Scaling up Local Search for Minimum Vertex Cover in Large Graphs by Parallel Kernelization

Wanru Gao[1], Tobias Friedrich[1,2], Timo Kötzing[2], and Frank Neumann[1]

[1] School of Computer Science, The University of Adelaide, Australia
[2] Hasso Plattner Institute, Potsdam, Germany

Abstract. We investigate how well-performing local search algorithms for small or medium size instances can be scaled up to perform well for large inputs. We introduce a parallel kernelization technique that is motivated by the assumption that graphs in medium to large scale are composed of components which are on their own easy for state-of-the-art solvers but when hidden in large graphs are hard to solve. To show the effectiveness of our kernelization technique, we consider the well-known minimum vertex cover problem and two state-of-the-art solvers called NuMVC and FastVC. Our kernelization approach reduces an existing large problem instance significantly and produces better quality results on a wide range of benchmark instances and real world graphs.

1 Introduction

Local search algorithms belong to the most successful approaches for many combinatorial optimization problems [1,2]. The general problem of local search approaches is that they can be trapped in local optima. Since often these approaches include a random initialization or random components, running an algorithm several times on a given instance might help with finding a global optimum. However, if the probability of getting stuck in a local optimum is high, then even repeated runs might not help to evade local optima.

In this paper, we present a new approach for scaling up existing high-performing local search solvers in order to perform well on large graphs. Our approach builds on the assumption that large graphs are composed of different (hidden) substructures. Substructures are often found in large social network graphs as social networks usually consist of (loosely connected) sub-communities. In large graphs, the issue of local optima might occur in the different substructures of the given problem instance; having a large number of these substructures where an algorithm even just fails with a small probability might make it very hard for local search approaches to obtain the optimal solution. We present a simple parallel kernelization approach that builds on theoretical investigations regarding substructures on large graphs.

Kernelization approaches have been shown to be very effective for designing algorithms which have a good performance guarantee [3,4]. Recently, a technique of incorporating kernelization in evolutionary algorithms is proposed to the NP-hard independent set problem [5]. The key idea of this approach is to pre-process a given instance by making optimal decisions on easy parts of the given input such that the overall problem

instance is reduced. There are several kernelization techniques available for the minimum vertex cover problem which perform well if the number of vertices in an optimal solution is small. However, the applicability to difficult instances which are usually dense graphs is limited as the pre-processing does not significantly reduce the problem instance size. As proposed by [6], identifying the 'backbone' solution component and then making use of this information is beneficial for the local search algorithm.

We present a new way of reducing the problem instance size by parallel kernelization (note that this is not a kernelization in the theoretical sense). In Section 2 we present theoretical investigations which assume that small substructures can be solved effectively by a local search heuristic; we then turn these observations into a parallel kernelization technique. The approach uses existing local search solvers to deal with large graphs. The key idea is to do μ parallel runs of such a solver and reduce the given instance by fixing components that have been selected in all μ runs and reducing the instance afterwards. The resulting *reduced* instance is then solved by an additional run of the local search solver and the combined result is returned as the final solution.

We consider the NP-hard minimum vertex cover (MVC) problem to illustrate the effectiveness of our approach. Popular local search approaches for tackling MVC include PLS [7], NuMVC [8], TwMVC [9], COVER [10] and FastVC [11]. Recently a branch-and-reduce algorithm for MVC is proposed [12]. Although this exact algorithm gets stuck in solving some well-known benchmark problems, it has shown good performance in dealing with sparse real world graphs.

The MVC algorithms are usually evaluated on standard benchmarks and (in more recent years on) large real world graphs. We take NuMVC and FastVC as the baseline local search solvers for our new kernelization approach. These two algorithms belong to the best-performing approaches for MVC. Our experimental results show that our new kernelization technique does not do any harm on instances where NuMVC and FastVC are already performing well while improving results on graphs containing benchmark instances as connected components. Furthermore, on real world graphs the kernelization reduces the large graphs significantly such that only $10\% - 20\%$ of vertices remain in the kernelized instance; hence, the kernelizing algorithm significantly outperforms the plain version of NuMVC and FastVC on most large graphs considered in our experimental investigations.

The outline of the paper is as follows. In Section 2, we present the theoretical motivation for our parallel kernelization technique that is based on the assumption that large graphs are composed of substructures. Section 3 outlines the resulting local search approach with parallel kernelization for the vertex cover problem. We evaluate the performance of our new approach on two state-of-the-art MVC solvers in Section 4 and 5 on combinations of classical benchmark instances and large real world graphs. Finally, we finish with some concluding remarks.

2 Substructures in large Graphs

Large graphs originating for example from social networks consist of a large number of vertices and edges. Our approach builds on the assumption that these graphs are composed of different substructures which on their own and at a small scale would

not be hard to handle by current local search approaches. This is for example the case for social networks which are composed of different communities. The difficulty arises through the composition of substructures which are not known to the algorithm and which are hard to extract from the given instances.

We would like to illustrate the problem by the following simple observations. Assume that you are running some (randomly initialized) local search algorithm on an instance that consists of different subparts s_i, $1 \leq i \leq k$, where each part s_i has the probability p_i of failing to obtain that optimal sub-solution independently of the other components. Then the probability of obtaining the optimal solution is

$$\prod_{i=1}^{k}(1-p_i).$$

Even if there is only a constant probability $p' = \min_{i=1}^{k} p_i$, $0 < p' < 1$ of failing in each of the k components, the probability that the local search algorithm would solve the overall instance would be exponentially small in k, meaning, we only succeed with probability

$$\prod_{i=1}^{k}(1-p_i) \leq \prod_{i=1}^{k}(1-p') = (1-p')^k \approx e^{-p'\cdot k}. \tag{1}$$

In our kernelization, we run μ instances of the same local search algorithm (randomly initialized). After some time t_1 for each of these runs, we stop the algorithm. After all μ solutions are computed, we *freeze* the setting for all those components which are set the same way in all μ runs; then we run the local search algorithm on the *reduced instances* with the frozen components removed.

