CSE6140 Final Project
Team 10

Tompkins, John R
Computer Science
jtompkins8
tompkins.jt@gatech.edu

Chatterjee, Anirban
Computational Science & Engineering
achatterjee36
a.chatterjee@gatech.edu

Xiong, Feng
Civil & Environmental Engineering
fxiong32
Cycling_xf0913@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 DEFINITION

The Minimum Vertex Cover Problem, which can be described as select the minimum number of vertices which can cover all the edges in a graph.

3 RELATED WORK

The structure of our folder can be described as below. We have four different solvers correspond to four different algorithms, and also a template Solver.java file which is used to be inherited to generate other solvers.

And we have a self created Graph.java class which used to better describe the structure of the data we obtain. The Program.java file give us an example about how to run different algorithms for a graph, like toy.graph. And the run.sh can really run all the graphs and generate the result files we need.

4 ALGORITHMS

We have four different algorithms to deal with this Minimum Vertex Cover problem. Below

4.1 Branch-and-Bound

For each vertex in the graph the program assigns one of three values: included into the cover, excluded from the cover, or undecided. A mapping of all vertices to one of these three values is a 'state.' The root node of the search tree is just the state for which all vertices are undecided. From the root node, all possibilities exist. The program keeps track of the 'frontier', a data structure that keeps track of the children of states we have already seen. The frontier is a priority queue, and the first thing to be popped off is the state for which the most amount of edges are covered. The incumbent solution is the best valid vertex cover seen so far during program execution.

The program first pushes the root state onto the frontier. Then it loops until the frontier is empty. In the loop, a state if popped off the frontier. If the state is a valid vertex cover, we check if it is better than the incumbent solution. If so, we update the incumbent to this state. If the state is not a vertex cover, we then check if we can prune its children from the search tree. Otherwise, we look at the first undecided vertex in the state, and push onto the frontier two children: one child for inclusion and one for exclusion.

With regards to pruning, the first check if to see if the number of vertices included in the state's partial cover are greater than the size of the incumbent solution. If so, we can prune by not adding the state's children to the frontier. But we can do better than this. We know how many edges are uncovered in the current state. And we know which vertices may not be included in the state's cover. If we assume that each uncovered edge is only incident to one vertex which is not included in the cover, we can increase the lower bound for the current state. Sort the vertices which are not included in the cover in descending order by their degree. We keep adding the degree of the vertices who are not included into the cover onto the number of edges covered by the current state until the number equals or exceeds the size of the graph. Each time we add the degree of a vertex onto the number of edges covered, we are pretending that it is being included in the cover, so increment the bound by one each time.

There is a second form of pruning done. Suppose that the first prune check did not prune, and we are about to push the two children onto the frontier. For the vertex that we decided to include in the cover, we check all its neighbors. If they are all included in the cover, there is no reason to include it in the cover, because all its edges are already covered. And for the vertex who we are going to exclude from the cover, we look at its neighbors too. If it has one neighbor who is excluded from the cover too, we cannot exclude this one too because we would have an edge whose endpoints are not in the cover. For the children who meet these criteria, we can avoid pushing them onto the frontier to be searched.

4.2 Heuristic Algorithm

A greedy approach has been used to approximate a decent vertex cover. The overall concept of the Greedy algorithm is as follows:

(1) Initialize the vertex cover as an empty set.
(2) Keep randomly choosing an edge whose vertices are not in the vertex cover. Add both vertices to the vertex cover.

Algorithm 2 shows the pseudocode for this algorithm.

4.3 Local Search

The minimal vertex cover problem can be converted to the following decision problem:

CSE6140, Fall 2017, Atlanta, GA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Algorithm 1 Branch-and-Bound Vertex Cover

```
B ← V                                              ▷ Initialize best solution
frontier ← empty priority queue of vertex sets ordered by size
frontier.push(ϕ)                                   ▷ Insert empty set
while frontier is not empty AND time remains do
    C ← frontier.pop()                             ▷ get smallest set in frontier
    if C is a vertex cover then                    ▷ update bound
        if |C| < |B| then
            B ← C
    else                                           ▷ Check bound
        edgesMissing ← number of edges not covered by C
        vertexCount ← |C|
        degrees ← sorted list of degrees of vertices not in C
        for degree in degrees do
            if edgesMissing ≥ 0 then
                vertexCount ← vertexCount + 1
                edgesMissing ← edgesMissing − 1
        if vertexCount ≥ |B| then
            continue                               ▷ branch
        i ← vertex chosen for inclusion
        frontier.push(C ∪ {i})
        e ← vertex chosen for exclusion
        frontier.push(C ∪ {e})
return B
```

Algorithm 2 Greedy Vertex Cover

```
function VC(G(V,E))
    VC ← ∅
    while E ≠ ∅ do
        e ← randomlyPick(E)
        E ← E − {e}
        u, v ← getVertices(e)
        if u ∉ VC AND v ∉ VC then
            VC ← VC ∪ {u, v}
    return VC
```

> Given a graph $G(V, E)$ does there exist a vertex cover of size k?

