*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology — March 23, 2018
Instructors: Srini Devadas, Jason Ku, and Nir Shavit — Problem Set 6

# Problem Set 6

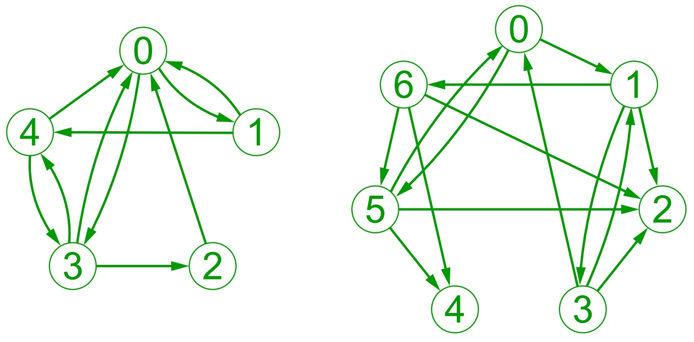
**All parts are due on April 5, 2018 at 11PM**. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `py.mit.edu/6.006`.

**Problem 6-1.** [30 points] **Graph Practice** A graph $G = (V, E)$ has two common representations. An *adjacency matrix* is a $|V| \times |V|$ matrix of boolean values, where the matrix element in row $i$ and column $j$ is 1 if edge $(v_i, v_j)$ is in $E$, and 0 otherwise. An *adjacency list* is a $|V|$ length list of lists, where the $i$th list contains index $j$ if edge $(v_i, v_j)$ is in $E$.

**(a)** [8 points] Draw the **directed graph** associated with each of the following graph representations. $G_1$ is represented by an adjacency matrix, while $G_2$ is represented as an adjacency list. For consistency, please draw your vertices to lie roughly on a circle ordered clockwise by index.

```
G1 = [[0 1 0 1 0],        G2 = [[1,5],
      [1 0 0 0 1],              [2,3,6],
      [1 0 0 0 0],              [],
      [1 0 1 0 1],              [0,1,2],
      [1 0 0 1 0]]              [],
                                [0,2,4],
                                [2,4,5]]
```

**Solution:**

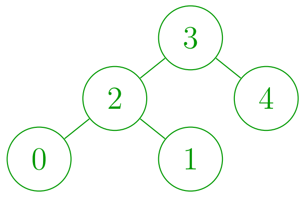


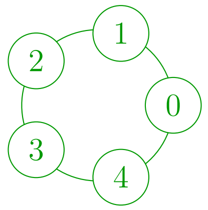
**Rubric:**

- 4 points for each correct drawing
- -1 point for each incorrect edge (max -4 for each graph)

**(b)** [12 points] Draw two different **connected** undirected graphs on the set of five vertices, $V = \{v_0, v_1, v_2, v_3, v_4\}$. One graph should be a tree and the other should contain exactly one cycle. Write down adjacency matrix **and** an adjacency list representations for each of your graphs.

**Solution:**

Tree: 

Cycle: 

```
M_1 = [[0 0 1 0 0],
       [0 0 1 0 0],
       [1 1 0 1 0],
       [0 0 1 0 1],
       [0 0 0 1 0]]

A_1 = [[2],
       [2],
       [0, 1, 3],
       [2, 4],
       [3]]
```

```
M_2 = [[0 1 0 0 1],
       [1 0 1 0 0],
       [0 1 0 1 0],
       [0 0 1 0 1],
       [1 0 0 1 0]]

A_2 = [[1, 4],
       [0, 2],
       [1, 3],
       [2, 4],
       [3, 0]]
```

**Rubric:**

- 2 points for each correct graph drawing (tree and with cycle)
- 2 points for correct corresponding adjacency matrix (should be symmetric)
- 2 points for correct corresponding adjacency list

**(c)** [10 points] The **separation between two vertices** in an undirected graph is the length of the shortest path between them. The **separation of an undirected graph** is the maximum separation between any two vertices in the graph. The **maximum separation of a vertex** in an undirected graph is the maximum separation between the vertex and any other vertex in the graph. A vertex is a **centroid** of an undirected graph if its maximum separation is minimum across all vertices. Let $G$ be a **connected** undirected graph, with $d$ the separation of $G$ and $c$ the maximum separation of a centroid of $G$. Prove that $c \leq d \leq 2c$.

