Group 5
CSE 6140/CX 4140
Final Project Checkpoint

Algorithm Details

Approximation algorithm

Three different approximation algorithms were implemented and tested to determine which gave the most accurate results for the dataset provided. The first of these algorithms is the classic maximal matching based algorithm. This algorithm was modified to select edges with endpoints of maximum degree. While quite fast and guaranteed to have an approximation ratio of two, the maximal matching edge deletion algorithm performed relatively poorly for the graphs provided. Relative errors for this algorithm were consistently above 0.2. The next approximation algorithm tested operates by running a simple depth-first search on the graph in order to compute a spanning tree. The non-leaf vertices of this tree are returned as vertex cover. This algorithm was slower than the maximal matching algorithm but still quite fast. The accuracy was also comparable to that of maximal matching. Finally, an algorithm based on the Greedy independent cover algorithm analysed by François Delbot and Christian Laforest in the paper *Analytical and Experimental Comparison of Six Algorithms for the Vertex Cover Problem* was implemented. The algorithm iteratively finds vertices of minimum degree then adds all neighboring vertices of this minimum degree vertex to the vertex cover. The independent cover algorithm returned optimal solutions for three of the problem instances and never exceeded 0.017 in relative error. While the two algorithms implemented prior have constant approximation ratios, the independent cover algorithm far out performs them in accuracy at the slight cost of more time. It seems robust for all graphs except ones especially designed to trip it up. Shown below is the pseudocode and evaluation results for this algorithm.

```
procedure approximateMVC:
  Input - Graph g: input graph with vertices V and edges E
  VC = {}
  while (|E| > 0):
      minDegree = |V|
      minVertex = -1
      for each v in V:
         degree = degree(v)
         if (minDegree < degree):
            minVertex = v
            minDegree = degree
      for each u adjacent to minVertex:
         VC = VC ∪ {u}
         E = E \ {all edges incident on u}
  return VC
```

Preliminary Results

	Approximation			**Branch and Bound**		
Dataset	**Time (s)**	**VC Value**	**RelErr**	**Time (s)**	**VC Value**	**RelErr**
as-22july06	13.80	3303	0.00	0	3303	0
delauney_n10	0.0091	715	0.017	12	716	0.0182
email	0.021	597	0.0051	11	595	0.0017
football	0.00036	95	0.011	11	94	0
hep_th	1.11	3930	0.0010	21	3937	0.028
jazz	0.0019	159	0.0063	11	158	0
karate	0.00	14	0.00	0	14	0
netscience	0.032	899	0.00	12	899	0
power	0.38	2204	0.00045	11	2203	0
star	1.19	6946	0.0063	20	7318	0.0568
star2	3.99	4572	0.0066	22	4679	0.0293

Local Search

The Local Search Algorithm explores the space of possible solutions in a sequential fashion, moving in one possible candidate of a solution to a "nearby" solution. We use a cost function to evaluate the quality of the current candidate and move to a neighbor which has a better quality.

Outline for Local Search Algorithm 1
In the first Algorithm we implemented a generic Algorithm where we define a neighborhood of the current candidate and move to one neighboring candidate based on a heuristic.
Given a Graph (V, E)
Candidate (set X): A candidate of the Minimum Vertex cover problem is a subset of the vertices.
X V
Neighborhood Relation: Adding or deleting one vertex from Candidate X.
Choosing the best candidate: If the current candidate X is a Vertex Cover pick an edge(u,v) which has both u,v X and delete the end with the lower degree(u). X = X / {u}. If an uncovered edge exists (i,j), pick the node with the higher degree(i) and add it to X. X= X {i}.
PsuedoCode

```
int Quality(X) //quality of candidate X- return number of vertices in X
    return size of X
 PocedureLS1(Graph G, time cutofftime)
Generate random candidate X.
Set VC-Best solution initialize to null set.
int minVC = INT_MAX
While(cutofftime - running time)
        If candidate X a VC
                If(Quality(X) < minVC )
                    VC = X
                    minVC = Quality(X)
                Pick an edge(u,v) which has both u,v  X
                Select lower degree node (u) and delete it from X. X = X / {u}.
        If (uncovered edge exists)
                Get a random uncovered edge(u,v)
                Get the higher degree node(u) and add it to X. X= X  {u}.
```

Outline for Local Search Algorithm 2

Simulated Annealing.

