NP-Hard Problems

The purpose of this assignment is to familiarize yourself with different approaches to solving NP-hard problems in practice, especially via integer programming.

As in previous assignments, we will use NetworkX (https://networkx.github.io/) to manipulate graphs and PuLP (http://pythonhosted.org/PuLP/) to solve linear and integer programs.

In [7]:

```
import networkx as nx
import pulp
```

We will use two graph datasets for this assignment, both available in GML (https://en.wikipedia.org/wiki/Graph_Modelling_Language) format. The first is a well-known graph of the social network of a karate club [1]. The second is a network representing the western states power grid in the US [2].

In [8]:

```
karate = nx.read_gml('karate.gml')
power = nx.read_gml('power.gml')
```

[1] W. Zachary, An information flow model for conflict and fission in small groups, Journal of Anthropological Research 33, 452-473 (1977).

[2] D. J. Watts and S. H. Strogatz, Collective dynamics of 'small world' networks, Nature 393, 440-442 (1998).

Independent Set

Recall that an *independent set* in a graph is a set of nodes such that no two nodes are adjacent (connected by an edge). Finding the maximum size of an independent set in a graph is an NP-hard problem, so there is no known polynomial-time algorithm to solve it.

Problem 1

Recall the integer programming formulation for independent set given in class. Implement a function that solves the integer program given a graph as input. (The graph is denoted $G = (V, E)$. For this problem, you should use a binary variable $x_i \in \{0, 1\}$ for each node $i \in N$, and you shouldn't need any other variables.

In [9]:

```
def independent_set_ip(graph):
    """Computes the a maximum independent set of a graph using an integer program.

    Args:
      - graph (nx.Graph): an undirected graph

    Returns:
        (list[(int, int)]) The IP solution as a list of node-value pairs.

    """
    # TODO: implement function
    pass
```

The following code outputs the size of the sets computed by your function.

In [10]:

```
def set_weight(solution):
    """Computes the total weight of the solution of an LP or IP for independent set

    Args:
      - solution (list[int, float]): the LP or IP solution

    Returns:
      (float) Total weight of the solution

    """
    return sum(value for (node, value) in solution)
```

In [11]:

```
karate_ind_set = independent_set_ip(karate)
print "Size of karate set = ", set_weight(karate_ind_set)
power_ind_set = independent_set_ip(power)
print "Size of power set = ", set_weight(power_ind_set)
```

```
Size of karate set =

---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-11-fc0eceaa57df> in <module>()
      1 karate_ind_set = independent_set_ip(karate)
----> 2 print "Size of karate set = ", set_weight(karate_ind_set)
      3 power_ind_set = independent_set_ip(power)
      4 print "Size of power set = ", set_weight(power_ind_set)

<ipython-input-10-84ae48f6a398> in set_weight(solution)
      9
     10     """
---> 11     return sum(value for (node, value) in solution)

TypeError: 'NoneType' object is not iterable
```

Problem 2

Take the *linear programming relaxation* of your integer program and implement a function to solve it. This simply means that in your integer program, you should replace each constraint $x_i \in \{0, 1\}$ with $0 \leq x_i \leq 1$ (see also your slides from 11/19).

In [ ]:

```
def independent_set_lp(graph):
    """Computes the solution to the linear programming relaxation for the
    maximum independent set problem.

    Args:
      - graph (nx.Graph): an undirected graph

    Returns:
        (list[(int, float)]) The LP solution as a list of node-value pairs.

    """
    # TODO: implement function
    pass
```

Let's see how the LP solutions compare to those of the IP.

In [ ]:

```
karate_ind_set_relax = independent_set_lp(karate)
print "Value of karate set = ", set_weight(karate_ind_set_relax)
power_ind_set_relax = independent_set_lp(power)
print "Value of power set = ", set_weight(power_ind_set_relax)
```

A heuristic way to convert a fractional solution to an independent set is as follows. For each node i, include the node in the set if $x_i > 1/2$, and discard it otherwise. This will yield a set of a nodes which have b edges between them. If we remove at most one node for every edge, then we obtain an independent set of size at least $\max\{a - b, 0\}$.

Implement this rounding procedure.

In [ ]:

```
def round_solution(solution, graph):
    """Finds the subgraph corresponding to the rounding of
    a solution to the independent set LP relaxation.

    Args:
      - solution (list[(int, float)]): LP solution
      - graph (nx.Graph): the original graph

    Returns:
        (nx.Graph) The subgraph corresponding to rounded solution

    """
    # TODO: implement function
    pass
```

The following function assesses the quality of the heuristic approach.

In [ ]:

```
def solution_quality(rounded, optimal):
    """Computes the percent optimality of the rounded solution.

    Args:
      - rounded (nx.Graph): the graph obtained from rounded LP solution
      - optimal: size of maximum independent set

    """
    num_nodes = rounded.number_of_nodes() - rounded.number_of_edges()
    return float(num_nodes) / optimal
```

Let's check the quality of this approach compared to the optimal IP solutions.

In [ ]:

```
karate_rounded = round_solution(karate_ind_set_relax, karate)
karate_quality = solution_quality(karate_rounded, set_weight(karate_ind_set))
print "Quality of karate rounded solution = {:.0f}%".format(karate_quality*100)

power_rounded = round_solution(power_ind_set_relax, power)
power_quality = solution_quality(power_rounded, set_weight(power_ind_set))
print "Quality of power rounded solution = {:.0f}%".format(power_quality*100)
```