Minimum Vertex Cover (MVC)
CSE 6140 Final Project – Group 21

Jiaxing Su
Georgia Institute of Technology
Address
Atlanta, GA, 30339
+1 (404)-731-7826
jsu38@gatech.edu

Bo Xu
Georgia Institute of Technology
Address
Atlanta, GA, 30318
+1 (770)-880-5690
brodyxu@gatech.edu

Hantao Yang
Georgia Institute of Technology
Address
Smyrna, GA, 30080
+1 (734)-353-1530
hyang320@gatech.edu

1. INTRODUCTION

The Minimum Vertex cover (MVC) problem is a well-known NP-complete problem with numerous applications in computational biology, operations research, the routing and management of resources. In this project, we will treat the problem using different algorithms, evaluating their theoretical and experimental complexities on real world datasets.

2. PROBLEM DEFINATION

Given an undirected graph $G = (V, E)$ with a set of vertices V and a set of edges E, a vertex cover is a subset $C \subseteq V$ such that $\forall\, (u, v) \in E : u \in C \vee v \in C$. The Minimum Vertex Cover problem is therefore simply the problem of finding minimizing $|C|$. The following diagram shows an example of minimal vertex cover; the solution is represented by $\{a, c, f, g\}$:

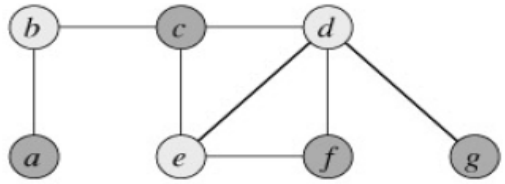

Fig 1. Minimum vertex cover of graph G = (*V*, *E*)

3. RELATED WORK

Many algorithms have been proposed to construct vertex cover in different contexts (offline, online, list algorithms, etc.) leading to solutions of different level of quality. This quality is traditionally measured in terms of approximation ratio, that is, the worst possible ratio between the quality of the solution constructed and the optimal one. For the vertex cover problem the range of such known ratios are between 2 (conjectured as being the smallest constant ratio) and Δ, the maximum degree of the graph. Based on this measure of quality, the hierarchy is almost clear (the smaller the ratio is, the better the algorithm is).

For heuristic with approximation guaranteed, there are current algorithms proposed, which are maximum degree greedy, greedy independent cover, depth first search, edge deletion, list left, and list right.

The implementation of branch-and-bound algorithms on minimum vertex cover is a well-studied subject. A traditional and successful approach here is to keep the number of live vertices in the search tree low "during the computations by means of tight bound calculations, even if these calculations are time-consuming. Since the bound calculation for each live vertex is independent of all other bound calculations, and a number of case studies have appeared, like parallel implementation.

Local search can be used on minimum vertex cover problem since it can be formulated as finding a solution maximizing a criterion among many of candidate solutions. Local search algorithms move from solution to its neighbor solutions in the space of candidate solutions, the search space, by applying local changes, until a solution fall within an acceptable quality close to the optimal or a cut-off time has elapsed or number of iterations has been reached. There are a variety of families of local search algorithms, i.e., Hill Climbing, Simulated Annealing, Genetic Algorithm, etc. To start a local search algorithm, heuristic approximation algorithms described above can be used as algorithm initialization.

4. ALGORITHMS

4.1 Branch and Bound

4.1.1 Description

A Branch & Bound algorithm searches the complete space of solutions for a given problem for the best solution. However, explicit enumeration is normally impossible due to the exponentially increasing number of potential solutions. The use of bounds for the function to be optimized combined with the value of the current best solution enables the algorithm to search parts of the solution space only implicitly.

4.1.2 Pseudo Code

Algorithm 1 Branch and Bound

Input: graph $G = (V, E)$
Output: minimum vertex cover of G
$C \leftarrow \emptyset$
$tempVC \leftarrow \emptyset$
covered edge set Covered $\leftarrow \emptyset$
explored vertex set Explored$\leftarrow \emptyset$
Lower bound = number of total vertices

Branch and Bound (tempVC, Covered, Explored**)**
if execution time is bigger than cutoff
    return
end if
if VC $covers\ all\ edges$ AND number of VC is smaller than Lower Bound **then**
    Lower Bound = number of VC
    $C \leftarrow VC$
end if

