CSE 6410 Project Report

AMELIA GLAESE, YONGRUI LIN, and WANYANG GUO

1 INTRODUCTION

The Minimum Vertex Cover problem is a classic NP-Complete problem with many practical applications such as network security, power grids, or online social networks. In this paper we first introduce the problem and review existing research to this problem. We then present and compare different approaches to solve the problem: the exact branch-and-bound algorithm, an approximation algorithm using construction heuristics with approximation guarantee and two local search algorithms using an iterated local search and a simulated annealing approach. The algorithms are evaluated and compared based on the size of the resulting vertex cover and the necessary computing time for a number of well known data sets of different sizes and structures.

2 PROBLEM DEFINITION

Given a graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices and $E = \{e_1, e_2, ..., e_m\}$ the set of edges connecting two vertices, $e_i = (v_j, v_k)$. A vertex cover $V_C \subseteq V$ is a subset of all vertices in V, such that all edges $\in E$ connect to at least one vertex in V_C.

$$V_C \subseteq V : \quad v_i \in V_C \quad \vee \quad v_j \in V_C \quad \forall \quad e = (v_i, v_j) \in E$$

The minimum vertex cover problem asks for the vertex cover with the smallest number of vertices:

$$V_{C_{min}} = \arg\min_{V_C} |V_C|$$

3 RELATED WORK

The minimum vertex cover problem is a well researched problem and exact, approximation, and local search algorithms for this problem have been focus of many publications. The branch-and-bound algorithms are a complete algorithms that guarantees to find the optimal solution to the minimum vertex cover problem. Because it recursively explores the tree of all possible solutions, the worst case time complexity is bounded by 2^n [12]. Here, existing research is mainly focused on calculating typical running times of branch-and-bound algorithms finding vertex covers of size k such as [18] and [13]. Approximation algorithms usually have polynomial running time and provide a guaranteed approximation ratio. The best known constant approximation ratio for the minimum vertex cover is 2; however, there are algorithms with non-constant approximation guarantees based on the maximum degree of the graph that can be lower than 2. The approximation algorithms proposed in literature are based on different heuristics, such as the Maximum Degree Greedy Algorithm [6], the Depth-First-Search Algorithm [14], the Left-List Algorithm [1] and the Dijkstra Algorithm [5]. The proposed algorithms have different theoretical time complexities, i.e. the worst case running time of the Maximum Degree Greedy Algorithm and the Left-List Algorithm is bounded by $O(n * m)$ while the Dijkstra Algorithm is bounded by $O(n^3)$. For the purpose of this paper, we implemented the Edge-Deletion Algorithm propoposed in [8], which has a worst-case time complexity of $O(m)$.

Local search algorithms have no approximation guarantees but can perform very well and achieve much better solutions on average than approximation algorithms. However, better solutions are often achieved at the cost of much longer runtime. In literature, different types of local search algorithms have been proposed. Many use an iterated local search approach: [4] presented the

Authors' address: Amelia Glaese; Yongrui Lin; Wanyang Guo.

NumVC algorithm based on two stage exchange and edge weighting with forgetting. [2] introduced a new FastVC algorithm with a smaller time Complexity per iteration by utilizing a Best from Multiple Selections approach. This algorithm results in better solutions for large graphs for shorter runtimes (the authors use 1000s) compared to the NumVC algorithm. Finally, [3] present two algorithms based on finding a partial vertex cover that provides a better upper bound on the size of the minimum vertex cover and extends it into a vertex cover. Other approaches include simulated annealing ([19]), genetic algorithms ([10], [11]), and ant-colony based algorithms ([9], [15]).

A few publications also analytically and empirically compare different algorithms for the minimum vertex cover with regard to time complexity and quality of solution. [12] and [17] compare different types of algoriths: the BnB algorithm, a genetic algorithm, the Maximum Degree Greedy Algorithm, the Primal-Dual Algorithm for weighted graphs, and Alom's algorithm. [7] and [16] compare the various 2-approximation algorithms presented above.

4 ALGORITHMS

4.1 Branch and Bound

The branch-and-bound algorithm implemented for this project begins with an empty set of vertices. The 2-approx is used to establish an initial crude upper bound. The algorithm selects the vertex with the highest degree of uncovered neighbors. The algorithm looks at this vertex and either adds the vertex to the partial solution, or leaves it out and adds its neighbors, depending on which is more promising. If a branch is determined not to be promising, it is not explored any further. Upon finding a vertex cover, the upper bound is updated to the size of the current best cover. The algorithm terminates when no promising branches remain.