The smallest value of k that receives the answer *YES* for this decision problem gives us the size of the minimal vertex cover. Using local search methods, we attempt to find a vertex cover for decreasing values of k, starting from $|V|$. The best solution obtained within the time limit is delivered as the proposed solution from this approach.

Two different methods for searching for the solution of the decision problem for a given k, were implemented, as explained below.

4.3.1 Random Search with Tabu Memory. In random search, given a random set of vertices C, we randomly add one vertex to C and remove another vertex from C until a vertex cover is reached. To avoid looping around to the same C, a tabu memory is maintained of the vertices recently removed. They are not allowed back in. The number of edges is covered is tracked throughout the algorithm, making each iteration fast. This algorithm is then employed as many times as possible within the set wall time and the best solution reached by the runs is accepted as the proposed solution of the algorithm. One run of the random search with tabu memory is explained in algorithm 3

The tabu memory is implemented as a queue with a maximum size. As a result, once it is fully populated, adding a new vertex to the tabu memory will remove the oldest vertex from the tabu memory (FIFO). A maximum size of 6 was tested in our implementation. To avoid the case of all remaining vertices getting locked in tabu memory, tabu memory is disabled if the number of remaining vertices is equal to or lesser than the tabu memory size.

Algorithm 3 Random Search with Tabu Memory

```
T ← queue of given maximum size
C ← V                                              ▷ Initialize current solution
while time remains AND |B| > 1 do
    if edges covered by C = |E| then
        B ← C
        i ← random vertex from C
        C ← C − {i}
        T.clear()
    if |V − C| ≤ T.size() then
        T.clear()
    i ← random vertex from C
    j ← random vertex from V − C − T
    C ← C − {i}
    C ← C ∪ {j}
    T.push(i)
return B
```

4.3.2 Hill Climbing. Hill climbing directs the random search explained earlier by moving to a neighboring solution that benefits the current solution the most by adding the most edges that would be covered. The pseudocode is given in algorithm 4.

Algorithm 4 Hill Climbing Vertex Cover

```
C ← V                                              ▷ Initialize current solution
while time remains AND |B| > 1 do
    if edges covered by C = |E| then
        B ← C
        i ← random vertex from C
        C ← C − {i}
    i ← vertex that will cover most edges from C
    j ← vertex that has least degree from V − C
    C ← C − {i}
    C ← C ∪ {j}
return B
```

5 EVALUATION

In order to evaluate the performance of our algorithms, we can use different output files(trace files) produced by our code to generate some meaningful figures and analyze them.