Find the vertex v that has the maximum edges, where vertex ∈ V \ tempVC and edge ∈ E \ Covered

Add this v to Explored
$G \leftarrow G \setminus \{v\}$
add all existed edges of v to Covered
if lower bound is bigger than number of VC plus **Two-approximation**(G) **then**
    add v to tempVC
    Recursively call **Branch and Bound(**tempVC, Covered, Explored**)**
end if

Remove the new added edges in Coverred
Remove v from the tempVC
Recursively call **Branch and Bound(**tempVC, Covered, Explored**)**
return C

Algorithm 2 Two-approximation

Input: graph $G = (V, E)$
Output: approximation minimum edge number of G
$C \leftarrow \emptyset$ (the vertex cover)
$E' \leftarrow E$(uncovered edges)
While $E' \neq \emptyset$ ***do***
    Arbitrary edge $(u, v) \notin E'$
    $C \leftarrow C \cup \{u, v\}$;
    Remove from E'every edge incident to u or v
return C

4.1.3 Data Structure used

- Array List
- Hash Set
- Hash Map

4.1.4 Time and Space Complexity

The time complexity is O(2^|V|) because the binary DFS is used to find the best solution. The explored vertex is stored in one hash set so that this algorithm will not explore it again. Therefore, every branch has two child nodes and the maximum depth is |V|.

The space complexity is O(|V|+|E|) because the original graph takes such space and the Covered and Explored sets are either O(|V|) or O(|E|). Meanwhile, the two-approximation algorithm also takes O(|V|) space. Therefore, the whole space complexity is O(|V|+|E|).

4.2 Heuristics with Approximation

4.2.1 Description

The Edge Deletion (ED) algorithm, proposed by Gavril and Johnson [1], returns the vertices of a maximal (but not necessarily maximum) matching. Its worst-case approximation ratio is 2 (Cormen et al. [2]). More precisely, the tight worst-case approximation ratio is asymptotic to min $\{2, \frac{1}{1-\sqrt{1-\epsilon}}\}$ for graphs with an average degree of at least ∈ n and to min $\{2, \frac{1}{\epsilon}\}$ for graphs with a minimum degree of at least ∈ n (Cardinal et al. [3]).

4.2.2 Pseudo Code

Algorithm 3 Approximation with Edge Deletion

Input: G = (V, E)

Output: a vertex cover of C

$C \leftarrow \emptyset$;

While $E \neq \emptyset$ ***do***

Select {u, v} $\in E$;

$C \leftarrow C \cup \{u, v\}$;

$V \leftarrow V - \{u, v\}$;

$E \leftarrow E$ – *all edges covered by {u, v};*

Return *C;*

4.2.3 Data Structure Used

- Hash Set
- Array List

4.2.4 Time and Space Complexity

Time complexity is $O(E + V)$ since the worst case is to include all vertex in our cover to delete all edges. Therefore, the total number of iteration is up to *O (E)*, and the total number of combining new vertices to our cover is $O(V)$.

The space complexity is $O(V)$, as for the extra space, we used a hash set to keep track of all vertices in the cover, and a int [] map to store remaining edges for each vertex. Thus, the overall space complexity is $O\ (V)$.

4.3 Local Search 1 (FastVC)

4.3.1 Description

Local search algorithms are efficient heuristic approaches to solve NP-hard problems in reasonable time. Although they do not guarantee to achieve global optimal solution, they are usually not far from the optimal. The first local search algorithm we adopted is call ***FastVC***, which is proposed by Cai [2015]. We basically followed the same pipeline as described in the paper.

4.3.1.1. FastVC

The main algorithm ***FastVC*** basically follows the common local search framework suggested in the problem description. Before going into details of the algorithm, we need to define two pricing function:

$\boldsymbol{loss(v)}$**:** the number of covered edges that would become uncovered by removing v from C.

$\boldsymbol{gain(v)}$**:** the number of uncovered edges that would become covered by adding v into C.

The algorithm starts with initialize a vertex cover C by calling function ***initVC***. The values of $loss(v)$ for all vertices in C are also calculated. The values of $gain(v)$ for the rest of vertices are set to 0 since adding those vertices to the initial vertex cover C would not help to gain more edge coverage (by the definition of vertex cover). Then we remove redundant vertices with minimum loss values when it doesn't affect the property of vertex cover C (covers all the edges). Then call function $chooseVertex(C)$ which randomly sample and remove one vertex from C. From the uncovered edge sets, randomly select an edge, add one of its vertices having bigger gain value into the vertex cover C. Repeat this process until time off, and return the current best vertex cover C^*.