More specifically, we have a graph $G = (V, E)$. The problem is to find a vertex cover with the smallest number of vertices in G.

SP: Given a partial solution C to the minimum vertex cover problem $G = (V, E)$, the resulting sub-problem is the minimum vertex cover problem for the subgraph $G' = (V', E')$, where $V' = V \setminus C$ and $E' = E \setminus \{\text{edges covered by } C\}$.

Choice: We can choose from the queue using the lower bound on the objective value for the subproblems.

Expand: We expand a subproblem by selecting the vertice $v_{maxDegree}$ with the highest degree and either adding it to the partial solution $C_{left} = C \cup v_{maxDegree}$ or adding its neighbors to the partial solution $C_{left} = C \cup \text{neighbors}(v_{maxDegree}, E)$. Here adding all neighbors is the logical branch, because if $v_{maxDegree}$ is not added to our partial solution, all edges incident to $v_{maxDegree}$ can only be covered by its neighbors.

LB: We calculate the lower bound of a partial solution C by finding the maximal matching M of the subgraph i.e. $LB(C) = |C| + |M|$

ALGORITHM 1: Branch and Bound

Data: V, E
Result: $minCover$ = smallest vertex cover found
$C = V$
$LowerBound = 2Approx(V, E)$
$minCover$ = `MinVertexCoverBnB(`C, V, E`)`

def `MinVertexCoverBnB(`C, V, E`)`**:**
  if $|C| + getLowerBound(V, E) > LowerBound$**:**
    return $\emptyset$
  if C *is vertex cover*:
    $minC = C$
    $LowerBound = |minC|$
    return $minC$
  v = getMaxDegreeVertex(V, E)
  $neighborsV$ = neighbors(v, V, E)
  $C_{left} = C \cup v$
  $V_{left} = V \setminus v$
  $E_{left} = E \setminus$ edges(v, E)
  $minC_{left}$ = `MinVertexCoverBnB(`$C_{left}, V_{left}, E_{left}$`)`
  $C_{right} = C \cup \{neighborsV\}$
  $V_{right} = V \setminus \{v\}$
  $E_{right} = E \setminus \{neighborsV\}$
  $minC_{right}$ = `MinVertexCoverBnB(`$C_{right}, V_{right}, E_{right}$`)`
  if C_{left} *is empty or* $|C_{right}| \leq |C_{left}|$**:**
1   **return** C_{right}
  else:
2   **return** C_{left}

4.2 Construction Heuristics

For the construction heuristics, we used a Greedy Edge Deletion Algorithm. This algorithm returns the vertices of a maximal (but not necessarily maximum) matching. It's worse-case approximation ratio is 2 (as proven in [8]). 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 $\in n$ and asymptotic to min $[2, \frac{1}{\epsilon}]$ for graphs with a minimum degree of at least $\in n$.
First, the algorithm randomly chooses a edge $e = (u, v) \in E$ from the initial edge set E, and adds both end nodes of the chosen edge to the minimum vertex cover set $minCover$. In the next step, it removes every edge incident to either u or v from the edge set E. This procedure is repeated until there are no edges in left $E = \emptyset$. The resulting $minCover$ is the approximated minimum set cover.

ALGORITHM 2: Construction Heuristics with Edge Deletion

Data: V, E
Result: $minCover$ = smallest vertex cover found
1 $minCover = \emptyset$
2 **while** $E \neq \emptyset$**:**
3 select a random edge $e = (u, v) \in E$
4 $minCover = minCover \cup \{u, v\}$
5 $E = E \setminus \{$edges incident to u and $v\}$
7 **return** $minCover$;

4.3 Local Search

4.3.1 Initial Vertex Cover. The function to construct the initial vertex cover used by both local search algorithms starts with an empty set and iterates over all $e = (v_1, v_2) \in E$. If an edge yet uncovered by the current set of nodes C is encountered, the node $v_i \in e$ with the higher degree is added to the set. The generation of this cover is computationally efficient but yields relatively small vertex covers.

