# Homework 3:

This homework is due by 4pm, November 10, 2015. They will be accepted without penalty until noon, November 11, 2015, after which no credit will be given. Your homework must be turned in as a PDF file and uploaded to the class blackboard page. As a reminder, while you may discuss problems with classmates, you are not allowed to leave such a conversation with any written (typed, etched, photographed, etc.) record of the conversation.

1. A Hamiltonian cycle of a graph G is a simple cycle that visits every vertex exactly once. Suppose someone gives you a function $HamCycle(G)$, which, given an undirected graph G, returns true if G has a Hamiltonian cycle and false if G does not. Use this function to create an algorithm that outputs the sequence of vertices that defines a Hamiltonian cycle, or correctly states that G does not have a Hamiltonian cycle. Your algorithm must make no more than O(E) calls to the HamCycle function, where E is the number of edges in G. If you describe your algorithm using pseudo-code, please also give a short english description of the main idea. Prove that your algorithm gives a valid Hamiltonian Cycle, including on graphs that contain multiple Hamiltonian Cycles.

## Solution 1:

**Description:** Run $HamCycle(G)$ and if false then G does not have a HamCycle. If $G$ does have a Hamiltonian Cycle then we do the following to find an example cycle.
For some vertex $v \in V$ and for an edge $e = (v, u) \in E$: remove $e$ from original graph, $G' = G - e$. Run $HamCycle(G')$. Remove all edges $e$ on $v$ until you return false. If false then the most recent edge is necessary to the Hamiltonian Cycle. Put this $e$ back into $G$. Put $v$ into a list, $H$, of vertices that form the Hamiltonian cycle. Set the endpoint $u$ of the most recent necessary edge equal to $v$, $v = u$. Test all edges until you return to the first explored vertex or until all edges in $G$ have been checked.

**Algorithm:** Here is some psuedo-code for a potential algorithm.

```
function HCVertices($G$)
    $H = []$
    Choose a starting edge $e = (v, u)$
    $next$
    while $G$ has unchecked edges do
        if HamCycle($G$) then
            for all edges on $v$ do
                $e \leftarrow (v, u)$
                $G' \leftarrow G - e$
                if HamCycle($G'$) then
                    $G = G'$
                else                              ▷ There isn't a Ham Cycle anymore keep $e$
                    add $v$ to $H$
                    mark $e$ as checked
                    $next \leftarrow u$
        $v \leftarrow next$                       ▷ The next vertex in the cycle connected to $v$
    return $H$
```

## Solution 1: (cont)

**Proof of Correctness:**

(a) We know that this algorithm will terminate.

- It will return an empty set of vertices if $G$ does not have a Hamiltonian Cycle.
- Otherwise it will return the vertices in order of a valid Hamiltonian Cycle.

(b) We know the algorithm will return the vertices in the correct order.

- After each necessary edge $e$ is found the next edges checked are connected to $e$.
- There must be a connecting edge among those searched because a Hamiltonion cycle is by definition a closed loop.
- Once all edges are checked, we return $H$, which is a list of connected vertices that results in a cycle that visits every vertex once. All unnecessary edges will not contribute to the order of vertices returned because unnecessary edges to the current Ham Cycle will be ignored, even if they are part of another Ham Cycle. They will be removed and HamCycle will return true because the edges for the returned Ham Cycle will be present.

2. Fair 3-coloring: Given a graph $G = (V, E)$, where $|V|$ is a multiple of 3, the Fair 3-coloring problem asks if G can be 3-colored so that there are the same number of nodes of each color ( $|V|/3$ ). Prove that Fair 3-coloring is NP-complete. You can prove this with a reduction from any NP-complete problem, including the standard 3-coloring problem.

## Review: Proving Similar Hardness

: if you can show that one problem reduces to another problem (with some time constraint) and vice versa (with the same time constraint) then those two problems are equally hard to solve.

## Review: Proving NP-Completeness

**Definition of NP-Complete:** A problem $U \in NP - Complete$ if:

- $U \in NP$
- For all $H \in NP$, $H \leq_p U$

**Proving NP-Completeness:** Based on Cook's Theorem we know that any problem in NP can be solved by SAT and that SAT is in NP. From there we have developed a longer list of NP-Complete problems. That's why we don't have to prove both directions of the reduction.