4.3.1.2. ***initVC***

Start an empty set of vertices, iterate through all the uncovered edges and add the endpoint of each of these edges that has more neighbor vertices into the set C. Initialize all loss values of the vertices in C to 0. Then for each edge, increment the loss value of the endpoint of it belonging to C. Finally, remove the redundant vertices that have zero loss.

4.3.1.3. ***chooseV***

In the main algorithm, we need to randomly remove a vertex from the current vertex cover solution. Instead of using the 'best-picking' strategy that could be very slow for large graphs, a more cost-efficient way was adopted. We randomly sample 50 vertices and choose the edge of best—lowest loss value and return it back to ***FastVC***.

4.3.2 Pseudo Code

Algorithm 4 FastVC

Input: graph $G = (V, E)$, cutoff time
Output: minimum vertex cover of G
$C \leftarrow initVC()$
$gain(v) \leftarrow 0$ initialize vertices $v \notin C$
while $elapsed\ time < cutoff$ **do**
    if C $covers\ all\ edges$ **then**
        $C^* \leftarrow C$
        $C \leftarrow C \setminus \{argmin(loss(v),\ v \in C)\}$
        continue
    $u \leftarrow chooseV(C)$
    $C \leftarrow C \setminus \{u\}$
    $e \leftarrow$ random uncovered edge, $e = (v_1, v_2)$
    $v \leftarrow argmax(gain(v_1), gain(v_2))$
    $C \leftarrow C \cup \{v\}$
return C^*

Algorithm 5 initVC

Input: graph $G = (V, E)$
Output: vertex cover of G
$C \leftarrow \emptyset$;
for $e \in E$ **do**
    if $e = (v_1, v_2)$ *is uncovered by* C then
        $C = C \cup \{argmax(neighbor(v_1), neighbor(v_2))\}$
$loss(v) \leftarrow 0, v \in C$
for $e \in E$ **do**
    if$(v_1 \in C, v_2 \notin C)$ *or* $(v_2 \in C, v_1 \notin C)$ **then**
        $v = arg(v_1, v_2 \in C)$, $loss(v)$++
for $v \in C$ **do**
    if $loss(v) = 0$ **then**
        $C \leftarrow C \setminus \{v\}$
        Update loss of $neighbor(v)$
return C

Algorithm 6 chooseV

Input: vertices set V, integer $k = 50$
Output: vertex of V
$best\_v \leftarrow$ randomly select a vertex from V
for $i \leftarrow 1$ to $k - 1$ **do**
    $r\_v \leftarrow$ randomly select a vertex from V
    if $loss(r_v) < loss(best\_v)$ **then**
        $best\_v \leftarrow r\_v$
return $best\_v$

4.3.3 Data Structure used

- Array List
- Hash Set
- Hash Map
- Concurrent Hash Map

4.3.4 Time and Space Complexity

Algorithm 5 and 6 are parts of algorithm 4, so we need to analyze the complexity of these two first. The time complexity of algorithm 5 is $O(E)$: there are 3 loops in the algorithm, the first two loop through all the edges, so $O(E)$. The last loop iterates over all vertices in C which at most has $|2E|$. Space complexity is $O(V)$ since the size of vertex cover is at most the same as the total vertex number. For algorithm 6, time and space complexities are $O(1)$ because the parameter k is always much smaller than the $|V|$. With these pieces, we can start analyzing algorithm 4: Initialization is performed by algorithm 5; In each iteration of the while loop, checking C covers all edges takes $O(E)$; Removing a vertex with minimum loss from C requires $O(V)$. In our implementation, randomly choosing an uncovered edge requires a construction of the uncovered edge set which requires $O(E)$ time and space; Finally, updating loss and gain values takes $O(degree) = O(1)$, and all the loss and gain values were stored in ArrayList's, so space complexity is $O(V)$.

All in all, the space complexity is $O(V + E)$, time complexity for initialization is $O(E)$, and time complexity for each iteration in while loop is $O(V + E)$. Note that in our implementation, we wrote the result to trace only when we got a better solution, so the total iterations of the while loop, and therefore the total runtime, depends on the random seed and the cut-off time limit.