ALGORITHM 3: Construct Initial Vertex Cover

Data: V, E
Result: C = constructed vertex cover, I = vertices not included in C
C, I = InitialVertexCover(V, E)
def `InitialVertexCover`(C, I, V, E)**:**
    $C = \emptyset$
    $I = V$
    for $e = (v_1, v_2) \in E$**:**
        if *e not covered by C*:
            $v_i = \max_{v_i}(\text{degree}(v_1), \text{degree}(v_2))$
            $C = C \cup v_i$
            $I = I \setminus \{v_i\}$
    return C, I

4.3.2 Iterated Local Search Approach. The first Local Search algorithm utilizes an Iterated Local Search approach. First, an initial vertex cover is constructed using the greedy algorithm described in section 4.3.1. Then the algorithm randomly picks k vertices (here $k = 10$) from the cover and computes the loss for each node. The loss is defined as the number of edges uncovered by the remaining set of nodes if the vertex was removed from the vertex cover:

$$\text{loss}(v_i) = C \setminus \{v_i\}, i \in k$$

In each iteration the algorithm removes the vertex with the least loss among the k nodes. This process is repeated until the resulting set of vertices is not a vertex cover. Then the algorithm starts exchanging vertices in the current set of vertices for vertices currently not included in the set. First the algorithm randomly picks k vertices $\in C$ and removes the node with the least loss. Then the algorithm randomly picks k vertices from the set I of vertices not included in C and computes the gain for each vertice. The gain is defined as the number of edges that are additionally covered by C, if the vertex was added:

$$\text{gain}(v_i) = C \cup \{v_i\}, i \in k$$

ALGORITHM 4: LS1 - Iterated Local Search

Data: V, E
Result: $minCover$ = smallest vertex cover found
$C, I = InitialVertexCover(V, E)$
$minCover$ = `MinVertexCoverLS1`(C, I, V, E)

def `MinVertexCoverLS1`(C, I, V, E)**:**
    while *(elapsed time ≤ cut off time)*:
        while *(C is vertex cover)*:
            $minC = C$
            $minV$ = minLoss(C, V, E, k)
            $C = C \setminus \{minV\}$
            $I = I \cup \{minV\}$
        $minV$ = minLoss(C, V, E, k)
        $C = C \setminus \{minV\}$
        $I = I \cup \{minV\}$ $maxV$ = maxGain(I, V, E, k)
        $C = C \cup \{maxV\}$
        $I = I \setminus \{maxV\}$
1   **return** *minC*

4.3.3 Simulated Annealing. The second local search algorithm is implemented using a simulated annealing approach, minimizing an objective function similar to the one described in ([19]). The objective value of a set of vertices $S \subseteq V, G = (V, E)$, is defined as:

$$O(S) = \text{number of edges} \in E \text{ uncovered} + |S|$$

Like in the first algorithm, an initial vertex cover is constructed using the algorithm described in section 4.3.1. Subsequently, a random node $v \in C$ random node is selected. If $C \setminus \{v\}$ is a vertex cover and its objective value is smaller than the objective value of the current best solution $minC$, v is removed from C and $minC$ is updated. If the objective value of the solution after removing f v, $C \setminus \{v\}$, is greater than the objective value of the current solution C, the probability $P(v)$ that v is added to the current solution is computed as follows:

$$P_{remove}(v) = e^{-\frac{\Delta O * (1 - \text{imp}(v))}{Temp}}$$

where $Temp$ is the cooling coefficient depending on the remaining running time of the algorithm

$$Temp = \text{rem. time} * 100$$

and imp(v) is the importance of the vertice v in Graph $G = (V, E)$, defined as the degree of v normalized by the number of edges in the graph

$$imp(v) = \frac{\text{degree}(v)}{|E|}$$

Based on the computed probability, v is either removed or not removed from C. The same process is implemented for adding a node to the current solution. A random node $u \in I$ is selected. If If $C \cup \{u\}$ is a vertex cover and its objective value is smaller than the objective value of the current best solution $minC$, u is added to C and $minC$ is updated. If the objective value of the solution after removing f v, $C \setminus \{v\}$, is greater than the objective value of the current solution C, the probability $P(v)$ that v is added to the current solution is computed as follows, where $Temp$ and imp(u) are computed as above.:

