v and neither x nor y. The reason behind this is that any vertex cover needs to either contain x or y. If we select one of them and remove the vertex along with all now covered incident edges, v becomes a degree 1 vertex. In such a case v would never be in an optimal vertex cover and thus, x and y are part of an optimal vertex cover. We therefore remove them completely and save v as an MIS vertex. As for approximate reductions, according to Asgeirsson and Stein [3], removing triangles is really effective for opening up possibilities for exact reductions. Since our approach is not to remove all triangles in the graph but only a few, we may select a vertex for removal that is not part of a triangle. In that case, we remove larger structures, namely 4-cycles and 6-cycles, if we find them. Although the cycles potentially create more low degree vertices after their removal, they take longer to find and are less likely. We later describe how to find the cycles during the implementation details. If our selected vertex is not part of these cycles either, we remove the neighbors if there are at most 3 as a last resort.

Our algorithm partitions the vertices into 3 subsets: undecided vertices, cut vertices that are completely removed, cut vertices that are in the MIS (in the following often called additional MIS vertices). For a visual representation, see Figure 3. Note that there can exist multiple different MIS and depending on the order of removal the last two subsets' contents may differ. Initially all vertices are undecided and we insert them into a bucket priority queue. Then follows the main loop which consists of the two sections *low degree reductions* and *structure removal*. We check after each cut if enough is removed from the graph and if that is the case we input the subgraph induced by the remaining undecided vertices into ILS. Also, we input the additional MIS vertices into ILS. So when ILS finishes computing the solution for the subgraph we simply add these additional vertices to the solution.

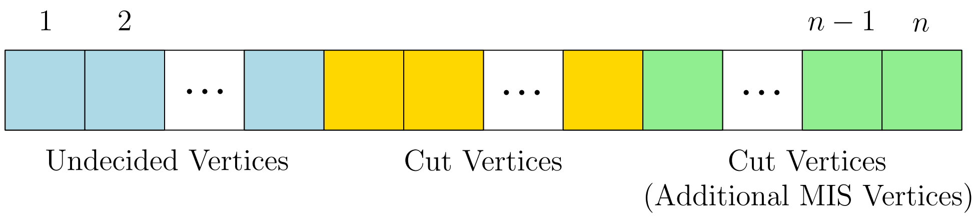

Figure 3: Vertex partition used by low degree reductions and approximate reductions cutting algorithm. Undecided vertices have not been investigated yet. Vertices from the second subset are cut entirely. Vertices from the third subset are added to the solution once ILS finished computation.

In the low degree reductions section we iterate over all undecided vertices and check their degree. If we iterate over all undecided vertices and successfully apply at least one exact reduction we start another iteration and otherwise go over to the structure removal. We are aware that the use of a priority queue capable of removing the lowest degree vertex speeds up the low degree reductions. However, as can be seen in Section 5, this cutting technique is not promising.

In the structure removal section we iteratively remove the maximum degree undecided vertex via the bucket queue's corresponding method. Since we are not aware of any rule for selecting a starting vertex we choose one with a low probability of being in an MIS. We then proceed to search for structures this vertex is part of. Even if we find none the vertex is still cut from the graph to make sure that we progress at all. If we find one, however, we immediately remove that structure and then continue with the next maximum degree vertex. First, we search for a triangle containing this vertex. If we do not find one we look for a 4-cycle, followed by a 6-cycle. These three are found by iterating over neighbors and their neighborhoods and finding a common neighboring vertex that connects two ends of a path. For example, to find a 4 cycle we look for a path of length 3 and a 4th vertex neighboring both ends of the path. If there is none of these structures but the vertex has only a degree of 2 or 3 we remove all of the neighbors. We continue these structure removals until we either removed a certain limit of structures or maximum degree vertices.

Algorithm 6: Low Degree Reductions and Approximate Reductions

```
Input: Graph $G = (V, E)$, cutting factor $f$
Output: Cut Graph $G' = (V', E')$, set of MIS vertices
1  $k \leftarrow |V| \cdot f$                                   // amount to be cut
2  $\Delta \leftarrow$ maximum degree in $G$
3  $bucketQueue \leftarrow$ new bucket_pq($\Delta$)              // call constructor
4  number of partition subsets $\leftarrow 3$
5  insert each $v \in V$ into subset $s_1$ and the bucket corresponding to deg($v$)
6  while less than $k$ vertices cut do
7      foreach $v \in$ subset $s_1$ do
8          if enough cut then leave main while loop
9          LowDegReduction ($v$)
10     if nothing cut in for each loop then
11         while not reached limit of iterations or removals do
12             StructureRemoval ()
13 return graph $G'$ induced by subset $s_1$, MIS vertices in subset $s_3$
```

Function LowDegReduction(v)

```
1  $d \leftarrow$ deg($v$)
2  switch $d$ do
3      case 0
4          put $v$ in partition subset $s_3$
5      case 1
6          put $v$ in partition subset $s_3$
7          put the sole neighbor of $v$ in partition subset $s_2$
8      case 2
9          if the neighbors $x$,$y$ of $v$ are adjacent then
10             put $v$ in partition subset $s_3$
11             put $x$,$y$ in partition subset $s_2$
```

Function StructureRemoval

```
1  $v \leftarrow$ bucketQueue.deleteMax()
2  search for a triangle $\lambda\{v, x, y\}$ and remove it from bucketQueue
3  if no triangle found then
4      search for a 4-cycle $\diamond\{v, w, x, y\}$ and remove it from bucketQueue
5      if no 4-cycle found then
6          search for a 6-cycle $\hexagon\{u, v, w, x, y, z\}$ and remove it from bucketQueue
7          if no 6-cycle found then
8              if deg($v$) $\in \{2, 3\}$ then
9                  remove neighbors of $v$ from bucketQueue
10 put all removed vertices in partition subset $s_2$
```

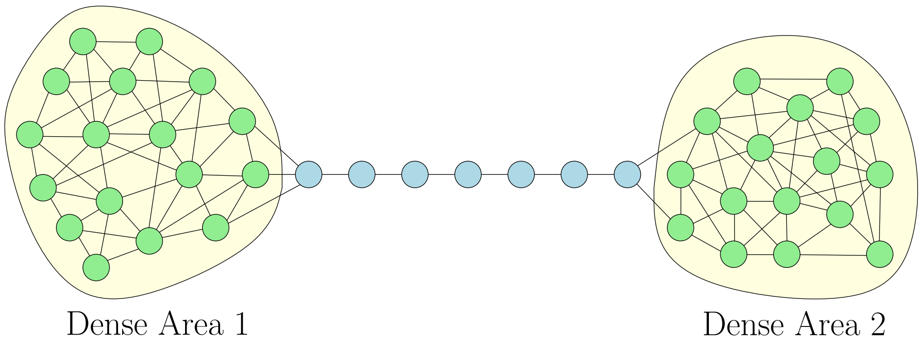

Figure 4: Two dense areas connected by a line of low degree vertices. Each blue vertex has a high betweenness centrality since they lie on every path between the dense areas. They are still very likely to be in the MIS.

Betweenness Centrality

The next algorithm is devoted to *betweenness centrality*. A vertex has a high betweenness centrality if it lies on many shortest paths between all of the graph's vertices. To be exact, the centrality value is

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the number of shortest paths between s and t and $\sigma_{st}(v)$ is the number of those paths that v lies on. We think that vertices with a high centrality have the potential to slow down ILS. Their central position in the graph may lead to ILS performing swaps more frequently on them if the local search circles in a central area. To hopefully prevent such circling we remove them in this experiment. However, the computation of all betweenness centrality values for general graphs is in $\mathcal{O}(n^3)$. It requires calculating the shortest paths between all pars of vertices with the Floys-Warshall algorithm [9]. This is too time consuming for large graphs. Since we aim at a low computation time for cutting vertices, we need a quick heuristic algorithm. Therefore, we use functionality of NetworKit by Staudt et al. [24], a toolkit for high-performance network analysis. To quickly calculate approximate betweenness centrality values we utilize NetworKit's *ApproxBetweenness2* class which performs Brandes' algorithm [11].

We try out two versions of this experiment. One is straight forward where we let NetworKit calculate the approximate centrality for all vertices and then cut a percentage of the most central vertices. We do so by obtaining a vector of vertex-centrality-pairs from NetworKit and then iterate over it. The other version focuses its cutting only among the highest degree vertices. So when iterating over the most central vertices we only remove those whose degree is high enough. A simple case where this should lead to much better results is a graph with two dense areas that are connected via a path of degree 2 vertices, see Figure 4. All of these vertices lie on all paths between the two areas. This causes their betweenness centrality to be very high. However, every second vertex of the path should be in an MIS so we do not want to remove the entire path.

Algorithm 7: Betweenness Centrality Cut

```
Input: Graph $G = (V, E)$, cutting factor $f$
Output: Cut Graph $G' = (V', E')$
1  $k \leftarrow |V| \cdot f$                                    // amount to be cut
2  betweenness ← ApproxBetweenness2($G$)    // constructor for betweenness algorithm
3  betweenness.run()
4  ranking ← betweenness.ranking()                  // vertex-centrality-pairs
5  for int $i$ from 1 to $k$ do
6      remove remaining vertex with highest centrality
7  return graph $G'$ induced by remaining vertices
```

Algorithm 8: High Degree Betweenness Centrality Cut

```
Input: Graph $G = (V, E)$, cutting factor $f$
Output: Cut Graph $G' = (V', E')$
1  $k \leftarrow |V| \cdot f$                                    // amount to be cut
2  betweenness ← ApproxBetweenness2($G$)    // constructor for betweenness algorithm
3  $\Delta \leftarrow$ maximum degree in $G$
4  bucketQueue ← new bucket_pq($\Delta$)                          // constructor
5  insert each $v \in V$ into the bucket corresponding to deg($v$)
6  $\lambda \leftarrow$ DegreeLimit()          // we cut only among 20% highest degree vertices
7  betweenness.run()
8  ranking ← betweenness.ranking()                  // vertex-centrality-pairs
9  for int $i$ from 1 to $k$ do
10     remove remaining vertex $v$ with deg($v$) $\geq \lambda$ and highest centrality
11 return graph $G'$ induced by remaining vertices
```

Function DegreeLimit

```
1  $\lambda \leftarrow 0$
2  sum ← 0
3  for int $i$ from 0 to $\Delta$ do
4      if sum + size of bucket $i > 0.8 \times |V|$ then
5          break
6      sum ← sum + size of bucket $i$
7      $\lambda \leftarrow i$
8  return $\lambda$
```

Clustering

Since all previous algorithms were applied to the graph as a whole we now perform the cutting on smaller components of the graph. We do this by clustering the graph. This means, that we partition the graph into dense subgraphs that are loosely connected. Each of these subgraphs, called clusters, may feature high degree vertices that can keep ILS in a local maximum. If we cut the highest degree vertices globally we may only cut vertices from specific sections of the graph while leaving other parts mostly untouched. To better distribute the cutting over the whole graph we cut a selected percentage of high degree vertices within each cluster. After running the clustering algorithm, we examine all vertices and determine their degree with regard to their cluster. According to these local degrees we construct a bucket priority queue for each cluster and cut the specified percentage of highest local degree vertices from every cluster. The subgraph induced by the remaining vertices is the input for ILS.

Algorithm 9: Cut in Cluster

```
Input: Graph $G = (V, E)$, cutting factor $f$
Output: Cut Graph $G' = (V', E')$
1 $clustering \leftarrow$ PLP($G$)                                  // constructor
2 $clustering$.run()
3 $clusters \leftarrow clustering$.getPartition()
4 foreach $c \in$ clusters do
5     HighDegreeClusterCutting ($c$)
6 return graph $G'$ induced by remaining vertices
```

Function HighDegreeClusterCutting(cluster c)

```
1 $k_c \leftarrow |c| \cdot f$                                        // amount to be cut in c
2 foreach $v \in c$ do
3     compute the degree $\deg_c(v)$ with regard to $c$
4 $\Delta_c \leftarrow$ maximum degree with regard to $c$
5 $bucketQueue \leftarrow$ new bucket_pq($\Delta_c$)                // constructor
6 insert each $v \in V$ into the bucket corresponding to $\deg_c(v)$
7 for int $i$ from 1 to $k_c$ do $bucketQueue_c$.deleteMax()
```

4.2. Inserting Cut Vertices

After discussing methods to simplify graphs by removing vertices before running ILS, we now have a look at inserting vertices back into the graph after ILS reached a decent result. When cutting vertices we may cut MIS vertices. Our goal for this section is to describe algorithms that take advantage of the cut, that is, achieving a speed up of ILS, and then consider the cut vertices. This boost should quickly deliver acceptable independent set sizes. When progression slows down we introduce cut vertices back into the graph such that ILS can then select from more vertices. We think that this method can lead to larger independent sets. We provide a larger input that contains both the cut graph that ILS starts with and the full graph as well as a list of the vertices that have been cut. This makes changing the input graph easier.

Insertion Only

First, we try different methods of inserting vertices. We compare the effects of inserting random cut vertices and inserting only the cut vertices with the lowest degrees. Inserting the lower degree vertices first has a higher chance of still leaving those vertices out that slow down ILS. It is unclear whether a gradual insertion or inserting everything at once is more effective to achieve our goal. On the one hand, inserting everything at once may cause ILS to instantly get stuck in a local maximum. On the other hand, gradual insertion may be too slow, canceling the advantage we get from cutting. For this reason, we try a number of different insertion speeds to find the sweet spot. After cutting a small portion of the graph we insert the cut vertices either randomly or ordered. An ordered insertion meaning that we choose the cut vertices with the lowest degree. The insertion speeds are described in detail in Section 5.

Algorithm 10: Insert Cut

```
Input: Graph $G = (V, E)$, Subgraph $G' = (V', E')$ of $G$,
    Set $C$ of cut vertices ordered by degree
Output: Independent Set $S$
1  compute cut graph $G'$
2  run ILS on $G'$ until progression slows down to get solution $S$
3  map IDs of $S$ from $G'$ to $G$
4  while more insertions to be done do
5      $G'' \leftarrow$ Insert($G'$, $C$)
6      map IDs of $S$ from $G$ to $G''$
7      if more insertions to be done then
8          run ILS on $G''$ starting with $S$ until progression slows down
9      else
10         run ILS on $G''$ starting with $S$ until stopping criterion met
11     map IDs of $S$ from $G''$ to $G$
12 return $S$
```

Function Insert(graph G', set of vertices C)

```
1  if using random insertion then
2      select vertices verts from $C$ at random                // selection increases in size
3  else
4      select lowest degree vertices verts from $C$            // selection increases in size
5  $G'' \leftarrow$ graph induced by $G' \cup verts$
6  return $G''$
```

Swapping In and Out

In the next algorithm we do not only insert vertices but also occasionally swap them with other cut vertices to hopefully escape local maxima. A random insertion potentially inserts all the vertices that keep ILS in one of these maxima. Thus, we think repeatedly changing which cut vertices are taken back into the graph eventually leads to a near-optimal composition of vertices. We try to exploit these coincidental benefits whenever possible. The algorithm is otherwise close to identical to the insertion-only algorithm. After the initial computation of ILS we insert most of the cut vertices randomly and keep this size until the end of the algorithm. However, instead of more insertions we take out the inserted vertices again and insert the same amount of cut vertices randomly yet again. If we remove vertices that have already been part of the best solution so far, we need to save that result in case it remains the best solution until the end.

Algorithm 11: Swap Cut

```
Input: Graph $G = (V, E)$, Subgraph $G' = (V', E')$ of $G$,
       Set $C$ of cut vertices ordered by degree
Output: Independent Set $S$
1  compute cut graph $G'$
2  run ILS on $G'$ until progression slows down to get solution $S$
3  map IDs of $S$ from $G'$ to $G$
4  $S' \leftarrow S$
5  while time limit not reached do
6      select vertices verts from $C$ at random              // always select the same amount
7      $G'' \leftarrow$ graph induced by $G' \cup verts$
8      map IDs of $S$ from $G$ to $G''$
9      run ILS on $G''$ starting with $S$ until progression slows down or time limit reached
10     map IDs of $S$ from $G''$ to $G$
11     if $|S| > |S'|$ then $S' \leftarrow S$
12 $S \leftarrow S'$
13 return $S$
```

4.3. Exact Reductions

The final set of algorithms is about performing exact reductions that can be reverted when done computing on the reduced graph. We use the exact reductions presented by Akiba et al. [1]. The reduction algorithm greatly reduces the size of the graph.

As for the simpler reductions, one of them is the removal of a vertex with degree 1 and its neighbor. As seen with the low degree reductions earlier, the degree 1 vertex is in an MIS. Moreover, if a vertex v with $\deg(v) = 2$ has two non-adjacent neighbors u,w then all three are reduced to a vertex v'. Now, $N(v') = N(u) \cup N(w) \setminus \{v\}$. If v' is in an MIS of the reduced graph then $\{u, w\}$ is in an MIS of the original graph. If v' is not in an MIS, however, then v is in one. Another simple reduction can be applied when a vertex v's neighborhood lies completely in another vertices' neighborhood. In this case v is called *dominated* and part of an MIS. These simple reductions are illustrated in Figure 5.

The more complex reductions have the names *LP relaxation*, *unconfined*, *twin*, *alternative* and *packing*.

(i) LP Relaxation:

As for LP relaxation, the MIS problem can be formulated as a linear programming relaxation with the solution values $x_v \in \{0, \frac{1}{2}, 1\}$ [20]. It can be solved with bipartite matching, that is, maximize $\sum_{v \in V} x_v$, where $x_u + x_v \leq 1$ for $u, v \in V$ and $x_v \geq 0$ for $v \in V$. A vertex v with $x_v = 1$ lies in the MIS and is therefore removed from the graph.

(ii) Unconfined [26]:

Whether a vertex v is *unconfined* is determined by the following small program: Let $S = \{v\}$ initially. Now, search for $u \in N(S)$ where $|N(u) \cap S| = 1$ and $|N(u) \setminus N[S]|$ minimal. If no such u can be found, then v is confined. If $N(u) \setminus N[S] = \emptyset$, then v is unconfined. If $N(u) \setminus N[S] = w$, then insert the single vertex w into S and repeat the program. If none of the above cases apply, then v is confined. Unconfined vertices can be removed from the graph as there exists an MIS that contains no such vertices.