4.4 Local Search 2 (Simulated Annealing)

4.4.1 Description

Simulated Annealing (SA) is another metaheuristic to approximate the global optimal solution in a large search space. The classical local search algorithms always accept new better neighbors, and thus, the algorithms terminate at their first local optima. However, SA allows to accept worse solutions with lower probabilities. Moreover, like the slow cooling process of annealing in metallurgy, as the solution space is explored, the acceptance probability of worse solutions decreases. We adopted a variant version of Efficient Simulated Annealing (ESA) proposed by Xu and Ma [2006]. we initial goal is to minimize the cost function defined as the sum of the vertex cover size and the number of uncovered edges:

$$\boldsymbol{cost} = \boldsymbol{|VC|} + \boldsymbol{|uncovered\ edges|}$$

because we want to have the minimum cover size and have all the edges covered. However, frequently checking the size of uncovered edges is costly. Thus, we simplified the cost function to vertex cover size:

$$\boldsymbol{cost} = \boldsymbol{|VC|}$$

and check the validity of the current candidate. In the algorithm, we initialized our candidate solution with function $initVC2$, which is based on the edge deletion method mentioned in Delbot and Laforest [2010]. Then we search to a neighbor of the current solution, and decide if we will accept this new solution based on the probability calculated in the following binary function:

$$p_i = \begin{cases} e^{-\Delta cost(1-d(v_i))/T}, & v_i\ was\ uncovered \quad (1a) \\ e^{-\Delta cost(1+d(v_i))/T}, & v_i\ was\ covered \quad (1b) \end{cases}$$

where v_i is the flipped vertex from the previous candidate solution to its neighbor, and $d(v_i) = \frac{degree(v_i)}{|E|}$.

If $\Delta cost < 0$, we got a better candidate and accept it.

If $\Delta cost > 0$ and v_i was not covered by previous candidate. In equation (1a), p_i would be large if $d(v_i)$ is large since we want to cover more edges. Otherwise, if the degree of the vertex v_i is small, a lower probability acceptance rate is assigned.

If $\Delta cost > 0$ and v_i v_i was covered previously. In this situation, we are facing worse solutions, so a large degree of the vertex v_i will result in a low acceptance probability.

4.4.2 Pseudo Code

Algorithm 7 initVC2 (Edge Deletion)

Input: graph $G = (V, E)$
Output: vertex cover of G
$C \leftarrow \emptyset$
while $|E| > 0$ **do**
  $e \leftarrow$ random uncovered edge, $e = (v_1, v_2)$
  $C \leftarrow C \cup \{u, v\}$
  $E \leftarrow E \setminus \{e\}$
return C

Algorithm 8 Simulated Annealing

Input: graph $G = (V, E)$, $cutoff$ time
Output: minimum vertex cover of G
$C \leftarrow initVC2()$
while $elapsed\ time < cutoff$ & $T > 0$ **do**
  $initialize\ T, cooling\ rate$
  $v \leftarrow$ randomly flip one vertex v_i
  if $v \in C$ **then**
    if $C \setminus \{v\}$ covers all the edges **then**
      $C' = C \setminus \{v\}$
  else
    $C' = C \cup \{v\}$
  $\Delta \leftarrow cost(C') - cost(C)$
  if $\Delta < 0$ **then**
    $C \leftarrow C'$
  else
    if $v_i \in C$ **then**
      $C \leftarrow C'$ with p_i in (1a)
    else
      $C \leftarrow C'$ with p_i in (1b)
  $T = T \times colling\ rate$
return C

4.4.3 Data Structure used

- Array List
- Hash Map

4.4.4 Time and Space Complexity

The edge deletion method initialization (algorithm 7) terminates when the edges are all covered. The worst case is that in each iteration, only one edge is deleted. So, the time complexity is $O(E)$, and space complexity is $O(V)$;

In algorithm 8, after initialing the vertex cover candidate, in each iteration, we check if the neighbor of previous candidate $C \setminus \{v\}$ covers all the edges, it loops through the neighbors of v, and it takes $O(degree) = O(1)$ time and the neighbor instance takes $O(V)$ space. The calculation of the cost value only takes $O(1)$ since we simplified the cost function to the size of the current vertex cover candidate.