$$P_{add}(v) = e^{-\frac{\Delta O * (1 - \text{imp}(u))}{Temp}}$$

ALGORITHM 5: LS2 - Simulated Annealing

Data: V, E
Result: *minCover* = smallest vertex cover found
C, I = InitialVertexCover(V, E)
minC = `MinVertexCoverLS2(`C, I, V, E`)`

def `MinVertexCoverLS2(`C, I, V, E`)`**:**
  while *elapsed time* $\leq$ *cut off time*:
    if C *is vertex cover and better solution than minC*:
      $minC = C$
    $randV$ = random vertex $v \in I$
    if qual($C \cup \{randV\}$) > qual(C)**:**
      $C = C \cup \{randV\}$
      $I = I \setminus \{randV\}$
    else:
      $Temp$ = remaining time $*100$
      ΔQ = qual($C \cup \{randV\}$) − qual(C)
      $probV = e^{-\frac{\Delta Q * (1 + \text{imp}(randV))}{Temp}}$
      if $probV$ *is met*:
        `/* remove node */`
        $C = C \cup \{randV\}$
        $I = I \setminus \{randV\}$
    $randV$ = random vertex $v \in C$
    if qual($C \setminus \{randV\}$) > qual(C)**:**
      $C = C \setminus \{randV\}$
      $I = I \cup \{randV\}$
    else:
      $Temp$ = remaining time $*100$
      ΔQ = qual($C \setminus \{randV\}$) − qual(C)
      $probV = e^{-\frac{\Delta Q * (1 - \text{imp}(randV))}{Temp}}$
      if $probV$ *is met*:
        $C = C \setminus \{randV\}$
        $I = I \cup \{randV\}$
    if *new* C *is vertex cover and better solution than minC*:
      $minC = C$
  return *minC*

5 EMPIRICAL EVALUATION

To empirically evaluate our algorithms, we applied them to 11 different well known graphs of different sizes and different structures. The edge-deletion algorithm, the iterated local search algorithm, and the simulated annealing algorithm were applied at least 10 times to the same graph instance using different seeds to obtain the average performance values. Each algorithm was executed until the optimal solution was found; if the optimal solution was not found within 10 minutes of computation, the execution was stopped and the best solution found was reported. For each algorithm, we recorded the (average) running time, the (average) size of the resulting solution and the (average) relative error of the resulting solution. The results for the 11 instances are reported in 1. For the larger instances, the branch-and-bound algorithm was not able to find any solution better than the upper bound within a reasonable time frame. In these cases we report "NA". The algorithms were implemented in Python 2.7 and executed on a computer with a 2.2 GHz Intel Core i7 and 16 GB 1600 MHz DDR3.

	Branch-and-Bound			Construction Heuristics			LS1: Iterated Local Search			LS2: Simulated Annealing		
Dataset	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr
as-22july06	600	NA	NA	141.31	5998	0.81	649.90	3380.82	0.02	601.93	3381.45	0.02
delaunay_n10	600	753	0.07	582.34	898	0.28	601.97	763.91	0.09	600.08	764.89	0.09
email	600	633	0.07	488.2	792	0.33	603.29	636.73	0.07	600.07	637.36	0.07
football	2.07	94	0	463.28	100	0.06	600.10	94.36	0.00	600.00	98.00	0.04
hep-th	600	NA	NA	642.49	5736	0.46	647.87	3995.00	0.02	601.48	3994.45	0.02
jazz	4.11	159	0.01	548.27	170	0.07	600.51	160.91	0.02	600.02	159.25	0.01
karate	0.01	14	0	14.22	14	0	600.00	14.00	0.00	600.00	14.00	0.00
netscience	50.24	967	0.07	89.88	1182	0.31	601.86	899.00	0.00	600.06	899.00	0.00
power	600	NA	NA	291.29	3574	0.62	623.61	2339.64	0.06	600.67	2339.00	0.06
star	600	NA	NA	681.96	10192	0.48	1323.09	7453.00	0.08	608.31	7452.27	0.08
star2	600	NA	NA	149.98	6806	0.49	1869.68	5075.00	0.12	622.61	5074.64	0.12

Table 1. Empirical Results

From the smaller graph instances (*football* and *karate* instances) in table 1, we can see that the branch-and-bound algorithm time described in section 4.1 will find the optimal solution of to any minimum vertex cover problem. However, this can take a long time. For graphs with more than 1000 edges, the time to get to the any solution better than the initial lower bound already exceeds 10 minutes.