Consider a component s_i again where the probability of failing is p_i. The probability that a run obtains the optimal solution for this component is $(1-p_i)$ and the probability that μ random runs identify an optimal solution is $(1-p_i)^\mu$. As long as the failure probability p_i is only a small constant and μ is not large, this term is still a constant that is sufficiently large, which shows that the kernalization will likely be successful as well. Let $|s_i|$ be the size of component s_i. Furthermore, we assume that the whole instance s is composed of the k subcomponents and we have $|s| = \sum_{i=1}^{k} |s_i|$.

The expected decrease in size of the original problem consisting of the components s_i is given by

$$\sum_{i=1}^{k}(1-p_i)^\mu |s_i|$$

Assuming $\hat{p} = \max_{i=1}^{k} p_i$, then we get

$$\sum_{i=1}^{k}(1-p_i)^\mu |s_i| \geq (1-\hat{p})^\mu \sum_{i=1}^{k} |s_i| = (1-\hat{p})^\mu \cdot |s|, \tag{2}$$

which reduces the whole instance by a fraction of at least $(1-\hat{p})^\mu$.

Algorithm 1: Local Search with Parallel Kernelization

1 Initialize P with μ solutions after μ different independent runs of MVC solver with cutoff time t_1.
2 Let set V_a be the set of vertices which are selected by all solutions in P.
3 Construct an instance I with vertices $v \notin V_a$ and edges which are not adjacent to any vertex in V_a.
4 Run MVC solver on instance I with cutoff time t_2 to get a minimum vertex cover V_s.
5 Construct the final solution $V = V_a \cup V_s$.

We now consider the probability that one of the different components has not achieved an optimal sub-solution in at least one of the μ runs. In such a case our algorithm could potentially reduce the instance and fix vertices of that component which do not belong to an optimal solution. In this case, the kernelization step would fail and prevent us from obtaining the overall optimal solution. Consider component s_i. The probability that all μ runs do not obtain the optimal sub solution for this component is p_i^μ. The probability that at least one of them obtains the optimal sub-solution is therefore at least

$$1 - p_i^\mu$$

and the probability that all components have at least one run where the optimal sub-solution is obtained is therefore at least

$$\prod_{i=1}^{k}(1 - p_i^\mu) \geq (1 - \hat{p}^\mu)^k \approx e^{-\hat{p}^\mu \cdot k}. \tag{3}$$

As an example, assume that the probability of the original approach failing on each subcomponent is 10%, $\mu = 3$, and $k = 50$. Then the expected reduction according to Equation 2 is $(1 - 0.1)^3 \cdot |s| = 0.729 \cdot |s|$, that is, the resulting instance has only 27.1% of the original number of vertices. The probability of not failing in the reduction step according to Equation 3 is $(1 - 0.1^3)^k = 0.999^k$ whereas the probability of a single run of the original approach not failing in at least one component according to Equation 1 is $(1 - 0.1)^k = 0.9^k$. For $k = 50$ we get a probability of not failing in the kernelization step of $0.999^{50} \approx 0.95$ and a probability of not failing in the original algorithm of $0.9^{50} \approx 0.005$.

The user can control μ, and from our calculations we observe a trade-off between reducing the number of vertices and the probability of fixing the wrong vertices in at least one of these components, depending on μ.

3 Parallel Kernelization for Minimum Vertex Cover

We now show how to use the ideas discussed in the previous section in an algorithmic sense. As mentioned previously, our approach assumes that there is already a good local search solver for the given problem P for small to medium size instances. Our goal is to

use parallel kernelization to make it work for large instances. While we taking the well-known NP-hard minimum vertex cover problem as an example problem, we expect that our approach is applicable to a wide range of other problems as well.

The minimum vertex cover (MVC) problem can be defined as follows. Given an undirected graph $G = (V, E)$ where V denotes the set of vertices and E denotes the set of edges, the goal is to find a smallest subset $C \subseteq V$ such that for all edge $e \in E$ there is at least one endpoint included in C.

The main idea is to kernelize the vertex set and form a smaller instance for the MVC solver to solve. Firstly the MVC solver is run μ times on the given graph $G = (V, E)$ with a cutoff time t_1 for each run to achieve a set of μ solutions. The vertices which are selected in all μ solutions, are added to a separate set V_a and the edges that are covered by the vertices of V_a are removed from the edge set. The new instance $G' = (V', E')$ is formed by the vertices that are not selected in all μ solutions and the edge set after deletion, meaning, we have $V' = V \setminus V_a$ and $E' = E \setminus \{e \in E \mid e \cap V_a \neq \emptyset\}$. The MVC solver is run on the new instance G' to obtain a minimum vertex cover V_s. The overall solution for the original graph G is

$$VC = V_a \cup V_s$$

and consists of the set of vertices which are selected in all μ initial solutions and the minimum vertex cover achieved by the MVC solver running on the new instance G'. It should be noted that it is crucial that the cutoff time t_1 allows the μ runs to have obtained at least nearly locally optimal solutions. A detailed description of our approach is given in Algorithm 1.

For our experimental investigations we use NuMVC [8] and FastVC [11] as the MVC solvers. Both algorithms are based on the idea of iteratively solving the deterministic problem from MVC.

NuMVC is one of the best performing local search approaches for MVC and has an advantage over TwMVC [9] in that it does not require parameter tuning for different types of benchmark instances. The authors introduce techniques to select the vertices for exchange in two separate stages, which enable the local search in a wider neighbourhood. The NuMVC algorithm keeps track of the exchange in order to avoid reversing behavior. These two techniques make NuMVC perform well in dealing with most MVC benchmark problems.

