WFCVC: A New WFC-based Heuristic for Vertex Cover

Yanran Ma[1] and David Perkins[2]

[1]Nanjing Foreign Language School, Nanjing, China
[2]Hamilton College, New York, United States

Abstract

Wave Function Collapse (WFC) is a greedy search algorithm. In this paper, we apply the WFC framework to the minimum vertex cover problem (MinVC) and propose a new algorithm Wave Function Collapse Vertex Cover (WFCVC). Our experiments show that WFCVC performs well on both DIMACS benchmark and real-world massive graphs. Keeping the worst approximation ratio within 1.02, WFCVC is so efficient that it successfully outputs results for graphs with 500,000 vertices in 50 seconds. On hard instances of real-world massive graphs the median speed of WFCVC is 30 times faster than that of FastVC, one of the state-of-the-art heuristics for MinVC in large-scale graphs. As a simple, fast and robust algorithm, WFCVC achieves a desirable balance between efficiency and accuracy.

1 Introduction

The Minimum Vertex Cover (MinVC) problem is an NP-complete problem. Given an undirected graph, a vertex cover (VC) of a graph is a subset of vertices in which every edge in the graph has at least one endpoint. The minimum vertex cover problem is to find the minimum-sized vertex cover. Due to its wide range of applications, a lot of researchers have done researches on minVC problem.

MinVC problem is a NP-complete problem. Exact algorithms contained in [10] [19] [27] for the MinVC problem have high time complexity. For this reason, people started to search for approximation algorithms, which can output acceptable results within reasonable times at the cost of sacrificing optimization [12]. The earliest approximation algorithms with polynomial complexity [4] [13] were developed in the 1980s. In recent years, more work has been done. DLS [23] is a stochastic phased local search, whose vertex selection techniques are dynamically adjusted during the search. COVER [24] is a k-vertex solution which maintains k vertices and iteratively exchanges two vertices. EWLS [5] updates the edge weighting when faced with a local optimum. EWCC [6] is another local search with edge weighting and configuration checking heuristics for minimum vertex cover. NuMVC [7] is based on two-stage exchange and edge weighting with forgetting.

Some algorithms, based on structures of graphs, are simple and fast. NOVCA [11] is motivated by the simple fact that vertex cover candidates are those that are adjacent to minimum degree vertices. VSA designs with the concept called "support of vertices", which equals the sum of the degrees of adjacent vertices [2]. MVSA [14] and AVSA [17] are improvements of this algorithm. In short, many approximation algorithms recently invented can output relatively accurate results while keeping much lower time complexity than exact algorithms.

However, these heuristics are mainly tested on the benchmark with small-scale data. According to results published in previous essays, we find that most algorithms cannot finish within a reasonable time on instances with over 2000 vertices. FastVC [8] [9] is a new algorithm specially designed to solve MinVC on massive sparse graphs. FastVC, having been tested on a broad range of real-world massive graphs, is by far the most competitive algorithm. In this paper, we proposes a new WFC-based heuristic for minimum vertex cover problem: the WFCVC algorithm. Our experiments on both DIMACS benchmark and massive graphs demonstrate that our algorithm

shows no worse performance than FastVC.

2 Wave Function Collapse (WFC)

2.1 Wave Function Collapse (WFC) Overview

Wave function collapse algorithm was developed by Gumin Maxim [20]. The concept of “wave function collapse” derives from quantum mechanics. An object with many potential possibilities has high **entropy**. Upon observed, the object would have fewer possibilities. This process is called "collapse". When an object has a definite state, its entropy equals zero.

Wave Function Collapse algorithm is initially designed to generate bitmaps that are locally similar to input bitmaps. Local similarity includes 2 standards:

- **Standard 1**: Each N*N pattern of pixels in the output should occur at least once in the input.
- **Standard 2**: The distribution of the output bitmap is similar to that of the input bitmap. To specify, when a N*N pattern is randomly chosen from the output bitmap, the probability of meeting a particular pattern approximately is approximate to the density of that pattern in the input bitmap.

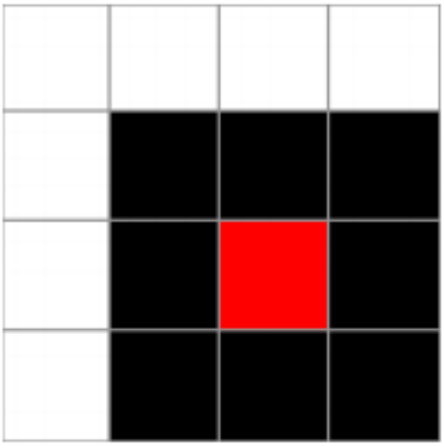

Figure 1: Input Bitmap

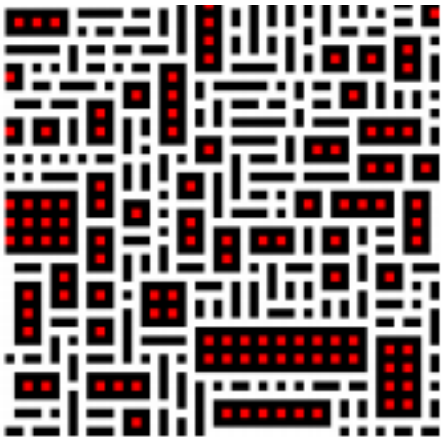

Figure 2: Output Bitmap Generated

2.2 Wave Function Collapse (WFC) Algorithm

We should know the size of the output bitmap. Every N*N region has its own **“wave function”**. Wave function is a data structure that stores information about which patterns can be chosen for a N*N small region. We use a window to extract all N*N patterns from the input bitmap. To satisfy standard 1, all N*N regions in the output bitmap can only be chosen from patterns extracted from the input. Therefore, at first each small region’s wave function stores the information that all extracted patterns are valid. It is worth noting that now all N*N small regions have high entropy, because they have many potential states [18]. After completing steps above, we start to allocate patterns to locations. Do following steps repeatedly until every N*N region in the output bitmap has obtained a definite state, i.e. The entropy of every region equals zero.