All in all, the space complexity is $O(V)$, time complexity for initialization is $O(E)$, and in each iteration of the while loop, time complexity is $O(1)$. Note that the total runtime also depends on how many loops to run, which is determined by the initial temperature T and the cooling rate. And we only wrote candidate solutions to the file when smaller vertex cover is found.

5. EMPERICAL EVALUATION

5.1 Platform

Device 1 was used for producing trace files, solution file, and result table of approximation algorithm.

Device 2 was used for producing trace files, solution file, result table and the plots files of local search algorithms.

Device 3 was used for producing trace files, solution files, result table of B&B algorithm.

Table 1: Platforms

	Device 1	Device 2	Device 3
Processor	2.4 GHz Intel Core i7	3.6 GHz Intel Core i7	2.7 GHz Intel Core i5
Ram	16 GB	32 GB	8 GB
Language	Java	Java	Java
Compiler	GCC	javac	javac
System	Windows 7	Windows 8	Mac OS

5.2 Experimental Procedure

For each of the algorithms, we ran 10 tests with different random seeds for each of the eleven graphs. The runtime and relative error results shown in the appendix are averaged over these 10 tests for each graph each algorithm.

In terms of the two local search algorithms, the boxplots, *QRTDs*, and *QSDs* were also based on these 10 tests for each graph. The cutoff time for both algorithms is 600 seconds. Detail of these results will be described in section 5.3.

5.3 Results

To evaluate the performance of these algorithms, the author uses Qualified Runtime, Solution Quality and Solution Variance as the major Criteria.

To compare the performance of different algorithm provided above, the author provided a Result Table in the appendix. The table contains the VC result of each algorithm, relative error (%) and running time. Note that, the cut-off time used in the local search algorithm is 10 minutes (600s), and the results are averaged over 10 tests.

5.3.1 Branch & Bound

The total structure of the algorithm is binary DFS, so the execution time is extremely smaller than traverse every vertex in every recursively function. This Branch and Bound algorithm could run out the seventh large graph in 30s.

The solution quality is not very good because BnB should run out a true value. This error is caused by the lower bound in pruning the branch. To calculate the lower bound of the remaining graph, two-approximation algorithm is used in Branch and Bound. This approximation result might cause the error. Therefore, this process might prune the branch that containing the true solution.

Because the branch and bound should traverse all the possible branches, it cannot come out a result for the four largest graph because of running time.

5.3.2 Heuristics with Approximation

The upper bound of VC problem using approximation algorithm (edge deletion) is two times of the optimal solution. The average relative error for different instance graph is typically less than 0.5 since it randomly selects the edge for each deletion, it might include a few non-optimal selections and thus has a relative large error.

5.3.3 Local Search

5.3.3.1 Boxplot

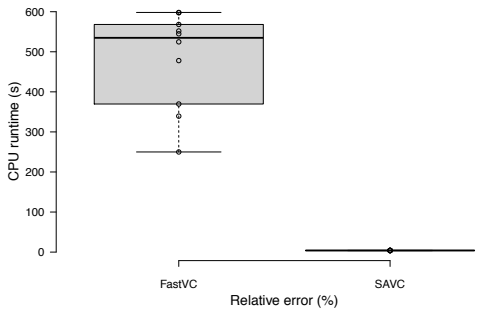

Fig 2. Boxplots for *FastVC* and *SAVC* ran on graph star2

Figure 2 shows the comparison between runtimes for best solution before within cutoff time of *FastVC* and *SAVC* (Simulated Annealing) on graph star2. The runtime difference is huge: *FastVC* has much larger mean and *IQR* than *SAVC*. This is coincident with our runtime analysis in last section. One thing to be noted is that in the simulated annealing algorithm, when temperature is cool, the acceptance rate for new neighbors is extremely low, that's why the candidate solution wouldn't update (we only wrote better solutions to trace files) after several seconds after start.

