# Problem Set 5

Please write your solutions in the LaTeX and Python tem-plates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct.

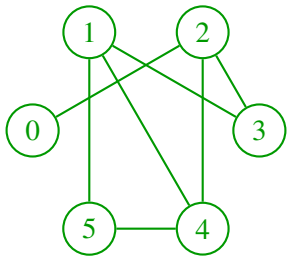
---

**Problem 5-1.** [15 points] **Graph Practice**

**(a)** [3 points] Draw the undirected graph described on the right by **adjacency matrix** `Adj`: a direct access array Set mapping each vertex $u \in \{0, \ldots, 5\}$ to an adjacency list `Adj[u]`, where each adjacency list is also implemented using a direct access array Set such that `Adj[u][v]` $= 1$ if and only if vertices `u` and `v` are connected by an edge.

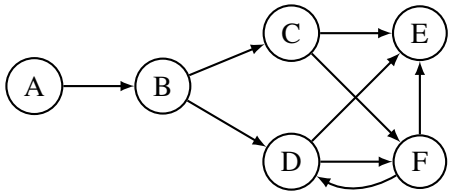
```
#        0  1  2  3  4  5
Adj = [[0, 0, 1, 0, 0, 0], # 0
       [0, 0, 0, 1, 1, 1], # 1
       [1, 0, 0, 1, 1, 0], # 2
       [0, 1, 1, 0, 0, 0], # 3
       [0, 1, 1, 0, 0, 1], # 4
       [0, 1, 0, 0, 1, 0]] # 5
```

**Solution:**




**Rubric:** -1 point per edge error; minimum 0 points

**(b)** [3 points] Write down the adjacency list representation of the graph on the right: where a Python Dictionary maps each vertex `v` to its adjacency list `Adj[v]`, and each adjacency list is a Python List containing `v`'s outgoing neighbors **listed in alphabetical order**.




**Solution:**

```
Adj = {
    'A': ['B'],
    'B': ['C', 'D'],
    'C': ['E', 'F'],
    'D': ['E', 'F'],
    'E': [],
    'F': ['D', 'E']
}
```

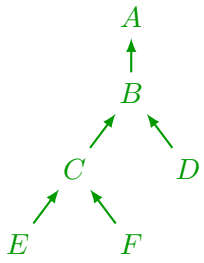
**Rubric:** -1 point per listing error; minimum 0 points

**(c)** [6 points] Run both BFS and DFS on the graph from part (b), starting from node A. While performing each search, visit the outgoing neighbors of a vertex in alphabetical order. For each search, draw the resulting tree and list vertices in the order in which they were first visited.

**Solution:**

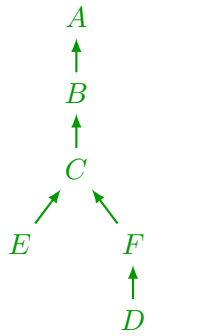
**BFS:**

`[A, B, C, D, E, F]`




**DFS:**

`[A, B, C, E, F, D]`




**Rubric:**

- 2 points per tree
- 1 point per order
- Partial credit may be awarded

**(d)** [3 points] It is possible to remove a single edge from the graph in (b) to make it a DAG. State every edge with this property, and for each, state a topological sort order of the resulting DAG.

**Solution:** There is only one cycle in the graph between vertices D and F, so we can remove either the edge (D, F) or (F, D) to make the graph acyclic. Removing edge (D, F) results in a DAG with a unique topological ordering (A, B, C, F, D, E). Removing (F, D) results in a DAG with two topological orderings: specifically (A, B, C, D, F, E) and (A, B, D, C, F, E).

**Rubric:**

- 1 point for the two edges
- 1 point for each topological order

**Problem 5-2.** [10 points] **Power Plants**

The Boston electrical network contains $n$ power plants and $n^2$ buildings, pairs of which may be directly connected to each other via bidirectional wires. Every building is powered by either: having a wire directly to a power plant, or having a wire to another building that is recursively powered. Note that the network has the property that no building could be powered by more than one power plant (or else accounting would be ambiguous). Boston has secured funds to install an emergency generator in one of the power plants, which would provide backup power to all buildings powered by it, were the power plant to fail. Given a list $W$ of all the wires in the network, describe an $O(n^4)$-time algorithm to determine the power plant where Boston should install the generator that would provide backup power to the most buildings upon plant failure.