1. **Observation**. In this step, we choose a N*N region with the lowest non-zero lowest entropy, which means this region has the fewest valid patterns to choose from. We denote this region as $Region_{observed}$. Then, we **collapse** the state of $Region_{observed}$ into a definite state. To specify, we assign a pattern for $Region_{observed}$ and modify information in wave functions. To

meet standard 2, we choose a pattern to be assigned according to the distribution of N*N patterns in the input.

2. **Propagation**. Since the state of $Region_{observed}$ has changed, states of nearby regions will be influenced, which is called "**propagation**". Based on pattern assigned to $Region_{observed}$, we update neighbor $N * N$ regions' wave function and its entropy. If the entropy of a region turns to zero, we put information about the region in a data structure (for example, a queue) which will be further used for updating. We loop steps above until the data structure is empty.

Eventually, according to pattern we choose for each N*N region in the output bitmap, we generate a new bitmap, which is locally similar to the input bitmap.

Algorithm 1: Framework of WFC

Input a bitmap;
Output the bitmap generated ;
Initialize every N*N small region's wave function ;
extract patterns from the input bitmap ;
while *Regions with non-zero entropy* $\neq \varnothing$ **do**
  $Observe()$;
  $Propagate()$;
end

3 WFCVC: A New WFC-based Algorithm for Vertex Cover

Description of minVC problem :Let $G = (V, E)$ be a simple, undirected, graph with a vertex set V and an edge set $E \subseteq V \times V$. Given a Graph G, we find a vertex cover Vc, a set of vertices so that $u \in Vc$ or $v \in Vc$ for each edge $(u, v) \in E$. If at least one endpoint of an edge is chosen, we can say that the edge is covered; otherwise, the edge is uncovered. The vertices in Vc can cover all edges in a graph. Minimal vertex cover problem is to find a vertex cover set Vc to ensure that the size of the vertex cover is as small as possible.

3.1 Definitions and Initialization

3.1.1 Construction of Patterns

In WFC algorithm, each N*N small region is considered as a unit and consideration of which patterns are valid is required. In WFCVC algorithm, we take each edge as a unit, and construct patterns for edges. Each edge $e = (u, v)$ has 3 potentially valid patterns: $[u], [v], [u, v]$. $[u]$ means that u is in the vertex cover while v is not in the vertex cover. $[u, v]$ means that u and v are both in the vertex cover. It is worth noting that it is impossible that neither u nor v is in the vertex cover. Therefore, we can guarantee that the vertex cover obtained eventually must be legal as long as the algorithm finds a valid pattern for each edge.

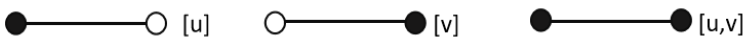

Figure 3: the Patterns for Edge

3.1.2 Construction of Wave Functions

Like the WFC algorithm, we construct wave functions for edges. For an edge $e = (u, v)$, $wave(e)$ is defined as a set of patterns still available. Entropy is the number of patterns still available, which equals to the size of $wave(e)$. At the beginning, e has three patterns in its $wave$ function. The entropy of e equals 3.

$$wave(e) = \{[u], [v], [u, v]\}$$

In the WFC algorithm, after looping observation and propagation, the states of all regions are definite. Similarly, in our algorithm, for each edge e=(u, v), it must have only one pattern available in its wave function. For instance, $wave(e)$ may be $\{[u]\}$ eventually. The entropy of e equals 1.

3.1.3 Construction of $\widehat{Deg}(v)$

The degree of a vertex equals the number of edges incident to it, noted as $deg(v)$. $\widehat{Deg}(v)$ is defined as the number of uncovered edges associated with v. At the beginning, since all edges are uncovered, for any vertex v, $\widehat{Deg}(v)$ equals to $deg(v)$, the degree of v.
Algorithm 2 will complete the initial construction.

Algorithm 2: Initialization

Input: $G = (V, E)$;
for *each* $e = (u, v) \in E$ **do**
  $wave(e) \leftarrow \{[u], [v], [u, v]\}$;
end
for *each* $v \in V$ **do**
  $\widehat{Deg}(v) \leftarrow deg(v)$;
end

As shown in figure 4, after initialization: $\widehat{Deg}[1..7] = \{2, 4, 3, 3, 3, 2, 1\}$ $wave(i) = \{[5], [7], [5, 7]\}$. Similar to edge i, other edges have 3 elements in $wave$.

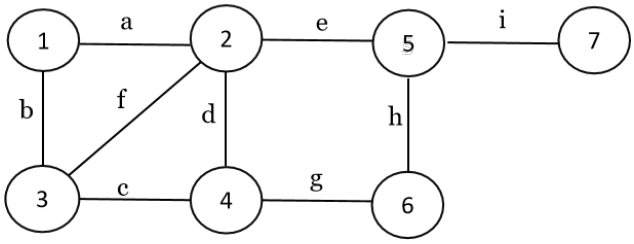

Figure 4: Example of Initialization

3.2 WFCVC Framework

We imitate the WFC algorithm. Loops of observation and propagation are performed in the WFCVC algorithm. Through observation, we determine states of one or more edges, which in turn determines chosen states of vertices. Through propagation, we use the vertices whose chosen states are newly determined to update nearby edges' wave functions. If an edge does not have a definite state (the entropy of that edge is larger than 1), it is in the set of $E_{uncollapsed}$. When every edge has a definite state (the set of $E_{uncollapsed}$ is empty), the loop stops and the result is outputted.

1. **Observation**. Collect information of G and push the vertices of the state-determined edge into the queue Q according to the selection strategy. For WFCVC, the selection strategy is derived from the NOVCA algorithm [11], which preferentially selects the neighbors of the vertex with the smallest degree.
2. **Propagation**. Obtain a vertex from queue Q and propagate states according to the adjacencies of graph G. When the queue Q is empty, it means that the current round of propagation is over.

