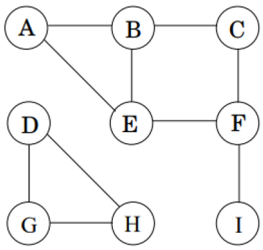
CSCI 2300 — Introduction to Algorithms
Homework 1 (document version 1.1) — DUE January 24, 2020
Recurrence Relations and Breadth First Search (BFS)

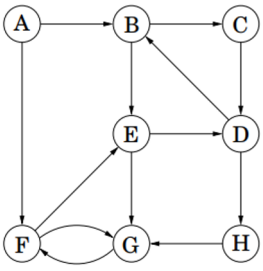
- This homework is due by 11:59PM on the due date above and must be submitted in Submitty as a single PDF called `upload.pdf`.
- Just like the labs, please avoid using Google to find suggestions or solutions. You are allowed to use the textbooks and class notes as references. We strongly encourage you to use the office hours, labs/recitations, and Discussion Forum to successfully complete this assignment.
- Also, you are allowed to consult with your classmates, collaborating in line with the policies set forth in the syllabus. You must write up your own separate solution; do not copy-and-paste anything.
- Remember the goal is to use your own brain to work these problems out, which will help you develop the skills to do well on the exams and, more importantly, become a substantially better computer scientist.
- For your solutions and proofs, please write clearly and concisely, and use rigorous formal arguments and pseudocode that mimics the style used in the Dasgupta textbook. Be consistent.
- In general, if you are asked to provide an algorithm, the best thing to do is to provide a clear description in English (e.g., "Use BFS, but with the following change..." or "Run BFS two times as follows...").
- Be as concise as you can in your answers. Long answers are difficult to grade.

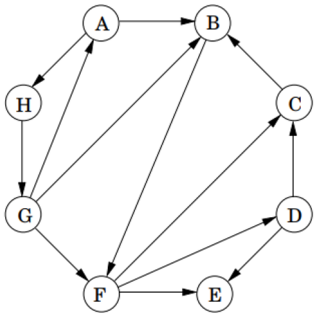
Warm-Up Problems (do not hand in for grading)

Work out the problems below as practice problems before going on to the next section. Note that warm-up problems might sometimes show up on an exam!

1. DPV Problem 0.1 (as many as you can)
2. DPV Problem 0.3 (Fibonacci follow-up)
3. Walk through the BFS algorithm starting from node A in the graphs below. Repeat but start from node G.







Required Problems (hand in for grading)

Work out the problems below and submit for grading before the deadline.

1. Consider the following pseudocode of a function that takes integer $n \geq 0$ as input.

```
function umeme(n):
  print '*'
  if n == 0: return
  for i = 0 to n - 1:
    umeme(i)
  return
```

Let $T(n)$ be the number of times the above function prints a star (`'*'`) when called with valid input $n \geq 0$. What is $T(n)$ exactly, in terms of n only (i.e., remove any reference of $T()$ on the right-hand side). Prove your statement.

SOLUTIONS AND GRADING GUIDELINES:
$\implies$ 3 points for determining $T(n)$
$\implies$ 3 points for proof

Observe the pattern of `umeme()` calls to identify that the number of stars printed will be one initial star plus $n-1$ recursive calls, which we can write as $T(n) = 1 + \sum_{i=0}^{n-1} T(i)$, with $T(0) = 1$.

This sum is a known sum that we can rewrite as $T(n) = 1 + (2^n - 1)$, or just $T(n) = 2^n$.

Alternatively, expanding this out a few terms, observe that each subsequent term expands out to twice as many stars as the previous term. We can write this as $T(n) = T(n-1) \times 2$, which simplifies to $T(n) = 2^n$.

Given either of the above, we propose and must prove that $T(n) = 2^n$. We can use induction (or more specifically, strong induction) to prove our claim.

Base case. For $n = 0$, $T(0) = 2^0 = 1$.

Induction step. Assume that for all $k < n$, $T(k) = 1 + \sum_{i=0}^{k-1} = 2^k$ is True.
We must prove $T(k) \to T(k+1)$.
Starting with $1 + \sum_{i=0}^{k-1} T(i) = 2^k$,
multiply both sides by 2 to obtain $2 \times (1 + \sum_{i=0}^{k-1} T(i)) = 2 \times 2^k$,
which simplifies via the inductive hypothesis to $2 \times (2^k) = 2 \times (2^k)$ or simply $2^{k+1} = 2^{k+1}$.

Using strong induction, we have thereby shown that $T(n) = 1 + \sum_{i=0}^{n-1} = 2^n$ is True for all $n \geq 0$.

Be careful: proving $T(k+1) \to T(k)$ is incorrect; in other words, do not start with $T(k+1)$.

2. Repeat the previous problem with the pseudocode below.

```
function memeo(n):
  if n == 0: return
  for i = 0 to n - 1:
    print '*'
  memeo(n - 1)
  return
```

SOLUTIONS AND GRADING GUIDELINES:
$\Longrightarrow$ 3 points for determining $T(n)$
$\Longrightarrow$ 3 points for proof

Observe the pattern of `memeo()` calls to identify that the number of stars printed will be $T(n) = n + T(n-1)$. Expanding this out a few terms, the pattern emerges as simply the sum of integers from 1 to n. More specifically, $T(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$. Proof by induction could be done here, but stating that this is proven by definition is fine.