**Solution:** Construct undirected graph $G$ whose vertices are the buildings and power plants, and whose edges are wires between them. There are $n^2 + n$ vertices and $|W| = O(n^4)$, under the assumption that at most one wire connects any pair of vertices.

Since every building is powered and no building is powered by the same power plant, there is exactly one power plant in each connected component of $G$. So the number of buildings that would benefit by having a generator installed at plant $p$ would be the size of the connected component containing $p$ minus one (since the plant is not a building). So we want to install the generator at the plant that powers the largest connected component.

So run Full-BFS or Full-DFS to count the size of each connected component in the graph, and return the plant contained in the largest one. Constructing $G$ takes $O(n^4)$ time, running Full-BFS or Full-DFS also takes $O(n^4)$ time, and finding the plant in a connected component can be done in $O(n)$ time, so this algorithm runs in $O(n^4)$ time.

**Rubric:**

- 2 points for graph construction
- 3 points for description of a correct algorithm
- 1 points for correctness argument for a correct algorithm
- 1 points for running time argument for a correct algorithm
- 3 points if the algorithm is efficient, i.e., $O(n^4)$
- Partial credit may be awarded

**Problem 5-3.** [10 points] **Short-Circuitry**

Star-crossed androids Romie-0 and Julie-8 are planning an engagement party and want to invite all their robotic friends. Unfortunately, many pairs of their friends can't be in the same room without short-circuiting. Romie-0 has an idea to throw *two* parties instead: inviting some friends to the first party, and the rest to the second. Ideally, everyone would get invited to a party, but nobody would go to the same party as someone that would make them short-circuit. Julie-8 points out to Romie-0 that this might not be possible, for example if they have three friends who all make each other short-circuit. Given the $n$ short-circuiting pairs of friends, describe an $O(n)$-time algorithm to determine whether Romie-0 and Julie-8 can invite all of their friends to two peaceful parties, without anyone short-circuiting.

**Solution:** Construct a graph $G$ where each vertex is a friend who might short-circuit, with an undirected edge between a pair of friends if they make each other short-circuit. Note that we can ignore any friend not participating in short-circuiting because they have no restrictions on which party they can attend. We can assign the friends to two peaceful parties if and only if there is a coloring of the vertices of $G$ with two colors such that no edge connects vertices of the same color, i.e., the graph must be bipartite. To determine whether the graph is bipartite, we will either find a valid two-coloring, or find an odd-length cycle in the graph (if there is an odd-length cycle in the graph, there certainly cannot be a valid two-coloring of the cycle or the graph).

To find a possible two-coloring of $G$, our approach will be to find a spanning forest $F$ of $G$, two-color it, and then check to see whether any edge in $G$ violates the coloring. If there is an edge $\{u, v\}$ such that both vertices have the same color, then the path between $u$ and $v$ in $F$ has even length (since it has a valid two-coloring), so adding $\{u, v\}$ would be an odd-length cycle.

Running either Full-BFS or Full-DFS on $G$ can construct a spanning forest $F$. We choose to run Full-BFS, marking each vertex with its shortest path distance to the arbitrarily chosen source within its connected component. To two-color $F$, assign one color to all vertices with even distance to their source, and second

color to all vertices with odd distance to their source (since shortest paths alternate distance parity along the path). Lastly, check whether every edge in the graph has endpoints assigned opposite colors. If not, return that two parties is impossible. Otherwise the coloring confirms the graph is bipartite, so return that two parties is possible.

$G$ has size $O(n)$ since it has $n$ edges and at most $2n$ vertices, and can be constructed in as much time. Running Full-BFS on $G$ takes time $O(n)$ (linear in the size of $G$), and then for each of the $n$ edges we perform a constant-time check. So this algorithm runs in $O(n)$ time.

**Rubric:**

- 2 points for graph construction
- 3 points for description of a correct algorithm
- 1 points for correctness argument for a correct algorithm
- 1 points for running time argument for a correct algorithm
- 3 points if the algorithm is efficient, i.e., $O(n)$
- Partial credit may be awarded

**Problem 5-4.** [10 points] **Ancient Avenue**

The ancient kingdom of Pesomotamia was a fertile land near the Euphris and Tigrates rivers. Newly-discovered clay tablets show a map of the kingdom on an $n \times n$ square grid, with each square either:

- part of a river, labeled as `'euphris'` or `'tigrates'`; or
- not part of a river, labeled with a string: the name of the farmer who owned the square plot of land.