(iii) Twin [26]:

Two vertices u, v are called a *twin* if their neighborhoods are equal and u, v have degree 3. If the graph induced by $N(u)$ has any edges, add u, v to the independent set and remove $u, v, N(u)$ from the graph. Otherwise, reduce $u, v, N(u)$ to the new vertex w that is connected to the remains of the set of vertices at distance 2 from u. These are all the vertices that are adjacent to $u, vorN(u)$ but not part of them. If w is in the independent set for the reduced graph, then $N(u)$ is in the independent set for the original graph. Otherwise, u, v are in the independent set.

(iv) Alternative [26]:

Two sets of vertices A, B are called *alternative* if $|A| = |B| \geq 1$ and there exists an MIS S with $S \cap (A \cup B) = A$ or $S \cap (A \cup B) = B$. Akiba et al. [1] introduce two structures that are alternatives, namely *funnel* and *desk*. The vertex sets $\{u\}$ and $\{v\}$ are called a funnel if u, v are adjacent and $N(v) \setminus \{u\}$ induces a complete graph. The sets $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$ are called a desk if the following things hold: $a_1 b_1 a_2 b_2$ is a chordless 4-cycle, the 4 vertices have degree of at least three, $N(A) \cap N(B) = \emptyset$, $|N(A) \setminus B| \leq 2$ and $|N(B) \setminus A| \leq 2$. Given two alternative sets A, B, the sets A, B and $N_\cap = N(A) \cap N(B)$ can be removed from the graph. Afterwards, each $a \in \bar{A} = N(A) \setminus N_\cap$ is connected with each $b \in \bar{B} = N(B) \setminus N_\cap$ via an edge. If $\bar{A}$ has vertices in an MIS of the reduced graph, then B has vertices in an MIS of the original graph, analogous for $\bar{B}$ and A.

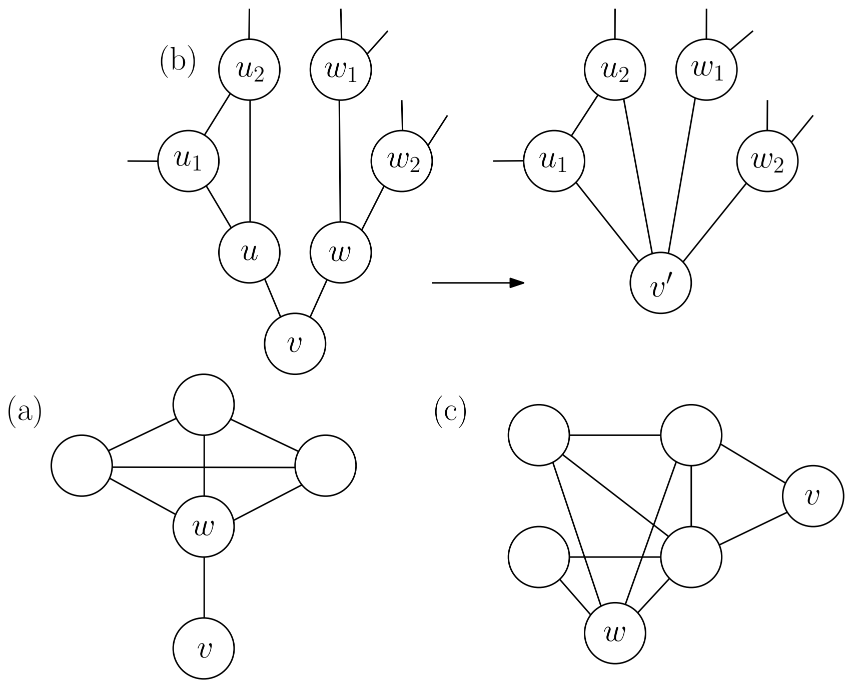

Figure 5: Simple reductions. (a) Degree-1 vertex v is in an MIS while neighbor w is in a minimum VC. (b) Degree-2 vertex v has two non-adjacent neighbors u, w. We remove the three vertices and introduce the new vertex v' which has the combined neighborhoods of u and w. If v' is in an MIS for the reduced graph then u, w are in an MIS of the original graph. Otherwise v is in an MIS. (c) v's neighborhood lies completely in w's neighborhood so v is in an MIS.

(v) Packing:

Finally, for the packing reduction, Akiba et al. [1] introduce constraints $\sum_{v \in S} x_v \leq k$, where vertex set S is non-empty and $x_v = 0$ if v is part of an MIS and 1 otherwise. A package constraint is added when a vertex is definitely excluded or included from the independent set (e.g., if the vertex is unconfined or confined). If vertex v is excluded from it, x_v is removed from the constraints and k reduced by 1. In the easy case $k = 0$, the graph induced by S must only contain vertices of degree 0 and therefore S is in an MIS and can be removed alongside $N(S)$. The other cases are complex and described in detail in the paper by Akiba et al. [1].

Many of the instances tested can be reduced to an empty graph. In this case, simply reverting the reductions automatically leads to an optimal independent set. Like the previous set of algorithms about insertions, we combine the reductions algorithm with our code of ILS.

Reductions Only

This basic algorithm consists of reducing either the complete graph or a cut version and then letting ILS run on the reduced graph. The reduction reduces the graph to its kernel such that ILS does not waste time on parts of the graph that can easily be evaluated. Cutting beforehand may further simplify the graph to increase the speed up of ILS. We then proceed to run ILS normally until the specified time limit is reached or the main loop repeats a certain number of times. Afterwards, we extend the independent set found for the reduced graph to an independent set for the whole graph by reverting the exact reductions.

Algorithm 12: Exact Reductions

```
Input: Graph $G = (V, E)$
Output: Independent Set $S$
1 read the graph $G$
2 $adjVector \leftarrow$ adjacency vector for $G$
3 $redAlg \leftarrow$ reductions_algorithm($adjVector$)          // constructor
4 $G' \leftarrow redAlg$.reduce_graph($G$)
5 get ID map from $G'$ to $G$
6 run ILS on $G'$ until stopping criterion is met to get solution $S$
7 map IDs of $S$ from $G'$ to $G$
8 return $S$
```

Reduction Swap

Expanding on the previous algorithm we now let ILS run until it reaches results similar to those of the reductions-only algorithm. Afterwards, we perturb the current solution by taking vertices out of it to hopefully leave local maxima. After the perturbation we select a subgraph for which we find the MIS exactly and add it to the perturbed solution. These steps are repeated until a stopping criterion is met and are now described in more detail: We perturb the current solution (with one of four techniques described later) and then construct the subgraph that contains all of the reduced graph's vertices that are neither in the current solution nor its neighborhood. The neighborhood cannot contain new solution vertices as their tightness is greater than zero. This way, an independent set computed for this subgraph is guaranteed not to be adjacent to the perturbed solution. Thus, we can simply add it to the perturbed solution without any complications. We get the subgraph's MIS exactly by using the reduction algorithm which reduces the subgraph to its kernel. After adding this MIS to the perturbed result we check whether our new solution translates to a larger independent set in the original graph. We extend it with the reduction algorithm to check its true size by reverting the original reductions. If we reached a new best size, we store the large independent set of the original graph explicitly and then repeat the loop on the reduced graph. In short, the difference to the reductions-only algorithm is to change the strategy after reaching similar result sizes. We think that difficult instances have kernels that can keep ILS in local maxima in such a way that our previous strategies are not enough to escape. For this reason we make larger leaps to more distant solutions and use the reduction algorithm for computing a subgraph's MIS instead of ILS. We try the following four variations for the solution perturbation.

k-Core Removal

The first removal technique iteratively removes a percentage of vertices with the highest degree from the solution. This is very similar to the second cutting technique we introduced earlier. We think that these vertices are the unlikeliest of the current solution vertices to be true MIS vertices. The perturbation repeatedly selects the highest degree solution vertex, marks it as a non-solution vertex and then updates the neighbors' degrees. Note that in this context, removal means taking a vertex out of the solution and not cutting it completely from the graph.

Neighborhood Removal

The second version additionally removes solution vertices that are two steps away from the vertices cut in the first version. We cannot remove the direct neighbors since they cannot become part of the solution. If we only remove the vertices we consider difficult to process, like in the first technique, the resulting subgraph may be too small. A small graph has a low chance of containing more solution vertices than what we had before. Therefore, the additional vertices added to the subgraph in this version may open up more possibilities for detecting new solution vertices.

(1,2)-Swaps

The third version, in addition to what we do in the second one, performs a couple of (1,2)-swaps. The swaps decrease the subgraph's size and add some vertices to the perturbed solution prior to running the reduction algorithm. This is to reduce the computation time required for exactly finding the subgraph's MIS. While the first reduction swap may remove too few vertices from the solution, the second one may remove too many. We therefore try to find a middle ground here. We chose to perform half as many swaps as we removed high degree vertices.

Random Removal

Finally, the forth version does none of the above but only removes a number of random solution vertices. We cannot predict the structure of the reduced graph. Thus, the other three variations potentially construct subgraphs which's MIS contains exactly the vertices they removed. If this is the case, the three techniques repeat the same steps without ever progressing. To avoid this from happening we try a probabilistic approach.

Algorithm 13: ReductionSwap

```
Input: Graph $G = (V, E)$, Type $type$ of swapping, solution cutting factor f
Output: Independent Set $S$
1  read graph $G$
2  $adjVector \leftarrow$ adjacency vector for $G$
3  $redAlg \leftarrow$ reductions_algorithm($adjVector$)                    // constructor
4  $G' \leftarrow redAlg$.reduce_graph($G$)
5  run ILS on $G'$ until progress slows down significantly to get solution $S$
6  $S^* \leftarrow S$                                                        // best solution so far
7  map IDs of $S^*$ from $G'$ to $G$                                         // extend the solution
8  while time limit not reached do
9      Swapping($type$)
10     $V'' \leftarrow$ all non-solution vertices $v \notin N(S)$
11     $E'' \leftarrow (V'' \times V'') \cap E'$                             // $E'$ is edge set of $G'$
12     $localAdjVector \leftarrow$ adjacency vector for $G'' = (V'', E'')$
13     $localRedAlg \leftarrow$ reductions_algorithm($localAdjVector$)
14     $S_{add} \leftarrow localRedAlg$.solve()                             // get exact MIS for subgraph
15     $S \leftarrow S \cup S_{add}$
16     $S' \leftarrow S$
17     map IDs of $S'$ from $G'$ to $G$                                      // extend the solution
18     if $|S'| > |S^*|$ then $S^* \leftarrow S'$
19 $S \leftarrow S^*$
20 return $S$
```

Function Swapping(int *type*)

```
if $type \neq 4$ then
    $highDegVerts \leftarrow$ SwapA ()
    if $type \in \{2, 3\}$ then
        SwapB ()
        if $type = 3$ then
            SwapC ()
else
    SwapD ()
```

Function SwapA

```
$k \leftarrow |S| \cdot f$                                      // amount to be cut
$\Delta \leftarrow$ maximum degree in $G'$
$bucketQueue \leftarrow$ new bucket_pq($\Delta$)                // constructor
insert each $v \in S$ into the bucket corresponding to deg($v$)
$highDegVerts \leftarrow \emptyset$
while less then $k$ vertices removed do
    $v \leftarrow bucketQueue$.deleteMax()
    mark $v$ as non-solution vertex
    $highDegVerts \leftarrow highDegVerts \cup \{v\}$
    put each $w \in N(v)$ in the next lower bucket
return $highDegVerts$
```

Function SwapB

```
$TwoStepNeighbors \leftarrow \emptyset$
foreach $v \in S$ do
    if $\exists w \in highDegVerts$: $v$ is two steps away from $w$ then
        $TwoStepNeighbors \leftarrow TwoStepNeighbors \cup \{v\}$
foreach $v \in TwoStepNeighbors$ do
    mark $v$ as non-solution vertex
```

Function SwapC

```
perform (1,2)-swaps until predefined limit reached or no more swaps possible
```

Function SwapD

```
$k \leftarrow |S| \cdot f$                                      // amount to be cut
mark $k$ random solution vertices as non-solution vertices
```

5. Experiments

In this section we present experiments aiming at evaluating the current algorithms for the MIS problem as well as our presented techniques. We first compare the two leading algorithms for this subject in depth. Following are cutting techniques to ease the computation of large independent sets in difficult instances and methods to merge selected cut vertices back into the graph to obtain larger independent sets. Lastly, we perform exact reductions to simplify graphs as another method to improve computation time. We discuss the cutting algorithms in the same order as in the previous section. Each experiments consists of 5 runs with a time limit of 2 hours each unless stated otherwise. The seeds for pseudo-random numbers run from 1 to 5 during the presented experiments. For converting graphs from the DIMACS format to the Metis format used by our cutting algorithms, we apply a conversion algorithm by Sebastian Lamm. We compute average value curves as follows: We measure the time whenever an algorithm improves its solution size during each run and merge the results. The values are then ordered by result size and we compute the mean time value for each result size. The curve represents these mean values. Steps that would decrease the curve's current result size are omitted such that the curve is monotonous. All implementations are written in C++ and compiled with gcc version 4.9.2 (using the -03 flag). Each run was performed on one core of a 4 x AMD Opteron 6168 1.9 Ghz (12-Core) CPU with 256GB of RAM running Ubuntu 12.04.

5.1. Graph Instances

The graph instances used in our experiments are from four different families, namely ROAD, WEB, MESH and SOCIAL. All graphs are undirected and unweighted.

The ROAD family consists of road networks from various countries and are thus large but very sparse graphs. They are unlikely to have high degree vertices. They are from the work of Andrade et al. [2] as well as the 9th DIMACS Implementation Challenge for Shortest Paths[1]. The importance of road networks with regard to the MIS problem lies in map labeling algorithms.

Next, the WEB family contains very large real-world graphs used in the works of Akiba et al. [1]. They are, among others, provided by the Stanford Large Network Dataset Collection[2] and the Laboratory for Web Algorithmics[3]. This family of graphs is especially important for our experiments as it contains instances we consider difficult for the ILS algorithm such as `cnr-2000` and `in-2004`.

The MESH family contains dual-graphs of well-know meshes that have been triangulated. A dual graph has a vertex for each of the original graph's faces which in this case are always triangles. Whenever two faces of the original graph share the same incident edge, the corresponding vertices are adjacent in the dual graph. This family of graphs was also used by Andrade et al. [2]. Their importance with respect to the MIS problem lies in the processing of triangulations. This requires to find a small set of triangles such that every edge is incident to at least one of the set's triangles. Almost all vertices of this family's graphs have degree three. If our hypothesis about high degree vertices slowing ILS down is correct we should be able to detect that these graphs do not cause a slowdown.

Finally, the SOCIAL family consists of social networks which are increasingly promi-

[1]http://www.dis.uniroma1.it/challenge9/download.shtml
[2]https://snap.stanford.edu/data/
[3]http://law.di.unimi.it/datasets.php

nent in real-world data sets. The family features graphs from the Stanford Large Network Dataset Collection[4] and the works of Akiba et al. [1].

As a side note, we also performed preliminary tests on graphs from the 2nd DIMACS Implementation Challenge which are used by Andrade et al. [2]. However, they are mostly very small graphs that are not very interesting for our main experiments since they can mostly be solved exactly and do not cause a slowdown of ILS. We present the final results for the graphs for which we run all experiments in Tables 2, 3, 4, 5 and 6.

5.2. Comparison of ILS and NuMVC

In order to properly compare the two algorithms we need to invert NuMVC's result from a vertex cover to an independent set. To compare the result sizes we simply output $|V| - |S|$ for NuMVC where S is its resulting vertex cover. We first run ILS and NuMVC on the graph instances without any preconditioning for two hours each. Then, the measured result sizes and computation times are the new stopping criteria. This means, if ILS reached size x and NuMVC reached y, we run ILS until result size y and NuMVC until x to better compare computation times. If the result size is not reached within the two hour timelimit the algorithm stops. Then, we do the same with the time values we got from the first set of experiments to better compare results sizes. These values are the average computation time needed for a run's final result size. Note that for time measurement in any experiment, the time for reading the graph in both ILS and NuMVC is excluded. We did the comparison on the mesh `dragon`, the road map `ny` and the social network `libimseti`.

Figures 6, 7 and 8 show the average progression of both algorithms over time. The plots for our 3 different stopping criteria show hardly any differences so we only show the plots for the 2-hour time limit. It is easy to spot that the computation of the initial solution is usually quicker in ILS. ILS also often reaches a decent solution quicker than NuMVC computes the initial solution. Once ILS plateaus, no more significant jumps in solution size are to be expected. In comparison, NuMVC abruptly reaches a first plateau worse than the one of ILS and struggles to get out of it. Only after a longer time span, more improvements are detected that eventually lead to similar solution sizes as ILS. Our assumption is that the lack of diversification methods keep NuMVC within local maxima. Each iteration of NuMVC's main loop only exchanges single vertices. On the other side, ILS occasionally forces a larger number of non-solution vertices into the solution for greater perturbation. This can be seen in function TwoStageExchange, line 6 of NuMVC and function Perturb, line 12 of ILS. As for the result sizes, we found out in preliminary tests that in small graphs, that is, graphs with less than 5,000 vertices, NuMVC was slightly ahead of ILS. These were graphs of the 2nd DIMACS Implementation Challenge as mentioned earlier. NuMVC was more likely to reach optimal results or results slightly better than those of ILS. However, with growing numbers of vertices ILS takes the lead and outperforms its competitor on very large graphs. In addition, the much quicker computation of comparable result sizes makes ILS our clear winner in this comparison. We therefore focus on ILS in all of the following experiments.