3. DPV Problem 3.6(a-c) – **(v1.1)** For part (c), if yes, prove your answer; if no, provide a counter-example (i.e., prove by contradiction).

SOLUTIONS AND GRADING GUIDELINES:
$\Longrightarrow$ 3 points each for parts (a), (b), and (c)

(a) Each edge (u, v) contributes exactly 1 to $d(u)$ and 1 to $d(v)$. Therefore, each edge contributes exactly 2 to the sum $\sum_{u \in V} d(u)$, which gives $\sum_{u \in V} d(u) = 2|E|$.

(b) Let V_o be the set of nodes with odd degree and V_e be the set of nodes with even degree. Then we have:

$$\sum_{v \in V_o} d(v) + \sum_{v \in V_e} = 2|E|$$

We can rewrite this as:

$$\sum_{v \in V_o} d(v) = 2|E| - \sum_{v \in V_e}$$

The RHS of this second equation is even since both terms are even and subtracting two even numbers yields an even number. The LHS is a sum of odd numbers, which can only be even if it is the sum of an *even number of odd numbers*. Therefore, the number of nodes in V_o (i.e., $|V_o|$) must be even.

(c) No. As a counter-example, consider directed graph $G = (V, E)$ with $V = A, B$ and edge $e = (A, B) \in E$ (i.e., one directed edge from nodes A to nodes B). In G, only nodes B has odd indegree.

4. Given two connected undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, with $V_1 \cap V_2 = \emptyset$, and nodes $s \in V_1$ and $t \in V_2$. Also given set of edges E' that you are to build with the constraint that each edge in E' has one endpoint in V_1 and one endpoint in V_2. Therefore, adding any edge of E' would connect graph G_1 with G_2; it would also form paths connecting nodes s and t.

   Design a linear-time algorithm to determine an edge $e \in E'$ whose addition to graphs G_1 and G_2 would result in the shortest possible distance (i.e., the smallest number of edges) between s and t among all edges in E'. **HINT:** Make use of BFS as a subroutine to help solve this problem.

   Remember that linear-time here means linear in the size of the input, which in this case is the number of nodes plus the number of edges in graphs G_1 and G_2, plus the number of edges in E'. More concisely, this is $|V_1| + |V_2| + |E_1| + |E_2| + |E'|$.

   SOLUTIONS AND GRADING GUIDELINES:
   $\Longrightarrow$ 6 points for a correct algorithm
   $\Longrightarrow$ if correct, 3 points for linear-time algorithm

   First note that adding all edges in E' at the same time and then running BFS on the resulting graph would not work, as the shortest path you find might include more than one edge of E'.

   A solution using BFS is as follows. Run BFS from s, then run BFS from t. For every node $x \in V_1$, we have distance $d_s[x]$ from s to x in G_1. Similarly, for every node $y \in V_2$, we have distance $d_t[y]$ from t to y in G_2.

   Any path from s to t after the addition of some edge $e = (u, v) \in E'$ must be the concatenation of a path from s to u in G_1, followed by edge e, followed by a path from v to t in G_2. Since this is an undirected graph, the length of the shortest path is simply $d_s[u] + 1 + d_t[v]$.

   Running BFS takes $O(|V_1| + |E_1|)$ time for G_1 and $O(|V_2| + |E_2|)$ time for G_2. Computing the length of the shortest $s - t$ path after adding edge e requires only constant time for each edge in E'; therefore, the time required is $O(|E'|)$. All of the above is linear time and is additive.

5. DPV Problem 3.7(a) – Assume graph G is connected. Use BFS to solve this problem.

SOLUTIONS AND GRADING GUIDELINES:
$\Longrightarrow$ 6 points for a correct solution

First we annotate sets V_1 and V_2 with two distinct labels or colors, e.g., nodes in set V_1 will be colored red, while nodes in set V_2 will be colored blue. (Other coloring/labeling schemes here are fine, as long as they alternate; further, using a distance and identifying even versus odd distances should also work.)

Next, we perform BFS on graph G from arbitrary node s and color alternate levels of the BFS tree as red and blue. The graph is bipartite if and only if there is no monochromatic edge, i.e., an edge with both endpoint nodes having the same color (or same label).

To check for this, note that all edges in G must have endpoint nodes either in the same layer or in adjacent layers of the BFS tree. Therefore, the property of a monochromatic edge can be checked during the BFS itself as such an edge must be an edge to an already visited node with the same distance from s as the current node.

To prove this (though not required), both implications of "if and only if" must be proven (i.e., $A \rightarrow B$ and $B \rightarrow A$). First, suppose that there is no monochromatic edge, i.e., all edges in G have one red endpoint and one blue endpoint. Then, by definition, graph G is bipartite. Next, for the other direction, suppose G is bipartite. Then the first layer of the BFS tree must be entirely in V_1, the second layer entirely in V_2, the third entirely in V_1, etc. Thus the above alternating process would result in no monochromatic edges.