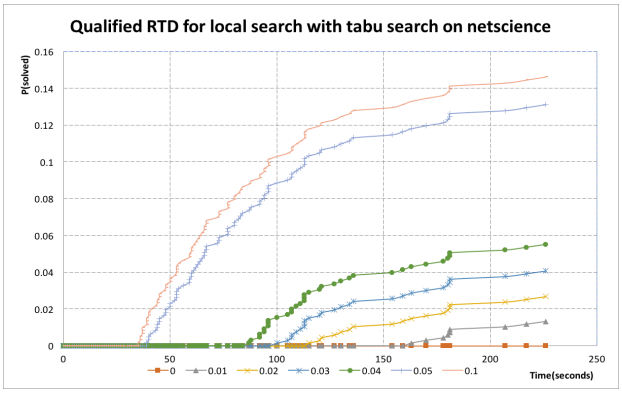

Figure 1: QRTD curve for netscience graph

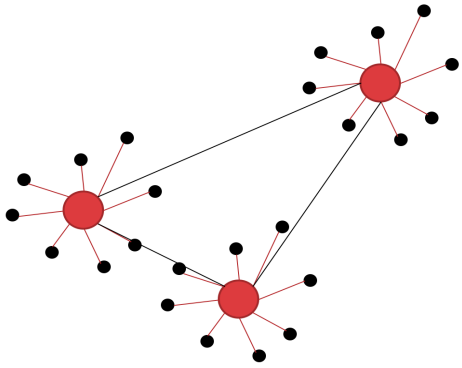

Figure 3: Possible view of the star2 graph

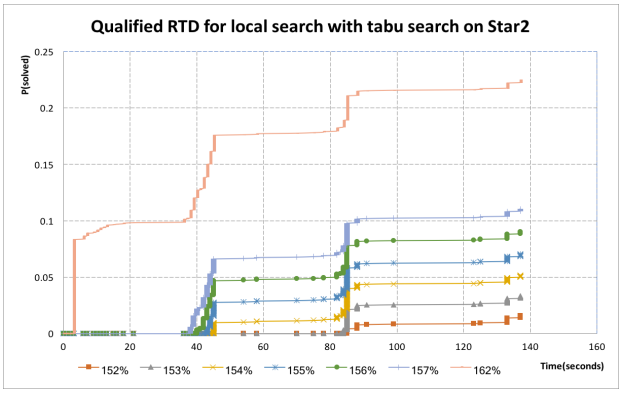

Figure 2: QRTD curve for star2 graph

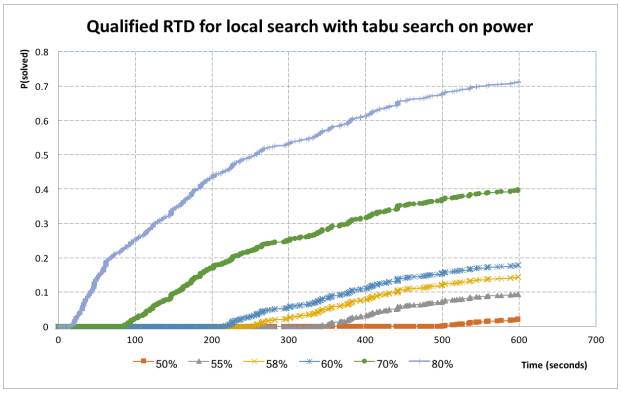

Figure 4: QRTD curve for power graph

In our project, since we have two local search algorithms, but only one algorithm, local search with tabu search can have valid running time series, so all the plots are based on this algorithm, or we can just say after we running our two local search algorithms, we choose to select LS2.

5.1 Qualified Runtime for various solution qualities (QRTDs)

(1) netscience graph as example.

For all of the output files we have, the result of the netscience graph is the best graph which has suitable size and get the number of vertices exactly as the optimal number of vertices, so we use this graph as an example to show how the algorithms work

(2) star2 graph QRTD curve.

We can see that for the star2 graph, the Real Error compare to the Optimal solution is large, which reach like over 150% and with the increase of the running time, the percentage of the problems solved increase in a ladder shape, means there are some vertices which have higher degree compare to others, when these vertices are included, the qualify of the solution can be hugely increased.

Then we can just assume the possible shape of the graph can be viewed as picture below.

(3) power graph QRTD curve.

Then for the power graph, from the QRTD curve we can see that the percentage of the problems solved increase in arcs, kind of like the vertices are distributed relatively uniform, while some verticse may have higher degrees, but not as high as the star2 graph, according to the name of "power" we can assume this graph can be looked like radiation, from a center point expand to the whole space. So the possible shape of the graph can be viewed as picture below.

5.2 Solution Quality Distributions for various run-times (SQDs)

(1) star2 graph SQD curve.

From this plot, we can see that with the increase of the quality, the percentage of the problems be solved can also increase, basically increase in a linear relationship for all the time series, which indicate that no matter what the time is, the quality of the solutions and the percentage of the solutions hold positive correlation, which means higher the