FastVC is a fast local search algorithm designed for dealing with large MVC problems. It involves some relaxation in selecting the candidate vertices for exchange, which accelerate the search process. It has performed well in some large real world graphs.

In the following sections, we discuss our experiments carried out with Algorithm 1 compared with the single run of the MVC solver; the total time budget that both approaches can use is the same.

All of the experiments are executed on a machine with 48-core AuthenticAMD 2.80GHz CPU and 128GByte RAM; note that the program uses only a single core. The memory consumption depends on the instance size and the MVC solver. The runtime will benefit from multi-threading and other parallel execution techniques.

	Instance			NuMVC-PK				NuMVC			Comparison			
Name	OPT	$\|V\|$	$\|E\|$	$\|V'\|$	$\|E'\|$	$VC_{\min}$	VC_{avg}	$VC_{\min}$	VC_{avg}	t_{avg}	Δ	Δ'	p-value	
frb40-19-1_10	7,200	7,600	413,140	576.0	1469.1	7,200	7,200.0±0	7,200	7,200.0±0	294.10	0	0.0	NA	no diff.
frb40-19-2_10	7,200	7,600	412,630	1,538.7	12,595.2	7,202	7,203.6±1.174	7,204	7,206.2±0.919	1,498.15	2	2.6	0.0005	better
frb40-19-3_10	7,200	7,600	410,950	1,201.8	7,466.7	7,200	7,200.3±0.483	7,200	7,201.9±0.994	1,645.52	0	1.6	0.0015	better
frb40-19-4_10	7,200	7,600	416,050	1,369.5	10,071.3	7,200	7,201.7±0.949	7,201	7,203.2±1.135	1,853.66	1	1.5	0.0083	better
frb40-19-5_10	7,200	7,600	416,190	1,599.5	13,801.8	7,203	7,205.5±1.269	7,205	7,206.7±1.252	1,438.47	2	1.2	0.0669	no diff.
frb45-21-1_10	9,000	9,450	591,860	1,722.6	14,822.9	9,000	9,001.6±0.966	9,001	9,002.1±0.994	1,929.33	1	0.5	0.3215	no diff.
frb45-21-2_10	9,000	9,450	586,240	1,808.0	16,134.1	9,002	9,003.5±1.269	9,002	9,003.9±0.994	2,050.30	0	0.4	0.5058	no diff.
frb45-21-3_10	9,000	9,450	582,450	1,875.2	17,610.1	9,004	9,007.0±1.247	9,006	9,007.1±0.738	1,936.15	2	0.1	0.8390	no diff.
frb45-21-4_10	9,000	9,450	585,490	1,735.1	14,302.0	9,001	9,003.1±1.197	9,004	9,004.5±0.707	1,606.35	3	1.4	0.0085	better
frb45-21-5_10	9,000	9,450	585,790	1,843.3	17,119.2	9,002	9,004.7±1.767	9,003	9,005.4±1.897	1,889.51	1	0.7	0.4900	no diff.
frb50-23-1_10	11,000	11,500	800,720	2,127.5	20,889.9	11,006	11,008.9±1.729	11,009	11,010.8±1.033	2,124.59	3	1.9	0.0164	better
frb50-23-2_10	11,000	11,500	808,510	2,079.4	19,760.5	11,009	11,011.0±1.414	11,010	11,011.8±1.033	2,035.62	1	0.8	0.1301	no diff.
frb50-23-3_10	11,000	11,500	810,680	2,133.2	21,310.8	11,010	11,011.5±1.269	11,010	11,012.6±1.075	1,795.17	0	1.1	0.0512	no diff.
frb50-23-4_10	11,000	11,500	802,580	2,116.1	20,679.4	11,003	11,005.6±1.506	11,004	11,006.3±1.252	2,219.85	1	0.7	0.2963	no diff.
frb50-23-5_10	11,000	11,500	800,350	2,097.7	20,008.5	11,006	11,007.7±1.337	11,009	11,010.2±0.632	1,885.96	3	2.5	0.0002	better
brock800_2_10	7,760	8,000	1,114,340	930.2	11,653.2	7,790	7,779.0±0	7,790	7,790.0±0	205.93	0	0.0	NA	no diff.
brock800_2_20	15,520	16,000	2,228,680	1,850.3	23,019.0	15,580	15,580.0±0	15,580	15,580.0±0	1,219.59	0	0.0	NA	no diff.
brock800_2_30	23,280	24,000	3,343,020	2,822.9	35,894.6	23,370	23,371.0±0.667	23,370	23,370.7±0.823	2,349.51	0	-0.3	0.3486	no diff.
brock800_4_10	7,740	8,000	1,119,570	888.0	10,866.5	7,790	7,790.0±0	7,790	7,790.0±0	285.43	0	0.0	NA	no diff.
brock800_4_20	15,480	16,000	2,239,140	1,842.5	23,371.4	15,580	15,580.0±0	15,580	15,580.2±0.422	1,880.70	0	0.2	0.1675	no diff.
brock800_4_30	23,220	24,000	3,358,710	2,836.8	36,873.8	23,372	23,374.0±0.943	23,372	23,373.7±1.767	2,248.83	0	-0.3	0.4157	no diff.
C1000.9_10	9,320	10,000	494,210	2,603.0	24,603.4	9,321	9,323.3±1.252	9,323	9,324.9±1.101	1,345.04	2	1.6	0.0111	better
C2000.5_10	19,840	20,000	9,991,640	739.1	10,880.3	19,842	19,843.9±0.876	19,840	19,840.0±0	1,003.06	-2	-3.9	0.0001	worse
C2000.9_10	19,200	20,000	1,994,680	3,339.6	42,224.1	19,239	19,243.8±2.394	19,232	19,239.6±3.471	1,664.31	-3	-4.2	0.0030	worse

Table 1: This table contains instances that have been tested on, which are generated by duplicating one existing hard instance in BHOSLIB and DIMACS benchmark. The instance name contains the name of original instance and the number of copies. The cutoff time of single NuMVC is set to $3,000$ seconds. The parameters for NuMVC-PK are set to $\mu = 5$, $t_1 = 500$ and $t_2 = 500$. The average time for NuMVC to find the local optima is reported in column t_{avg}. The p-value is labelled as NA if the results from the 10 independent runs of the two algorithms are the same.