[4]ttps://snap.stanford.edu/data/

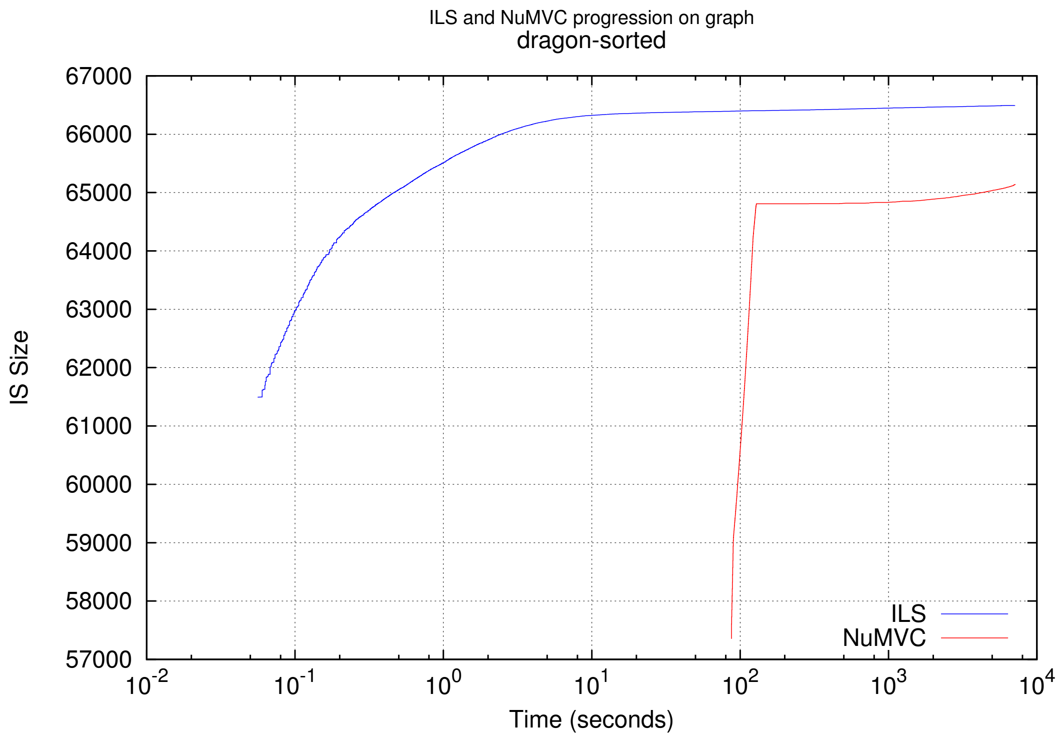

Figure 6: Comparison of ILS and NuMVC on mesh `dragon`.

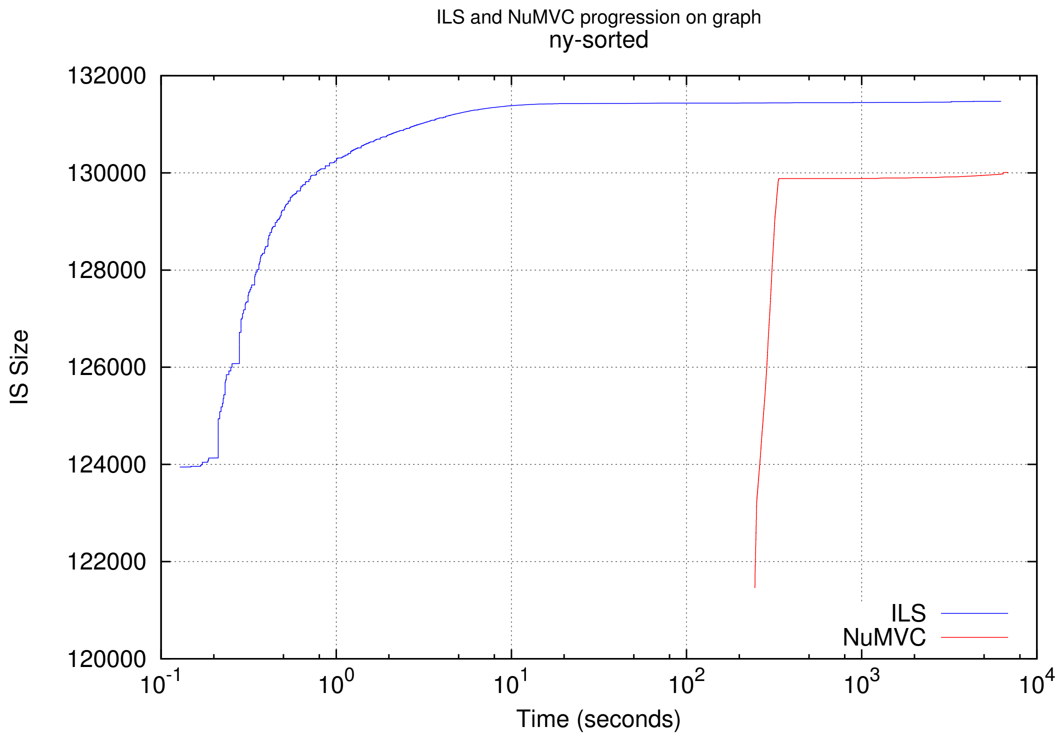

Figure 7: Comparison of ILS and NuMVC on road map `ny`.

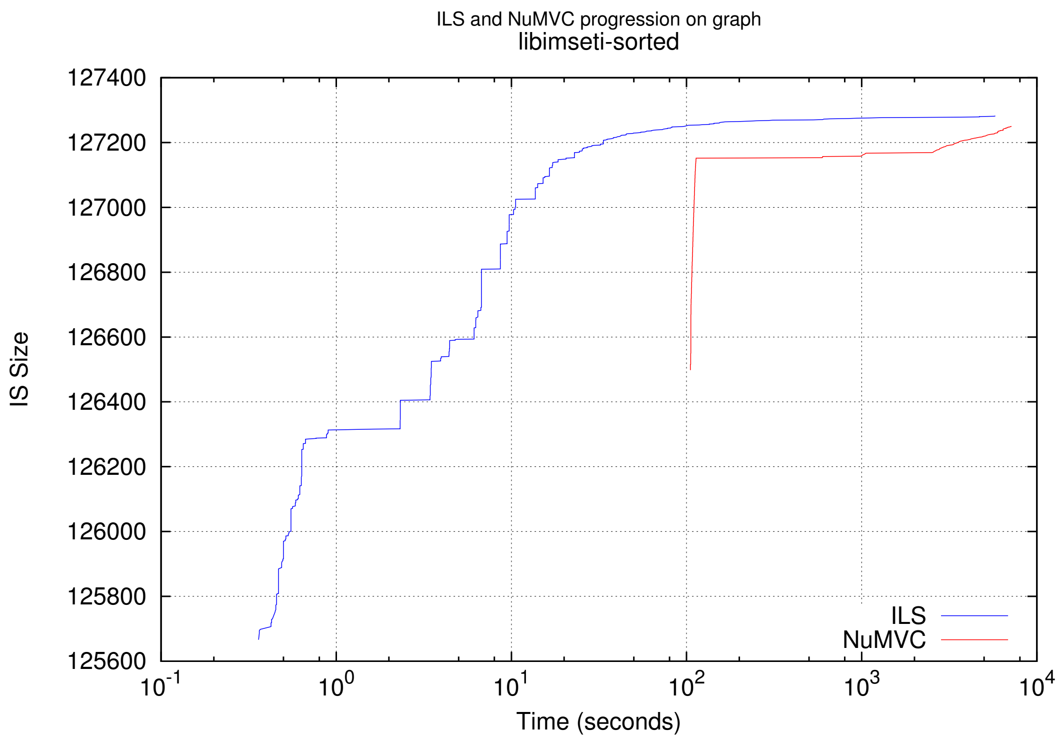

Figure 8: Comparison of ILS and NuMVC on social network `libimseti`

5.3. Cutting Vertices and Structures

The experiments of this section are performed in two parts. First, we cut a percentage of the graph's vertices according to the techniques we discussed. They are the naive cut of high degree vertices, the k-core cut, approximate reductions, betweenness centrality cut and the cut in clusters. Afterwards, we input the cut graph into the ILS algorithm and measure the progression of the solution size over time. We performed preliminary experiments indicating that cutting too many vertices reduces the final independent set's size too much. Hence we focus on the cutting percentages of 1%, 5% and 10%. However, in most cases even 10%-cuts cause unsatisfactory outcomes and are thus excluded from most plots. Note that in all experiments that use cutting techniques, including experiments from the later sections, the cutting time is included in our time measurements.

Naive Removal of High Degree Vertices:

As anticipated, ILS has problems with progressing on some instances with very high degree vertices. The various runs may differ much more than in easier cases and the representing curves are not as steep. For better insight we show the 5 runs in separate curves for this experiment to show the differences between them. On some graphs like `web-Stanford`, `in-2004` and `cnr-2000`, the cutting makes the runs of ILS much more consistent. On the other side, the runs of ILS without cutting beforehand have varying speeds of progression. In most cases the different runs' curves cannot be distinguished at all. Since the direct influence of the cutting on the progression is easily visible, we conclude that very high degree vertices are the main reason for ILS's worse progression on some very large graphs. The amount of removed vertices is also directly correlated with the final result sizes. While cutting one percent on difficult instances leads to results close to the optimal or best know

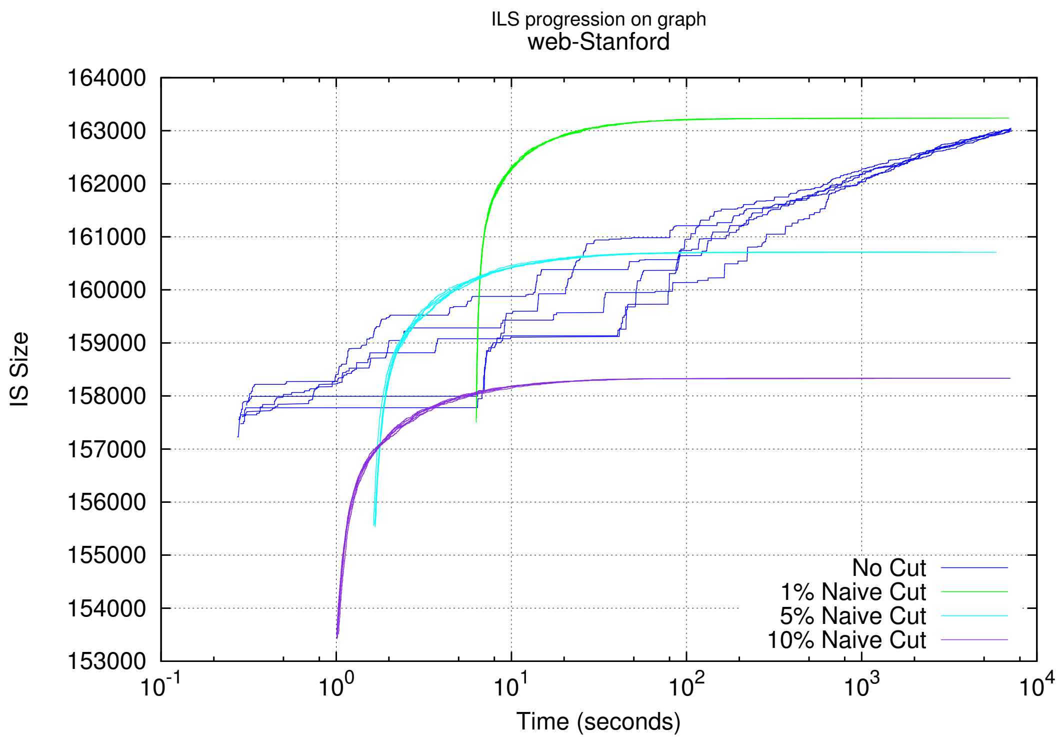

Figure 9: Improvement of ILS on graph `web-Stanford` after naive cutting. The 5 runs are much more consistent after the cut and reach their plateau earlier.

results, cutting 5% or more clearly removes many MIS vertices. In some plots we do not display the curve for 10%-cutting as it has much lower quality, and makes the plots difficult to interpret. This shows that even the set of 5% highest degree vertices contains thousands of MIS vertices in the tested graphs. The final results of the 1%-cut are decently sized but still leave room for improvement. We aim to get closer to the best known results with our other cutting techniques.

Although improvements on many graphs are promising, not all graphs are suited for this cutting method. The results for the mesh `dragon` and the road map `ny` worsen for each cut. Running ILS without cutting does not show any significant slowdown and the runs are already very consistent. The average degrees of these two graphs are 3.0 and 2.8 respectively. All of `dragon`'s vertices have degree 3 and degrees in `ny`'s vertex set are not higher than 8. For this reason, any of the vertices has a similar probability of being part of an MIS. As a result, any cutting removes MIS vertices and worsens the size of the final independent set without causing any speed up. This further supports our hypothesis that very high degree vertices are the main source of ILS's slowdown. Preliminary tests showed that even graphs with a highest degree of over one thousand, such as `amazon-2008` (1,077) or `citationCiteseer` (1,318), do not benefit from cutting. To effectively use our methods, a graph should have hundreds of vertices with a degree higher than one thousand. For instances following a power law mostly seen in real-world data this roughly translates to highest degrees of well over 10,000 or higher.

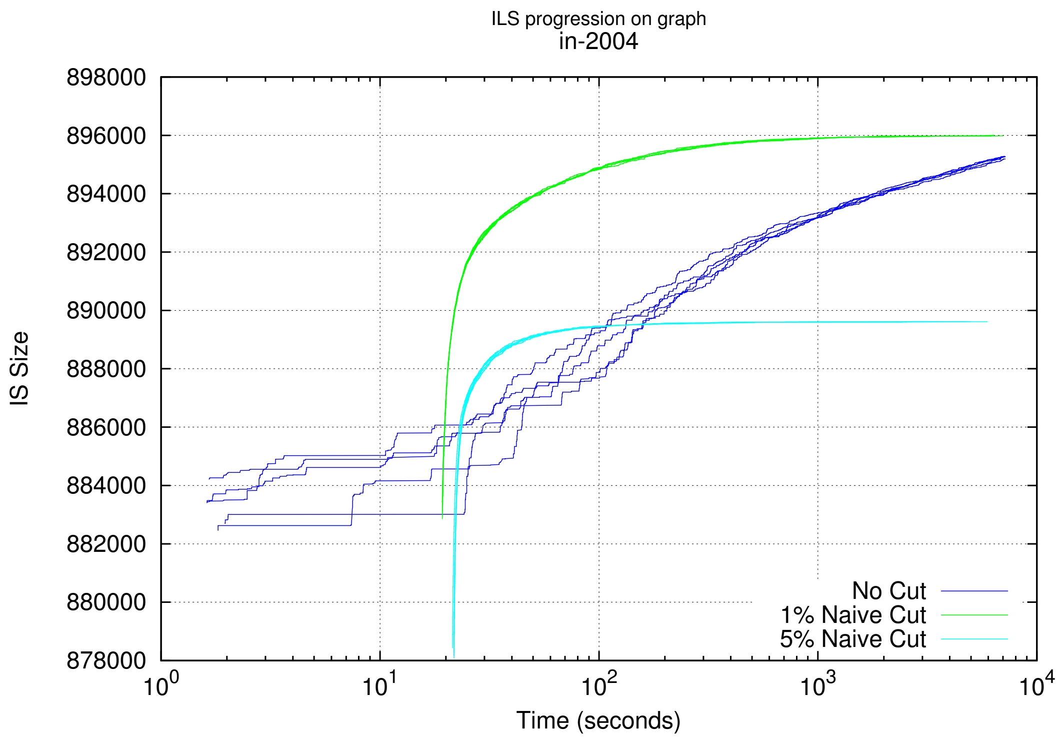

Figure 10: Improvement of ILS on web graph `in-2004` after naive cutting. The 5 runs are much more consistent after the cut and reach their plateau earlier. Cutting 10% leads to unsatisfactory results.

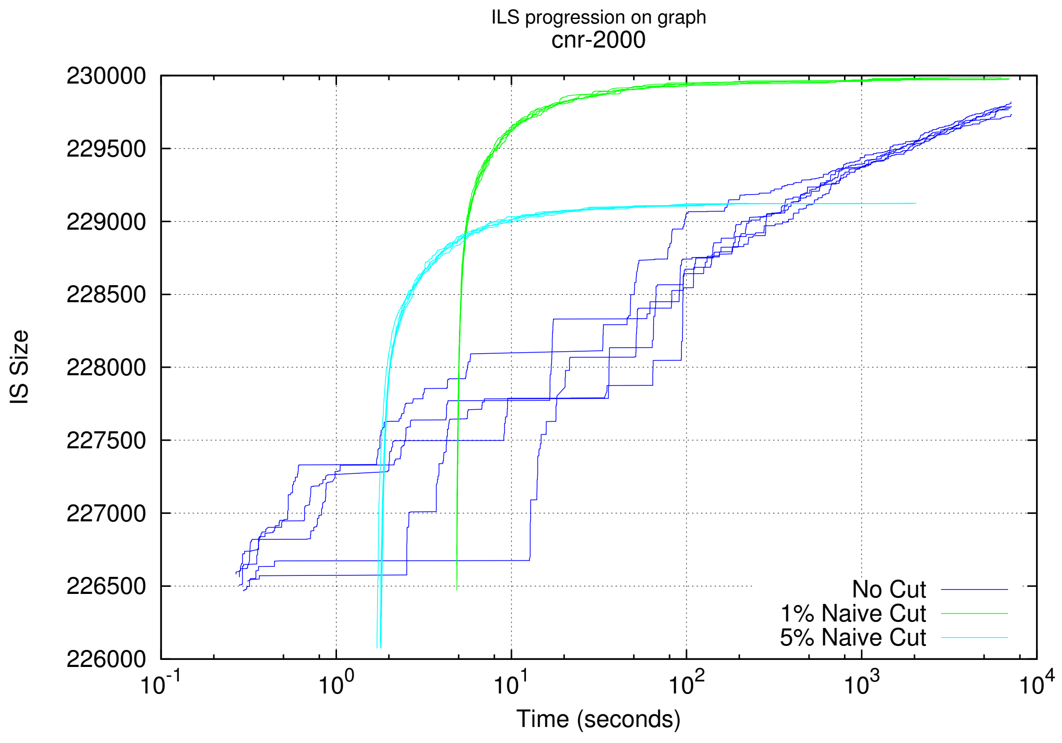

Figure 11: Improvement of ILS on web graph `cnr-2000` after naive cutting. The 5 runs are much more consistent after the cut and reach their plateau earlier. Cutting 10% leads to unsatisfactory results.