In the second Local Search Algorithm we plan to implement a Simulated Annealing approach for finding the Vertex Cover. Where we jump to candidates with bad quality if a factor ‘T’ is high.

Given a Graph (V, E)

Candidate (set X): A candidate of the Minimum Vertex cover problem is a subset of the vertices.

X V

Neighborhood Relation: Adding or deleting one vertex from Candidate X.

Define a constant k And Mapping Function T which maps time to a range which is monotonically decreasing.

PsuedoCode

```
int Quality(X) //quality of candidate X- return number of vertices in X
    return size of X
int Cost(X) //number of edges left uncovered by X
    return number of edges left uncovered by X
int T (time t) //maps a time to smoothly decreasing function T Mimicking Temperature
    return T
 PocedureLS1(Graph G, time cutofftime)
Generate random candidate X.
Set VC-Best solution initialize to null set.
int minVC = INT_MAX
While (cutofftime - running time)
    T = T (cutofftime - running time)
            If candidate X a VC
                If(Quality(X) < minVC )
            VC = X
            minVC = Quality(X)
    Let Z be any random candidate which is not X.
    Probability P =
    With probability P
        X = Z
    Let Y be any neighbor of X.
    If( Cost(S) < Cost (X) )
        X = S
```

Branch and Bound:

The search part of algorithm is based:

For one vertex, this vertex will be in the minimum vertex cover set or all its neighbor should be push into the set. During the search, we also choose the vertex with maximum degree to create the branch.

For cutting the branch, points below are used:

1. For edge that degree of its one ends is 1 and the other end's degree is above 1. We can put the end with degree above 1 to our minimum vertex cover set. Since we need to put at least one of its end to minimum vertex cover set, while the one with more than 1 can bring more increase to our objective. This can be done as a preprocess of each search.
2. Lower bound is the half of an approximate result(always choose the one with maximum degree).

Psuedo Code:

```
size_t lowerBoundOfMVC(Graph g) {
   VCTYPE vc
   while(not all edges covered) {
      vc.push(the vertex with biggest degree)
   }
   return vc.size()/2
}

void searchMVC(Graph g, VCTYPE vc, VCTYPE mvc) {
   if(till the cutofftime)
      return

   if(all edges covered) {
      if(vc is better than mvc) {
         mvc = vc
      }
      return
   }
```

```
    // put all neighbors of the vertices with one degree to vc
    screenOutPartOfVertices(g, vc)

    //start to search
    maxDegreeVertexID = g.findVertexWithBiggestDegree(maxDegreeVertexID)

    /*
     * search the route with the vertex with largest degree
     * */
    vc.push(maxDegreeVertexID)
    //store and then delete vertex
    g.removeVertex(maxDegreeVertexID)
    if((lowerBoundOfMVC(g)+vc.size()) < mvc.size()) {
        searchMVC(g, vc, mvc, start, cutoffTime)
    }
    g.recover();
    //recover after search
    vc.pop()

    /*
     * search the route with adding all neighbors of the largest degree vertex
     * */
    g.removeNeighborsOfVertex(maxDegreeVertexID)
    if(lowerBoundOfMVC(g) < mvc.size()) {
        searchMVC(g, vc, mvc, start, cutoffTime)
    }
    //recover after search
    vc.recoverToBeginOfThisCalling()
    g.recover();
}

void branchAndBound(Graph g, VCTYPE mvc) {
    //init process
    if(mvc.size() != 0) {
        cout << "In branch and bound: wrong start position to search" << endl
        return
    }

    stack<pair<size_t, set<size_t>>> verticesDelete
    g.screenOutPartOfVertices(mvc, verticesDelete)

    VCTYPE vc(mvc)
    g.findOneVerticesCover(mvc)

    searchMVC(g, vc, mvc)
}
```