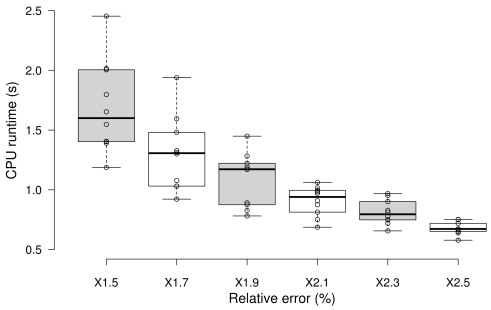

Fig 3. Boxplots for *FastVC* ran on graph power

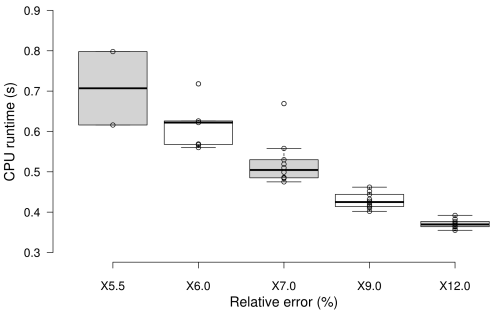

Fig 4. Boxplots for *SAVC* ran on graph power

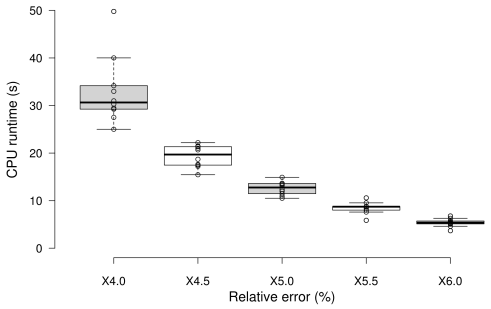

Fig 5. Boxplots for *FastVC* ran on graph star2

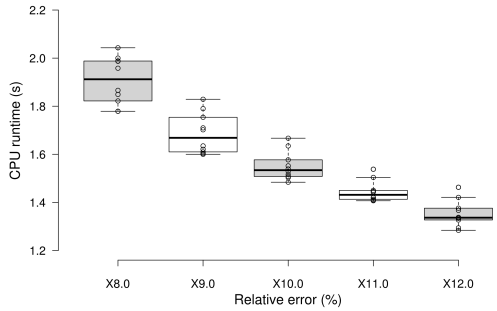

Fig 6. Boxplots for *SAVC* ran on graph star2

Figure 3 to 6 show the boxplots of the two local search algorithms ran on graph power and star2. Generally, bigger graph costs longer runtime. On the other hand, for the same graph, with the decrease in relative error, the runtime is higher and the variation is also monotonically increasing. Moreover, like Figure 2, we can observe that, at any given quality, *FastVC* is more sensitive to the randomness induced by different random seeds. Note that in Figure 4, with given quality 5.5%, not all tests can reach this quality. The targeting quality also influence on the shape of the qualified *RTDs* that we will show in the following section.