**Solution:** Clearly since $d$ is the maximum separation between any two vertices, and the centroid is a vertex minimizing its maximum separation, then $c \leq d$. To show $d \leq 2c$, let $v$ be a centroid of $G$. For any pair of vertices $a$ and $b$, there is a path of length at least $c$ from $a$ to $v$, and a path of length at least $c$ from $v$ to $b$. Thus there exists a path from $a$ to $b$ of length at least $2c$, so the separation between $a$ and $b$ cannot be larger or else the path from $a$ to $v$ to $b$ would be shorter. So every pair of vertices has a path between them with length at least $2c$, so the minimum path length between any pair ($d$) cannot be greater than this number.

**Rubric:**

- 3 points for a correct argument for $c \leq d$
- 7 points for a correct argument for $d \leq 2c$
- Partial credit may be awarded

**Problem 6-2.** [30 points] **Consulting**

**(a)** [5 points] **Security Breach:** SicroMoft, a software company, has experienced a security breach in its company email system. An employee's computer has been infected with a computer virus, that copies itself to another computer if an email is sent to it from the infected computer. Security analysts have identified the computer that was first infected with the virus, and has obtained a log of all emails sent between company computers since the time of first infection. Logged emails are sorted by the time at which the email was sent from one computer to another. Given this log, describe a linear time algorithm to determine which company computers have been infected by the virus.

**Solution:** Assume each of the $n$ computers is labeled with a unique ID in the log. Maintain a growing frontier of infected computers, initialized by inserting the ID of the first infected computer into a hash table (or direct access array if the IDs are within a linear $O(n)$ range). For each of the $m$ log items ordered by time, check if the origin computer is in the infection frontier. If it is, insert the IDs of any receiving computers into the frontier. This operation maintains the set of infected computers, from before each email to after the email by induction. Assuming only one computer receives email during any log item, this algorithm takes $O(n + m)$ time to complete, which is linear in the number of computers and log items (the size of the input). Note that, while you **can** use a graph to answer this problem, constructing a graph is not necessary.

**Rubric:**

- 4 points for a correct algorithm description
- 1 points for running time analysis
- Partial credit may be awarded

**(b)** [5 points] **Gamer Latency:** Four gamers, AceArezu34, BenignBilly56, ClaudiaThe-Crusher94, and DastardlyDarius73, regularly engage in intense games of competitive Dandy Drush$^{\text{TM}}$. One gamer hosts the game, while the others connect to the host's machine over the internet. For elite gamers, **latency**, or the amount of time it takes for an update to travel from the game's server's to a player's screen, is a big concern! Assume that latency is directly proportional to the number of internet routers the data must pass through to reach its destination. The four gamers decide that the user who hosts their games should be the user who minimizes the sum of latencies of all four users. Note that the host will always have zero latency. Given a map of $n$ network routers from their internet service provider (obtained via questionable means), describe a linear time algorithm to compute who should host their game.

**Solution:** Run a breadth-first search from each player to find the latency from each player to every other player, which can be performed in linear time with respect to the number of routers and connections in the mapped router network. Then, compute the average latency for each player by summing the latency to the other gamers and divide by four. Choose a player with the minimum average latency to host the game.

**Rubric:**

- 4 points for a correct algorithm description
- 1 points for running time analysis
- Partial credit may be awarded

**(c)** [10 points] **Party Separation:** Haddy Ceron has gotten into trouble with some of her, shall we say, difficult friends: her friends do not get along with each other. She would like to invite them all to a party, but that would cause too much drama. Haddy decides instead to throw two separate parties, one on Friday and one on Saturday night. She wants to invite each of her friends to exactly one of the parties, but not to both. Haddy is specifically worried about conflicting pairs of friends who intensely dislike each other. Given a list of conflicting pairs among her friends, describe a linear time algorithm to determine whether Haddy can assign friends to parties, without assigning any conflicting pair of friends to the same party.

**Solution:** Construct a graph on Haddy's friends with an edge between friend $a$ and friend $b$ if they are a conflicting pair. We must determine whether this graph is **bipartite**: every vertex can be assigned one of two colors, either red or blue, such that no edge connects to vertices of the same color. Pick an arbitrary vertex $v \in G$ and color it blue. Run a breadth-first search from $v$ and color red all vertices reachable from $v$ in one step. Continue the breadth-first search, coloring nodes in even levels blue and odd levels red. Then loop through the edges of the graph. If there exists an edge connecting two vertices $a$ and $b$ having the same color, the shortest paths from $v$ to $r$ and $b$, together with the edge, form a cycle with odd length. This cycle is not two-colorable, so friends in this cycle cannot be assigned to parties without some conflicting pair being invited to the same party. Otherwise, if no edge between same colored vertices exists, then invite red friends on Friday and blue friends on Saturday, free from drama!