- In order to prove that a new problem, $U$, is NP-Complete we first show that it is in NP. If in NP we know that it reduces to any know NP-Complete problem.
- Then we show a reduction from $H$ (some known NP-Complete Problem) to $U$. Because we know $H$ is NP-Complete it is reducible from any problem $P \in NP$ thus $\forall_{P \in NP} P \leq_p H \leq_p U$.

## Prove $U \in$ NP:

- We suspect this problem is hard and can't be solved in polynomial time.
- To show that it is at least in NP we need to show that we can verify some input to the problem in polynomial time. Thus we design an algorithm which can verify in polynomial time whether an instance is a solution to the problem or not.

## Prove for all $H \in NP$, $H \leq_p U$

**H $\leq_p$ U:** Reduce from known hard problem $H$ to unknown problem $U$. If $H$ reduces to $U$, it means we can use $U$ to solve $H$. When we prove this reduction we show that $H$ cannot be harder than $U$ because a solution to $U$ provides a solution to $H$. Thus we prove $U$ is just as hard as $H$, which we know is NP-Hard. This process is outlined below:

(a) Assume for contradiction that the problem, $U$ (the one you're trying to prove is NP-Hard), is actually solvable in polynomial time.
(b) Show that some known NP-Complete problem, $H$, can be solved in polynomial time using $U$ as a subroutine.
(c) Get a contradiction to your original assumption that $U$ is solvable in polynomial time because we know that $H$ is not solvable in polynomial time.
(d) Thus $U$ is also NP-Hard

## In Practice

The steps we have done in class are acceptable shorthand for the above formal proof structure. Those steps are:

(a) Prove $U \in$ NP: Let C be a certificate that describes an answer to an arbitrary instance problem U. Argue that you can verify in polynomial time whether or not C is a valid answer to that instance.

(b) Prove U is NP-HARD: Pick some problem H that is known to be NP-Complete. Show how to reduce an instance of problem H to be an instance of problem U. Prove (a) your reduction runs in polynomial time, and (b) the answer for any input to the original problem H is the same as the answer when you ask U about the transformed input.

## Solution 2:

**Reduction: 3-Color $\leq_p$ Fair3-Color** (reduce from known NP-Complete to unknown)

(a) Assume, for contradiction, that FAIR3-COLOR is solvable in polynomial time.

  i. Assumptions are like magic boxes.

  ii. Basically we now can say in polynomial time if any graph with $|V| \mod 3 = 0$, is Fair3-colorable or not.

(b) Show that 3-COLOR can be solved in polynomial time using FAIR3-COLOR as a subroutine given any graph $G$.

  i. Any given graph might have $|V| \mod 3 > 0$, in that case it wouldn't be solvable by FAIR3-COLOR but still could be 3-Colorable. Thus we have to transform $G$ in polynomial time such that we can use FAIR3-COLOR on the new graph $G'$ to prove the original graph $G$ is 3-Colorable. It is important this transformation is in polynomial time because we're trying to show that a polynomial solution exists for 3-COLOR when we assume one exists for FAIR3-COLOR.

  **Transformation:** Given a graph $G$ transform it in polynomial time such that FAIR3-COLOR(G') proves 3-COLOR(G)

  - For each vertex $v$ in $G$ add 2 new vertices to and connect them to each other and to $v$.

  **Proof of Correctness:**

  - FAIR3-COLOR(G') $\rightarrow$ 3-COLOR(G)
    - $G' \subset G$ thus by definition it must be 3-Colorable if it is Fair 3-Colorable
  - 3-COLOR(G) $\rightarrow$ FAIR3-COLOR(G')
    - If $G$ is 3-Colorable then for each vertex $v$ in $G$ is able to be colored with one color while the 2 vertices added in $G'$ represent the 2 other colors.
    - Thus every node in G has a corresponding triplet of nodes in G'. Each of these 3 nodes have a different color, so G' has n-nodes of each color.

3. A circular communication network consists of n nodes on a cycle, numbered 0 through $n-1$ clockwise around the cycle. This problem considers routing a collection of calls through a circular connection. A set C of calls is given; each call is a pair $(i,j)$ originating at node i and destined to node j. That call has a cost $c(i,j)$ which is the load of that call (e.g. how much data that call requires). The call can be routed either clockwise or counterclockwise around the cycle.