5.3.3.2 Qualified RTDs

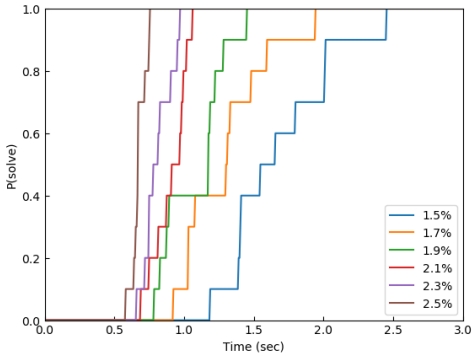

Fig 7. *QRTD* for *FastVC* ran on graph power

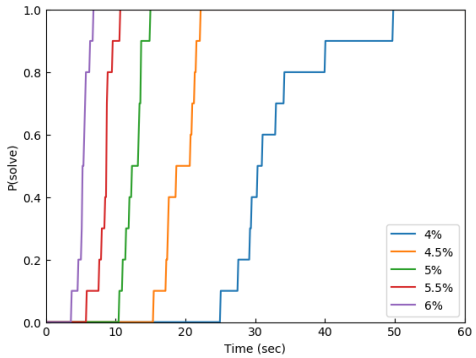

Fig 8. *QRTD* for *FastVC* ran on graph star2

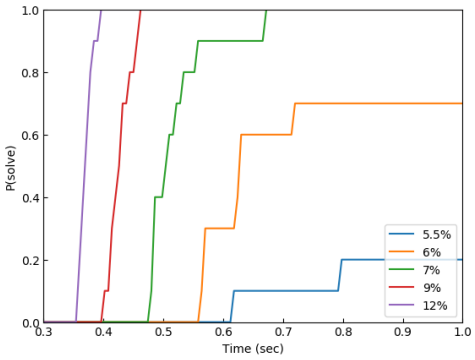

Fig 9. *QRTD* for *SAVC* ran on graph power

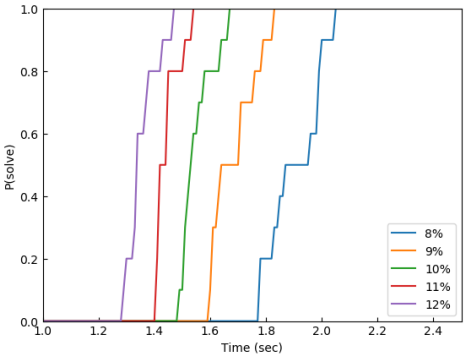

Fig 10. *QRTD* for *SAVC* ran on graph star2

From Figure 7 to 10, we can observe that better quality solutions require longer runtime on average, which is intuitively reasonable. In addition, shown as the left most curves in each figure, given a low quality (high relative error), random seed doesn't make much difference on the runtime of the 10 tests. However, when we asked for better qualities (curves toward the right of the x axis), variation on runtime becomes more apparent, especially when we are aiming for a quality that the algorithm couldn't consistently reach. Figure 9 shows this situation, for relative error at 5.5% and 6%, the corresponding curves didn't reach the ratio 1.0 since the algorithm doesn't yield these small errors for all 10 tests and the targeting quality is beyond the ability of our *SAVC* algorithm on graph power.

From Figure 11 to 14, we can generalize that as the runtime increases, the probability to get a better solution becomes larger. On the other hand, with a given runtime, the probability to get a solution is higher when the aiming quality is low.

5.3.3.3 SQDs

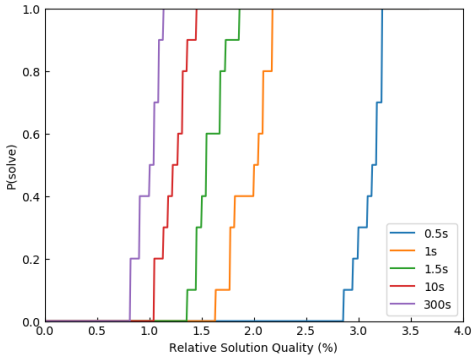

Fig 11. *SQD* for *FastVC* ran on graph power

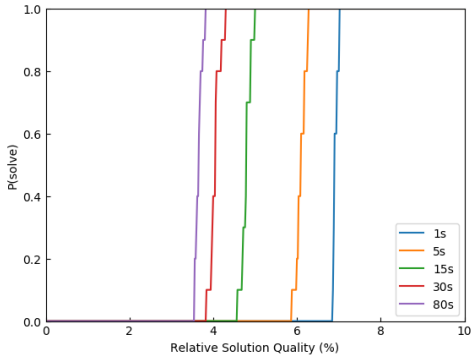

Fig 12. *SQD* for *FastVC* ran on graph star2

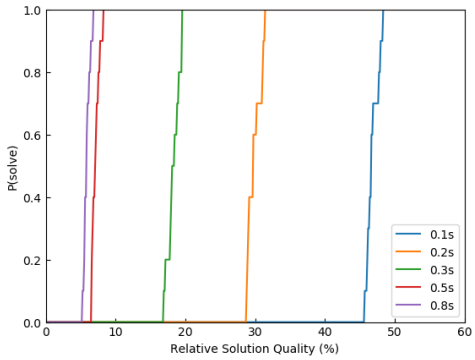

Fig 13. *SQD* for *SAVC* ran on graph power

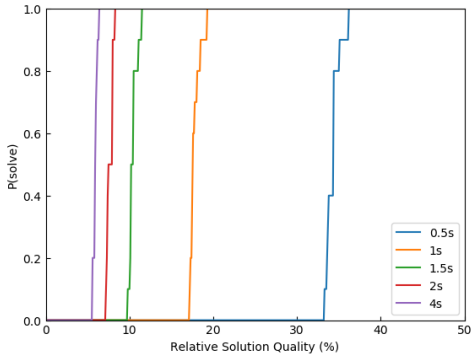

Fig 14. *SQD* for *SAVC* ran on graph star2

6 CONCLUSIONS

6.1 Branch & Bound

As for the Branch and Bound algorithm, it performs well when solving the small graph, like the karate graph, the error rate is 0.00%. However, when the graph becomes larger, the lower bound plays a significant role in pruning the branch. The two-approximation in this algorithm is not a tight lower bound so that it causes increasing error rate growing, like power graph is 3.36%. The branch and bound algorithm should be precise because it traverses all the probability and pick the optimal value based on a tight lower bound.