4 Experimental Results from NuMVC with Parallel Kernelization

The implementation of NuMVC is open-source and implemented in C++. We compiled the source code with g++ with '-O2' option. The parameter setting follows what is reported in [8]. We take NuMVC as the MVC solver in Algorithm 1 The new approach to solve MVC is referred to as NuMVC-PK, since it is strongly based on the original NuMVC program.

Each experiment on a certain instance for each algorithm is executed 10 times in order to gather statistics. The cutoff time for the first run in NuMVC-PK is set based on initial experimental investigations on the different classes of instances considered. Based on our theoretical investigations carried out in Section 2, it is important that each of the μ runs obtains at least a nearly locally optimal solution for the given problem. This implies that a too small cutoff time t_1 might have detrimental effects.

4.1 DIMACS and BHOSLIB Benchmarks

There are some well-known MVC benchmarks which have been used to evaluate the performance of different MVC solvers. Two of the benchmarks are the *DIMACS* and the *BHOSLIB* benchmark sets.

The *BHOSLIB* (Benchmarks with Hidden Optimum Solutions) problems are generated from translating the binary Boolean Satisfiability problems randomly generated

	Instance		NuMVC-PK				NuMVC		Comparison			
Name	$\|V\|$	$\|E\|$	$\|V'\|$	$\|E'\|$	$VC_{\min}$	VC_{avg}	$VC_{\min}$	VC_{avg}	Δ	Δ'	p-value	
dImacs10-citationCiteseer	268,495	1,156,647	47,588.5	35,638.2	118,175	118,188.2±8.817	118,329	118,345.5±13.640	154	157.3	0.0002	better
dImacs10-coAuthorsCiteseer	227,320	814,134	63,325.5	37,318.9	129,193	129,193.1±0.316	129,193	129,195.1±1.197	0	2.0	0.0005	better
dImacs10-cs4	22,499	43,858	12,607.1	18,548.6	13,376	13,379.7±2.869	13,364	13,380.6±6.703	-12	0.9	0.2383	no diff.
dImacs10-cti	16,840	48,232	14,788.0	41,176.5	8,752	8,782.4±26.082	8,752	8,774.9±14.813	0	-7.5	0.6478	no diff.
dImacs10-delaunay-n15	32,768	98,274	9,991.4	11,339.9	22,460	22,464.4±4.033	22,460	22,463.1±2.025	0	-1.3	0.6185	no diff.
dImacs10-delaunay-n16	65,536	196,575	24,120.0	28,304.3	44,995	45,007.7±6.430	45,045	45,070.4±19.115	50	62.7	0.0002	better
dImacs10-delaunay-n17	131,072	393,176	53,422.6	65,915.5	90,356	90,373.1±12.306	90,639	90,677.8±23.522	283	304.7	0.0002	better
dImacs10-delaunay-n18	262,144	786,396	92,241.1	83,516.9	183,791	183,876.3±48.413	181,481	181,541.4±32.695	-2,310	-2,334.9	0.0000	worse
dImacs10-delaunay-n19	524,288	1,572,823	183,692.3	166,235.6	367,552	367,662.8±48.385	370,869	371,033.8±119.084	3,317	3,371.0	0.0002	better
dImacs10-fe-body	45,087	163,734	12,001.0	13,040.01	31,361	31,364.6±3.062	31,346	31,348.0±1.633	-15	-16.6	0.0002	better
dImacs10-fe-rotor	99,617	662,431	36,987.1	73,892.5	78,058	78,193.5±77.739	78,171	78,272.5±59.642	113	79.0	0.0211	better
dImacs10-fe-tooth	78,136	452,591	7,298.1	5,461.8	50,347	50,347.8±0.789	50,346	50,348.0±0.943	-1	0.2	0.4933	no diff.

Table 2: Experimental results on instances from some instances in *DIMACS10* benchmark set. The cutoff time of the single NuMVC run is set to $1,000$ seconds. The parameters for NuMVC-PK are set to $\mu = 3$, $t_1 = 200$ and $t_2 = 400$.