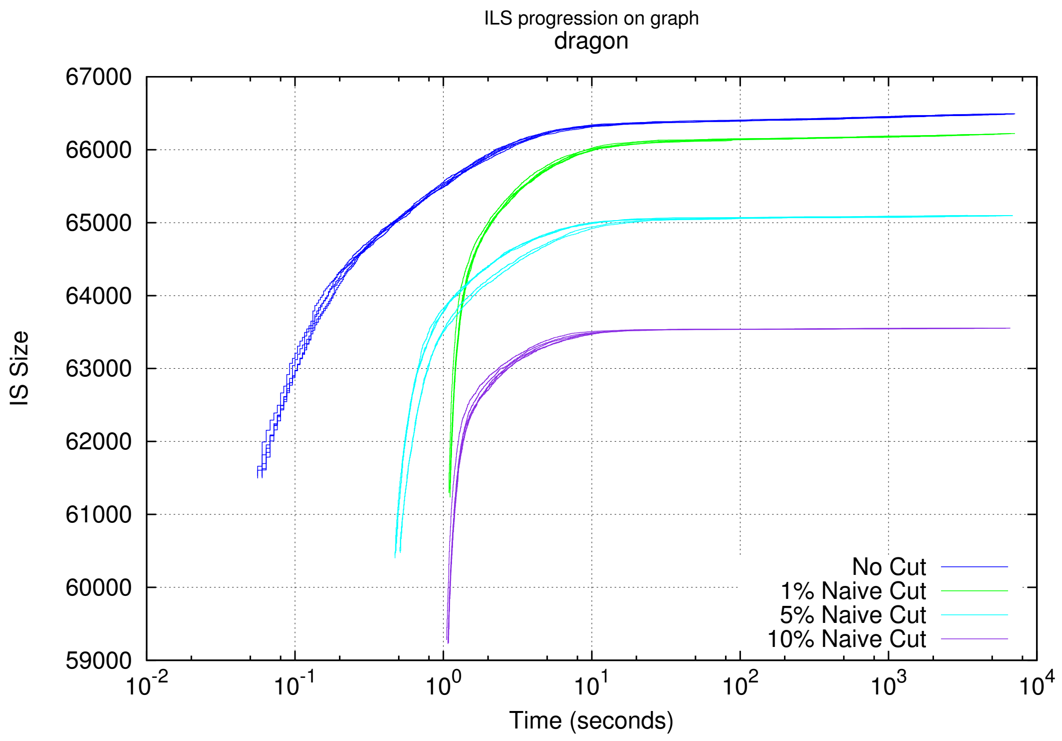

Figure 12: Naive cutting on mesh `dragon` removes too many MIS vertices. The runs were already consistent before the cut.

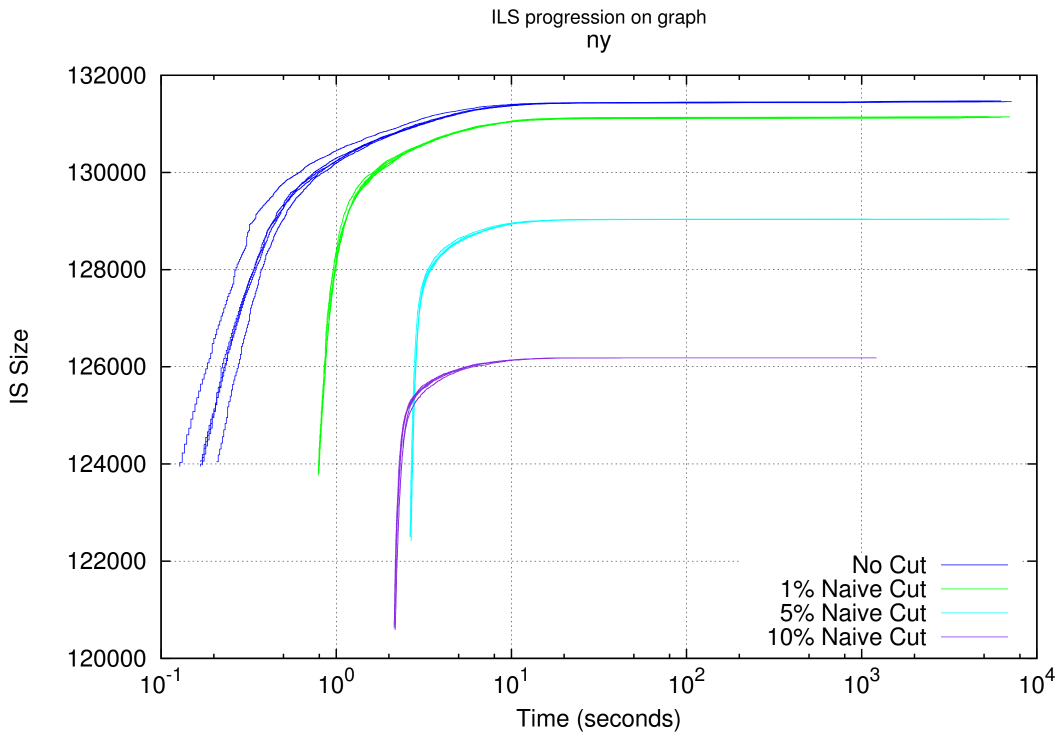

Figure 13: Naive cutting on road map `ny` removes too many MIS vertices. The runs were already consistent before the cut.

Cutting Vertices According to Their Core Number:

From now on, we focus only on 1%- and 5%-cuts and show average value curves instead of one curve per run. The k-core cutting method usually surpasses its naive cut counterpart's solution by several hundreds of vertices. This mostly closes the gap to best known or optimal results on the difficult instances. The speed up of ILS is also larger in most cases making this cutting technique a clear victor compared to naive cutting. The only difficult instance where the k-core result size is below our expectations is `cnr-2000`. We do not know the exact reason for this yet. Note, that the k-core cut does not require to actually compute the full k-core but only remove the percentage of vertices.

Continuing with our dense neighborhood tiebreak, a clear victor cannot be detected. The dense neighborhood tiebreak leads to a slightly larger speed up of ILS on difficult instances when cutting only 1%. When cutting 5%, the first version is quicker. As for easier instances like `dragon` or `bay`, the new tiebreak worsens both solution size and speed up. Since the tiebreak does not deliver clear advantages we do not favor it over the first version of k-core.

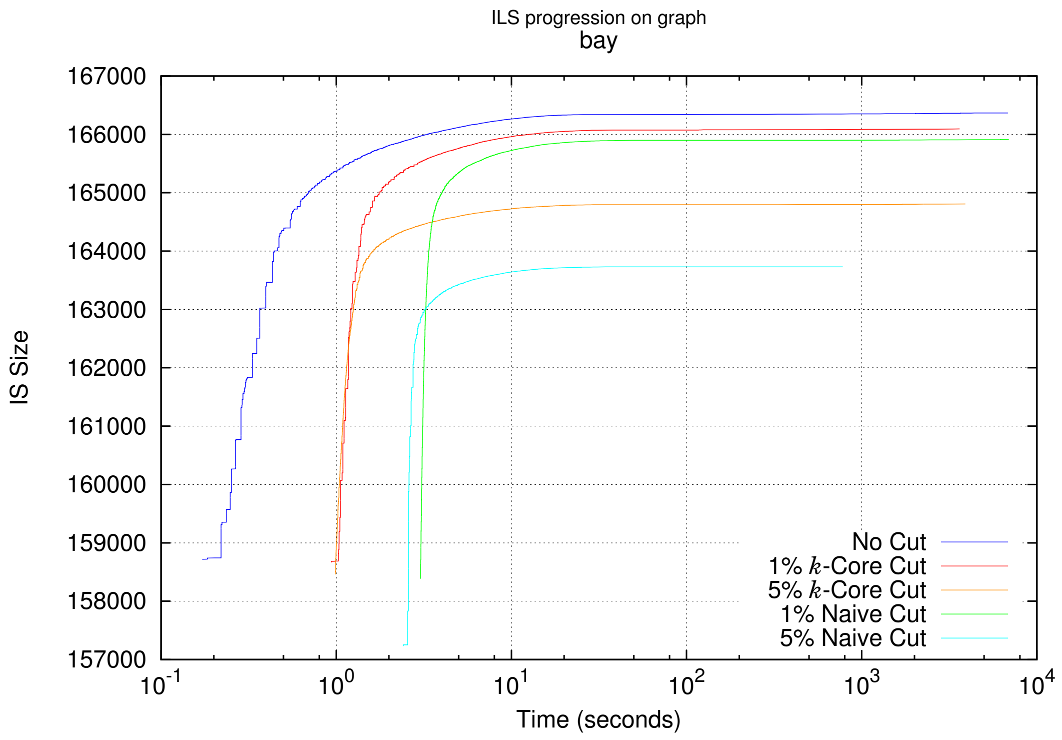

Figure 14: The k-core cut leads to much better results than the naive cut on road map `bay`. However, not cutting is preferred on this graph.

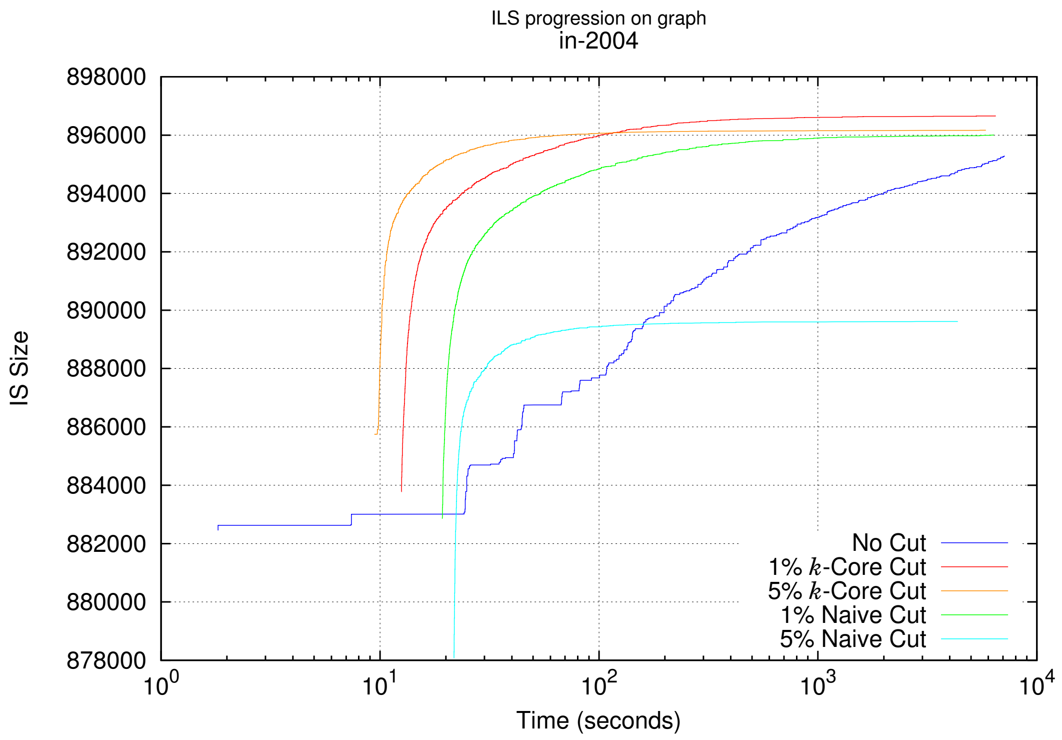

Figure 15: The k-core cut leads to a better speed up and better final results than the naive cut on web graph `in-2004`.

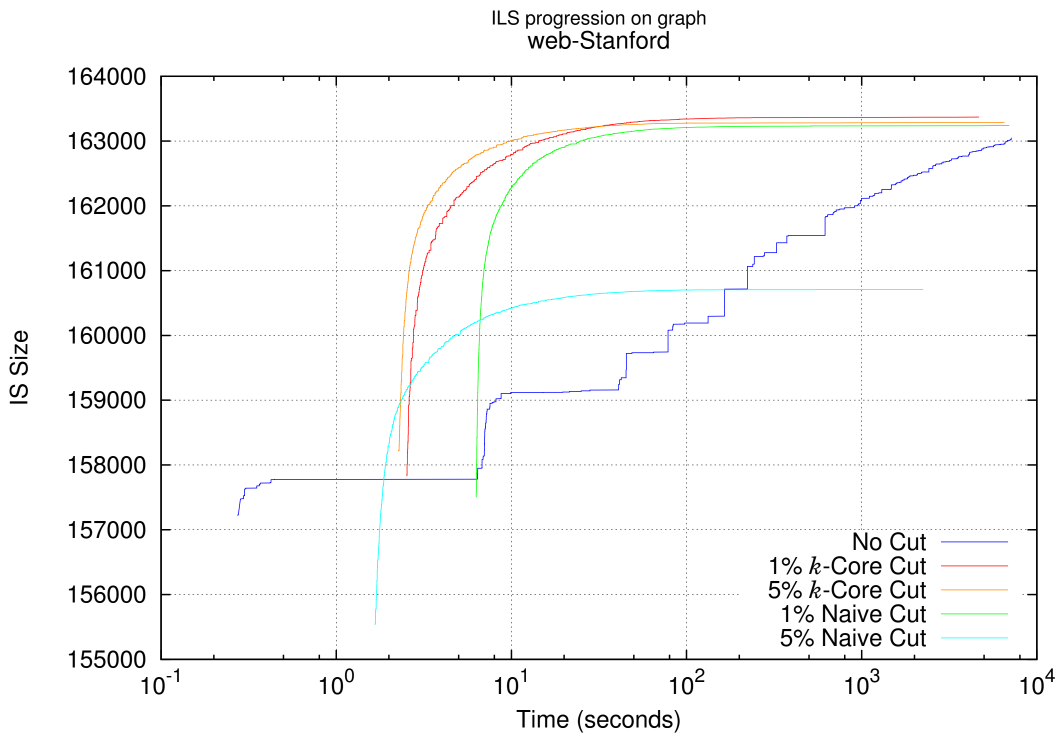

Figure 16: The k-core cut leads to better final results overall and a better speed up for the 1%-cut than the naive cut on social network `web-Stanford`.

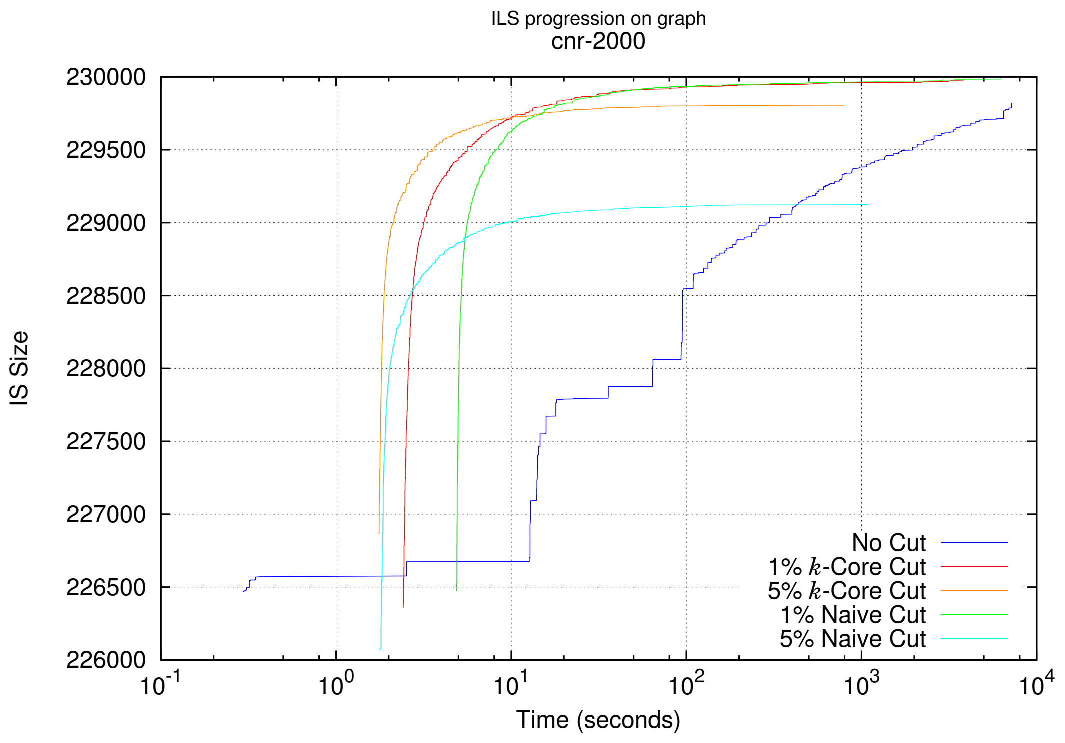

Figure 17: The 5% k-core cut leads to better final results than the 5% naive cut on web graph `cnr-2000`. The 1% cuts have very similar final results.

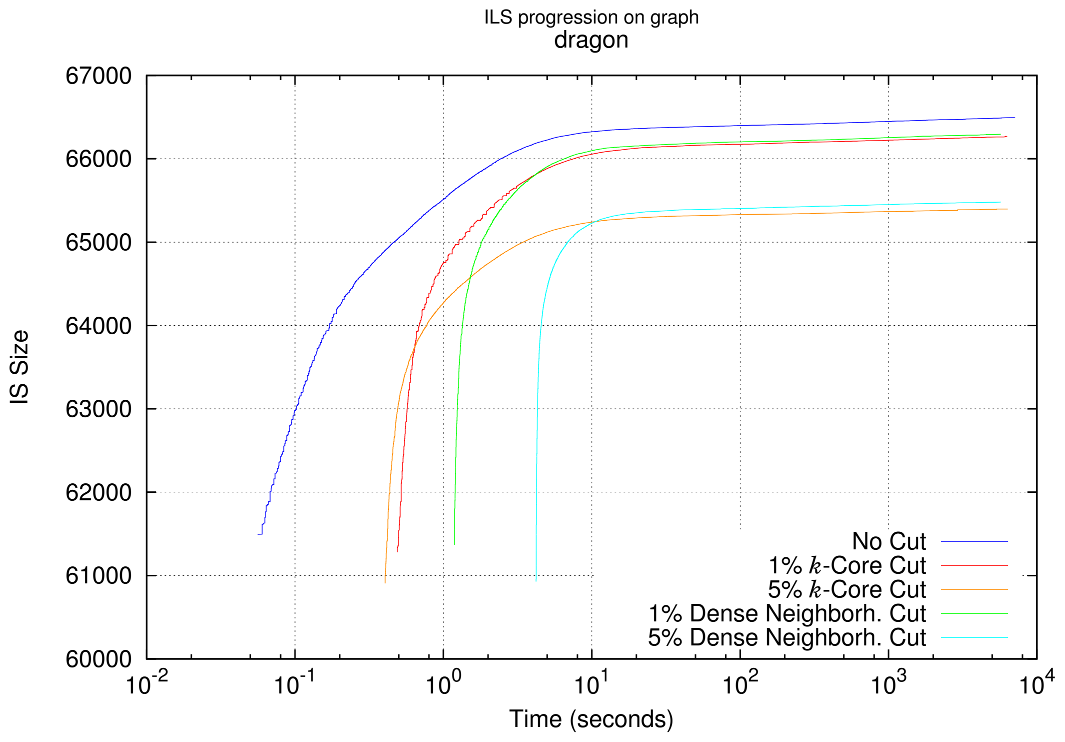

Figure 18: The k-core cut with the dense neighborhood tiebreak on mesh `dragon` leads to slightly larger independent sets but takes longer. Not cutting in preferred on this graph.