6.2 Heuristics with Approximation

For heuristic with approximation, we only tested on the edge deletion algorithm, which showed relatively 27.81% error (majority of cases are 20% - 50%, quite acceptable than the worst-case approximation ratio of 2). However it trades the approximating quality for a faster processing time as we can see from the running time in result table among others. Besides, the edge deletion algorithm works has a better performance on the relatively small sets of vertex.

6.3 Local Search

Generally, the two local search algorithms we implemented are efficient and stable. Although optimal solutions are not guaranteed, acceptable relative errors are observed for both algorithms. The averaged relative errors of the solutions with *FastVC* are mostly negligible ranging from 0 – 3.54%. Thus, *FastVC* always yields solutions very close to the optimal one. On the other hand, the Simulated Annealing methods returns relative error ranging from 0.82 – 9.65%, which is also acceptable since the sacrifice on solution exactness was traded off with relative low runtime comparing to other algorithms.

7 REFERENCES

[1] GAREY, M. AND JOHNSON, D. 1979. Computers and Intractability. Freeman and Co., New York.

[2] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2009. Introduction to Algorithms 3rd Ed. MIT Press, Cambridge, MA.

[3] CARDINAL, J., LABBE´, M., LANGERMAN, S., LEVY, E., AND MELOT', H. 2005. A tight analysis of

the maximal matching heuristic. In Proceedings of the 11th Annual International Conference on Computing and Combinatorics. Springer–Verlag, Berlin, 701–709.

[4] CAI, S. 2015. 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, Buenos Aries, 747-753.

[5] XU, X. AND MA J. 2006. An Efficient Simulated Annealing Algorithm for the Minimum Vertex Cover Problem. Neurocomputing, Vol.69, 913-916.

[6] CLAUSEN, J. AND LARSSON T., J. 1991. Implementation of parallel branch-and-bound algorithms – experiences with the graph partitioning problem. Annals of Operations Research, Vol.33(5), 329-349.

APPENDIX

Table 1. Performance of implemented algorithms

Dataset	Branch & Bound			Approximation		
	Time (s)	Result	RelErr	Time (s)	Result	RelErr
jazz	0.27	159	0.63%	6.26E-4	165	4%
karate	0.01	14	0.00%	3.16E-5	14	0%
football	0.05	96	2.13%	8.47E-5	98	4%
as-22july06	null	null	null	0.0209	5218	58%
hep-th	null	null	null	0.0238	5244	33%
star	null	null	null	0.0341	8378	21%
star2	null	null	null	0.0298	6587	45%
netscience	1.59	899	0.00%	0.0031	1088	21%
email	1.25	605	1.85%	0.0024	739	24%
delaunay_n10	0.96	737	4.84%	0.0017	855	22%
power	27.67	2277	3.36%	0.0063	3307	50%

Dataset	FastVC			Simulated Annealing		
	Time (s)	Result	RelErr	Time (s)	Result	RelErr
jazz	2.03	158.00	0.00%	0.01	162.90	3.10%
karate	0.00	14.00	0.00%	0.01	14.80	5.71%
football	0.058	94.00	0.00%	0.01	96.70	2.87%
as-22july06	150.43	3307.50	0.14%	5.41	3330.00	0.82%
hep-th	179.76	3932.90	0.18%	2.13	4097.70	3.91%
star	439.98	7040.70	2.01%	5.31	7567.70	9.65%
star2	482.21	4702.70	3.54%	4.04	4809.70	5.89%
netscience	0.01	899.00	0.00%	0.12	934.40	3.94%
email	85.86	596.20	0.37%	0.09	628.20	5.76%
delaunay_n10	49.73	712.80	1.39%	0.07	764.80	8.79%
power	464.28	2222.60	0.80%	0.81	2332.3	5.87%