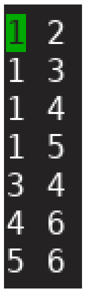
Programming Project
Unweighted Vertex Cover

Joshua Tymburski 001162007

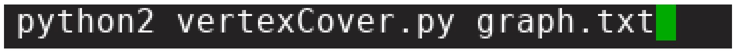
For my project, I chose to utilize the python programming language to program a greedy implementation of the unweighted vertex cover problem. This is backed up via a Linear Programming implementation of the vertex cover problem. Inputs for this program are of the following format:



```
1 2
1 3
1 4
1 5
3 4
4 6
5 6
```

Each line is an edge corresponding between the vertex number in slot one and the vertex number in slot 2. For the sake of consistency, it is requested that these vertices are numbered 1 to n where n is the number of vertices.

The following command instantiates the program where the final argument is the name of the file that is being fed into the program.



```
python2 vertexCover.py graph.txt
```

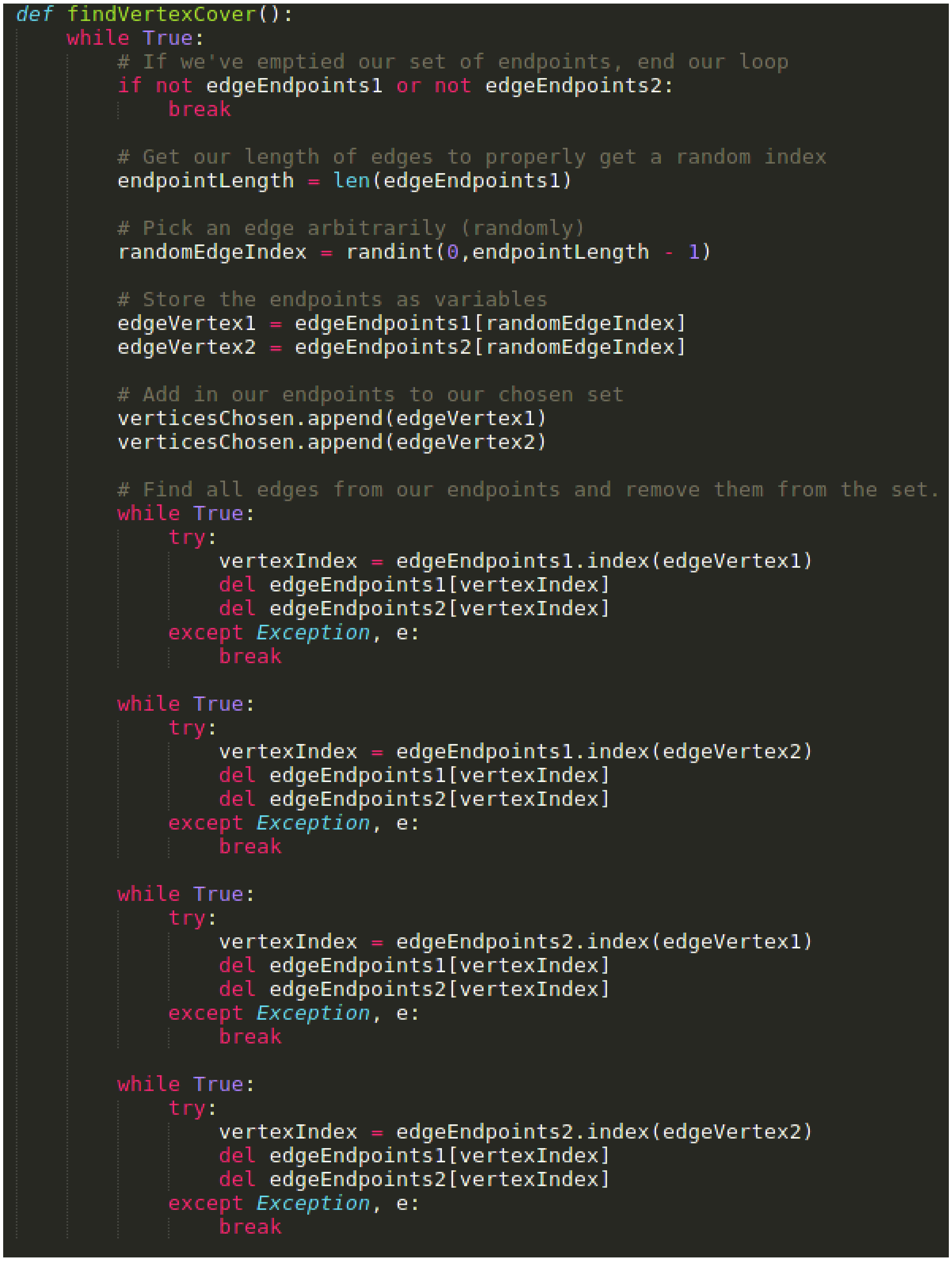
This command was specific for arch linux since the default is python version 3. On other distributions, it may be sufficient to just use the python command. You can check your python version with python –version