The edge deletion algorithm described in section 4.2 is faster than the Branch-and-Bound algorithm. However, if the number of edges is very large, iterating over all edges $e \in E$ and checking whether e should be removed may take a long time. In this case, the run time may be well over 10 minutes, as is the case for the *star* instance. As can be seen in the table above, the solutions provided by the edge-deletion algorithm can have a very large relative error, especially for large instances. The theoretical approximation ratio of the edge-deletion algorithm is 2 corresponding to a relative error of 1. The highest reported relative error in our experiments is 0.81 but the average relative error for the evaluated instances is 0.335 and therefore significantly lower.

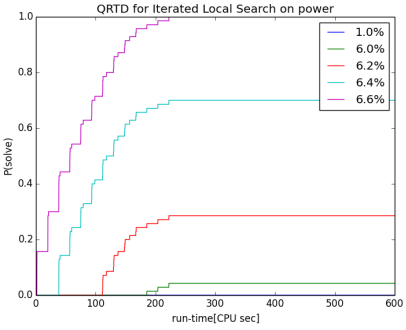
Compared to the edge-deletion algorithm, the local search algorithm provide solutions much closer to the optimal solution than the edge-deletion algorithm. After 10 min of computation, the relative errors of the provided solutions are consistently much lower. For small instances, the algorithms often reach the optimal solution, for larger instances, the resulting relative errors are still comparatively low (<12 %). Overall, the Iterated Local search algorithm performs slightly better on smaller graph instances while the simulated annealing algorithm seems to perform slightly better on larger graph instances. This is due to the structure of the Iterated Local Search algorithm. As we can see, the algorithm takes much longer that the cut off time of 600s. This is due to the fact that for one iteration, the algorithm needs to $2k$ times over all m edges. For the large graph instances, this only allows for one iteration and even that takes much longer.

To analyze the local search algorithms further, we generated QRTD and SQDS plots to visualize the relation between computation time and quality of solution for the *power* and the *star2* instances for each of the local search algorithms. We also generated Boxplots of the computation time needed to get to the best solution found.

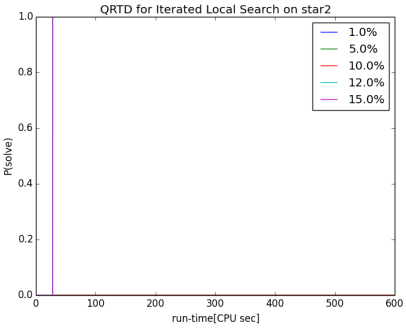
5.1 Iterated Local Search

From Fig. 1, we can see that the Iterated Local Search algorithm produces reasonable results for the *power* instance, in all runs a solution with a relative error of $\leq 6.6\%$ is found, and more than half of all solutions produced have a relative error of $\leq 6.4\%$. We can also see that after 250s of computation no further improvement is achieved, indication that the algorithm has run into a local minimum. For the *star2* instance, the Iterated Local Search algorithm can not find a solution better than the initial vertex cover. This is due to the fact that the initial vertex cover already finds a very

good solution with a relative error of $\leq 15\%$. This is also reflected in the run time distributions in Fig. 3, as the only the solution found is the initial solution found by the greedy algorithm.