**Rubric:**

- 8 points for a correct algorithm description
- 2 points for running time analysis
- Partial credit may be awarded

**(d)** [10 points] **Ground Transportation:** Tim the Beaver needs to get from MIT to a recruiting event in San Francisco, but the airlines no longer allow beavers to fly. Tim cannot drive, so he decides to travel to his destination via bus and train. Tim vastly prefers trains over buses, and wants his trip to be **bus-sparse**: the trip should not contain a transfer from a bus to another bus, only from a bus to a train, a train to a bus, or a train to a train. Given a map of bus and train routes across the country, describe a linear time algorithm to find a bus-sparse itinerary containing the fewest vehicle transfers, or tell Tim that no bus-sparse itinerary exists to his destination.

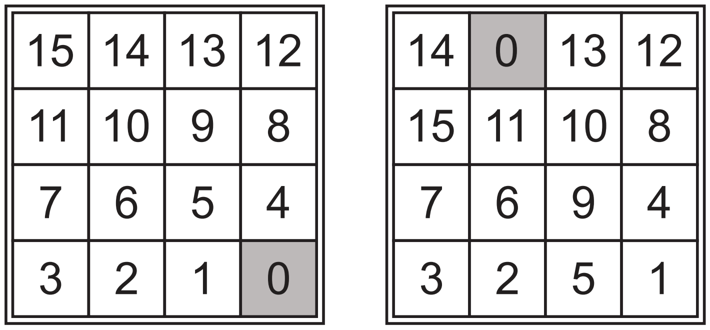
**Solution:** Construct a graph having two vertices per transit station $v$: vertex $v_b$ representing arrival at the station by bus, and vertex $v_t$ representing arrival at the station by train. For each train route from station $u$ to station $v$, add directed edges $(u_t, v_t)$ and $(u_b, v_t)$ to the graph, and for each bus route from station $u$ to station $v$, add directed edge $(u_t, v_b)$ (not $(u_b, v_b)$ as such an edge would mean taking two buses in a row). Let $s$ be the starting station and $t$ be the ending station. Use breadth-first search to determine whether a path exists from $s_b$ or $s_t$ to $t_b$ or $t_t$. If there is, return the itinerary associated with the path; otherwise return that no such itinerary exists. Let $n$ be the number of stations and $m$ be the number of routes. The graph has two vertices for each station and at most two edges for each route, so the size of the graph is $O(n + m)$. Running breadth first search twice (once from $s_b$ and once from $s_t$) takes $O(n + m)$ time, so this algorithm will answer TIM's query in linear time. (Note that running BFS once from $s_t$ is sufficient, since $s_t$ is connected to all vertices that $s_b$ is connected to.)

**Rubric:**

- 8 points for a correct algorithm description
- 2 points for running time analysis
- Partial credit may be awarded

**Problem 6-3.** [40 points] **Number Puzzles**

A **15-number puzzle** consists of 15 numbered slide-able square **pieces** placed on a $4 \times 4$ grid called a **board**, with one grid square left uncovered. Pieces may slide into the empty grid square, swapping the piece with the empty space; we call such an operation a **move**. The figure below depicts the puzzle in a solved configuration [left] and an unsolved configuration [right]. The goal of the puzzle is to transform an unsolved configuration to a solved configuration by performing a sequence of moves.



In this problem, we generalize the 15-number puzzle to an $(n \times m - 1)$-number puzzle, containing $n \times m - 1$ numbered pieces in an $n \times m$ board, with a $0$ marking the empty square. A move of a piece up, down, left, or right into the empty square will be denoted by the characters '`U`', '`D`', '`L`', and '`R`' respectively. For example, the unsolved configuration above can be solved via the sequence of moves `(R, U, L, L, U, U, L)`.

**(a)** [2 points] **Configurations:** Compute the number of possible placements of $n \times m - 1$ pieces within an $n \times m$ board.