The greedy algorithm that was implemented comes as follows:
Let G = (V, E) be a graph with V as the set of vertices and E as the set of edges.

```
while E is not empty:
    pick any edge in E arbitrarily where E = (i, j) where i and j are vertices in V.
    put both i and j into the list of vertices chosen.
    for all vertices u in V where there is an edge (i, u), delete this edge from E
    for all vertices v in V where there is an edge (v, j), delete this edge from E
```

As we saw in class, this is a 2-approximation algorithm.

This is implemented with the following set of code:



```python
def findVertexCover():
    while True:
        # If we've emptied our set of endpoints, end our loop
        if not edgeEndpoints1 or not edgeEndpoints2:
            break

        # Get our length of edges to properly get a random index
        endpointLength = len(edgeEndpoints1)

        # Pick an edge arbitrarily (randomly)
        randomEdgeIndex = randint(0,endpointLength - 1)

        # Store the endpoints as variables
        edgeVertex1 = edgeEndpoints1[randomEdgeIndex]
        edgeVertex2 = edgeEndpoints2[randomEdgeIndex]

        # Add in our endpoints to our chosen set
        verticesChosen.append(edgeVertex1)
        verticesChosen.append(edgeVertex2)

        # Find all edges from our endpoints and remove them from the set.
        while True:
            try:
                vertexIndex = edgeEndpoints1.index(edgeVertex1)
                del edgeEndpoints1[vertexIndex]
                del edgeEndpoints2[vertexIndex]
            except Exception, e:
                break

        while True:
            try:
                vertexIndex = edgeEndpoints1.index(edgeVertex2)
                del edgeEndpoints1[vertexIndex]
                del edgeEndpoints2[vertexIndex]
            except Exception, e:
                break

        while True:
            try:
                vertexIndex = edgeEndpoints2.index(edgeVertex1)
                del edgeEndpoints1[vertexIndex]
                del edgeEndpoints2[vertexIndex]
            except Exception, e:
                break

        while True:
            try:
                vertexIndex = edgeEndpoints2.index(edgeVertex2)
                del edgeEndpoints1[vertexIndex]
                del edgeEndpoints2[vertexIndex]
            except Exception, e:
                break
```

The randint function of python is used to simulate the picking of an edge arbitrarily by choosing a random index of the edgeEndpoints.

This is complimented via the following Linear Program. Recall that our Linear program for unweighted vertex cover is as follows:

$$minimize \sum_{i=1}^{n} x_i$$

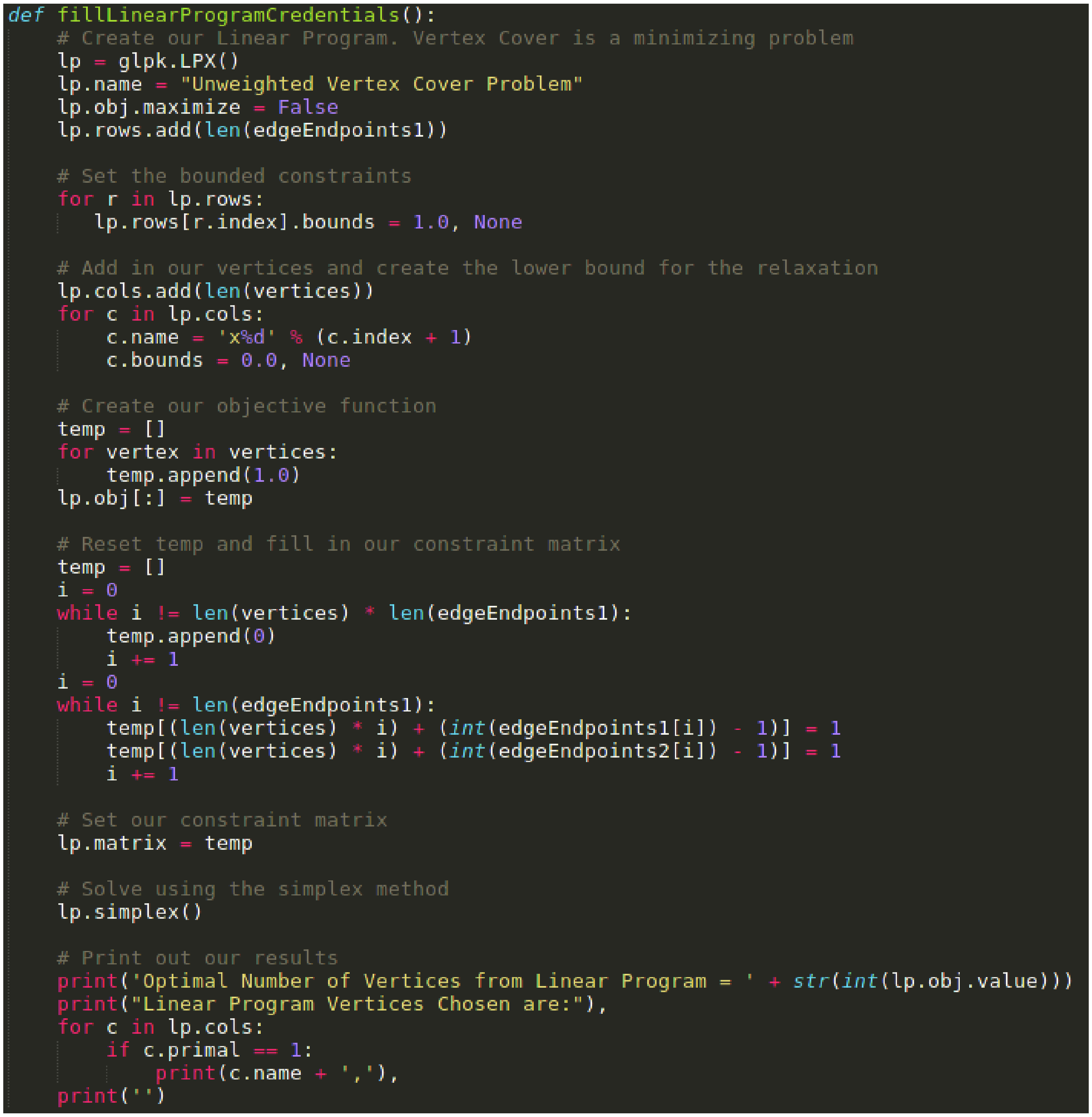
$$\text{subject to constraints} \quad x_i + x_j \geq 1 \ \forall (i, j) \in E$$

$$x_i, x_j \geq 0$$

Using the GNU Linear Programming Kit, we can implement a version of this in Programming Code. Python itself has a specific package called pyglpk which allows for instantiation of this. The link to the source code of said package is here:

https://github.com/bradfordboyle/pyglpk

And the implementation in the code is on the next page:

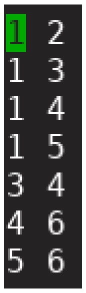


```python
def fillLinearProgramCredentials():
    # Create our Linear Program. Vertex Cover is a minimizing problem
    lp = glpk.LPX()
    lp.name = "Unweighted Vertex Cover Problem"
    lp.obj.maximize = False
    lp.rows.add(len(edgeEndpoints1))

    # Set the bounded constraints
    for r in lp.rows:
       lp.rows[r.index].bounds = 1.0, None

    # Add in our vertices and create the lower bound for the relaxation
    lp.cols.add(len(vertices))
    for c in lp.cols:
        c.name = 'x%d' % (c.index + 1)
        c.bounds = 0.0, None

    # Create our objective function
    temp = []
    for vertex in vertices:
        temp.append(1.0)
    lp.obj[:] = temp

    # Reset temp and fill in our constraint matrix
    temp = []
    i = 0
    while i != len(vertices) * len(edgeEndpoints1):
        temp.append(0)
        i += 1
    i = 0
    while i != len(edgeEndpoints1):
        temp[(len(vertices) * i) + (int(edgeEndpoints1[i]) - 1)] = 1
        temp[(len(vertices) * i) + (int(edgeEndpoints2[i]) - 1)] = 1
        i += 1

    # Set our constraint matrix
    lp.matrix = temp

    # Solve using the simplex method
    lp.simplex()

    # Print out our results
    print('Optimal Number of Vertices from Linear Program = ' + str(int(lp.obj.value)))
    print("Linear Program Vertices Chosen are:"),
    for c in lp.cols:
        if c.primal == 1:
            print(c.name + ','),
    print('')
```

Since linear programs can be represented via matrices, glpk uses this fact to create matrices. The number of rows in the lp is the number of constraints and the number of columns is the number of variables (IE, in this case, the number of vertices). We tell the program that this is a minimization problem, and tell it to solve this with the simplex method. For the sake of clarity, we've named the LP as well.