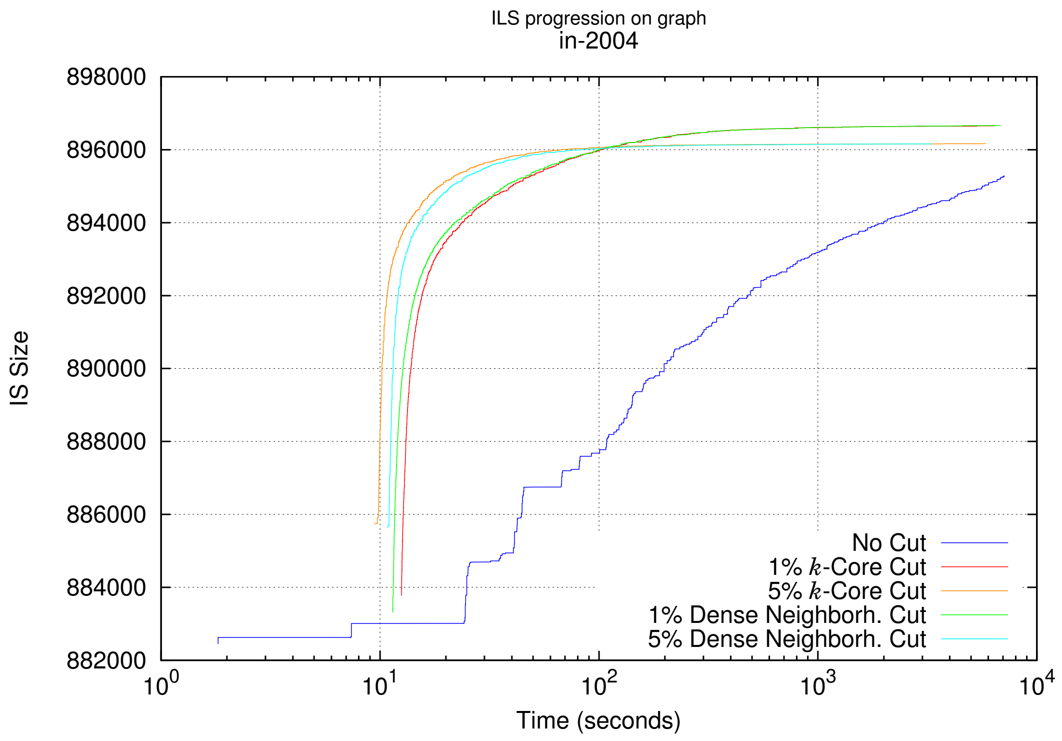

Figure 19: The two variations of the k-core cut show no significant differences on web graph `in-2004`.

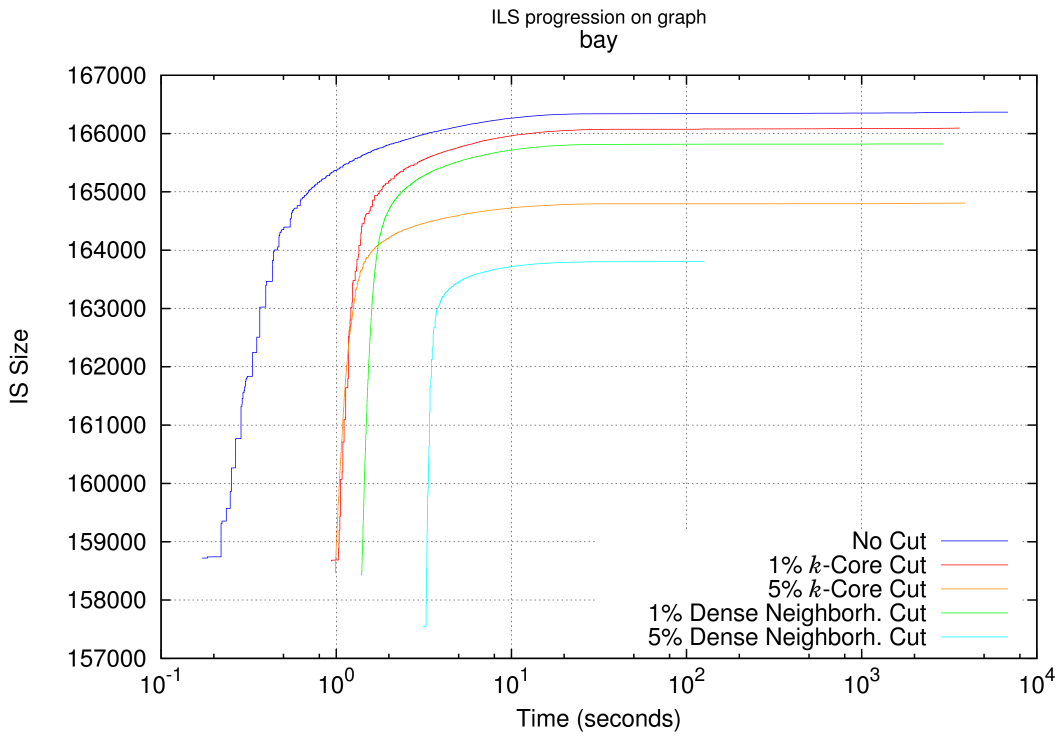

Figure 20: The k-core cut without the dense neighborhood tiebreak on road map `bay` leads to larger independent sets. Not cutting in preferred on this graph.

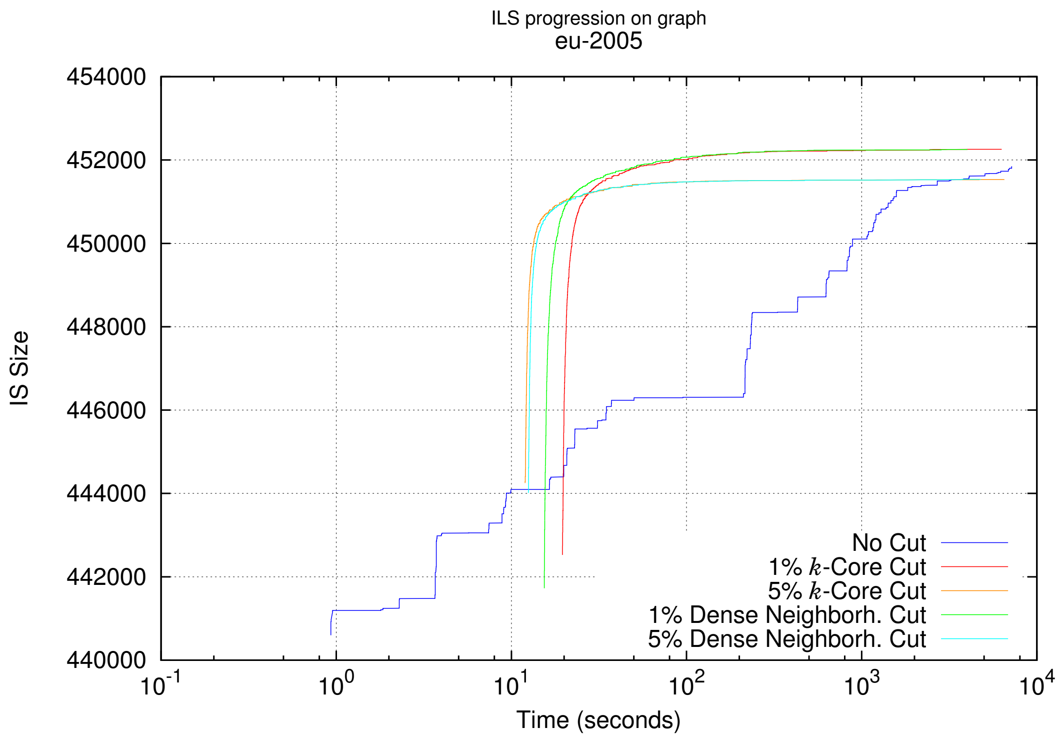

Figure 21: The dense neighborhood tiebreak is slightly faster for the 1%-cut on social network `eu-2005` but the final results are indistinguishable.

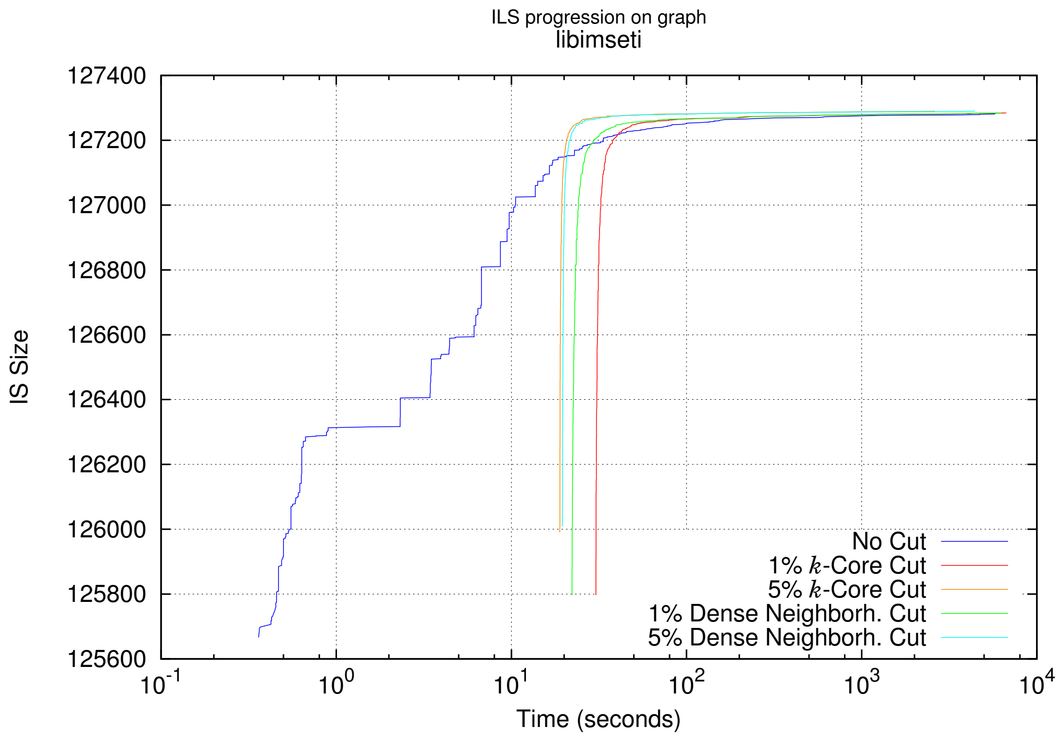

Figure 22: There is no clear winner among the tiebreaks result-wise on social network **`libimseti`**. The 5% cuts have slightly better result sizes. We think this is due to the high percentage of very high degree vertices in this graph, see Figure 2.

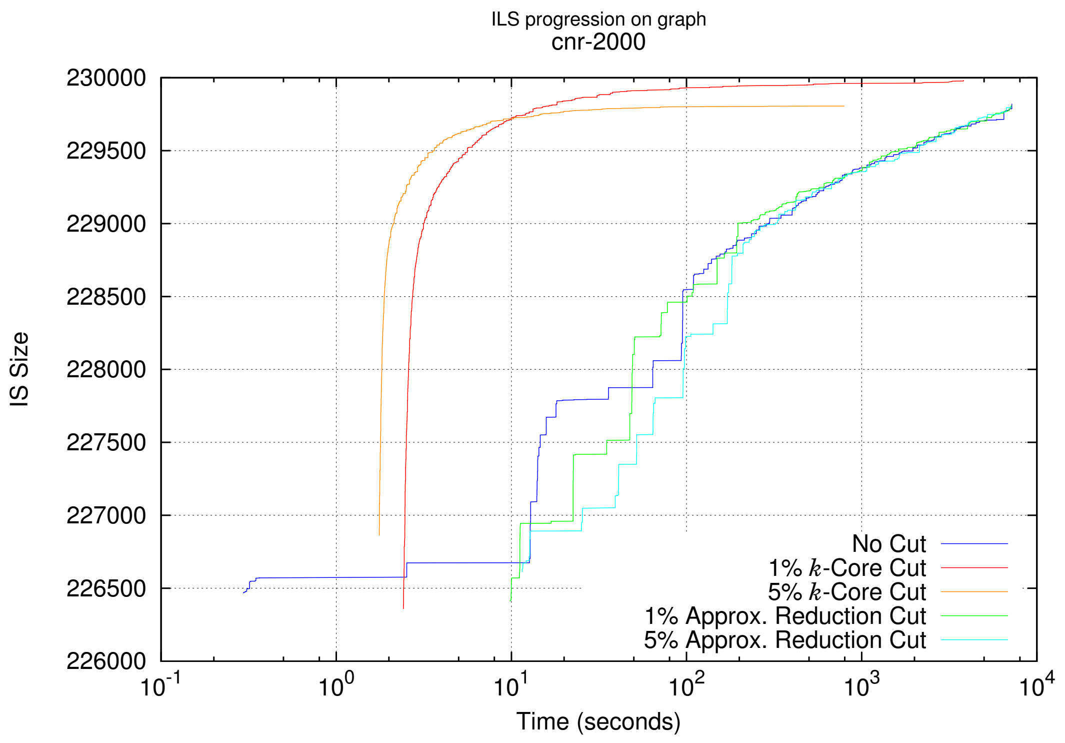

Figure 23: Approximate reductions do not remove the vertices responsible for the ILS slowdown on web graph `cnr-2000`. There is no improvement over not cutting.

Low Degree Reductions and Approximate Reductions:

We set the limit for structure removals to the average degree within the original graph. The limit for removing maximum degree vertices is set to three times that value. Other parameters may lead to better results but this cutting technique did not prove to be successful on the difficult instances. The speed up of ILS is insignificant to non-existent and the problems we try to solve are still apparent, see `cnr-2000`. Apparently structures like triangles, four-cycles and the like have little to no negative effect on the running time of ILS. The approximate reductions may even cut more MIS vertices than the previous methods. In addition, the cut for graph `in-2004` leads to a slightly worse outcome than not cutting in the first place.

Betweenness Centrality:

The running time of NetworKit's betweenness centrality algorithm is linear in its parameter *nSamples*. We tried parameter values of 25, 50, 100 and 200 which all lead to very similar results. However, the higher values resulted in running times too long to be interesting for our aims, so we use a value of 25. The experiments show that, like with the approximate reductions, there is no clear connection between the slowdown of ILS and vertices with a high betweenness centrality. The plotted curves for it mostly have a shape similar to the computation without any cutting. Even worse, the cutting leads to smaller independent sets with a longer computation time, see `in-2004`. Cutting only among the vertices with the 20% highest degrees slightly improves the outcome but not in any significant way (marked as "High Deg. Betw. C. Cut" in figures). Although this method is better than not cutting on `cnr-2000`, it is still far off from reaching the results of our other methods. All in all, this cutting method proves not to be successful.

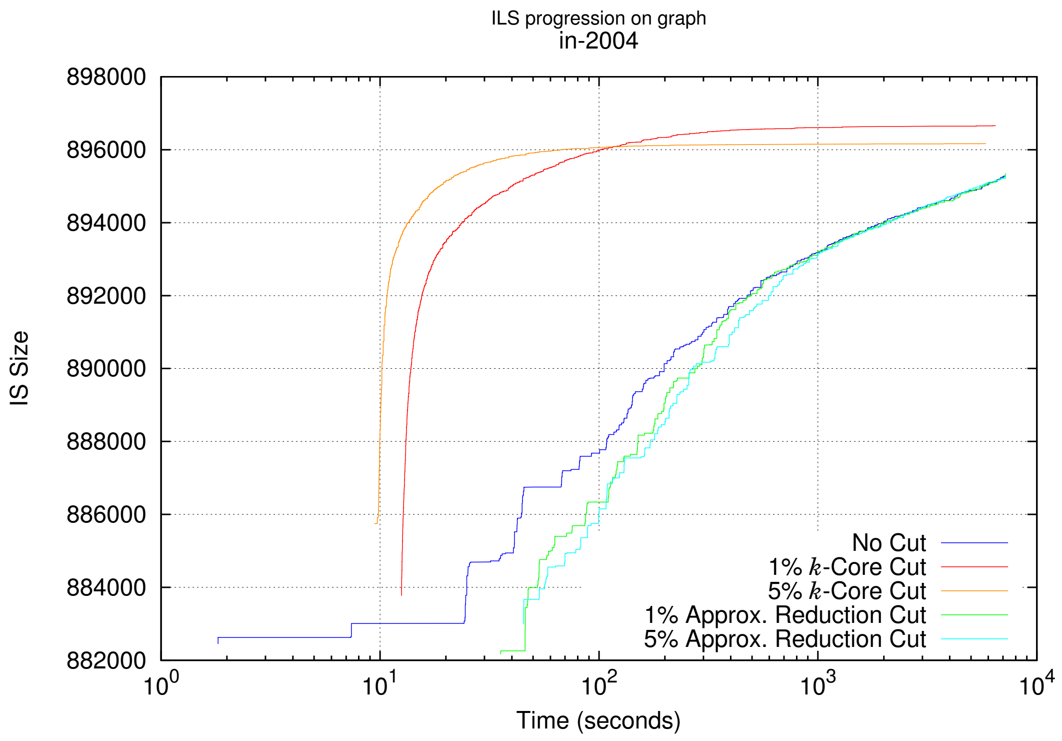

Figure 24: Approximate reductions do not remove the vertices responsible for the ILS slowdown on web graph `in-2004`. It even leads to further slowdown.