**Solution:** For each board position, place a tile (or the free location) without replacement. The number of possible placements is then $\prod_{i \in nm} i = (nm)!$.

**Rubric:**

- 2 point for exactly $(nm)!$.
- 1 point partial credit for a bound that is $\Theta((nm)!) \neq (nm)!$.

**(b)** [3 points] **Implicit Graph:** Given a board configuration $C$, another board configuration $C'$ is a **neighbor** of $C$ if $C$ can be transformed into $C'$ via a single piece move. What is the minimum and maximum number of neighbors of any $(n \times m - 1)$-puzzle configuration? Describe how to compute all neighbors of a given configuration in $O(nm)$ time.

**Solution:** Each board configuration allows four possible moves: {`U, D, L, R`}, so maximum degree of a configuration is **four**. Moves are limited when the empty square is on the boundary of the board, allowing only **two** moves when in one of the four corner positions. To compute a neighbor of a configuration, make a copy of the board in $O(nm)$ time and swap the empty square with its top, bottom, left, or right adjacency in constant time.

**Rubric:**

- 1 point for maximum number of neighbors.
- 1 point for minimum number of neighbors.
- 1 point for neighbor computation in $O(nm)$ time.

**(c)** [10 points] **Shortest Solve:** Given a board configuration, describe an $O((nm + 1)!)$ time algorithm to output a shortest sequence of moves that solves the board, if such a sequence of moves exists.

**Solution:** Perform a breadth-first search from a solved configuration of the board, producing parent pointers and expanding a frontier of configurations until the input configuration is found (can alternatively search from the input configuration until the solved configuration is found). When the target is found, follow parent pointers to produce a sequence of moves for return. If the search completes before finding the target configuration, the board is not solvable. Because breath-first search finds shortest paths in an unweighted graph, this algorithm will find a shortest sequence of moves. Breadth-first search runs in linear time with respect to graph size. By part (a), there are at most $(nm)!$ vertices in the graph. Each vertex has constant out-degree, so the number of edges is also $O((nm)!)$. By part (b), each configuration takes $O(nm)$ time to compute, so the breadth-first search takes $O((nm)! \cdot nm) = O((nm + 1)!)$ time.

**Rubric:**

- 8 points for a correct algorithm description
- 2 points for running time analysis
- Partial credit may be awarded

**(d)** [25 points] **Implement:** Write the Python function `solve_puzzle(config)` that implements your puzzle solving algorithm. An input configuration will be a length $m$ tuple of length $n$ tuples storing the integers 0 to $n \times m - 1$ in some permutation, e.g., the right board would be `((14,0,13,12),(15,11,10,8),(7,6,9,4),(3,2,5,1))`. The function should return a tuple of moves from the character set {'`U`', '`D`', '`L`', '`R`'} that when applied to the input configuration solves the puzzle, or return `None` if no such sequence exists. Submit your code online at `py.mit.edu/6.006`.

**Solution:**

```
def move(mv, config):
    n, m = len(config[0]), len(config)
    for i in range(n):
        for j in range(m):
            if config[j][i] == 0:
                k, l = i, j
                if   mv == 'U' and j != m - 1:  l = j + 1
                elif mv == 'D' and j != 0:      l = j - 1
                elif mv == 'L' and i != n - 1:  k = i + 1
                elif mv == 'R' and i != 0:      k = i - 1
                else:
                    return None
                return tuple(tuple(
                        config[j][i] if el == config[l][k] else (
                        config[l][k] if el == config[j][i] else el)
                    for el in row) for row in config)
    raise TypeError('No zero to move configuration')

def solve_puzzle(config):
    n, m = len(config[0]), len(config)
    sol = tuple(tuple(n * (m - j) - i - 1 for i in range(n)) for j in range(m))
    parent = {sol: None}
    frontier = [sol]
    while 0 < len(frontier):
        current = []
        for source in frontier:
            for mv in ('U', 'D', 'L', 'R'):
                neighbor = move(mv, source)
                if (neighbor is not None) and (neighbor not in parent):
                    parent[neighbor] = source
                    current.append(neighbor)
                if config in parent:
                    moves = []
                    while config != sol:
                        for mv in ('U', 'D', 'L', 'R'):
                            if move(mv, config) == parent[config]:
                                moves.append(mv)
                                config = parent[config]
                                break
                    return tuple(moves)
        frontier = current
    return None
```