The tablets accompanying the map state that ancient merchants built a **trade route** connecting the two rivers: a path along edges of the grid from some grid intersection adjacent to a `'euphris'` square to a grid intersection adjacent to a `'tigrates'` square. The tablets also state that the route:

- did not use any edge between two squares owned by the same farmer (so as not to trample crops); and
- was the shortest such path between the rivers (assume a shortest such path is unique).

Given the labeled map, describe an $O(n^2)$-time algorithm to determine path of the ancient trade route.

**Solution:** Let $M$ be the $n \times n$ labeled map, where $M[r][c]$ is the label of the grid square in row $r$ and column $c$. Construct graph $G$ with $(n+1)^2$ vertices $(r, c)$ for all $r, c \in \{0, \ldots, n\}$, and an undirected edge between:

- vertex $(r, c-1)$ and vertex $(r, c)$ for $r \in \{0, \ldots, n\}$ and $c \in \{1, \ldots, n\}$, except when:
  - $r \in \{1, \ldots, n-1\}$ (not a boundary edge),
  - $M[r][c]$ is not `'euphris'` or `'tigrates'` (is owned by a farmer), and
  - $M[r][c-1] = M[r][c]$ (owned by the same farmer); and
- vertex $(r-1, c)$ and vertex $(r, c)$ for $c \in \{0, \ldots, n\}$ and $r \in \{1, \ldots, n\}$, except when:
  - $c \in \{1, \ldots, n-1\}$ (not a boundary edge),
  - $M[r][c]$ is not `'euphris'` or `'tigrates'` (is owned by a farmer), and
  - $M[r-1][c] = M[r][c]$ (owned by the same farmer).

This graph has the property that a path between any two vertices corresponds to a route on the map that does not trample crops. It remains how to find the shortest such route between any vertex adjacent to a `'euphris'` square to a vertex adjacent to a `'tigrates'` square. For each `'euphris'` square $M[r][c]$, mark vertices $\{(r, c), (r+1, c), (r, c+1), (r+1, c+1)\}$ as `'euphris'` vertices (possible starting intersection of the route), and mark `'tigrates'` vertices similarly.

We could solve SSSP from each `'euphris'` vertex using BFS, but that would take too much time (potentially $\Omega(n^2)$ `'euphris'` vertices, with each BFS taking $\Omega(n^2)$ time.) So instead, add a supernode $s$ to $G$ with an undirected edge from $s$ to each `'euphris'` vertex in $G$, to construct graph $G'$. Then any possible trade route will correspond to a path from $s$ to a `'tigrates'` vertex in $G'$ (without the first edge), and the trade route will be the shortest among them. So run BFS from $s$ to every `'tigrates'` vertex, and return the shortest path to the closest one by traversing parent pointers back to the source.

Graph $G'$ has $(n + 1)^2 + 1$ vertices and $O(n^2) + O(n^2)$ edges, so can be constructed in $O(n^2)$ time, and running BFS once from $s$ also takes $O(n^2)$ time, so this algorithm runs in $O(n^2)$ time in total.

**Rubric:**

- 3 points for graph construction
- 2 points for description of a correct algorithm
- 1 points for correctness argument for a correct algorithm
- 1 points for running time argument for a correct algorithm
- 3 points if the algorithm is efficient, i.e., $O(n^2)$
- Partial credit may be awarded

**Problem 5-5.** [15 points] **Statum Quest**

Liza Pover is a hero on a quest for free pizza in the Statum Center, a large labyrinth with some of the best free food in the entire Technological Institute of Mathachusetts. Many unsuspecting victims[1] have ventured into the labyrinth in search of free food, but few have escaped victorious. Luckily, Liza has downloaded a map from `⌊plans⌋.tim.edu` consisting of **locations** $L$ (including rooms, hallways, staircases, and elevators[2]) and **doors** $D$ which connect pairs of locations. One location $e \in L$ is marked as the **entrance/exit**: Liza's path must begin and end here. A subset $\Pi \subset L$ of locations each contain a free pizza. Liza's goal is to enter the maze, grab one pizza, and leave as quickly as possible.