Algorithm 3: Framework of WFCVC

Input: a graph $G = (V, E)$;
Output: a vertex cover set Vc of G;
$E_{uncollapsed} \leftarrow E$;
Create an empty queue Q ;
Initialization( ) ;
while $E_{uncollapsed} \neq \varnothing$ **do**
  Observe(G, Q);
  Propagate(G, Q);
  if *no edge is uncovered* **then**
    break;
  end
end

3.3 Observe

If we decide not to put a vertex v in the vertex cover, all its neighbors must be in the vertex cover. Thus, the cost of excluding v in the vertex cover is the number of all its neighbors with indefinite states (entropy greater than 1), which equals the number of uncovered edges that are incident to v, $\widehat{Deg}(v)$.

When observing, we find the vertex with the minimum $\widehat{Deg}$, noted as v_{minDeg}. In order to make the cost as low as possible, we decide not to put v_{minDeg} into the vertex cover. Since for each edge $e = (v_{minDeg}, u)$, patterns containing v_{minDeg} are illegal, we collapse $wave(e)$ into $\{[u]\}$. Then, we push v_{minDeg} and u to the queue Q, which stores vertices which have definite states (we know exactly whether the vertices are in the vertex cover) but have not been used to propagate.

Algorithm 4: Observe(G, Q)

$v_{minDeg} \leftarrow$ the vertex with the minimum non-zero $\widehat{Deg}$
for *each* $e = (u, v_{minDeg}) \in E$ **do**
  add u to Vc ;
  remove patterns that contain v_{minDeg} from $wave(e)$;
  $push(Q, u)$;
  $push(Q, v_{minDeg})$;
end

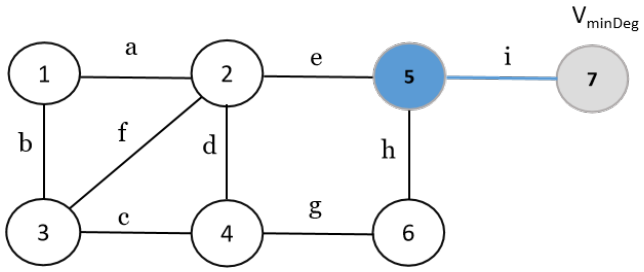

Figure 5: Example of Observe

As shown in the graph 5, blue vertices are in Vc; grey vertices are not in Vc; states of white vertices are not determined. Blue edges are covered. Now perform function Observe(G, Q). Then find $v_{minDeg} = 7$, and collapse edge i. $Vc = \{5\}$ $Q = \{5, 7\}$ $wave(i) = \{[5]\}$.

3.4 Propagate

Propagating

The function $propagate()$ determines the impact of deciding whether to put a vertex into the vertex cover set Vc on the edges that are incident to that vertex [21]. When the state of an edge becomes definite (there is only one pattern available in its wave function), we add its endpoints to Q. The propagation lasts until we no longer have edges whose wave functions have only one pattern.
To avoid unnecessary repeated propagation, we make an optimization that the vertices which were once in queue Q are no longer pushed into the queue Q.

Algorithm 5: Propagate(G, Q)

while *Q is not empty* **do**
  $v \leftarrow pop(Q)$;
  for *each* $e = (u, v) \in E_{uncollapsed}$ **do**
    remove invalid patterns from wave(e) ;
    if $|wave(e)| = 1$ **then**
      remove e from $E_{uncollapsed}$;
      $p \leftarrow$ the only pattern in $wave(e)$;
      add each vertex contained in p to vertex cover ;
      $push(Q, u)$;
    end
  end
  Renew $\widehat{Deg}(v)$;
end

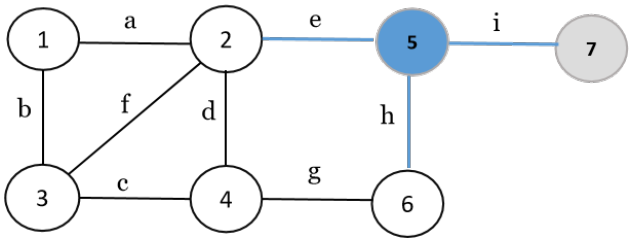

Figure 6: Example of propagate

At this point, the first element acquired from queue Q is 5. In following steps, renewing $\widehat{Deg}(v)$ may cause further propagation. $Vc = \{5\}$ $Q = [7]$ $wave(e) = \{[5], [2, 5]\}$ $wave(h) = \{[5], [5, 6]\}$.

Renew $\widehat{Deg}$

For each vertex v in Q, if v is in the vertex cover Vc, the edges that are incident to v will be covered, resulting in the changes of $\widehat{Deg}$. We update $\widehat{Deg}(v)$ by subtracting the number of edges incident to v that are newly covered. Updated $\widehat{Deg}$ will be used in the next observation.
After renewing $\widehat{Deg}$, some special vertices whose states can be determined directly will go into the fast collapse path. For example, the vertex v with $\widehat{Deg}(v) = 1$ is definitely not to be chosen in

Vc.

Algorithm 6: Renew $\widehat{Deg}(v)$

if *v is in Vc* **then**
  $\widehat{Deg}(v) \leftarrow 0$;
  for *each u that is the neighbour of v* **do**
    if $\widehat{Deg}(u) > 0$ **then**
      $\widehat{Deg}(u) \leftarrow \widehat{Deg}(u) - 1$;
      Fast Collapse(u);
    end
  end
end

Algorithm 7: Fast Collapse(v)

if $\widehat{Deg}(v) = 1$ **then**
  $u \leftarrow$ the only neighbor of v ;
  $e \leftarrow (u, v)$;
  remove patterns that contain v from $wave(e)$;
  add u to Vc ;
  $push(Q, u)$;
  $push(Q, v)$;
end

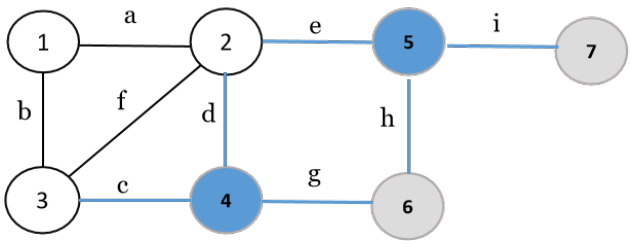

Figure 7: Example of Fast Collapse

After renewing $\widehat{Deg}(v)$, $\widehat{Deg}(v)[1..7] = \{2, 3, 3, 3, 0, 1, 0\}$. We find out that $\widehat{Deg}(6) = 1$. So perform $FastCollapse(6)$.The only neighbor of vertex 6 is vertex 4, so vertex 4 must be in Vc. $Q = [7, 4, 6]$ $Vc = \{5, 4\}$ $wave(h) = \{[5]\}$ $wave(g) = \{[4]\}$ $wave(c) = \{[4], [3, 4]\}$ $wave(d) = \{[4], [2, 4]\}$. So far, edges i, e, h, g, c, d are covered. $\widehat{Deg}(v)[1..7] = \{2, 2, 2, 0, 0, 0, 0\}$. After propagating the vertex 5, the next vertex from the queue is 7. Since vertex 7 no longer has neighbors, it is directly popped. Propagation goes on at the next vertex in the queue Q.