	Instance		NuMVC-PK				NuMVC		Comparison			
Name	$\|V\|$	$\|E\|$	$\|V'\|$	$\|E'\|$	$VC_{\min}$	VC_{avg}	$VC_{\min}$	VC_{avg}	Δ	Δ'	p-value	
soc-BlogCatalog	88,784	2,093,195	6,071.8	3,871.1	20,752	20,752.0±0	20,752	20,753.0±0.816	0	1.0	0.0021	better
soc-brightkite	56,739	212,945	9,717.3	6,157.2	21,190	21,190.0±0	21,194	21,196.7±2.003	4	6.7	0.0001	better
soc-buzznet	101,163	2,763,066	9,104.1	6,328.5	30,614	30,615.5±1.080	30,614	30,614.4±0.516	0	-1.1	0.0182	worse
soc-delicious	536,108	1,365,961	29,325.7	40,112.0	85,477	85,494.0±13.275	85,553	85,587.6±19.744	76	93.6	0.0002	better
soc-digg	770,799	5,907,132	24,412.2	15,814.8	103,239	103,240.7±0.823	103,297	103,323.4±10.793	58	82.7	0.0001	better
soc-douban	154,908	327,162	304.2	153.5	8,685	8,685.0±0	8,685	8,685.0±0	0	0.0	NA	no diff.
soc-epinions	26,588	100,120	4,730.8	2,864.2	9,757	9,757.0±0	9,757	9,757.0±0	0	0.0	NA	no diff.
soc-flickr	513,969	3,190,452	62,566.8	44,148.2	153,277	153,283.2±3.676	153,346	153,353.7±5.438	69	70.5	0.0002	better
soc-flixster	2,523,386	7,918,801	4,123.2	2,283.0	96,317	96,330.1±14.083	96,318	96,321.5±1.900	1	-8.6	1	no diff.
soc-FourSquare	639,014	3,214,986	10,305.2	7,374.8	90,110	90,111.8±1.398	90,131	90,136.9±3.635	21	25.1	0.0002	better
soc-gowalla	196,591	950,327	39,962.5	27,978.6	84,252	84,260.3±6.237	84,309	84,332.0±10.176	57	71.7	0.0002	better
soc-lastfm	1,191,805	4,519,330	3,738.3	2,149.1	78,688	78,689.2±1.033	78,694	78,696.2±2.098	6	7.0	0.0002	better
soc-LiveMocha	104,103	2,193,083	12,452.3	9,062.6	43,428	43,432.1±3.178	43,437	43,440.7±3.889	9	8.6	0.0003	better
soc-slashdot	70,068	358,647	11,063.8	6,839.8	22,373	22,373.0±0	22,376	22,379.1±1.912	3	6.1	0.0001	better
soc-twitter-follows	404,719	713,319	42.0	21.0	2,323	2,323.0±0	2,323	2,323.0±0	0	0.0	NA	no diff.
soc-youtube	495,957	1,936,748	65,727.2	42,496.9	146,469	146,500.9±16.093	146,453	146,469.3±8.667	-16	-31.6	0.0006	worse
soc-youtube-snap	1,134,890	2,987,624	160,650.9	88,012.0	277,828	277,857.6±21.752	278,542	278,603.7±31.049	714	746.1	0.0002	better
ca-citeseer	227,320	814,134	68,049	41,230	129,193	129,193.0±0	129,194	129,194.8±0.919	1	1.8	0.0001	better
ca-coauthors-dblp	540,486	15,245,729	93,319	69,523	472,250	472,257.3±5.832	472,324	472,334.8±7.540	74	77.5	0.0002	better
ca-dblp-2010	226,413	716,460	65,615	39,019	121,969	121,969.5±0.527	121,971	121,974.4±1.955	2	4.9	0.0001	better
ca-dblp-2012	317,080	1,049,866	78,406	43,526	164,951	164,953.2±1.687	164,956	164,958.2±2.486	5	5.0	0.0005	better
ca-MathSciNet	332,689	820,644	78,800	48,359	139,955	139,958.7±2.163	139,981	139,988.1±4.175	26	29.4	0.0002	better
web-arabic-2005	163,598	1,747,269	26,120	19,511	114,444	114,448.1±2.759	114,468	114,475.3±3.529	24	27.2	0.0002	better
web-baidu-baike-related	415,641	3,284,387	67,615	74,066	143,581	143,629.8±26.318	144,155	144,190.2±19.424	574	560.4	0.0002	better
web-google-dir	875,713	5,105,039	117,405	95,026	347,783	347,795.6±12.842	347,771	347,826.4±55.674	-12	30.8	0.1403	no diff.
web-it-2004	509,338	7,178,413	71,247	64,798	415,017	415,043.3±21.505	414,861	414,895.6±14.447	-156	-147.7	0.0002	worse
web-sk-2005	121,422	334,419	35,409	26,361	58,179	58,181.9±2.424	58,201	58,206.4±3.718	22	24.5	0.0002	better

Table 3: Experimental results on instances from some real world graphs about social networks, collaboration networks and websites. The cutoff time of the single NuMVC run is set to $1,000$ seconds. The parameters for NuMVC-PK are set to $\mu = 3$, $t_1 = 300$ and $t_2 = 100$. The p-value is labelled as NA if the results from the 10 independent runs of the two algorithms are the same.

based on the model RB [13]. These instances have been proven to be hard to solve, both theoretically and practically. The *DIMACS* benchmark is a set of challenge problems which comes from the Second *DIMACS* Implementation Challenge for Maximum Clique, Graph Coloring and Satisfiability [14]. The original Max Clique problems are converted to complement graphs to serve as MVC problems.

With the same overall time budget, both NuMVC and NuMVC-PK have good success rate for most of the instances.

4.2 Multiple Copies of the Well-Known Benchmark Problems

Most *BHOSLIB* instances and *DIMACS* instances can be solved with good success rates by NuMVC [8]. We propose some simple combinations of these existing benchmarks as new test cases. These will serve as very simple first test cases for our kernelization method. The new instances are composed of several sub-graphs and large in size of both vertices and edges. In particular, we construct a new instance by considering independent copies of an existing instance. Each single copy is easy to be solved by the MVC solver with a high sucess rate, while the combined instance is much harder to solve.

Some examples of these kinds of instances are given in Table 1. The original instances are selected from the *BHOSLIB* benchmark set or the *DIMACS* benchmarks. The last number in the instance name after the underscore denotes the number of copies of the given instance indicated by the first part of the instance name. Although the original instances can be solved by NuMVC in reasonable time, it takes much longer time for NuMVC to solve the multiplicated new instances. NuMVC may get trapped at local optima, which are far away from the global optima in search space. Table 1 shows the comparison between results from NuMVC-PK and single run of NuMVC. The basic information about the instances is included in Table 1 in the column Instance. The OPT column lists the optimal (or minimum known) vertex cover size. The numbers in the $|V|$ and $|E|$ columns are the numbers of vertices and edges in the corresponding instances. NuMVC-PK is executed with parameters $\mu = 5$, $t_1 = 500$ and $t_2 = 500$, which means 5 independent runs of NuMVC to get initial solutions after 500 seconds and then the original instance is processed based on the information gathered from the five solutions to generate a new instance. As the last step, NuMVC is run for another 500 seconds on the newly generated instance to achieve the final solution. The information of the generated reduced instance is listed in columns $|V'|$ and $|E'|$, where the average number of non-isolated vertices and edges of the new instance are listed. NuMVC is executed for $3,000$ seconds to be compared to NuMVC-PK, which had the same time budget. We report the average time for NuMVC to find the best solution in each run in the column t_{avg}.