Each door $d = (\ell_1, \ell_2)$ connects two locations $\ell_1$ and $\ell_2$ and may be **one-way** (can be traversed only from $\ell_1$ to $\ell_2$) or **two-way** (can be traversed in either direction). In addition, each door $d$ may require **card access** to one of the four Statum Center labs — See-Sail, TOPS, S3C[3], and DSW[4]. A card-access door of type $t \in \{\text{SeeSail}, \text{TOPS}, \text{S3C}, \text{DPW}\}$ can only be traversed after Liza acquires the matching key card, where the key card of type $t$ is in a known location $\ell_t$.

Describe an $O(|L| + |D|)$-time algorithm to find a viable path from $e$ to $e$ that collects at least one pizza from $\Pi$ (and possibly some of the key cards along the way to open some doors), and minimizes the number of times that Liza has to walk through a door.

[1] Formerly known as freshmen

[2] We model an entire elevator column as one location, with a door to each floor location it can access

[3] Super Secret Spider Consortium

[4] Department of Speechlore and Wisdomlove

**Solution:** While Liza traverses the Statum Center, which doors she can enter depends on which key cards she has in her possession. It would be useful to know which cards she has in her possession while traversing the building. Since there are only $4$ types of key card, there are at most $2^4$ possible sets of key cards she could have at any given time. We will use graph duplication to represent these $2^4 = O(1)$ possible states.

Let $T = \{\text{SeeSail}, \text{TOPS}, \text{S3C}, \text{DPW}\}$, let $C(\ell, k_t)$ equal $1$ if $\ell = \ell_t$ and $k_t$ otherwise, and let $P(\ell, p)$ equal $1$ if location $\ell$ contains a pizza and $p$ otherwise. If Liza enters any location $\ell$ having at least $p$ pizzas and $k_t$ key cards of type $t$, she can leave with at least $P(\ell, p)$ pizzas and $C(t, \ell, k_t)$ key cards of type $t$, for any $t \in T$. Construct a graph $G = (V, E)$ as follows.

- For each location $\ell \in L$, construct $2^5$ vertices $v(\ell, p, k_{\text{SeeSail}}, k_{\text{TOPS}}, k_{\text{S3C}}, k_{\text{DPW}})$ where $p \in \{0, 1\}$ and $k_t \in \{0, 1\}$ for all $t \in T$. Such a vertex represents arriving at location $\ell$ while possessing at least $p$ pizzas and $k_t$ of key card $t$ for $t \in T$.
- For each one-way door $(\ell_1, \ell_2)$ if the door does not require card access or the door requires card access of type $t$ and $C(t, \ell_1, k_t) = 1$ (i.e., Liza can leave $\ell_1$ with a card of that type), then add a directed edge from:
  - vertex $v(\ell_1, p, k_{\text{SeeSail}}, k_{\text{TOPS}}, k_{\text{S3C}}, k_{\text{DPW}})$ to
  - vertex $v(\ell_2, P(\ell_1, p), C(\ell_1, k_{\text{SeeSail}}), C(\ell_1, k_{\text{TOPS}}), C(\ell_1, k_{\text{S3C}}), C(\ell_1, k_{\text{DPW}}))$,

  for all $p \in \{0, 1\}$ and for all $k_t \in \{0, 1\}$ for all $t \in T$.
- For each two-way door $(\ell_1, \ell_2)$ construct two one-way edges in either direction as above.

This graph exactly encodes all possible state transitions from any location in the Statum Center while keeping track of the number of pizzas and key cards that Liza can be holding. Thus a path from vertex $s = v(e, 0, 0, 0, 0, 0)$ to any vertex $q \in Q = \{v(e, 1, k_{\text{SeeSail}}, k_{\text{TOPS}}, k_{\text{S3C}}, k_{\text{DPW}}) \mid \forall k_t \in \{0, 1\} \, \forall t \in T\}$ represents a path that enters and leaves the Statum Center at $e$, while also procuring a pizza. Since Liza would like to minimize the number of doors she must cross, running BFS in $G$ from $s$ to each $q \in Q$ finds the minimum doors crossed to each of them, so we can return a shortest path to any of them by traversing parent pointers back to the source.