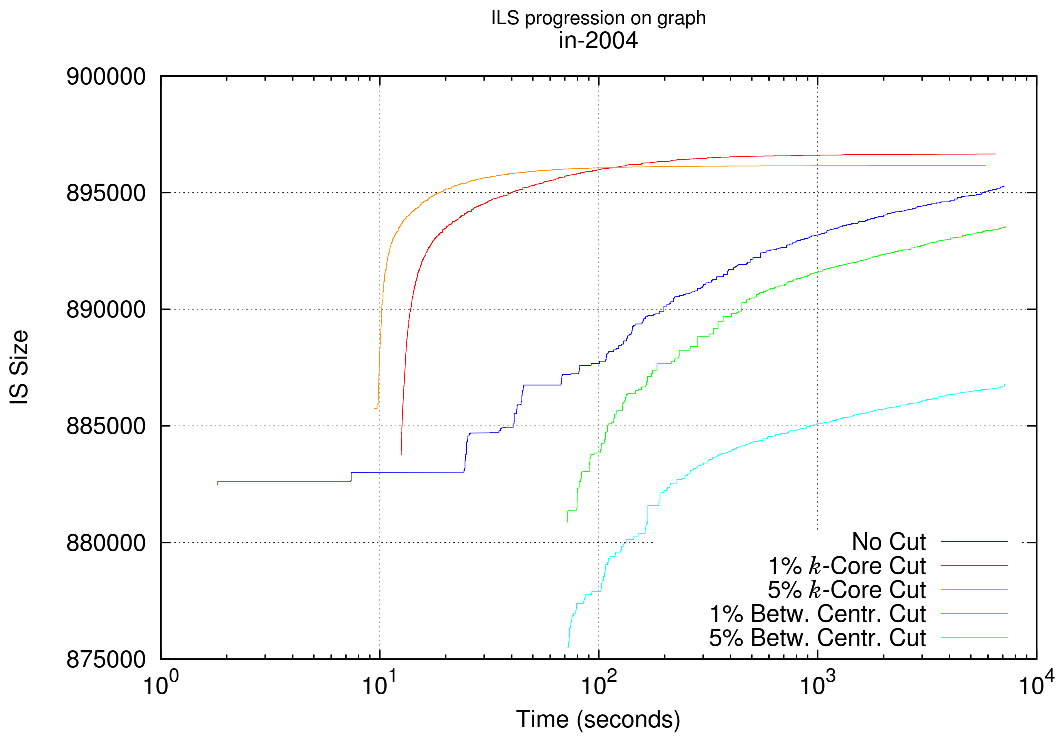

Figure 25: The betweenness centrality cut worsens the outcome on web graph `in-2004` both in time and result size.

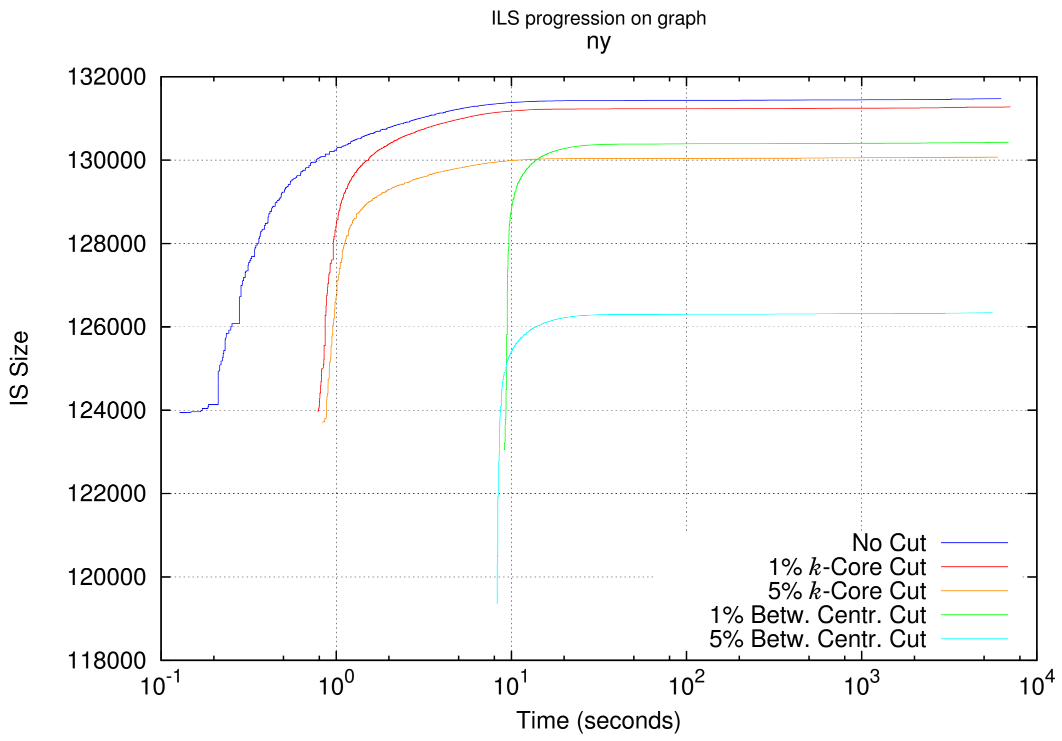

Figure 26: The betweenness centrality cut worsens the outcome on road map `ny` both in time and result size.

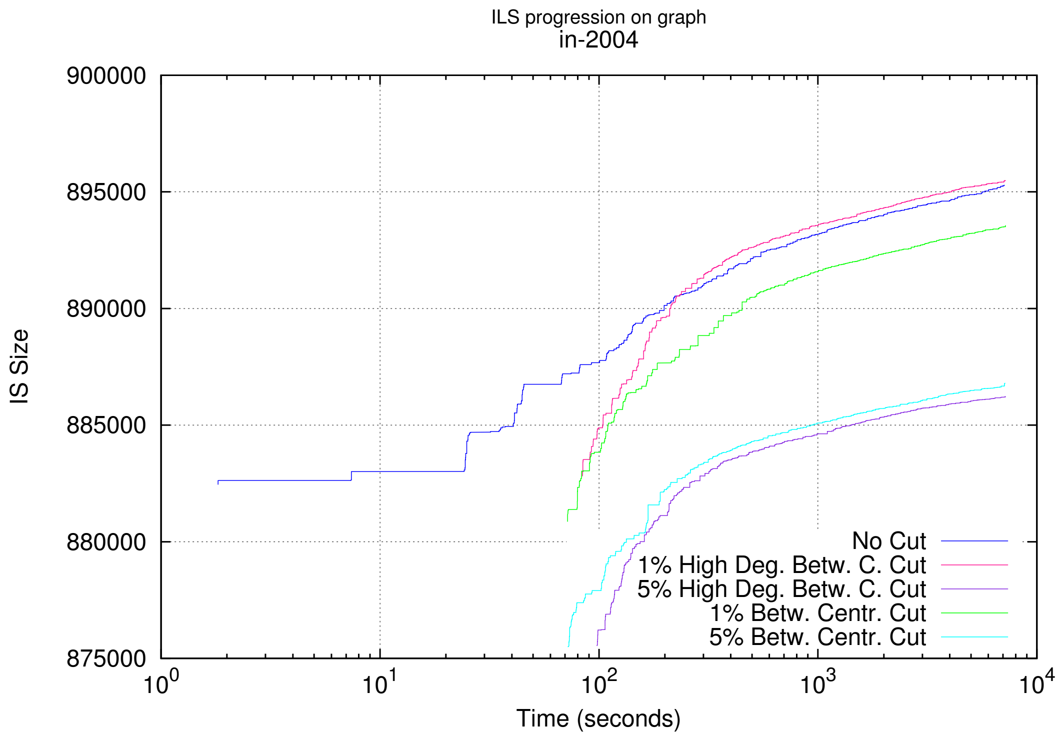

Figure 27: Restricting the betweenness centrality cut to the 20% highest degree vertices does not lead to promising results on web graph `in-2004`.

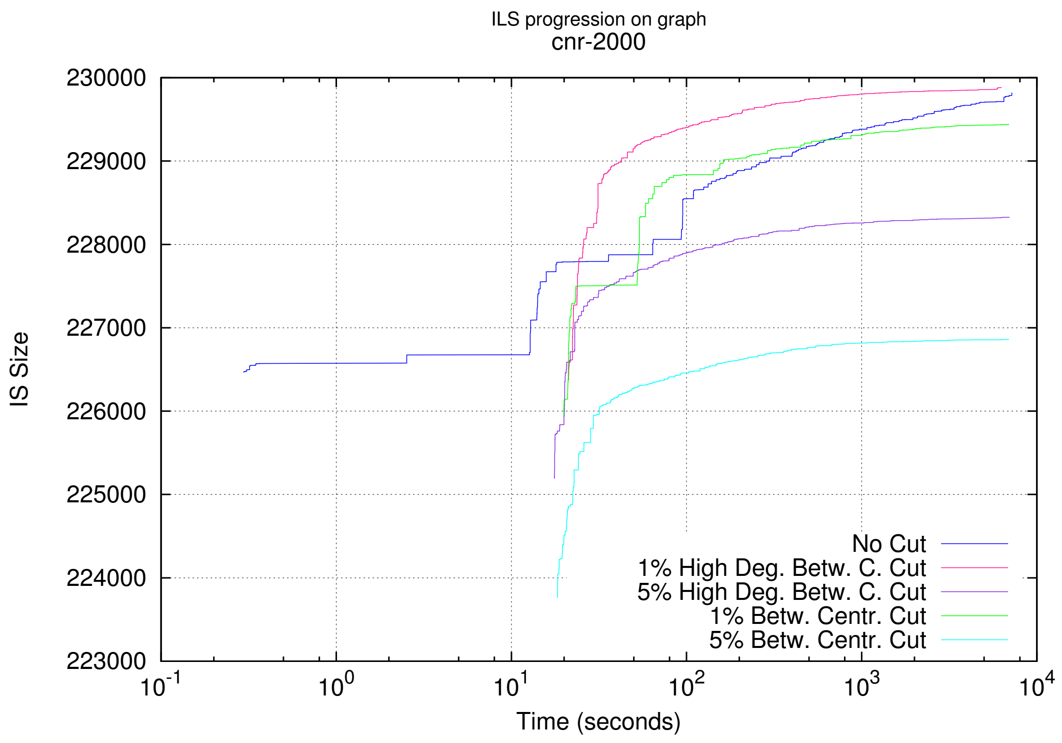

Figure 28: Restricting the betweenness centrality cut to the 20% highest degree vertices greatly increases the result sizes on web graph `cnr-2000`. However, they are still far behind the k-core cut.

Clustering:

To obtain a clustering quickly we make use of NetworKit's functionality again. We utilize NetworKit's *PLP* class to get an approximate clustering of the graph. PLP is a label propagation community detection algorithm. First, it assigns a label to each vertex. The labels propagate between neighbors to assign them to the same cluster. The denser a subgraph the more likely the vertices are to accept the same label. Over time the label changes come close to a halt and the clustering is determined. Each label corresponds to a cluster.

While visibly making the runs more consistent, cutting high degree vertices in clusters turns out not to be promising either. The result sizes are worse than those of the naive cut and especially inferior to those of the k-core cut. Cutting 5% per cluster is already worse than not cutting at all. In order to reach results similar to the ones of our preferred technique the distribution of high degree vertices needs to be similar in most clusters. Real world data usually does not have this property. If we cut the specified percentage within a cluster of mostly low degree vertices, the probability of removing MIS vertices is high.

Concluding this set of experiments, the preferred method of cutting is definitely the removal of vertices according to their core number. All other methods failed to achieve similar qualities. The naive cutting lead to a significant speed up but failed to reach the desired near-optimal solution sizes. The approximate reductions by Asgeirsson and Stein [3], [4] betweenness centrality cut failed at tackling the slowdown problem. Lastly, cutting in clusters removed too many MIS vertices. However, cutting of any kind in graphs with no or only very few vertices of degree of about 1,000 or higher has no benefits.

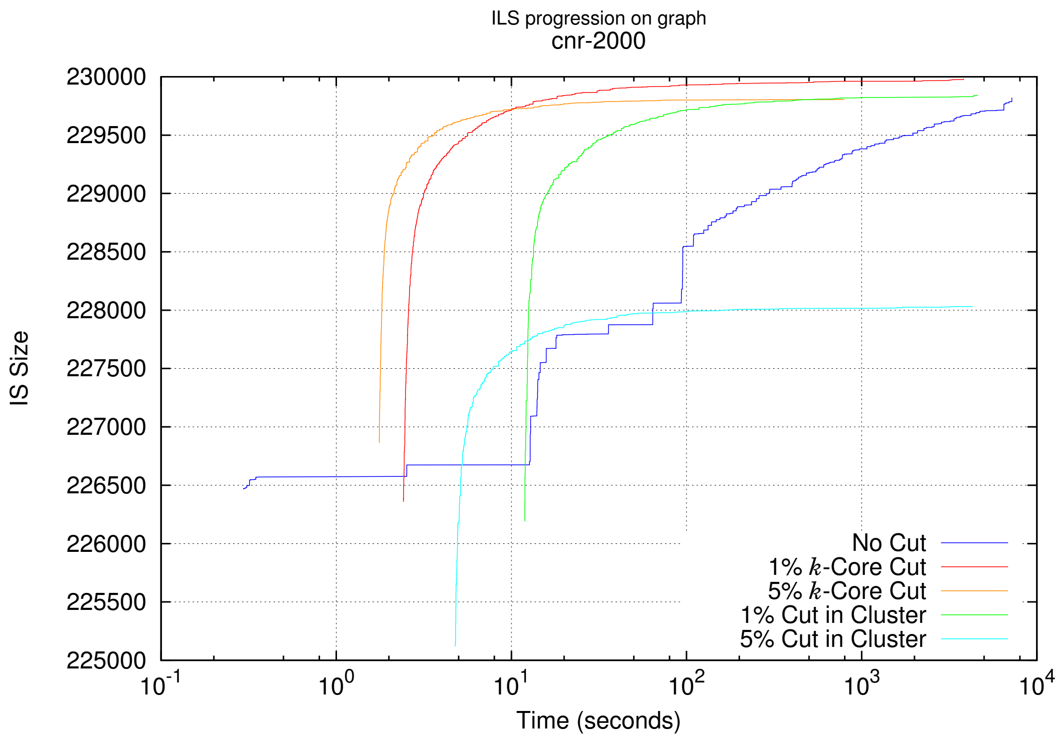

Figure 29: The k-core cut in clusters on web graph `cnr-2000` is both slower and leads to worse results than the global k-core cut.

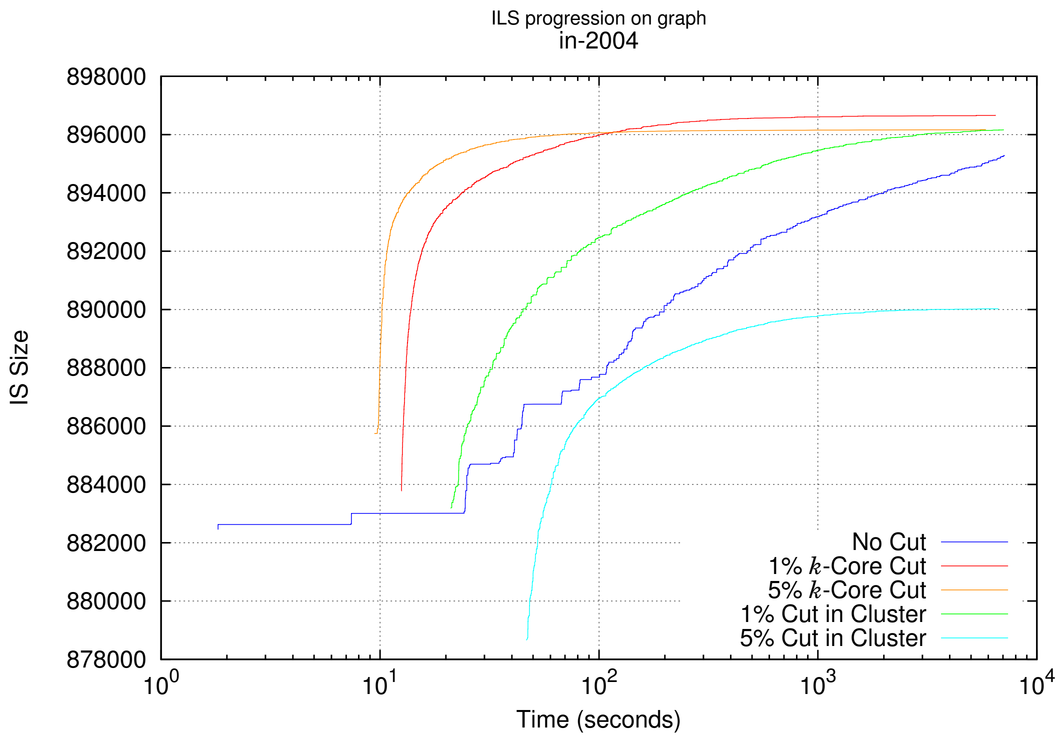

Figure 30: The k-core cut in clusters on web graph `in-2004` is both slower and leads to worse results than the global k-core cut.

5.4. Inserting Cut Vertices

For this set of experiments we use the cutting technique where we cut the highest degree vertices with a bucket priority queue. The size of removal is 5% of the graph's vertices. Preliminary experiments showed that this percentage leads to a large speed up and acceptable independent set sizes. We use the algorithm without the dense neighborhood tiebreak since the tiebreak computation is usually a bit slower for this cut size. After the cutting we transmit the cut graph along with the cut vertices to ILS. The algorithm runs until there are 100,000 itertions of the main loop without improvements and we then perform the insertion techniques. In between insertions ILS runs its local search until there are 500,000 iterations without any progress before we move on with the next insertion. We mark every insertion in the plots with a cross.

Insertion Only:

We investigate 5 different versions of inserting vertices. Preliminary tests showed the clear winner being insertion steps in decreasing sizes. So, while describing the outcome of all 5 experiments, note that the proper experiments were only made with the last version. Also note, that we did not include the computation time for cutting in the preliminary tests. In the first experiment we insert 4/5 of the cut vertices randomly and then run ILS until the time limit is reached. The problem with random insertion is that potentially most of the vertices responsible for ILS's slowdown get back into the graph. This cancels out the speed up and may even result in independent sets smaller than when we do not insert at all.

