1. Log in to your Sage Cloud account.
   
   (a) Start Firefox or Chrome browser.
   (b) Go to http://cloud.sagemath.com
   (c) Click “Sign In”.
   (d) Click project Classroom Worksheets.
   (e) Click “New”, call it c34, then click “Sage Worksheet”.

Now we will define a new graph function for computing the hard-to-compute stability number of a graph. The stability number of a graph is the largest number of vertices in the graph that have no edges between them.

So our problem of finding a maximum stable set in a graph can be reduced to the problem of finding a maximum stable set in two smaller subgraphs: (1) the graph formed by removing vertex $v$ and (2) the graph formed by removing $v$ and its neighbors. In this case, we assume that $v$ is in the maximum stable set.

Here’s our function to remove a vertex and its neighbors from a graph and produce a new graph with $v$ and its neighbors removed.

```python
def remove_vertex_and_neighbors(g,v):
    S2=g.vertices()
    S2.remove(v)
    for w in g.neighbors(v):
        S2.remove(w)
    return g.subgraph(S2)
```

2. So now let’s fix up our algorithm. We’ll make $V$ pop a vertex off the end. That won’t effect out original graph. Now we know how to define $S1$ and $S2$ to be what we want.

```python
def tt_maximum_stable_set(g):
    V = g.vertices()
    S1 = V
    v = S1.pop()
    S2 = remove_vertex_and_neighbors(g,v)
    g1 = g.subgraph(S1)
    g2 = g.subgraph(S2)
    Max1 = tt_maximum_stable_set(g1)
    Max2 = tt_maximum_stable_set(g2)
    if len(Max1) > len(Max2):
        return Max1
    else:
        return Max2
```
Evaluate. Let $g=$graphs.PetersenGraph() and then try tt\_maximum\_stable\_set(g). We get a pop from empty list error. Ooooh. Yeah, we didn’t think about that. At some point, after we keep removing vertices, we’ll get an empty list of vertices. At that point we should return that empty list.

```python
def tt\_maximum\_stable\_set(g):
    V = g.vertices()
    if V == []:
        return V
    S1 = V
    S2 = copy(V)
    v = S1.pop()
    S2 = remove\_vertex\_and\_neighbors(g,v)
    g1 = g.subgraph(S1)
    g2 = g.subgraph(S2)
    Max1 = tt\_maximum\_stable\_set(g1)
    Max2 = tt\_maximum\_stable\_set(g2)
    if len(Max1) > len(Max2):
        return Max1
    else:
        return Max2
```

Evaluate. Then try tt\_maximum\_stable\_set(g). What happened? We know that’s not the answer.

3. We forgot that when we remove $v$ and its neighbors that we are assuming $v$ is in a maximum stable set. So somehow we need to keep track of our maximum stable set and include $v$ in that. Let’s let StableSet be the (current) maximum stable set. That’s also what we should return if the vertex set is empty.

```python
def tt\_maximum\_stable\_set(g, StableSet):
    V = g.vertices()
    if V == []:
        return StableSet
    v = V.pop()
    S1 = V
    S2 = remove\_vertex\_and\_neighbors(g,v)
    g1 = g.subgraph(S1)
    g2 = g.subgraph(S2)
    Max1 = tt\_maximum\_stable\_set(g1, StableSet)
    Max2 = tt\_maximum\_stable\_set(g2, StableSet+[v])
    if len(Max1) > len(Max2):
        return Max1
    else:
        return Max2
```

Try tt\_maximum\_stable\_set(g, []). Now test this function with a variety of other graphs where you know the answer. Does it work?

4. Now use the timeit() command to compare the speed of the naive\_maximum\_stable\_set() and tt\_maximum\_stable\_set() functions. Use a variety of graphs for test purposes.