Approximating Minimum Vertex Cover

Zaymon Foulds-Cook s5017391
School of Information Communication Technology
Griffith University, Gold Coast Campus
Gold Coast, QLD, Australia

***Abstract*—This paper defines a real coded approach to searching for the minimal vertex cover of dense graphs.**

I. Introduction

A. Problem Statement

The vertex cover (vc) is given by:

$$\text{vc} = V\prime \subset V \text{ in graph } G(V,E)$$
$$\text{such that } uv \in E \Rightarrow u \in V\prime \vee v \in V\prime$$

A minimal vertex cover is the smallest possible vertex cover for a graph G.

That is to say the minimal set of vertices where every edge in graph $G(V,E)$ has at least one endpoint to a vertex in the vertex cover. Developing efficient algorithms for the mvc problem has applications in various fields. The SNP assembly problem in computation biochemistry can be resolved by finding the minimum vertex cover in order to resolve conflicts between sequences in a sameple (Pirzada, 2007). Another use for finding the mvc is in computer networking security as a team of computer scientists affiliated with the 'Virology and Cryptology Lab' and the French Navy, 'ESCANSIC' "... have recently used the vertex cover algorithm [6] to simulate the propagation of stealth worms on large computer networks and design optimal strategies for protecting the network against such virus attacks in real-time." (Pirzada, 2007).

The vertex cover problem can be applied to the traveling salesman problem. Vertex cover was also used as the main algorithmic engine in a method of nearest-neighbor data classification applicable in intelligent systems and data analysis (Gottlieb, Kontorovich, & Krauthgamer, 2014).

The mvc problem is classified in the NP-hard class of problems and was proven to be NP-Complete in a landmark paper in 1972 (Karp, 1972). The mvc problem is a classical optimization problem in the field of computer science and finite combinatorics. The minimum vertex cover can also be represented as:

$$\text{all } V \notin \text{Maximum Independent Set of } G'(V,E)$$

II. Literature Review

A multitude of approximation algorithms exist for finding approximate vertex covers in various graphs. One such algorithm is the Djikstra based algorithm proposed by Chen (Chen, Kou, & Cui, 2016). An advantage of this algorithm is the bounding exponential time complexity of $\mathcal{O}(n^3)$ where n is the number of vertices in the graph. A disadvantage to the approach is the approximation ratio is 2. This approximation, while fast, is not effective at finding vertex covers close to the optimal solution.

Another approach which has been investigated is a local search method with greedy heuristics. An approach proposed by Balaji uses heuristics to generate quality solutions for a variety of graphs (Balaji, 2013). The approach when compared experimentally with hBOA and Dual-LP algorithms produces solutions closer to optimality, especially diverging from the comparison algorithms as the size of the graph increases. The algorithm uses guiding heuristics in a greedy manner to determine which vertex to add to the expanding vertex cover. Another greedy heuristic driven approach proposed by Tomar, uses the maximum degree of vertices in the current evaluated neighborhood as the guiding heuristic (Tomar, 2014).

An approach proposed by Pullan leverages a new maximum clique algorithm called Phased Local Search (PLS) (Pullan, 2006) and applies it to the maximum independent set/minimum vertex cover problem. "PLS is a stochastic reactive dynamic local search algorithm that interleaves sub-algorithms which alternate between sequences of iterative improvement" (Pullan, 2009). The algorithm applies a series of sub-algorithms alternating between phases of incremental growth and plateau searching. The incremental growth component will rapidly increase the size of the maximum independent set or vertex cover until there are no more candidates to be added to the set. Then a phase of plateau search uses different methods and operators to swap and evaluate different combinations of nodes with countermeasures to avoid stagnation caused by cycling or nodes which confuse the initial heuristic (Pullan, 2009).

III. Algorithm Description

The algorithm used in this experiment attempts to use phased local search. The search algorithm uses iteritive phases of set building, plateau search and set reduction in an attempt to improve a solution. The algorithm initially utilizes a greedy search to build the initial set where the heuristic is the number of nodes adjacent to a prospective node subtracted by the number of adjacent nodes which are already contained within the cover set. The set implementation uses quick lists which provide methods for find(), insert() and delete in constant time without the overhead of hashing and searching within buckets. This data structure is well suited to the problem and will result in a higher level of performance in comparison with hash based unordered sets. The complexity of generating the degree lists is $\mathcal{O}((|V|-|V\prime|)|E|)$. The degree lists are constructed by iterating through the adjacency lists of nodes not currently contained within the cover set. The score for each node is defined as $H(v) = |\text{adjacent}[v] \cap V\prime|$. The complexity of determining if a current set is a vertex cover is at worst case $\mathcal{O}(|E|)$ as the algorithm iterates through the upper triangular matrix of the adjacency matrix for the graph and if an edge exists where either nodes u or v are not in the current cover set returns false.