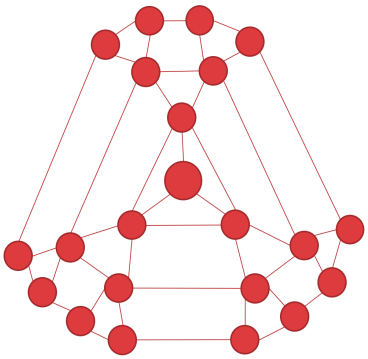

Figure 5: Possible view of the power graph

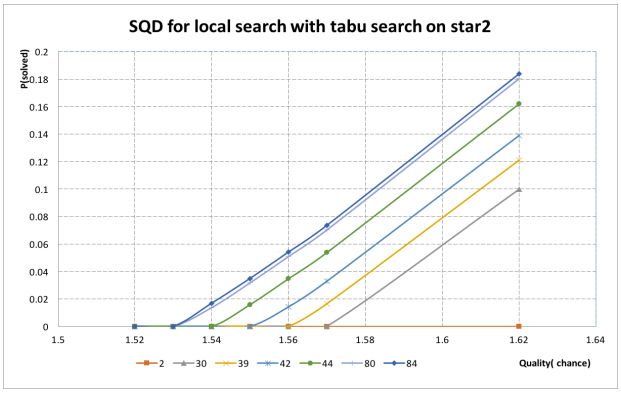

Figure 6: QRTD curve for power graph

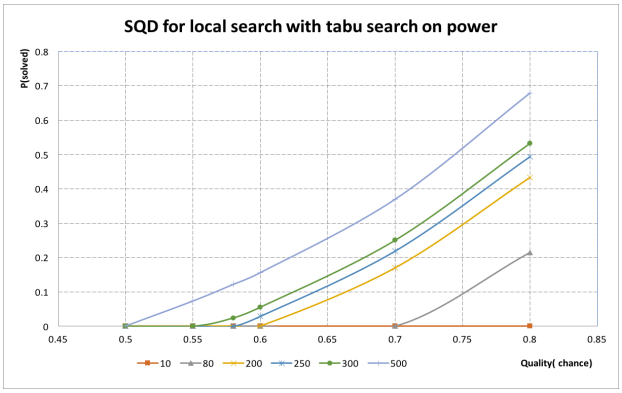

Figure 7: QRTD curve for power graph

qualified runtime is, more problems will be solved.

(2) power graph SQD curve.

Same as the curve for the star2 graph, the SQD curve for the power graph also show the same positive correlation between the quality and the percentage of problems solved. But obviously for the power graph, the solutions are closer to the optimal solution, reflected on the percentage of the problem solved is more close to 1.

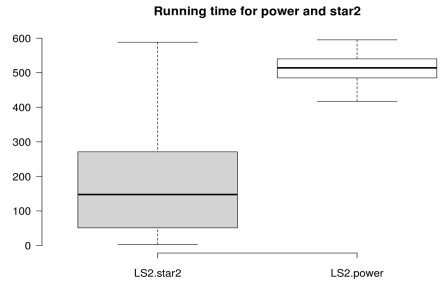

Figure 8: QRTD curve for power graph

5.3 Box plots for running times

Obviously, for different algorithms, we have different running times series which are caused by the random seeds we use. So at this time, we use seeds from 1 to 10 for the star2 and power graph, the distribution of the running times can be showed in the box plot as in Figure8. We can see that for star2 graph, the running time is less than the running time for the power graph, but actually the size of star2.graph is larger than the power.graph, this abnormal phenomenon tell us the running time may have no relationship with the size of the dataset.

What need to be noticed is that in this plot, we assume the quality of the solution for the star2 graph is around 248% and the quality of the solution for the power graph is around 155% because during the cutoff time, it is hard to get more closer to the optimal solution.

6 DISCUSSION

A comprehensive table of the running time and corresponding vertex cover size achieved by each algorithm for each input is given in table 1. Some observations are:

- The heuristic algorithm takes very little time to complete for all graphs thanks to its greedy approach. In our implementation, the initial sorting takes $O(|V|log|V|)$ time. This is followed by a main loop which fires at most $|V|$ times. The complexity of each iteration is proportional to the degree of the vertex, so the total complexity of the main loop is given by $O(|V| + |E|)$. Thus, the total complexity is $O(|E| + |V|log|V|)$.
- Search methods expectedly have mixed results based on the seed and input. For example, for the `football` input, the hill climbing method was able to achieve the best vertex cover in less than a second but took longer than the branch-and-bound approach in some other graphs.

Table 1: Comprehensive table

	Branch-and-bound			Heuristic			Random Search			Hill Climbing		
Dataset	Time (s)	Avg VC	RelErr	Time (s)	Avg VC	RelErr	Time (s)	Avg VC	RelErr	Time (s)	Avg VC	RelErr
as_22july06	0	3429	0.038	0	11981	2.627	2	22565	5.832	599	18643	4.644
delaunay_n10	10	798	0.135	0	809	0.151	0	996	0.417	195	703	0
email	0	668	0.125	0	759	0.278	0	1097	0.847	268	594	0
football	77	101	0.074	0	97	0.032	0	109	0.160	0	94	0
hep-th	25	4042	0.030	0	5252	0.338	0	8328	1.121	592	6474	0.649
jazz	2	165	0.044	0	179	0.133	0	191	0.209	0	158	0
karate	0	15	0.071	0	30	1.143	0	16	0.143	0	14	0
netscience	9	899	0	0	1081	0.202	0	1582	0.760	226	899	0
power	17	2506	0.138	0	3104	0.409	0	4902	1.225	599	3270	0.484
star	8	8396	0.216	0	7643	0.107	0	8200	0.188	596	8095	0.173
star2	17	5481	0.207	0	10970	1.415	1	7384	0.626	137	11404	1.511

- The branch-and-bound approach has the lowest relative error on average (0.098) while the random search has the highest average error (1.048).

7 CONCLUSION

In this project, four approaches for solving the NP-Complete problem of minimum vertex cover were explored: branch-and-bound, a greedy heuristic algorithm, random search with tabu memory and hill climbing. Branch-and-bound gave the lowest average relative error among the algorithms, while the heuristic algorithm gave the fastest speed.

Through this project our team was able to get deeper insight into these approaches for coping with NP-Complete problems and develop a greater appreciation for algorithm design, its challenges, and applications.