Graph $G$ has $|L| \cdot 2^5 = O(|L|)$ vertices and at most $|D| \cdot 2^5 = O(|D|)$ edges, so can be constructed in $O(|L| + |D|)$ time, and running BFS once from $s$ also takes time linear in the size of the graph, so this algorithm runs in $O(|L| + |D|)$ time.

**Rubric:**

- 5 points for graph construction
- 2 points for description of a correct algorithm
- 2 points for correctness argument for a correct algorithm
- 2 points for running time argument for a correct algorithm
- 4 points if the algorithm is efficient, i.e., $O(|L| + |D|)$
- Partial credit may be awarded

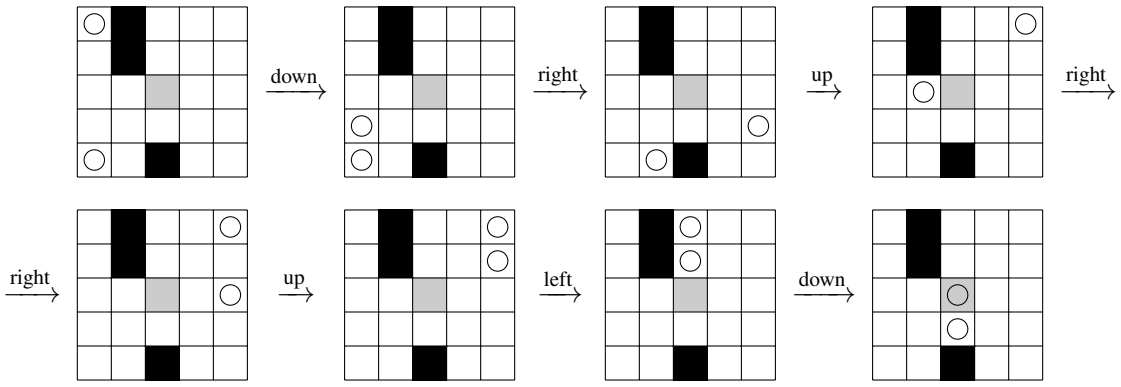
**Problem 5-6.** [40 points] **6.006 Tilt**

6.006 Tilt[5] is a puzzle game played on a Tilt **board**: an $n \times n$ square grid, where grid square contains either a fixed **obstacle**, a movable **slider**, or neither (the grid square is **empty**).

A Tilt **move** consists of tilting a board in any of the four cardinal directions (up, left, down or right), causing each slider to slide maximally in that direction until it hits either: the edge of the board, an obstacle, or another slider that cannot slide any further. A Tilt move results in a new Tilt board **configuration**.

Given a Tilt board $B$ with one grid square $t$ labeled as the **target**, a sequence of $k$ Tilt moves **solves** 6.006 Tilt **puzzle** $(B, t)$ if applying the moves in sequence to $B$ results in a Tilt board configuration $B'$ that contains a slider in square $t$.

The figure below shows a small 6.006 Tilt puzzle and a solution using the fewest possible moves $k = 8$. Obstacles are shown in black, movable sliders are circles, and the target square is shaded gray.




We represent an $n \times n$ board configuration `B` using a length-$n$ tuple of length-$n$ tuples, each representing a row of the configuration, where the grid square in row $y$ (from the top) and column $x$ (from the left) is `B[y][x]`, equal to either character `'#'` (an obstacle), `'o'` (a slider), or `'.'` (neither). We represent the target square by a tuple `t = (xt, yt)` where the puzzle is solved when `B[yt][xt] = 'o'`. Your code template has a function `move(B, d)` which computes a move from board `B` in direction `d` in $O(n^2)$ time.

**(a)** [2 points] Given a Tilt puzzle with starting $n \times n$ board $B$ containing $b$ fixed obstacles and $s$ movable sliders, argue that the number of board configurations reachable from $B$ is at most $C(n, b, s) = \frac{1}{s!}\prod_{i=0}^{s-1}(n^2 - b - i)$.