4 Experiments of WFCVC Algorithm

DIMACS is a standard benchmark for MinVC related research, which is established in the second DIMACS Challenge [26]. Therefore, we select DIMACS to test WFCVC algorithm. The DIMACS benchmark is originally used to test maximum clique [16]. For each set of data, we only know the accurate value of the size of maximum clique. Since the size of the maximum clique of an graph equals the size of minimum vertex cover of its complementary graph, we run the WFCVC algorithm on the complementary graph and compare results outputted with the given value of the size of maximum clique of the original graph. In addition, we test our algorithm's performance on massive graphs acquired from real world including social network like Facebook, technological

networks, and infrastructure network [22]. All data used for testing are downloaded from the Network Data Repository Website.

Experiment results are analysed by two indicators: running time and approximation ratio. For the NP-hardness approximation algorithm, approximation ratio is defined as the ratio of the value of the solution generated to the value of the optimal solution (sometimes this value equals the correct value) [1].

The programming language is C++. We use the Dev C++ companion MinGW64 compiler. Our experiments run on a PC machine with a 3.00 GHz CPU and 16 GB RAM.

4.1 Performance on DIMACS Benchmark

Table 1 shows the performance of WFCVC on DIMACS. 41 instances have been taken into account. All Instances, except C4000-5 (with 4000 vertices), successfully finish in less than 1 second. 8 instances attain optimal values. The average approximation ratio is 1.006 and the average running time is only 0.222 second. On about half instances, running time is less than 0.05 second.

WFCVC is dedicated to reducing the running time. Leading MinVC heuristics such as COVER and NuMVC aim to obtain more optimal values by running many turns, while WFCVC algorithm, using a deterministic vertex selection strategy, only runs once to get the result. Since we want to know how much WFCVC loses in accuracy while shortening the running time, we pick some extremely hard instances listed in NuMAC paper [7] from DIMACS Benchmark for comparison. Experimental results of NuMVC and COVER are extracted from the original literature [24] [7]. For comparison, we use the “Median Runtime”, which is median running time of all turns, to compare the differences in running time between algorithms.

Table 1: WFCVC Performance on DIMACS benchmark

Graph			WFCVC		
Instance	Vertices	|C*|	|Vc|	Time(sec)	|Vc|/|C*|
brock200_1	200	179	181	0.005	1.011
brock200_2	200	188	191	0.011	1.016
brock200_4	200	183	185	0.007	1.011
brock400_2	400	371	379	0.013	1.022
brock400_4	400	367	377	0.020	1.027
brock800_2	800	776	782	0.101	1.008
brock800_4	800	774	782	0.100	1.010
C250-9	250	206	208	0.001	1.010
C500.9	500	443	448	0.014	1.011
C1000-9	299	932	938	0.044	1.006
C2000.5	2000	1984	1987	0.884	1.002
C2000-9	2000	1920	1933	0.180	1.007
C4000-5	4000	3982	3986	3.613	1.001
c-fat200-5	200	142	142	0.011	1.000
c-fat500-10	500	374	374	0.068	1.000
DSJC5005	500	487	489	0.053	1.004
DSJC10005	1000	985	987	0.217	1.002
gen200_p0.9_44	200	156	156	0.092	1.000
gen400_p0.9_55	400	345	352	0.010	1.020
johnson16-2-4	120	112	112	0.001	1.000
johnson32-2-4	496	480	480	0.030	1.000
hamming8-4	256	240	240	0.010	1.000
hamming10-2	1024	512	512	0.010	1.000
hamming10-4	1024	984	988	0.071	1.004
keller4	171	160	160	0.010	1.000
keller5	776	749	753	0.061	1.005
keller6	3361	3302	3313	0.908	1.003
MANN_a27	378	252	253	0.001	1.004
MANN_a45	1035	690	693	0.005	1.004
MANN_a81	3321	2221	2225	0.449	1.002
p_hat300-3	300	264	267	0.015	1.011
p_hat500-1	500	491	492	0.081	1.002
p_hat700-1	700	689	692	0.150	1.004
p_hat700-2	700	656	658	0.105	1.003
p_hat700-3	700	638	641	0.056	1.005
p_hat1500-1	1500	1488	1490	0.729	1.001
p_hat1500-2	1500	1435	1438	0.487	1.002
p_hat1500-3	1500	1406	1415	0.235	1.006
sanr200_0.7	200	183	185	0.008	1.011
sanr400_0.7	400	379	382	0.022	1.008
san1000	1000	985	990	0.212	1.005

As shown in Table 2, the performance of WFCVC does not decline on hard instances. The approximation ratio remains within 1.01, except for two instances of brock400. However, the time optimization is quite amazing. WFCVC is the fastest of all existing algorithms on all hard instances: on half instances, it finishes successfully in less than 0.1 second. The worst running time is shown on instance C4000.5. Although the approximation value for C4000.5 is only 4 more than the optimal value, the running time is 249.194 seconds less than that of NuMVC and 617.767 seconds less than that of COVER. The most notable advantage in speed is shown in brock400_2: running time of our algorithm is 44,030 times less than that of NuMVC. The most significant reduction in seconds is achieved on C2000.9, on which WFCVC's running time is 1994.381 seconds less than that of NuMVC.