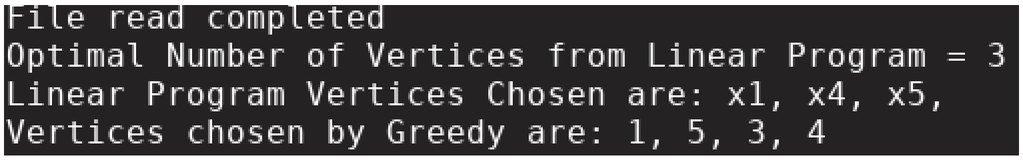
Some samples are shown below:

Example 1
Graph is:

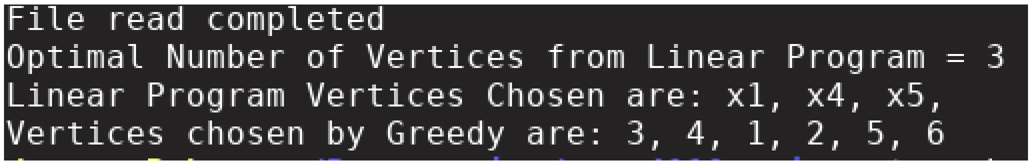


```
1 2
1 3
1 4
1 5
3 4
4 6
5 6
```

Sample outputs are:

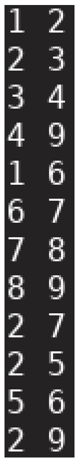


```
File read completed
Optimal Number of Vertices from Linear Program = 3
Linear Program Vertices Chosen are: x1, x4, x5,
Vertices chosen by Greedy are: 1, 5, 3, 4
```



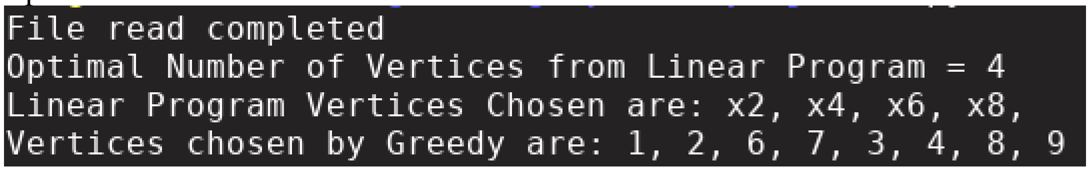
```
File read completed
Optimal Number of Vertices from Linear Program = 3
Linear Program Vertices Chosen are: x1, x4, x5,
Vertices chosen by Greedy are: 3, 4, 1, 2, 5, 6
```

Example 2
Graph is:

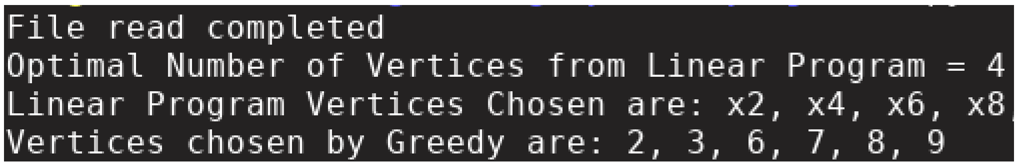


```
1 2
2 3
3 4
4 9
1 6
6 7
7 8
8 9
2 7
2 5
5 6
2 9
```

Sample outputs are:



```
File read completed
Optimal Number of Vertices from Linear Program = 4
Linear Program Vertices Chosen are: x2, x4, x6, x8,
Vertices chosen by Greedy are: 1, 2, 6, 7, 3, 4, 8, 9
```



```
File read completed
Optimal Number of Vertices from Linear Program = 4
Linear Program Vertices Chosen are: x2, x4, x6, x8
Vertices chosen by Greedy are: 2, 3, 6, 7, 8, 9
```

As would be expected, Greedy runs worse than the Linear program in all instances, sometimes up 2 times that of the Linear program which makes sense considering that the algorithm itself is a 2-approximation algorithm.

This can be clearly converted to the weighted vertex problem by creating a file that has the weights for every associated vertex. The linear program is easily converted by adding a w_i before every xi in our objective function.