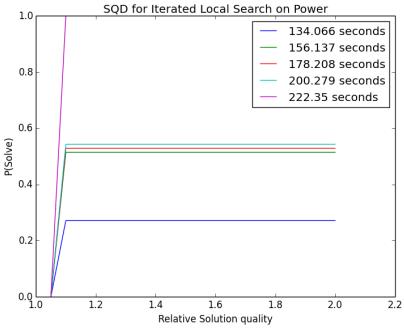


(a) *power*

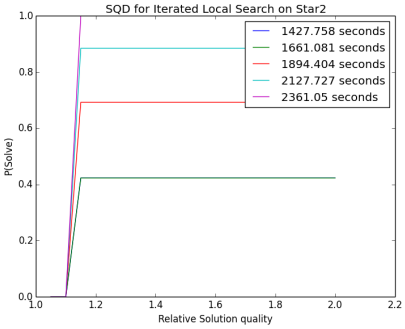


(b) *star2*

Fig. 1. QRTDs Iterated Local Search



(a) *power*



(b) *star2*

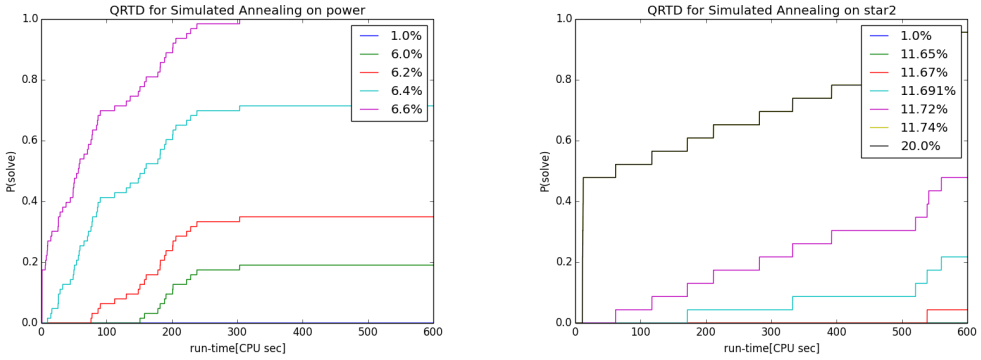
Fig. 2. SQDs Iterated Local Search

(a) *power* (b) *star2*

Fig. 3. Distribution of Runtimes Iterated Local Search

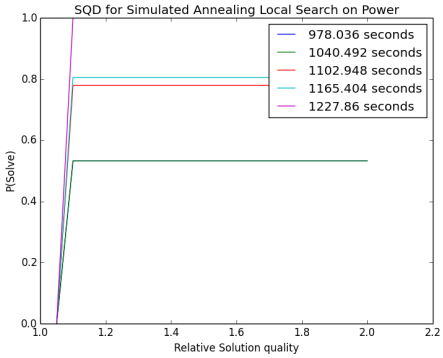
5.2 Simulated Annealing

For the simulated annealing algorithm, the QRTD for the *power* instance in Fig. 4 shows that, similar to the iterated local search algorithm, all produced solutions have a quality of $\leq$ 6.6%. However, compared to the iterated local search algorithm, some of the produced solutions will have a quality of $\leq$ 6.0%. Overall, the simulated annealing algorithm takes a little longer to come to its best solution, around 300s. For the *star2* instance, the simulated annealing approach seems to perform better as it is able to reduce the vertex cover found by the greedy algorithm. Judging from the QRTD graph in Fig. 4, in some of the runs, the algorithm does not converge to a local minimum during the run time of 10 min for the *star2* instance. Here, increasing the run time could lead to better results. This is also supported by the distribution of run times in Fig. 5.2 that shows a wide distribution of computation times for the best solution.

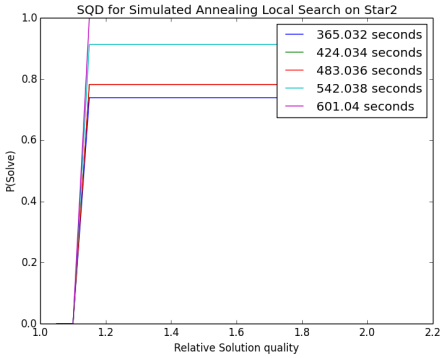


(a) *power* (b) *star2*

Fig. 4. QRTDs Simulated Annealing

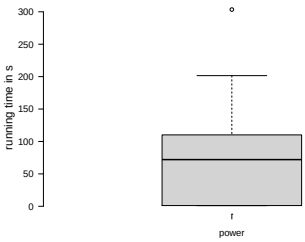


(a) *power*

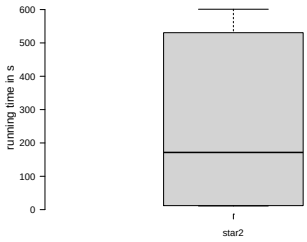


(b) *star2*

Fig. 5. SQDs Simulated Annealing



(a) *power*



(b) *star2*

Fig. 6. Distribution of Runtimes Simulated Annealing

6 DISCUSSION

Overall, the simulated annealing algorithm seems to produce the better solutions for very large graphs. Running this algorithm multiple times with random seeds is a good approach to find a vertex cover with very small relative error. Both local search algorithms perform significantly better that the edge deletion algorithm. However, both local search algorithm seem to converge to a local minimum - unable to find a better solution among the neighbors. Therefore, random restarts of the algorithms during the run time may improve the resulting solutions. However, given that the cut off time for the algorithms is 10 min and both algorithms show improvements in the quality of solutions for the first 5 min of computation, this will only lead to significantly better results if the cut off time is also increased.