Table 2: Comparison WFCVC with other Algorithms on DIMACS Hard Instances

Graph			WFCVC		NuMVC	COVER
Instance	|C*|	|Vc|	|Vc|/|C*|	Time(sec)	Time(suc time)	Median Runtime
brock400_2	371	379	1.022	0.013	572.390(512.906)	0.05
brock400_4	367	377	1.027	0.02	4.981	137.68
MANN_a45	690	693	1.004	0.005	86.362	0.28
brock800_2	776	782	1.008	0.101	n/a	0.98
brock800_4	774	782	1.010	0.1	n/a	1.35
p_hat1500-1	1488	1490	1.001	0.729	3.751	18.25
C2000.9	1920	1933	1.007	0.18	1994.561(1393.303)	323.11
MANN_a81	2221	2225	1.002	0.4492	1657.880(732.897)	30.89
keller6	3302	3313	1.003	0.908	2.51	15.18
C4000.5	3982	3986	1.001	3.613	252.807	621.38

To keep the continuity of the lines on the figure, the value marked as “n/a” has been changed to cut-off running time, which is set to be 1000 seconds on DIMACS instanhttps://www.overleaf.com/project/6131e8b48

The approximation ratio is proved to be at most $\theta(\frac{1}{\sqrt{logn}})$ [3] [15]. Here, the upper limit of approximation ratio is set to be 1.5 in the figure.

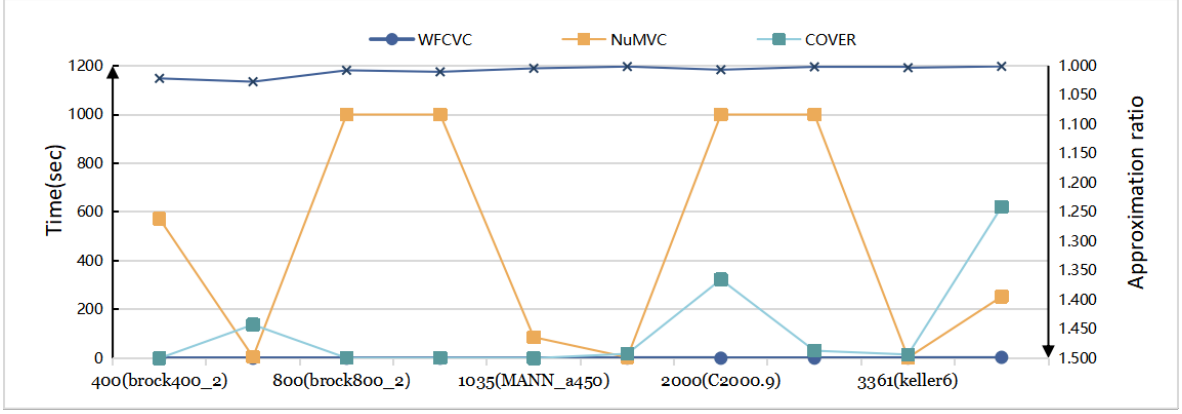

Figure 8: Comparison WFCVC with other algorithms on DIMACS Hard Instances

The experimental results demonstrate that WFCVC has a considerable advantage in speed. Although we sacrifice accuracy a little, the running time is greatly shortened. Moreover, the performance of WFCVC algorithm is stable.

4.2 Performance In Massive Graphs

To test the performance of WFCVC on massive graphs, we use the graphs on which FastVC [9] is tested. We select hard instances on which the running time of FastVC is over 8 seconds to test the performance of WFCVC on large-scale data. Due to the lack of accurate optimal values, values given by the FastVC are used as C* when calculating approximation ratios.

Table 3: Comparison WFCVC with FastVC on Massive Graphs

Graph			WFCVC			FastVC	
Instance	Vertices	Edges	|Vc|/|C*|	|Vc|	Time(sec)	|Vc|(min/avg)	Time(sec)
ia-infect-dublin	410	2765	1.020	299	0.007	293	480.11
socfb-MIT	6402	251230	1.014	4721	1.156	4657(4657)	41.13
socfb-Duke14	9885	506437	1.012	7778	2.754	7683(7683)	228.3
socfb-Stanford3	11586	568309	1.011	8607	3.484	8517(8517.9)	101
socfb-UCSB37	14917	482215	1.015	11428	0.673	11261(11263)	210.62
socfb-UConn	17206	604867	1.013	13406	2.695	13230(13231.5)	304.11
socfb-UCLA	20453	747604	1.014	15430	4.067	15223(15224.3)	297.92
socfb-Berkeley13	22900	852419	1.013	17426	1.322	17210(17212.7)	290.4
socfb-Wisconsin87	23831	835946	1.014	18646	1.303	18383(18385.1)	295.36
socfb-Indiana	29732	1305757	1.014	23645	2.041	23315(23317.1)	517.27
socfb-UIllinois	30795	1264421	1.014	24427	2.007	24091(24092.2)	477.69
socfb-UF	35111	1465654	1.014	27679	2.469	27306(27309)	459.23
socfb-Texas84	36364	1590651	1.013	28524	2.666	28167(28171.1)	495.91
socfb-Penn94	41536	1362220	1.013	31595	2.891	31162(31164.8)	552.92
sc-nasasrb	54870	1311227	1.003	51420	2.823	51244(51247.3)	517.13
socfb-OR	63392	816886	1.010	36905	4.651	36548(36549.2)	144.06
sc-pkustk13	94893	3260967	1.003	89446	4.384	89217(89220.6)	315.23
soc-buzznet	101163	2763066	1.002	30661	3.251	30625(30625)	15.95
soc-LiveMocha	104103	2193083	1.000	43432	2.644	43427(43427)	17.44
sc-shipsec1	140385	1707759	1.008	118312	8.362	117318(117337.5)	722.4
web-arabic-2005	163598	1747269	1.000	114462	6.023	114426(114427.2)	323.58
sc-shipsec5	179104	2200076	1.006	148092	14.47	147137(147173.8)	569.7
tech-RL-caida	190914	607610	1.005	75140	17.636	74930(74938.9)	9.62
soc-gowalla	196591	950327	1.002	84378	18.608	84222(84222.3)	64.58
sc-pwtk	217891	5653221	1.001	207846	14.626	207716(207719.9)	184.89
sc-msdoor	415863	9378650	1.001	381772	29.628	381558(381558.9)	18.71
web-it-2004	509338	7178413	1.001	415081	43.656	414671	9.91
soc-flickr	513969	3190452	1.001	153386	63.877	153272(153272)	18.51
ca-coauthors-dblp	540486	15245729	1.000	472229	106.451	472179(472179)	14.57
soc-FourSquare	639014	3214986	1.000	90112	6.691	90108(90109.2)	124.9
sc-ldoor	952203	20770807	1.000	856928	116.743	856755(856757.4)	218.94
ca-hollywood-2009	1069126	56306653	1.000	864109	592.224	864052(864052)	25.59
inf-roadNet-PA	1087562	1541514	1.011	561346	641.362	555220(555243)	652.14
soc-youtube-snap	1134890	2987624	1.000	276948	151.684	276945(276945)	12.74
soc-pokec	1632803	22301964	1.007	848993	1446.501	843422(843434.9)	772.77
tech-as-skitter	1696415	11095298	1.001	527888	535.484	527185(527195.9)	446.49
web-wikipedia2009	1864433	4507315	1.001	649081	1175.232	648317(648321.8)	692.3
inf-roadNet-CA	1957027	2760388	1.011	1012639	1964.978	1001273	896.33
socfb-B-anon	2937612	20959854	1.000	303048	46.344	303048(303048.9)	85.1
soc-orkut	2997166	106349209	n/a	n/a	n/a	2171296(2171380.5)	996.66
socfb-A-anon	3097165	23667394	1.000	375233	53.906	375231(375232.8)	128.55
soc-livejournal	4033137	27933062	n/a	n/a	n/a	1869046(1869051.1)	953.11
inf-road-usa	23947347	28854312	n/a	n/a	n/a	12049567(12050440.1)	1000
socfb-uci-uni	58790782	92208195	n/a	n/a	n/a	866768(866768)	37.05