**Solution:** Each of the $s$ sliders can be on any of the $n^2 - b$ empty cells, so there are

$$\binom{n^2 - b}{s} = \frac{(n^2 - b)!}{s!(n^2 - b - s)!} = \frac{1}{s!}\prod_{i=0}^{s-1}(n^2 - b - i)$$

possible configurations, though many of these configurations may be unreachable, as sliders may be blocked by obstacles from reaching some empty squares, or there may be gaps between sliders that cannot be created by tilting the puzzle.

**Rubric:**

- 2 points for a correct argument

[5] This is a simplification of the actual game ***Tilt***, shown here: http://www.youtube.com/watch?v=mFGevho4OLY

- Partial credit may be awarded

**(b)** [3 points] If Tilt board configuration $B$ can reach another Tilt board configuration $B'$ by a single move, we call $B'$ a successor of $B$, and $B$ a predecessor of $B'$. Argue that a board configuration $B$ may have $\Omega(n^s)$ predecessors, but has at most $r = O(1)$ successors.

**Solution:** A Tilt board configuration $B$ has at most $r = 4$ successors, since there are only 4 possible Tilt moves from the configuration, and each move results in at most a single new configuration. However, $B$ may have $\Omega(n^s)$ predecessors: consider a board with no obstacles but $s \leq n$ sliders, and let $B$ be a configuration with at most one slider in each column, with each slider at the bottom of its column. This board is reachable via a `'down'` move from any configuration having $s$ sliders in the same columns. Each slider could be in any of the $n$ squares in its column in a `'down'` predecessor of $B$, so there are at least $\Omega(n^s)$ predecessors of $B$.

**Rubric:**

- 1 point for a correct argument for successors
- 2 points for a correct argument for predecessors
- Partial credit may be awarded

**(c)** [10 points] Given a 6.006 Tilt puzzle $(B, t)$, describe an algorithm to return a move sequence that solves the puzzle in the fewest moves, or return that no such move sequence exists. If the puzzle's shortest solution uses $k$ moves ($k = \infty$ if the puzzle is not solvable), your algorithm should run in $O(n^2 \min\{r^k, C(n, b, s)\})$ time.

**Solution:** Consider the abstract graph $G$ defined by the $C(n, b, s)$ configurations possibly reachable from $B$. Our approach will be to run breadth-first search on $G$ from the input board configuration $B$, but only construct other reachable configurations as they are reached (and not try to construct the entire graph from the beginning). While processing configuration $B'$ during the search, compute the adjacent successor configurations from $B'$ in $O(n^2)$ time (using the move function provided), and return a shortest path to the first $B^*$ found where $B^*[y_t][x_t] =$ `'o'`. By definition of shortest paths, $B^*$ is reachable in $k$ moves, so since each configuration has at most $r = 4$ successors, at most $\sum_{i=0}^{k} r^i < r^{k+1} = O(r^k)$ configurations are explored during this search. If $B$ is not solvable, every non-solved configuration may need to searched, which is at most $C(n, b, s)$ configurations. Thus since it takes $O(n^2)$ time to process each configuration, this search takes at most $O(n^2 \min\{r^k, C(n, b, s)\})$ time to complete.

**Rubric:**

- 2 points for graph construction
- 2 points for description of a correct algorithm
- 1 points for correctness argument for a correct algorithm
- 3 points for running time argument for a correct algorithm
- 2 points if the algorithm is efficient, i.e., $O(n^4)$
- Partial credit may be awarded

**(d)** [25 points] Write a Python function `solve_tilt(B, t)` that implements your algorithm from part (d) using the template code provided. You can download the code template and some test cases from the website.

**Solution:**

```
def solve_tilt(B, t):
    '''
    Input:  B | Starting board configuration
            t | Tuple t = (x, y) representing the target square
    Output: M | List of moves that solves B (or None if B not solvable)
    '''
    M = []
    xt, yt = t
    P = {B: None}
    levels = [[B]]
    while levels[-1]:
        L = []
        for B1 in levels[-1]:
            for d in ('up', 'down', 'left', 'right'):
                B2 = move(B1, d)
                if B2 not in P:
                    P[B2] = (B1, d)
                    L.append(B2)
                    if B2[yt][xt] == 'o':
                        while P[B2]:
                            B2, d = P[B2]
                            M.append(d)
                        M.reverse()
                        return M
        levels.append(L)
    M = None
    return M
```

# 6.006 Introduction to Algorithms

Spring 2020