The size difference between the minimum vertex cover and the average value found by the two approaches is reported in the column of Δ and Δ' respectively. We used the Wilcoxon unpaired signed-rank test on the solutions from different runs of the algorithms for a given instance; the p-value is listed in Table 1. The difference is evaluated based on a significance level of 0.05.

Since *BHOSLIB* and *DIMACS* benchmarks are hard MVC problems, making sure all sub-graphs to be solved to optimality is hard for a single run of NuMVC, which easily gets trapped in some local optimum. On the other hand, NuMVC-PK shrinks the large instances and takes a fresh start on the reduced instance, thereby improving the performance of the local search.

From the results we see that NuMVC-PK is able to reduce the instance size. For the duplicated *BHOSLIB* and *DIMACS* instances, after 5 runs of NuMVC, the NuMVC-PK generates new instances which keep only 1% to 3% of the edges and 8% to 20% of the vertices. Unlike NuMVC, which usually makes no improvement after $2,000$ seconds, NuMVC-PK finds the global optimum for 4 instances where NuMVC ends up with

local optima after $3,000$ seconds in all 10 runs. The parallel kernelization mechanism significantly improves the performance of single run of NuMVC in 7 of the instances.

4.3 Real World Graphs

Now we turn our attention to comparing NuMVC-PK with NuMVC on large real world graphs as given by [15]. All of these selected graphs are undirected, unweighted and with a large number of vertices and edges. In contrast to the benchmark sets considered in Section 4.2, the global optima of these instances are unknown. The graphs examined are taken from the social network, collaboration network and web link (Miscellaneous) network packages. Some samples are also selected from the *DIMACS10* data sets which come from the 10th *DIMCAS* implementation challenge [16]. The graphs have a number of vertices in the range of $15,000$ to $2,600,00$ and a number of edges in the range of $40,000$ to $16,000,000$.

The experimental results are summarized in Tables 3 and 2. Just as in Table 1, the columns of $|V|$ and $|E|$ provide the brief information of the graphs (number of vertices and edges, respectively). The categories NuMVC-PK and NuMVC give the comparison between NuMVC-PK and single run of NuMVC. Since the large real world graphs are not designed to be as hard as the combined *BHOSLIB* instances, we use $\mu = 3$ to get the initial solution set. The minimum vertex cover found and the average number of minimum vertex cover in the ten independent runs is reported in the table. The standard deviation for each instance is also included to show the stability of the algorithms. NuMVC is run for $1,000$ seconds, corresponding to the total budget of NuMVC-PK. Some easy instances which can be easily solved by single run of NuMVC in short run time are omitted from the table since both algorithms have a 100% success rate.

For the real world graphs in social networks, collaboration networks and web link networks packages, NuMVC-PK reduces the instances size by more than 90% in the number of vertices and 70% in the number of edges. The size of the instance is one of the main factors that affect the performance of the MVC solvers for the real world graphs. The instances after shrinking have less than $200,000$ vertices and $100,000$ edges. For graphs in *dImacs10* package, the generated instances maintain around 20% vertices and 40% edges in most cases.

Regarding Tables 3 and 2, we make the following observations.

- NuMVC-PK finds smaller minimum vertex cover in the ten independent runs than NuMVC in 26 out of the 39 graphs.
- There are 4 graphs for which NuMVC-PK is not able to return a better solution than NuMVC. These graphs have the property that they are hard or large instances so local optima are not reached within time t_1 or even $1,000$ seconds.
- There are 10 graphs where NuMVC-PK finds a minimum vertex cover smaller by more than 50 than NuMVC.

For some large instances, the initialization process of NuMVC is very time consuming. Enough time should be given for NuMVC to get initial solutions at least near the local optima. For the same time limit, longer single initial runs are more beneficial than shorter initial runs and longer runs after the freezing phase. Therefore, a combination of larger t_1 and smaller t_2 may result in a better solution for these instances.

	Instance			FastVC-PK				FastVC		Comparison			
Name	$\|V\|$	$\|E\|$	$\|V'\|$	$\|E'\|$	$VC_{\min}$	VC_{avg}	$VC_{\min}$	VC_{avg}	Δ	Δ'	p-value	
soc-BlogCatalog	88,784	2,093,195	6,128.0	3,952.0	20,752	20,752.0±0	20,752	20,752.0±0	0	0.0	NA	no diff.
soc-brightkite	56,739	212,945	9,673.3	6,167.4	21,190	21,190.0±0	21,190	21,190.1±0.316	0	0.1	0.3681	no diff.
soc-buzznet	101,163	2,763,066	9,014.2	6,200.0	30,625	30,625.4±0.516	30,625	30,625.4±0.516	0	0.0	1	no diff.
soc-delicious	536,108	1,365,961	23,939.2	23,483.8	86,121	86,139.5±19.363	86,215	86,230.8±15.303	94	91.3	0.0002	better
soc-digg	770,799	5,907,132	23,165.2	14,455.8	103,244	103,244.4±0.516	103,244	103,244.7±0.483	0	0.3	0.2039	no diff.
soc-douban	154,908	327,162	297.6	152.1	8,685	8,685.0±0	8,685	8,685.0±0	0	0.0	NA	no diff.
soc-epinions	26,588	100,120	4,841.7	3,032.7	9,757	9,757.1±0.316	9,757	9,757.6±0.966	0	0.5	0.2328	no diff.
soc-flickr	513,969	3,190,452	64,695.7	40,410.1	153,271	153,271.0±0	153,271	153,271.0±0	0	0.0	NA	no diff.
soc-flixster	2,523,386	7,918,801	3,807.5	2,068.8	96,317	96,317.0±0	96,317	96,317.0±0	0	0.0	NA	no diff.
soc-FourSquare	639,014	3,214,986	9,306.2	6,177.9	90,108	90,109.0±0.471	90,108	90,108.8±0.422	0	-0.2	0.3566	no diff.