   The objective is to route the calls so as to minimize the maximum load on any link in the network. We define the load $L_i$ on link $i$ to be the cost of all the calls routed through edge $(i, i+1)$ (wrapping around to 0 if i+1=n).

   (a) Give a 2-approximation algorithm for this problem. If it helps, you may assume that all call costs c(i,j) are 1 (but note that it is not necessary that there is a call between every pair of nodes i,j).

   (b) **Required for 541 students, bonus for 441 students**. Show that this problem is NP-complete (Hint: you may, but are not required, to try to reduce from Partition). I believe this is easier if you include the call costs as part of the problem.

## Solution 3a:

**Description:** Route the calls greedily by shortest path.

**Algorithm:** Here is some psuedo-code for a potential algorithm.

```
function CircRouting($G, C$)            ▷ $G \leftarrow V = n, E =$ connect all v in circle
    while $C$ is not empty do
        Remove some $c(i, j)$ from $C$
        Route $c$ through ShortestPath($(i, j)$)
    return highest cost edge $e$
```

**Proof of Correctness:** The algorithm will return a routing of all calls. The cost of that routing is defined by the edge which has the highest cost routed through it. Let us name this edge "L" and also use L to define the cost of the calls routed through it.
We now prove that L $\leq$ 2H*, where H* is the cost of the max load edge in the optimal routing.

- Define M to be the edge exactly opposite L. In our approximation, we know that all calls through L will not go through some edge M.
  - Our algorithm routes based on the shortest path from some $i$ to some $j$. The path that includes L and M is longer than half the circle. Our algorithm wouldn't have routed the call that way because there exists a shorter path.
  - This is true for both and even and odd number of vertices. In the case of an odd number you can pick either edge across from L as M.
- The optimal solution may route some calls over these special edges L, M. Let's define the cost of calls that OPT routes on these edges L*, M*.
  All the calls we routed through L in the approximation, go from one side of L to the other side of L (otherwise, the shortest path wouldn't cross L). Therefore, OPT must route those calls from one side to the other. Perhaps OPT routes some of those call through M, but all the calls must be routed through either L or M.
  Therefore L* + M* $\geq$ L.
  (this is $\geq$ because OPT may route other calls through these edges as well).
- Thus L* or M* has to be at least equal to half of L. Meaning some edge in the optimal solution is at least as big at $\frac{L}{2}$.
- If some edge in the OPT solution has cost at least $\frac{L}{2}$, then the max edge has cost $\frac{L}{2}$, so the cost of the optimal solution H* $\geq \frac{L}{2}$. We can rewrite this as L $\leq 2H*$ and we have proven our 2-Approximation.

## Solution 3b:

**Reduction: Partition $\leq_p$ CircRouting** (reduce from known NP-Complete to unknown)

**Transformation:** Given a list of $n$ integers $N$, where $S$ is the sum over all the integers, transform the input in polynomial time such that CircRouting(N') proves Partition(N)

- Create a circular routing with 4 nodes, $s, a, t, b$, in order around the circle. So there are 2 paths from $(s, t)$. There are $n$ calls $C$ between $s$ and $t$. For each call, it has a cost $c(s, t)$ that corresponds to the value of each integer in $N$.
- If you can achieve a maximum edge load of $\frac{S}{2}$ in the circular routing of these calls then you can prove there is a partition Partition(N)
-

**Proof of Correctness:** The algorithm will return highest cost edge L such that L $\leq$ 2H*, the cost of the max load edge in the optimal routing.

- CircRouting(N') $\rightarrow$ Partition(N)
  - If a max load of $\frac{S}{2}$ is possible for CircRouting(N') with call costs corresponding to the $n$ numbers in Partition(N) then the load of the second path in CircRouting(N') must also be $\frac{S}{2}$.
  - The calls can be split to create a max load of $\frac{S}{2}$, thus the integers can be partitioned into 2 sets with sum equal to $\frac{S}{2}$.
- Partition(N) $\rightarrow$ CircRouting(N')
  - If you can partition the integers in $N$ into 2 sets with sum $= \frac{S}{2}$. Then the calls of the circular routing problem can be split between the two paths of the problem in the same way giving a max load of $\frac{S}{2}$.