As shown in the Table 3, 45 instances, sorted by the number of vertices, are tested. 4 instances fail and are recorded as "n/a". The reason for the failure of soc-orkut and socfb-uci-uni is that the number of nearly 100,000,000 edges is too large and the memory limit is exceeded. The other two instances fail because the running time exceeds 2000 seconds, cut-off running time set for massive graphs. For example, soc-livejournalf runs for 7758.527 seconds, so we reckon that on this instance our algorithm fail.

On 30 out of 45 instances, proportionally two-thirds, WFCVC runs faster than FastVc.The median running time of WFCVC is 9.672 seconds, which is 30 times less than that of FastVC, which is 290.4 seconds.
The approximation ratios are also desirable, ranging from 1 to 1.020. On DIMACS benchmark, the value of average approximation ratio is 1.006. Regarding the deviation in the number of vertices, median(|Vc|-|C*|)=176, corresponding to instance is sc-nasasrb with 54,870 vertices and 1,311,227 edges. On sc-nasasrb instance the running time of WFCVC is 2.823 seconds, which is 515 seconds less than that of FastVC. On socfb-B-anon instance WFCVC obtains the optimal solution in 85.1 seconds, while FastVC obtains it in 722.4 seconds.

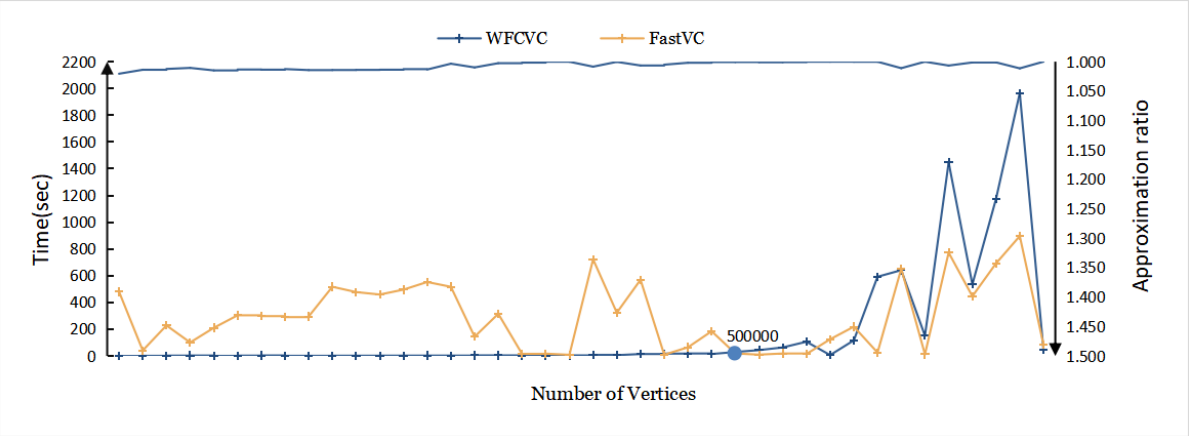

Figure 9: Comparison WFCVC with FastVC on Massive Grapshs

Figure 9 shows the performance of WFCVC algorithm is desirable on massive graphs. Approximation ratio is always in a smooth state, and the running time is excellent when the number of vertices is within 500,000.

4.3 Comparison with NOVCA Algorithm

For the WFCVC algorithm the selection strategy is derived from one core idea of the NOVCA algorithm. Based on this strategy, WFC algorithm is introduced to increase the search efficiency and improve the running speed. Therefore it is necessary to compare WFCVC with NOVCA.
The experiments are done on the benchmark of DIMACS and BHOSLIB [30] benchmarks except for the last graphs generated by the private custom scripts, on which NOVCA are tested on.

Table 4: Comparison with NOVCA based on statistics

	Time(sec)			Approximation Ratio		
	avg	median	worst	avg	median	worst
WFCVC	0.456	0.068	6.56	1.006	1.004	1.022
NOVCA	150.624	2.683	2013.667	1.008	1.004	1.045

From table 4 we can see that overall, the performance of the WFCVC is better than that of NOVCA. WFCVC's average running time is 330 times less than NOVCA's.
The detailed experiment results are shown in the table 5. All instances run within 1 second except for frb100-40, which is considered as "a problem that will not be solved on a PC for the next 20 years or more (from 2005)" [28]. On frb100-40 MVCFC it takes only 6.56 seconds while on NOVCA it takes 2013.667 seconds.
The instance number of optimal solutions of WFCVC is 2 more than that of NOVCA. For WFCVC

the worst approximation ratio is 1.022, with no reduction in accuracy despite the significant increase in speed.