A. The algorithm is as follows:

```
def ConstructVCGreedy(Graph g, QuickSet vSet):
  index <- index between 0 .. |V|
  vSet <-+ index

  while vSet is not vertex cover of g:
    degrees <- g.GetDegreeList(vSet)
    sort degrees
    vSet <-+ highestDegreeNode[degrees]


ALGORITHM MVC_Search(Graph g, int swapLimit, int
    ↪ operationsPerPhase, int iterationCap):
  QuickSet vSet <- QuickSet(g.vertices)
  ConstructVCGreedy(g, vSet)

  penalty <- list of -1, length = |V|

  iterations <- 0
  bestCover <- |vSet|

  while iterations < iterationCap:
    for operation < operationsPerPhase:
      depth <- random between 0...10

      // Reduce the number of nodes in the list
      for i in range 0...depth:
        u <- random between 0...|vSet|

        while vSet does not contain u:
          u <- random between 0...|vSet|
        remove u from vSet
        AddToPenalty(penalty, u)
        UpdatePenalty(penalty)

        if vertex cover break
      if vertex cover break

      // Rebuild - Stopping early if we find a better cover
      surface <- 0
      for i in range 0...depth:
        degree <- g.GetDegreeList(vSet)
        sort degree

        index <- 0
        while penalty[degree[index].node]:
          index <- index + 1
          if index <- |degree|:
            index <- random in range 0... |V| - |vSet|
            break

        vSet <-+ degree[index].node
        AddToPenalty(penalty, degree.at(index).second)
        UpdatePenalty(penalty)
          if g is vertex cover break
      if g is vertex cover:
        if (|vSet| < bestCover) bestCover <- |vSet|


      // --- Stabalisation Phase
      if odd iteration:
        v <- random between 0...|vSet|
        vSet.Remove(v)

      swapCount <- 0

      // Get Back to a vertex cover with swaps
      while vSet is not vertex cover of g and swapCount < swapLimit:
        degree <- g.GetDegreeList(vSet)
        sort degree

        index <- 0
        while penalty[degree[index].node]:
          index <- index + 1
          if index <- |degree|:
            index <- random between 0...|degree|
            break
        vSet <-+ degree[index].node

        while true:
          u <- random in range 0...|vSet|
          if penalty.at(u) > 0 continue

        AddToPenalty(penalty, degree.at(index).second)
        UpdatePenalty(penalty)
        swaps <- swaps - 1

      if g is vertex cover:
        if |vSet| < bestCover:
          bestCover <- |vSet|

      if iteration is odd
        degree <- g.GetDegreeList(vSet)
        sort degree
        vSet <-+ one node from top 5 in degree
      iterations <- iterations + 1
    return bestCover

def SearchRunner(Graph g):
  maximumSwaps <- 100
  operations <- 50
  iterations <- 2
  restarts <- 50

  best <- max int

  for i in range 0...restarts:
    score <- MVC_Search(g, maximumSwaps, operations, iterations)
    if score < best:
      best <- score
  return best
```

B. Plateau Search

Examining the pseudocode will reveal the methods used for different phases of the plateau search. The first method is a search which picks a random number n between 1 and 10 and then removes n nodes from the vertex set (providing that they are not currently under penalty). The search then picks nodes from a sorted list based on the nodes degree to repopulate the vertex set (providing the node is not under penalty). After each insertion the algorithm checks to see if the current vertex set results in a new vertex cover and if it does then the insertion loop breaks. The next phase of the plateau search removes an element and then performs a number of swaps to try and restore the property of the vertex set being a vertex cover. If no better cover is found then the loop reinserts a new element and goes back to the first phase of the plateau search. In each of the plateau search methods different selection methods were experimented with. In an attempt to find better solutions different mechanisms of node selection were used during different phases of the plateau search. Both random swaps and insertions were trialed along with swaps and insertions based off penalty and node degree. The parameters for number of swaps, and the number of iterations a node will be placed in the penalty list for were both changed experimentally.

IV. Results

Graph	Vertices	Best MVC Found	Optimal MVC
brock800_1.clq	800	782	777
brock800_2.clq	800	782	776
brock800_3.clq	800	781	775
brock800_4.clq	800	783	774
c2000.9.clq	4000	1941	1922
c4000.5.clq	2000	3988	3982
p_hat1500-1.clq	1500	1491	1488
MANN_a45.clq	1035	704	691

For all graphs the proposed algorithm did not find instances of minimal vertex covers. However, the algorithm found vertex covers within a small margin of the optimal for the graphs brock800_1, brock800_2, brock800_3, brock800_4 c4000.5, p_hat1500-1 and MANN_a45.

V. Conclusion

References

Balaji, s. (2013). A local search based greedy heuristic for minimum vertex cover problem. *Far East Journal of Mathematical Sciences*, *79*(2), 257-266.

Chen, J., Kou, L., & Cui, X. (2016). An approximation algorithm for the minimum vertex cover problem. *Procedia Engineering*, *137*, 180-185. doi: 10.1016/j.proeng.2016.01.248

Gottlieb, L.-A., Kontorovich, A., & Krauthgamer, R. (2014). Efficient classification for metric data. *IEEE Transactions on Information Theory*, *60*(9), 5750-5759. doi: 10.1109/tit.2014.2339840

Karp, R. (1972). Reducibility among combinatorial problems. In R. Miller & J. Thatcher (Eds.), *Complexity of computer computations* (pp. 85–103). Plenum Press.

Pirzada, S. (2007). Applications of graph theory. *PAMM*, *7*(1), 2070013-2070013.

Pullan, W. (2006). Phased local search for the maximum clique problem. *Journal of Combinatorial Optimization*, *12*(3), 303-323. doi: 10.1007/s10878-006-9635-y

Pullan, W. (2009). Optimisation of unweighted/weighted maximum independent sets and minimum vertex covers. *Discrete Optimization*, *6*(2), 214-219. doi: 10.1016/j.disopt.2008.12.001

Tomar, D. (2014). An improved greedy heuristic for unweighted minimum vertex cover. *2014 International Conference on Computational Intelligence and Communication Networks*. doi: 10.1109/cicn.2014.138