The remaining 4 experiments therefore only insert the cut vertices with the lowest degrees. First, we insert 4/5 of the cut vertices like in the previous experiment. In a second insertion we insert the remaining fifth of the vertices and let ILS run on the complete graph. This method of insertion combines a rapid initial improvement with a decently sized final independent set. One negative aspect is that after the second insertion progress decreases too quickly. The second version inserts a fifth of the cut vertices at a time until the full graph is reconstructed, meaning that there are 5 insertions. For some graphs, like `cnr-2000` or `in-2004`, several of these insertions are needed before any difference to not inserting at all is noticeable. In addition, the insertions happen to late to compete with the 1%-cut curve. We conclude that the first insertions need to be larger to cause an effect and that the optimal time for insertions varies depending on the method and graph. A better method of determining when to insert may be roughly approximating the curves' gradients. The third experiment inserts all cut vertices in 10 equally sized portions. As anticipated, the same features as the ones from the previous experiment are present.

Lastly, we decrease the sizes of insertions in the final experiment. The specified percentages refer to the size of the whole graph. Still, only the cut vertices are inserted and no others. Starting with an insertion of 4%, we continue with 0.5%, 0.3%, 0.1% and finally the last 0.1% to create the whole graph. This method shows the best results out of the insertion experiments. When ignoring the cutting time, it combines desired features of the 5%-cut curve and the 1%-cut curve. Fast initial result improvements and large final results are achieved.

However, when taking the cutting time into consideration, some of the advantages cancel out. This is due to the fact that the additional output required for the insertion (that is, the cut vertices and the complete graph) requires too much time. Thus, we lose the speed up advantage of the 5%-cut. We can see in the plots that ILS catches up with the better curves after insertion but we do not achieve noteworthy benefits from this.

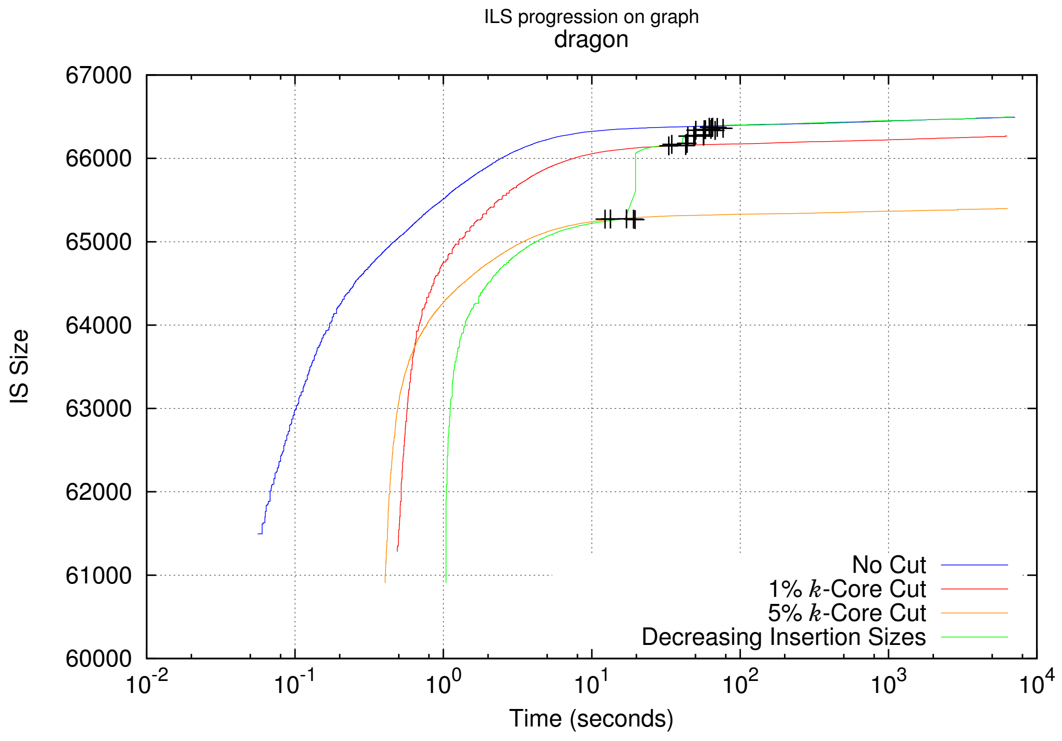

Figure 31: Ordered insertion of cut vertices in decreasing sizes on mesh `dragon`. Each cross marks an insertion in one of the runs. The result sizes of the other curves are reached after the insertion but not cutting is the better option on this graph.

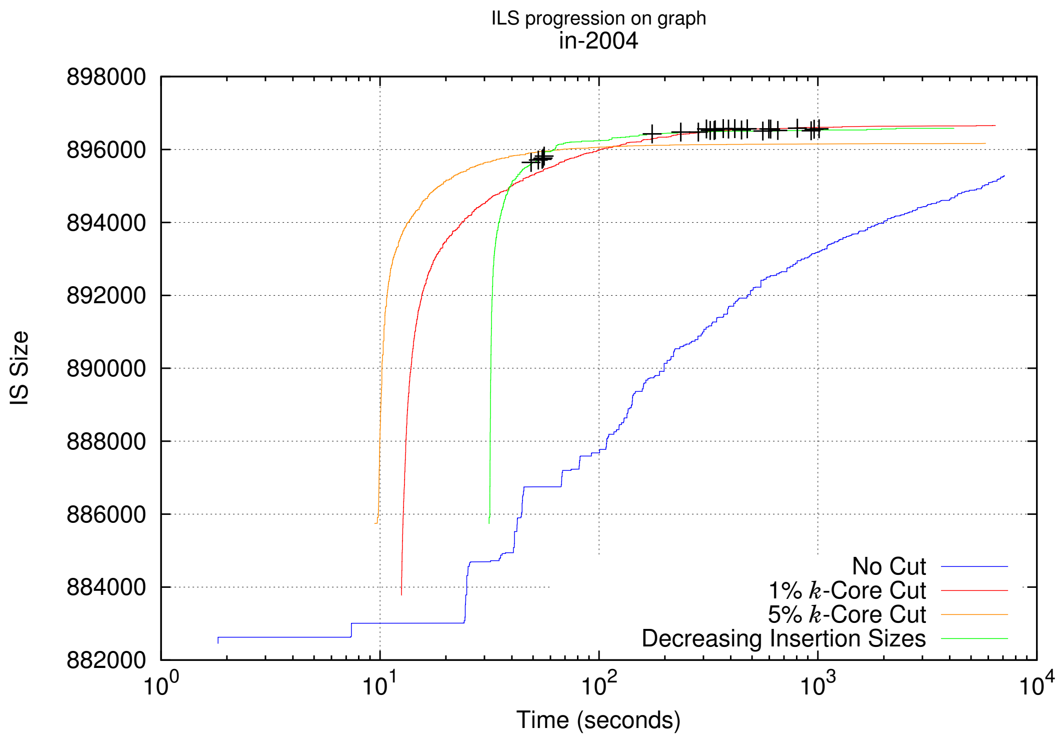

Figure 32: Ordered insertion of cut vertices into web-graph `in-2004`. Each cross marks an insertion. The curve almost catches up with the 1% k-core cut. The slower initial computation may be due to the larger output of the cutting algorithm.

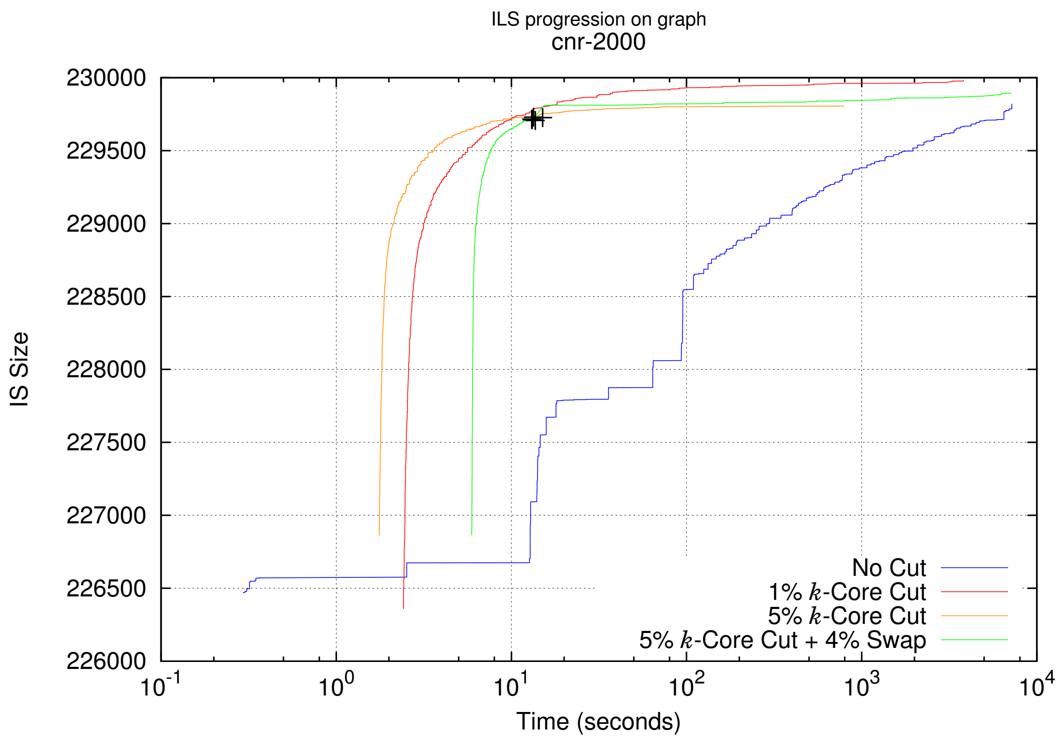

Figure 33: Swapping the cut vertices in and out does not work properly on web graph `cnr-2000`. After the first insertion (marked with a cross) the solution size slowly increases leaving the swap inactive.

Swapping In and Out:

Now, we randomly insert 4/5 of the cut vertices back into the graph and remove them again before we perform the next insertion of the same size. Note that when taking the inserted vertices out of the graph again we may lose a number of solution vertices. This fact as well as the shown downsides of random insertion lead to little to no improvements. On graph `cnr-2000`, ILS slowly improves its solution size leaving the swapping inactive entirely. The slow progression keeps the loop limit from activating a swap. The insertions of the previous paragraph show the same behavior on `cnr-2000`. As for the mesh `dragon`, the previous insertions are more successful then swapping since the swaps cannot exceed the 1%-cut's curve. Still, this is of no real interest considering that cutting is not helpful for this kind of graphs.

As a conclusion to this set of experiments, insertions of any kind are irrelevant when the cutting time is taken into consideration. The hope was to combine the advantages of different cuts' curves but this requires to keep the quick output of larger cuts which we could not achieve.

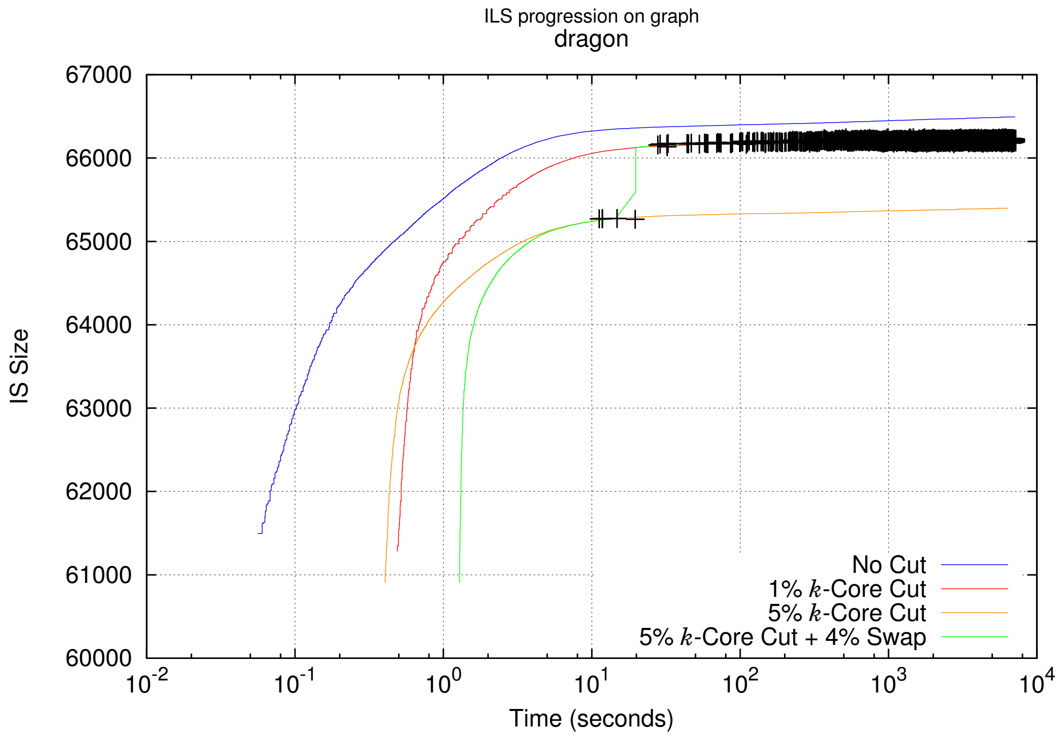

Figure 34: Swapping the cut vertices in and out leads to no improvement on mesh `dragon`. After the initial insertion (low crosses) the 1% k-core cut is reached but all swaps (all other crosses) have no effect.

5.5. Exact Reductions

This set of experiments does not use any of the cutting techniques except for the first experiment. In general, we reduce the graph with the exact reductions by Akiba et al. [1] and then run ILS on the reduced graph.

Reductions Only:

This experiment is performed on the original, uncut graphs. We also performed preliminary tests on cut graphs with a removal of either 0.2%, 0.5% or 1%. However, we did not spot any remarkable differences between the cutting sizes making the cutting potentially redundant. On many graphs this method leads to optimal or near-optimal results consistently. On the more difficult instances the results exceed those of our cutting and inserting experiments. Both cardinality of the final independent set and computation time are better. The algorithm often reaches the best known results. The speed up is also remarkable in many cases as the algorithm reaches very large results sizes shortly after the initial computation. This method also does not lead to worse results than only running ILS on graphs with few or no high degree vertices such as meshes and road maps. However, the reduction can take longer than the cutting when all degrees are roughly the same. This can be seen on mesh `dragon` which only has degree-3 vertices. Kernelization on such graphs is difficult as the vertices are too similar.

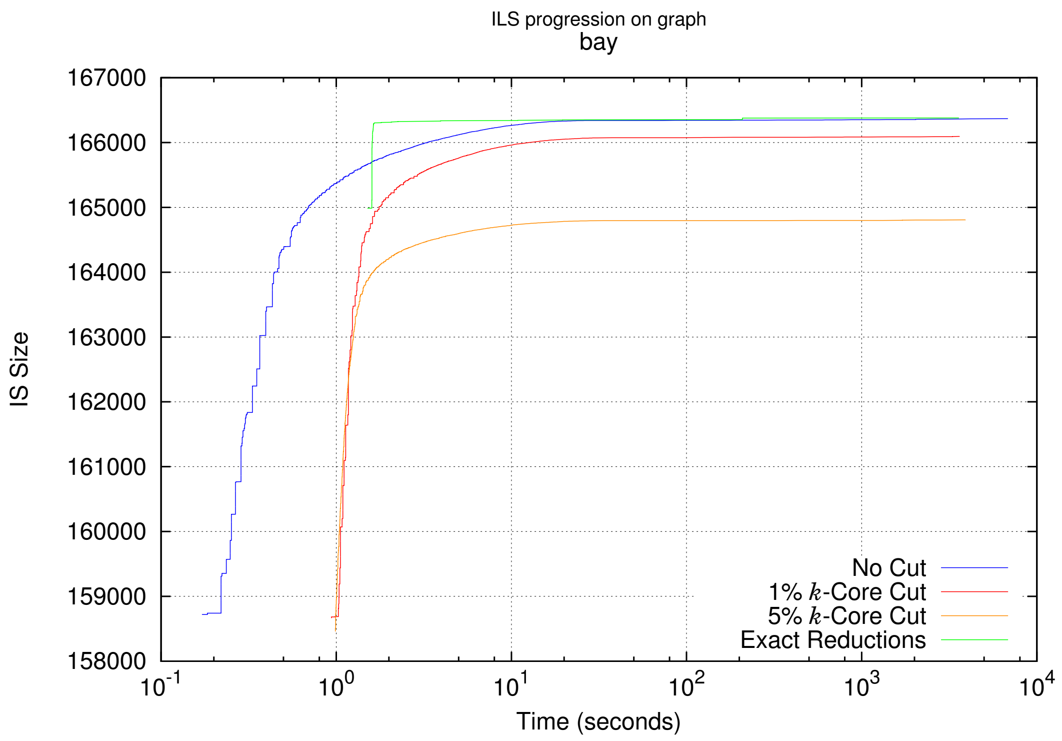

Figure 35: The exact reductions lead to a speed up on road map `bay` which we could not achieve with any of the cutting techniques.

Reduction Swap:

Note, that the following descriptions are from preliminary results. We did not repeat them with larger time limits as the outcome was not promising enough. In these final experiments we try removals of 1%, 5%, 10% and 25% of the solution. As a reminder, in this experiment we do not cut the vertices completely from the graph but just take them out of the solution. These percentages only apply to the first and forth version of reduction swap which remove the highest degree vertices with a bucket priority queue or remove random vertices. The second and third version continue their specific removals after the percentage of highest degree vertices is taken out of the solution. The initial solution is computed by running ILS after the reduction until its main loop repeats 20 million times without progress. The preliminary tests featured no distinguishable differences between cut sizes. Unfortunately, the improvements caused by the swaps were insignificant, that is, 0 to 3 vertices. The initial solutions are decisive while the swaps fail at enhancing them.

As last remarks for this section we consider the exact reductions to have the highest potential for improving current algorithms for the MIS problem. They are superior to our cutting methods on nearly all graphs both in terms of speed up and final result size.