Table 5: Comparison WFCVC with NOVCA

Graph		WFCVC			NOVCA		
Instance	|C*|	|Vc|	Time(sec)	|Vc|/|C*|	|Vc|	Time(sec)	|Vc|/|C*|
frb59-26-1	1475	1508	0.912	1.022	1485	80.258	1.007
frb59-26-2	1475	1508	0.906	1.022	1484	79.297	1.006
frb100-40	3900	3960	6.560	1.015	3917	2013.667	1.004
broc200_1	179	181	0.005	1.011	181	0.115	1.011
broc800_4	774	782	0.100	1.010	782	10.832	1.01
C2000.9	1922	1933	0.180	1.006	1932	207.06	1.005
c-fat200-5	142	142	0.011	1	142	0.092	1
c-fat500-10	374	374	0.068	1	374	2.117	1
gen200_p0.9_44	156	156	0.092	1	163	0.092	1.045
hamming10-2	512	512	0.010	1	512	10.297	1
hamming10-4	984	988	0.071	1.004	988	21.505	1.004
johnson16-2-4	112	112	0.010	1	112	0.076	1
johnson32-2-4	480	480	0.030	1	480	2.273	1
keller4	160	160	0.010	1	164	0.007	1.025
keller5	749	753	0.061	1.005	761	9.125	1.016
MANN_a27	252	253	0.001	1.004	253	0.493	1.004
MANN_a81	2221	2225	0.449	1.002	2225	773.963	1.002
p_hat500-1	491	492	0.081	1.002	492	2.683	1.002
p_hat1500-3	1406	1415	0.235	1.006	1414	74.991	1.006
san1000	985	990	0.212	1.005	991	22.901	1.006
sanr200_0.7	183	185	0.008	1.011	185	0.857	1.011
sanr400_0.7	379	382	0.022	1.008	382	1.03	1.008

In the NOVCA algorithm, if there are more than one vertices with the same minimum degree, the vertex with the maximum support value is selected [11]. The optimization is also verified by WFCVC. The experiment results show that WFCVC is not sensitive to the optimization. However, when the number of vertices is large, calculating the vertex support may significantly decrease the speed of the algorithm, while it does not significantly improve accuracy. Therefore, vertex support is not included in the core part of the WFCVC algorithm.

Figure 10 shows that the deviation of approximation ratio is very small between WFCVC and NOVCA. However, as the number of vertices exceeds 500, NOVCA's running time increases sharply, while WFCVC's running time remains stable.

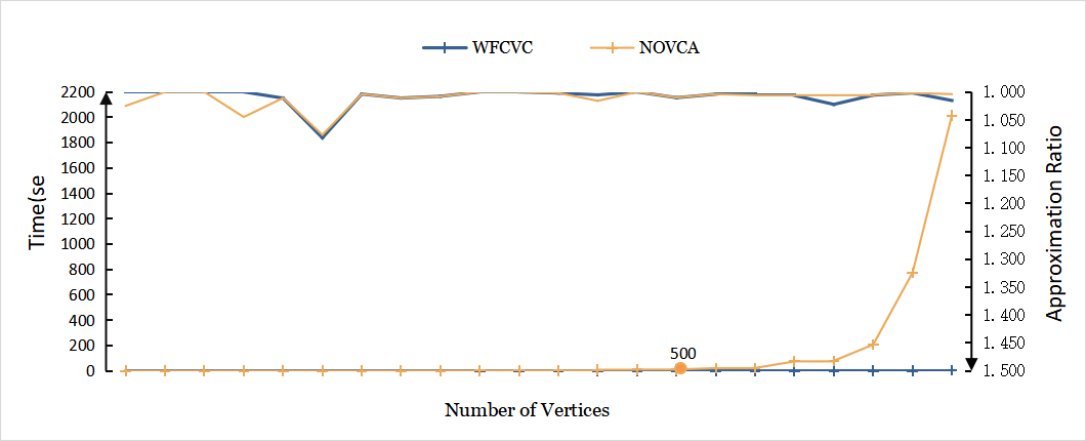

Figure 10: Comparison of WFCVC with NOVCA on Benchmark

It can be concluded that, with the introduction of the WFC framework, WFCVC can run significantly faster than NOVCA while maintaining a similar approximation ratio. The larger the number of vertices is, the more significant the performance in improvement will be.

4.4 Performance Trend Analysis

- 3,000 vertices: the program finishes in 1 second.
- 100,000 vertices: the program finishes in 5 seconds.
- 200,000 vertices: the program finishes in 20 seconds.
- 500,000 vertices: the program finished in 50 seconds.
- 1,000,000 vertices: the program finishes in 120 seconds.
- 500,000 vertices is a critical point beyond which the performance of WFCVC is inferior to that of FastVc.
- 1,000,000 vertices is a boundary point beyond which the performance of WFCVC deteriorates quickly.
- 3,000,000 vertices is a limit beyond which running may not be successful (mainly due to memory limitations).