Table 4: Experimental results on instances from some real world graphs about social networks. The cutoff time of the single FastVC run is set to $1,000$ seconds. The parameters for FastVC-PK are set to $\mu = 3, t_1 = 200$ and $t_2 = 400$. The p-value is labelled as NA if the results from the 10 independent runs of the two algorithms are the same.

	Instance			FastVC-PK				FastVC			Comparison			
Name	OPT	$\|V\|$	$\|E\|$	$\|V'\|$	$\|E'\|$	$VC_{\min}$	VC_{avg}	$VC_{\min}$	VC_{avg}	t_{avg}	Δ	Δ'	p-value	
frb40-19-1_10	7,200	7,600	413,140	1,523.6	11,972.5	7,208	7,212.0±3.091	7,213	7,217.2±2.098	537.57	5	5.2	0.0014	better
frb40-19-2_10	7,200	7,600	412,630	1,620.7	13,962.8	7,218	7,219.5±1.581	7,218	7,221.1±1.663	264.05	0	1.6	0.0507	no diff.
frb40-19-3_10	7,200	7,600	410,950	1,526.7	12,369.6	7,209	7,212.1±1.853	7,213	7,216.4±2.066	612.91	4	4.3	0.0006	better
frb40-19-4_10	7,200	7,600	416,050	1,590.0	13,536.9	7,217	7,219.8±1.549	7,221	7,222.6±1.075	236.07	4	2.8	0.0009	better
frb40-19-5_10	7,200	7,600	416,190	1,583.7	13,234.2	7,214	7,217.6±2.221	7,219	7,221.0±1.054	593.26	5	3.4	0.0005	better
frb45-21-1_10	9,000	9,450	591,860	1,833.4	16,832.7	9,015	9,023.8±4.341	9,024	9,026.6±2.011	872.19	9	2.8	0.0795	better
frb45-21-2_10	9,000	9,450	586,240	1,839.6	16,771.4	9,019	9,025.2±3.327	9,019	9,026.9±3.071	785.91	0	1.7	0.2151	no diff.
frb45-21-3_10	9,000	9,450	582,450	1,857.5	17,019.8	9,023	9,026.8±2.394	9,028	9,030.4±1.430	119.15	5	3.6	0.0029	better
frb45-21-4_10	9,000	9,450	585,490	1,825.9	16,049.0	9,019	9,022.9±3.281	9,024	9,025.9±1.524	459.29	5	3.0	0.0399	better
frb45-21-5_10	9,000	9,450	585,790	1,855.7	17,270.0	9,021	9,024.6±2.413	9,025	9,027.2±1.619	530.08	4	2.6	0.0198	better
frb50-23-1_10	11,000	11,500	800,720	2,068.0	19,489.4	11,029	11,034.9±3.542	11,032	11,036.9±2.378	434.04	3	2.0	0.2995	no diff.
frb50-23-2_10	11,000	11,500	808,510	2,061.0	19,335.2	11,033	11,035.1±1.912	11,032	11,035.9±2.025	842.49	-1	0.8	0.3555	no diff.
frb50-23-3_10	11,000	11,500	810,680	2,133.2	21,310.8	11,033	11,035.6±1.955	11,034	11,036.9±1.449	170.23	1	1.3	0.1532	no diff.
frb50-23-4_10	11,000	11,500	802,580	2,073.7	19,727.2	11,028	11,032.8±3.765	11,030	11,034.6±2.319	648.05	2	1.8	0.2538	no diff.
frb50-23-5_10	11,000	11,500	800,350	2,061.3	19,186.2	11,024	11,030.8±2.658	11,032	11,035.1±1.912	143.89	8	4.3	0.0008	better
brock800_2_10	7,760	8,000	1,114,340	915.3	11,355.5	7,793	7,797.1±2.923	7,799	7,802.0±1.247	866.12	6	4.9	0.0019	better
brock800_2_20	15,520	16,000	2,228,680	1,818.7	22,420.9	15,593	15,598.1±3.178	15,607	15,608.9±1.370	747.21	14	10.8	0.0002	better
brock800_2_30	23,280	24,000	3,343,020	2,728.7	33,696.8	23,398	23,404.7±4.572	23,413	23,418.6±4.222	187.10	15	13.9	0.0002	better
brock800_4_10	7,740	8,000	1,119,570	915.4	11,470.7	7,797	7,799.3±2.003	7,799	7,803.5±1.900	920.42	2	4.2	0.0017	better
brock800_4_20	15,480	16,000	2,239,140	1,813.4	22,549.1	15,598	15,604.6±3.627	15,608	15,612.4±2.503	1,015.30	10	7.8	0.0002	better
brock800_4_30	23,220	24,000	3,358,710	2,715.4	33,832.3	23,399	23,408.4±7.777	23,419	23,422.0±1.414	948.31	20	13.6	0.0006	better
C1000.9_10	9,320	10,000	494,210	2,507.6	22,457.3	9,324	9,329.6±2.459	9,328	9,331.1±1.853	920.80	4	1.5	0.1513	no diff.
C2000.5_10	19,840	20,000	9,991,640	690.0	9,468.4	19,848	19,851.7±2.003	19,857	19,858.6±1.350	745.16	9	6.9	0.0002	better
C2000.9_10	19,200	20,000	1,994,680	3,268.6	40,788.3	19,266	19,272.4±4.006	19,269	19,275.6±4.326	386.93	3	3.2	0.1195	no diff.