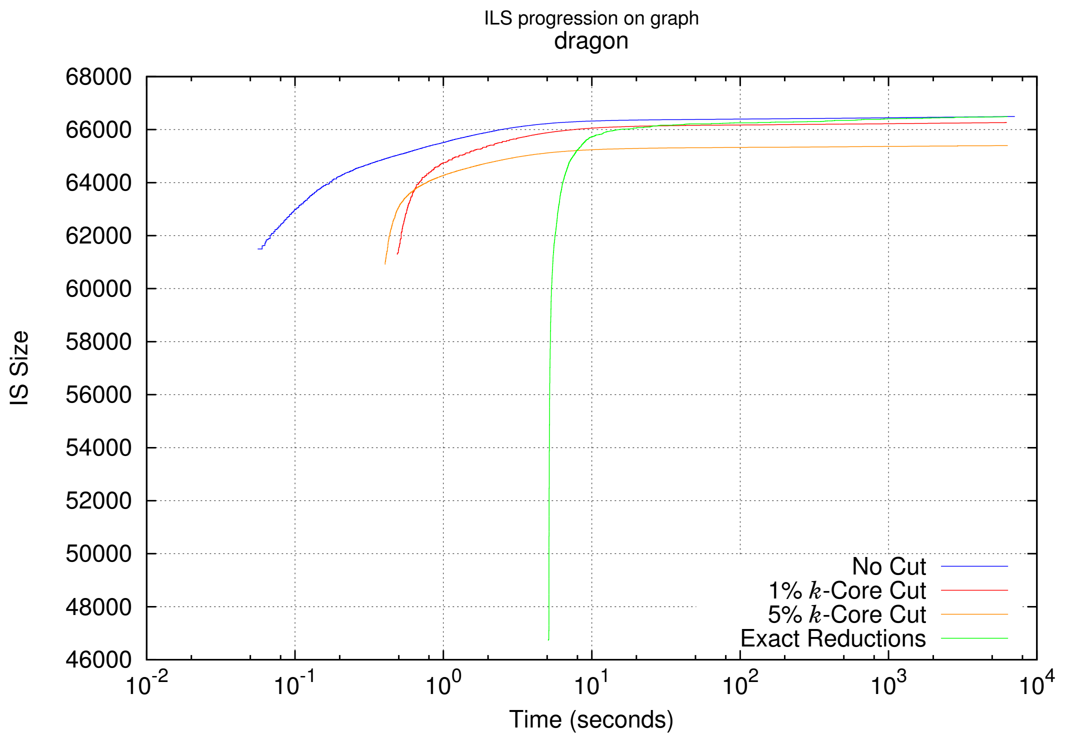

Figure 36: A kernelization on mesh `dragon` is difficult as all vertices have the same degree. Despite being slower than the cutting techniques, this method delivers result sizes on par with not cutting.

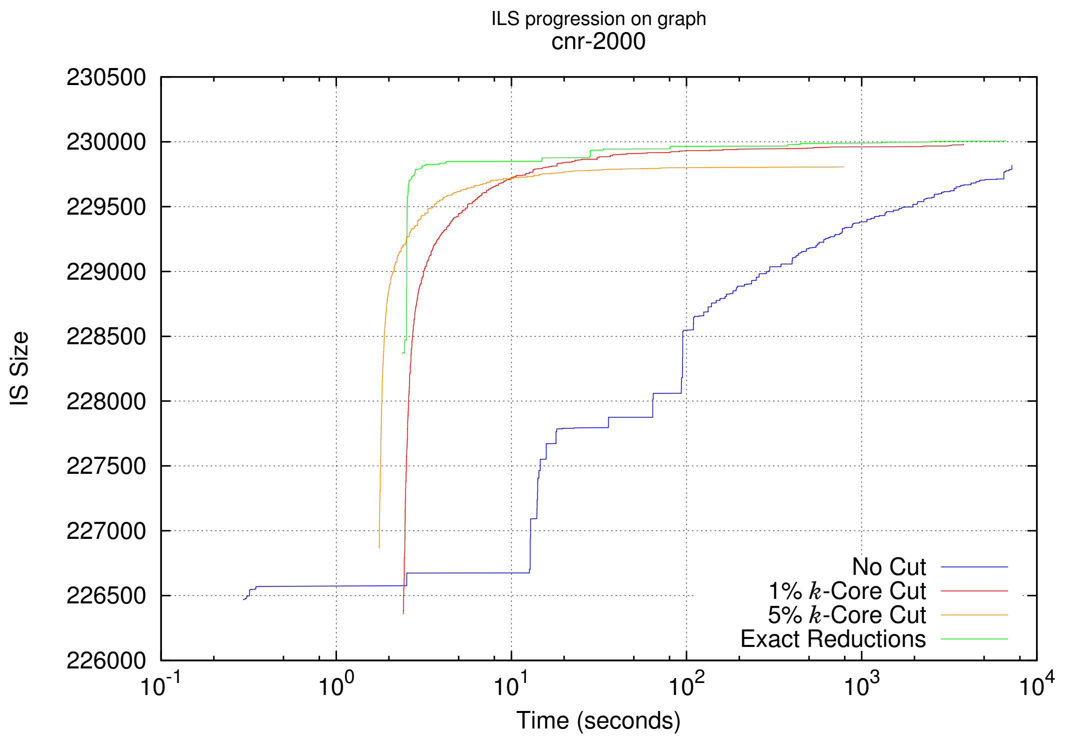

Figure 37: The exact reductions lead to both a better speed up and final result size on web graph `cnr-2000`.

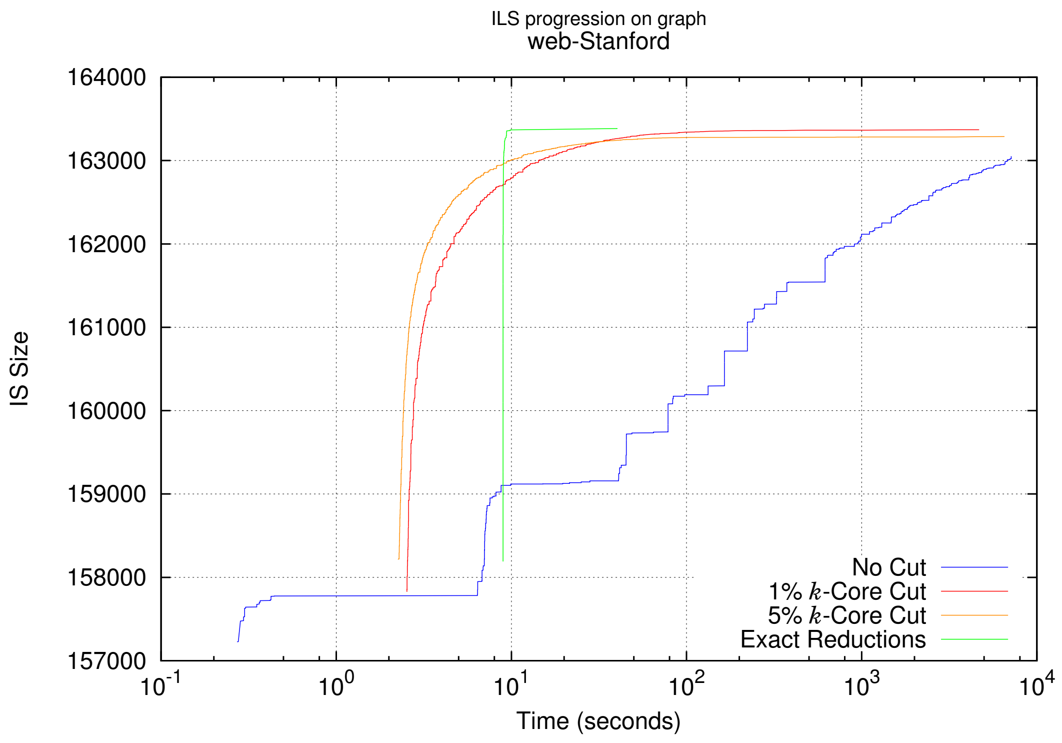

Figure 38: The exact reductions lead to very large independent sets very quickly on social network `web-Stanford` despite needing more time to compute the initial solution.

Table 1: Graphs used in every experiment.

NAME	*n*	*m*	*avg. deg.*
`cnr-2000`	325557	2738969	16.8264
`dragon`	150000	225000	3
`in-2004`	1382908	13591473	19.6564
`libimseti`	220970	17233144	155.977
`ny`	264346	365050	2.76191

Table 2: Final Results for all experiments on graph `cnr-2000`. We show the worst (MIN), average (AVG) and best (MAX) independent set found over 5 runs.

EXPERIMENT TYPE	*MIN*	*AVG*	*MAX*
No Cut	229733	229777.20	229821
Naive Cut (1%)	229973	229979.59	229986
K-Core Cut (1%)	229962	229971.80	229984
Dense Neighborhood Cut (1%)	229964	229971.00	229977
Approx. Reductions Cut (1%)	229776	229791.80	229809
Betweenness Centrality Cut (1%)	229431	229440.80	229446
High Degree Betw. C. Cut (1%)	229856	229865.59	229881
Cluster Cut (1%)	229830	229835.80	229844
Ordered Insert Cut	229956	229960.20	229965
Swap Cut	229876	229883.41	229893
Exact Reductions	**229986**	**229998.80**	**230014**

Table 3: Final Results for all experiments on graph **`dragon`**. We show the worst (MIN), average (AVG) and best (MAX) independent set found over 5 runs.

EXPERIMENT TYPE	*MIN*	*AVG*	*MAX*
No Cut	66487	66490.20	66495
Naive Cut (1%)	66214	66218.00	66222
K-Core Cut (1%)	66264	66268.20	66275
Dense Neighborhood Cut (1%)	66290	66294.20	66297
Approx. Reductions Cut (1%)	66105	66109.80	66115
Betweenness Centrality Cut (1%)	65921	65927.00	65938
High Degree Betw. C. Cut (1%)	65930	65933.60	65936
Cluster Cut (1%)	66163	66167.60	66176
Ordered Insert Cut	66483	66489.80	**66501**
Swap Cut	66236	66240.40	66243
Exact Reductions	**66493**	**66496.20**	66500

Table 4: Final Results for all experiments on graph **`in-2004`**. We show the worst (MIN), average (AVG) and best (MAX) independent set found over 5 runs.

EXPERIMENT TYPE	*MIN*	*AVG*	*MAX*
No Cut	895192	895260.38	895286
Naive Cut (1%)	895982	895991.00	896010
K-Core Cut (1%)	896646	896653.62	896657
Dense Neighborhood Cut (1%)	896650	896657.38	896662
Approx. Reductions Cut (1%)	895227	895269.81	895344
Betweenness Centrality Cut (1%)	893448	893515.38	893557
High Degree Betw. C. Cut (1%)	895424	895459.62	895489
Cluster Cut (1%)	896152	896159.00	896164
Ordered Insert Cut	896541	896560.81	896581
Swap Cut	896200	896230.81	896265
Exact Reductions	**896753**	**896757.00**	**896762**

Table 5: Final Results for all experiments on graph **`libimseti`**. We show the worst (MIN), average (AVG) and best (MAX) independent set found over 5 runs.

EXPERIMENT TYPE	*MIN*	*AVG*	*MAX*
No Cut	127276	127279.20	127282
Naive Cut (1%)	127281	127282.80	127284
K-Core Cut (1%)	127281	127282.20	127283
Dense Neighborhood Cut (1%)	**127282**	127282.40	127283
Approx. Reductions Cut (1%)	127273	127275.20	127277
Betweenness Centrality Cut (1%)	127279	127279.20	127280
High Degree Betw. C. Cut (1%)	127277	127278.20	127280
Cluster Cut (1%)	127281	127281.40	127282
Ordered Insert Cut	127253	127265.00	127273
Swap Cut	127273	127277.60	127281
Exact Reductions	127275	**127283.00**	**127289**

Table 6: Final Results for all experiments on graph **ny**. We show the worst (MIN), average (AVG) and best (MAX) independent set found over 5 runs.

EXPERIMENT TYPE	*MIN*	*AVG*	*MAX*
No Cut	131456	131463.41	131474
Naive Cut (1%)	131137	131144.20	131151
K-Core Cut (1%)	131250	131263.80	131272
Dense Neighborhood Cut (1%)	131009	131011.60	131015
Approx. Reductions Cut (1%)	131453	131463.41	131474
Betweenness Centrality Cut (1%)	130411	130420.60	130429
High Degree Betw. C. Cut (1%)	131011	131016.60	131025
Cluster Cut (1%)	131389	131398.00	131409
Ordered Insert Cut	131461	131467.00	131476
Swap Cut	131149	131152.41	131159
Exact Reductions	**131491**	**131494.41**	**131497**

6. Conclusion and Future Work

6.1. Conclusion

In this thesis we presented a number of methods to speed up local search algorithms for the maximum independent set problem. In this context, we evaluated a set of combinations of different algorithms to find the combinations that work best. First, we compared the current state-of-the-art algorithms usable for this problem, namely ILS and NuMVC. The comparison showed that ILS is superior to NuMVC on large sparse graphs. Additionally, we improved running times for ILS significantly especially on difficult instances. Our range of experiments tackles the removal of specific vertices that may lead to a slowdown of the algorithm as well as inserting them back later to assure that we did not cut MIS vertices. In addition, we utilized exact reductions to minimize a graph's complexity to further speed up the progression of ILS.

We showed that the removal of high degree vertices according to their core number boosts ILS's progress on difficult instances without a loss in solution size and makes the runs more consistent. Using this cutting technique makes the reinsertion of cut vertices redundant due to its computation time before running ILS. In comparison, the naive cut, approximate reductions and betweenness centrality cut failed to achieve similar advantages as the k-core cut. The best results were achieved by combining exact reductions of the input graph with ILS. This way, the optimal or best known results were reached very quickly. The combination of the exact reductions with ILS are therefore our recommendation for finding very large independent sets on difficult graphs. The benefits of our methods were especially large on the graphs `in-2004` and `cnr-2000`.

6.2. Future Work

As seen in Section 5, the improvements made by our techniques vary depending on the difficulty of the instance. The cuts and reductions are redundant on already easy-to-solve graphs. Therefore, developing methods to quickly determine whether applying our techniques on a graph is useful or not can further speed up general computations on a set of graphs. We already outlined the importance of the distribution of the highest degree vertices. Further investigating optimal thresholds and selecting the processing methods accordingly avoids wasting computation time. Moreover, there are other cutting methods that have not been tried yet. For example, KaHIP's partitioning utilities can be used to break extremely large graphs into smaller portions. Running our presented algorithms on the subgraphs may allow one to easily evaluate large independent sets on graphs difficult to handle with current computation powers. A similar approach is combining the exact reductions with cutting on the reduced graph. The reductions kernalize a graph and removing just very few of the kernel's vertices potentially further reduces the computation time. It needs to be tested whether this kernel cutting is too destructive on the MIS.

A. Data Structures for Cutting Algorithms

We store the graph in an adjacency list which is a vector containing a vector per vertex. A vertex's vector stores the vertex's neighbors. We also store edges in a vector where an edge is always directed and the only vertex stored per edge is its destination. A vertex stores its first outgoing edge's ID. This allows us to quickly iterate over a vertex v's neighbors in $\mathcal{O}(\deg(v))$ and easily insert and remove vertices from the graph. In addition, for large sparse graphs an adjacency list does not waste as much space as an adjacency matrix. Another useful data structure we use is a bucket priority queue. It provides containers for a discrete interval such that we can insert vertices in buckets corresponding to their degree. Since one of our main objectives is to remove high degree vertices, the bucket priority queue is very advantageous as it provides this exact functionality plus the ability to easily update the degrees of the removed vertices' neighbors.

References

[1] Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. In *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*, pages 70–81, 2015.

[2] Diogo Vieira Andrade, Mauricio G. C. Resende, and Renato Fonseca F. Werneck. Fast local search for the maximum independent set problem. *J. Heuristics*, 18(4):525–547, 2012.

[3] Eyjolfur Ingi Asgeirsson and Clifford Stein. Vertex cover approximations: Experiments and observations. In *Experimental and Efficient Algorithms, 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings*, pages 545–557, 2005.

[4] Eyjolfur Ingi Asgeirsson and Clifford Stein. Vertex cover approximations on random graphs. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, pages 285–296, 2007.

[5] Shaowei Cai, Kaile Su, and Qingliang Chen. EWLS: A new local search for minimum vertex cover. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[6] Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. NuMVC: An efficient local search algorithm for minimum vertex cover. *J. Artif. Intell. Res. (JAIR)*, 46:687–716, 2013.

[7] Shaowei Cai, Kaile Su, and Abdul Sattar. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artif. Intell.*, 175(9-10):1672–1696, 2011.

[8] Shaowei Cai, Kaile Su, and Abdul Sattar. Two new local search strategies for minimum vertex cover. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[9] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[10] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[11] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments, ALENEX 2008, San Francisco, California, USA, January 19, 2008*, pages 90–100, 2008.

[12] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Evaluation of labeling strategies for rotating maps. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pages 235–246, 2014.

[13] Andrea Grosso, Marco Locatelli, and Wayne J. Pullan. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *J. Heuristics*, 14(6):587–612, 2008.

[14] David S. Johnson and Michael A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

[15] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972.

[16] Sebastian Lamm, Peter Sanders, and Christian Schulz. Graph partitioning for independent sets. In *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*, pages 68–81, 2015.

[17] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Finding near-optimal independent sets at scale. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016.*

[18] Chu Min Li and Zhe Quan. Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 344–351, 2010.

[19] Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[20] George L. Nemhauser and Leslie E. Trotter Jr. Vertex packings: Structural properties and algorithms. *Math. Program.*, 8(1):232–248, 1975.

[21] Peter Sanders and Christian Schulz. High quality graph partitioning. In *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, pages 1–18, 2012.

[22] Pablo San Segundo, Fernando Matía, Diego Rodríguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 7(3):467–479, 2013.

[23] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & OR*, 38(2):571–581, 2011.

[24] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit:

An interactive tool suite for high-performance network analysis. *arXiv preprint arXiv:1403.3005*, 2014.

[25] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and Computation, 4th International Workshop, WALCOM 2010, Dhaka, Bangladesh, February 10-12, 2010. Proceedings*, pages 191–203, 2010.

[26] Mingyu Xiao and Hiroshi Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theor. Comput. Sci.*, 469:92–104, 2013.