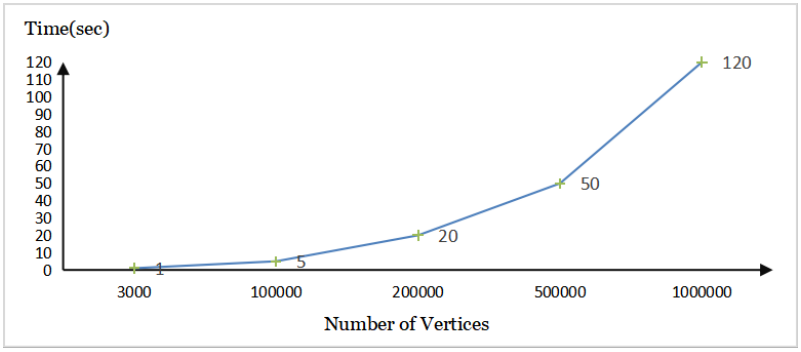

Figure 11: WFCVC Running Time Trend

5 Conclusion

In this paper, the WFC algorithm framework is introduced to the MinVC heuristic algorithm, which greatly improves the local search speed. Our experiments demonstrate that WFCVC algorithm is a near-optimal solution. Its performance is competitive not only on the small-scale

benchmark, but also on massive graphs. The running time remains stable as the number of vertices increases. Therefore, the performance of the WFCVFC algorithm is quite fast and robust. The current WFC framework considers one edge as a unit, on which WFCVC algorithm performs observation and propagation. Three vertices can be considered as a unit subsequently, which can further reduce the state space and increase the propagation speed. We may consider introducing other heuristic approaches. The optimal solution may be obtained by exchanging vertices under a certain strategy iteratively. It is also possible to extend WFC local constraints by local backtracking and weight calculation [25].

References

[1] Arora, S., Karger, D.,& Karpinski, M. (1999). Polynomial time approximation schemes for dense instances of NP-hard problems. Journal of computer and system sciences, 58(1), 193-210.

[2] Balaji, S., Swaminathan, V., & Kannan, K. (2010). Optimization of unweighted minimum vertex cover. World Academy of Science, Engineering and Technology, 43, 716-729.

[3] Bafna, V., Berman, P., &Fujito, T. (1999). A 2-approximation algorithm for the undirected feedback vertex set problem. SIAM Journal on Discrete Mathematics, 12(3), 289-297.

[4] Bar-Yehuda, R., & Even, S. (1981). A linear-time approximation algorithm for the weighted vertex cover problem. Journal of Algorithms, 2(2), 198-203.

[5] Cai, S., Su, K., & Chen, Q. (2010, July). EWLS: A new local search for minimum vertex cover. In Twenty-Fourth AAAI Conference on Artificial Intelligence.

[6] Cai, S., Su, K., & Sattar, A. (2011). Local search with edge weighting and configuration checking heuristics for minimum vertex cover. Artificial Intelligence, 175(9-10), 1672-1696.

[7] Cai, S., Su, K., Luo, C., & Sattar, A. (2013). NuMVC: An efficient local search algorithm for minimum vertex cover. Journal of Artificial Intelligence Research, 46, 687-716.

[8] Cai, S. (2015, June). Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In Twenty-Fourth International Joint Conference on Artificial Intelligence.

[9] Cai, S., Lin, J., & Luo, C. (2017). Finding a small vertex cover in massive sparse graphs: Construct, local search, and preprocess. Journal of Artificial Intelligence Research, 59, 463-494.

[10] Carraghan, R., & Pardalos, P. M. (1990). An exact algorithm for the maximum clique problem. Operations Research Letters, 9(6), 375-382.

[11] Gajurel, S., & Bielefeld, R. (2012). A Simple NOVCA: Near Optimal Vertex Cover Algorithm. Procedia Computer Science, 9, 747-753.

[12] Hochba, D. S. (Ed.). (1997). Approximation algorithms for NP-hard problems. ACM Sigact News, 28(2), 40-52.

[13] Hochbaum, D. S. (1982). Approximation algorithms for the set covering and vertex cover problems. SIAM Journal on computing, 11(3), 555-556.

[14] Imran, K., & Hasham, K. (2013). Modified Vertex Support Algorithm: A New approach for approximation of Minimum vertex cover. Research Journal of Computer and Information Technology Sciences ISSN, 2320, 6527.

[15] Karakostas, G. 2009. A better approximation ratio for the vertex cover problem. ACM Trans. Algor. 5, 4, Article 41 (October 2009), 8 pages.

[16] Johnson, D. S., Trick, M. A. (Eds.). (1996). Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993 (Vol. 26). American Mathematical Soc..

[17] Ahmad, I., & Khan, M. (2014). AVSA, modified vertex support algorithm for approximation of MVC. International Journal of Advanced Science and Technology, 67, 71-78.

[18] Karth, I., & Smith, A. M. (2017, August). WaveFunctionCollapse is constraint solving in the wild. In Proceedings of the 12th International Conference on the Foundations of Digital Games (pp. 1-10).

[19] Li, C. M., & Quan, Z. (2010, July). An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In Twenty-fourth AAAI conference on artificial intelligence.

[20] Maxim Gumin.(2016). https://github.com/mxgmn/WaveFunctionCollapse. GitHub repository.

[21] Mac, A., & Perkins, D. (2021). Wave Function Collapse Coloring: A New Heuristic for Fast Vertex Coloring. arXiv preprint arXiv:2108.09329.

[22] Network Graph Data. https://networkrepository.com. Network Repository.

[23] Pullan, W. (2006). Phased local search for the maximum clique problem. Journal of Combinatorial Optimization, 12(3), 303-323.

[24] Richter, S., Helmert, M., & Gretton, C. (2007). A stochastic local search approach to vertex cover. In Annual Conference on Artificial Intelligence (pp. 412-426). Springer, Berlin, Heidelberg.

[25] Sandhu, A., Chen, Z., & McCoy, J. (2019, August). Enhancing wave function collapse with design-level constraints. In Proceedings of the 14th International Conference on the Foundations of Digital Games (pp. 1-9).

[26] the Second DIMACS implementation Challenge. http://dimacs.rutgers.edu/programs/challenge/. DIMACS.

[27] Tomita, E., Tanaka, A., & Takahashi, H. (2006). The worst-case time complexity for generating all maximal cliques and computational experiments. Theoretical computer science, 363(1), 28-42.

[28] Vertex Cover Benchmark Instances.http://www.cs.hbg.psu.edu/benchmarks/vertex_cover.html.

[29] Wang, Z., Huang, D., & Pei, R. (2014). Solving the minimum vertex cover problem with DNA molecules in Adleman-Lipton model. Journal of Computational and Theoretical Nanoscience, 11(2), 521-523.

[30] Xu, K., Boussemart, F., Hemery, F., & Lecoutre, C. (2005). A simple model to generate hard satisfiable instances. arXiv preprint cs/0509032.