7 CONCLUSION

All four algorithms (branch-and-bound, 2-approximation edge-deletion, iterated local search, and simulated annealing) were implemented and result in vertex covers. The branch-and-bound finds the optimal solution for smaller graph instances, but the computation time for larger instances is too long to even find a vertex cover better than the maximum matching. The edge-deletion algorithm is much faster, but has large relative errors and its solutions often come close to double of the true minimum vertex cover. The two implemented local search algorithms have no approximation guarantees but result in much better solutions than the edge deletion algorithm in shorter computation time. Overall, the simulated annealing approach results in slightly better solutions especially for large graph instances. However both algorithms converge to local minima; given a longer cut off time, random restarts could improve the results even further.

REFERENCES

[1] David Avis and Tomokazu Imamura. 2007. A list heuristic for vertex cover. *Operations research letters* 35, 2 (2007), 201–204.

[2] Shaowei Cai. 2015. Balance between Complexity and Quality: Local Search for Minimum Vertex Cover in Massive Graphs.. In *IJCAI*. 747–753.

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

[4] Shaowei Cai, Kaile Su, and Abdul Sattar. 2012. Two new local search strategies for minimum vertex cover. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*.

[5] Jingrong Chen, Lei Kou, and Xiaochuan Cui. 2016. An Approximation Algorithm for the Minimum Vertex Cover Problem. *Procedia Engineering* 137 (2016), 180–185.

[6] Thomas H Cormen. 2009. *Introduction to algorithms*. MIT press.

[7] François Delbot and Christian Laforest. 2010. Analytical and experimental comparison of six algorithms for the vertex cover problem. *Journal of Experimental Algorithmics (JEA)* 15 (2010), 1–4.

[8] Michael R Garey and David S Johnson. 2002. *Computers and intractability*. Vol. 29. wh freeman New York.

[9] Raka Jovanovic and Milan Tuba. 2011. An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem. *Applied Soft Computing* 11, 8 (2011), 5360–5366.

[10] Sami Khuri and Thomas Bäck. 1994. An evolutionary heuristic for the minimum vertex cover problem. In *Genetic Algorithms within the Framework of Evolutionary Computation–Proc. of the KI-94 Workshop*. Saarbrücken, Germany, 86–90.

[11] Ketan Kotecha and Nilesh Gambhava. 2003. A Hybrid Genetic Algorithm for Minimum Vertex Cover Problem.. In *IICAI*. 904–913.

[12] K.V.R. Kumar. 2009. *Choosing the effiecient algorithm for vertex cover problem*. Ph.D. Dissertation.

[13] Joni Pajarinen. 2007. Calculation of typical running time of a branch-and-bound algorithm for the vertex-cover problem. (2007).

[14] Carla Savage. 1982. Depth-first search and the vertex cover problem. *Inform. Process. Lett.* 14, 5 (1982), 233–235.

[15] Shyong Jian Shyu, Peng-Yeng Yin, and Bertrand MT Lin. 2004. An ant colony optimization algorithm for the minimum weight vertex cover problem. *Annals of Operations Research* 131, 1 (2004), 283–304.

[16] Satoshi Taoka and Toshimasa Watanabe. 2012. Performance comparison of approximation algorithms for the minimum weight vertex cover problem. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*. IEEE, 632–635.

[17] Reshu Tyagi and Muskaan Batra. 2016. Implementation and Comparison of Vertex Cover Problem using Various Techniques. *International Journal of Computer Applications* 144, 10 (2016).

[18] Martin Weigt and Alexander K Hartmann. 2001. Typical solution time for a vertex-covering algorithm on finite-connectivity random graphs. *Physical review letters* 86, 8 (2001), 1658.

[19] Xinshun Xu and Jun Ma. 2006. An efficient simulated annealing algorithm for the minimum vertex cover problem. *Neurocomputing* 69, 7 (2006), 913–916.