Table 5: This table contains instances that have been tested on, which are generated by duplicating one existing hard instance in BHOSLIB benchmark. The instance name contains the name of original instance and the number of copies. The cutoff time of single FastVC is set to $3,000$ seconds. The parameters for FastVC-PK are set to $\mu = 5, t_1 = 500$ and $t_2 = 500$. The average time for FastVC to find the local optima is reported in column t_{avg}.

5 Experimental Results from FastVC with Parallel Kernelization

Like NuMVC, FastVC is also open-source and implemented in C++. The original code in version 2015.11 is compiled with g++ and '-O2' option. We use the parameters as reported in [11]. Following the same terminology used for NuMVC, we refer to the new algorithm with FastVC as the MVC solver in Algorithm 1 as FastVC-PK.

Both FastVC and FastVC-PK are tested on each certain instance for 10 times in order to gather statistics. The cutoff time for the initial runs in FastVC-PK is set based

on initial experimental investigations on the different classes of instances considered as FastVC. This step is important so that each of the μ runs should obtain at least a nearly locally optimal solution for the given problem according to the theoretical analysis in Section 2.

Similar to NuMVC, the integration of parallel kernelization into FastVC keeps the good performance of the original algorithm in solving *BHOSLIB* and *DIMACS* benchmarks. Some of the experimental results are reported in Table 4. Then we turn to test FastVC and FastVC-PK with the multiple copies of the benchmark problems. The results are presented in Table 5. The layout of the table follows the same rules of the tables in the previous section.

According to the statistics in Table 5, we can make the following observations:

- Among the 24 instances, FastVC-PK significantly improves the solution quality of FastVC in 16 instances.
- In the 8 graphs where both algorithms obtain similar solutions, there are 5 instances where FastVC-PK finds the minimum vertex cover in the 10 runs and the average solution size found by FastVC-PK is smaller than that from FastVC in 7 instances.
- There are 21 graphs where FastVC-PK finds a minimum vertex cover smaller than that from FastVC.

From the experimental results, we find that the instances generated after parallel kernelization have around 20% of the non-isolated nodes and less than 3% of the edges in the original graphs. In most instances shown in the table, FastVC gets stuck in some local optima after running for less than $1,000$ seconds and after that there is no improvement until the time limit is reached.

The graphs constructed from several independent copies of the existing instances are denser than the real world graphs. A single run of FastVC performs good in solving the real world graphs. Experiments are conducted on FastVC and FastVC-PK on these kinds of graphs as well. Since FastVC-PK is already able to find good solutions in a short time for most real world graphs from the benchmark set, both approaches obtain similar results for the sparse real world graphs. The parallel kernelization mechanism only improves FastVC in a few instances. The experimental results from some example testcases are shown in Table 4.

6 Conclusions

We have presented a new approach on scaling up local search algorithms for large graphs. Our approach builds on the theoretical assumption that large graphs are composed of different substructures which are on their own not hard to be optimized. Our approach is based on parallel kernelization and reduces the given graph by making μ parallel randomized runs of the given local search and fixing components that have been chosen in all μ runs. The resulting instance is then tackled by an additional run of the local search approach. Considering the Vertex Cover problem and the state-of-the-art local search solver NuMVC and FastVC, we have shown that our parallel kernelization technique is able to reduce standard benchmark graphs and large real world graphs to

about 10 – 20% of their initial sizes. Our approach outperforms the baseline local search algorithm NuMVC and FastVC in most test cases.

The parallel kernelization approach presented in this paper can be applied to a wide range of combinatorial optimization problems for which well performing local search solvers are available. We plan to investigate the application to other problems such as Maximum Clique and Maximum Independent Set in the future.

References

1. Aarts, E., Lenstra, J.K., eds.: Local Search in Combinatorial Optimization. Discrete Mathematics and Optimization. Wiley, Chichester, UK (1997)
2. Hoos, H.H., Stützle, T.: Stochastic local search : foundations and applications. Elsevier, San Francisco, CA (2005)
3. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. Texts in Computer Science. Springer (2013)
4. Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms. Springer (2015)
5. Lamm, S., Sanders, P., Schulz, C., Strash, D., Werneck, R.F.: Finding near-optimal independent sets at scale. In: Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016. (2016) 138–150
6. Zhang, W., Rangan, A., Looks, M.: Backbone guided local search for maximum satisfiability. In: IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence. (2003) 1179–1186
7. Pullan, W.: Phased local search for the maximum clique problem. Journal of Combinatorial Optimization **12**(3) (2006) 303–323
8. Cai, S., Su, K., Sattar, A.: Two new local search strategies for minimum vertex cover. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence. (2012)
9. Cai, S., Lin, J., Su, K.: Two weighting local search for minimum vertex cover. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence. (2015) 1107–1113
10. Richter, S., Helmert, M., Gretton, C.: A stochastic local search approach to vertex cover. In Hertzberg, J., Beetz, M., Englert, R., eds.: KI 2007: Advances in Artificial Intelligence. Volume 4667 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 412–426
11. Cai, S.: Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. (2015) 747–753
12. Akiba, T., Iwata, Y.: Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. Theor. Comput. Sci. **609** (2016) 211–225
13. Xu, K., Boussemart, F., Hemery, F., Lecoutre, C.: A simple model to generate hard satisfiable instances. In: IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence. (2005) 337–342
14. Johnson, D.J., Trick, M.A., eds.: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993. American Mathematical Society, Boston, MA, USA (1996)
15. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: AAAI. (2015) 4292–4293
16. Bader, D.A., Meyerhenke, H., Sanders, P., Schulz, C., Kappes, A., Wagner, D.: Benchmarking for Graph Clustering and Partitioning. In: Encyclopedia of Social Network Analysis and Mining. Springer New York, New York, NY (2014) 73–82