Problem Set 5

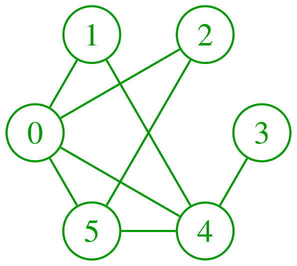
All parts are due on April 1, 2021 at 10PM. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. If you give an algorithm or a data structure, for full credit you must justify its runtime and correctness unless the problem says otherwise. Solutions should be submitted on Gradescope, and any code should be submitted for automated checking on `alg.mit.edu`.

Problem 5-1. [10 points] **Individual problem 1**

(1) [3 points] Draw the undirected graph G described on the right by the **adjacency matrix** `Adj`.
Recall that an adjacency matrix is a direct access array mapping each vertex `u` $\in \{0, \ldots, 5\}$ to a list of adjacencies `Adj[u]` that is also implemented using a direct access array, 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, 1, 1, 0, 1, 1], # 0
       [1, 0, 0, 0, 1, 0], # 1
       [1, 0, 0, 0, 0, 1], # 2
       [0, 0, 0, 0, 1, 0], # 3
       [1, 1, 0, 1, 0, 1], # 4
       [1, 0, 1, 0, 1, 0]] # 5
```

Solution:



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

(2) [3 points] What is the **adjacency list** representation of G? You can express this as a Python dictionary mapping each vertex `u` to a list containing `u`'s neighbors in increasing order.

Solution:

```
{0: [1, 2, 4, 5],
 1: [0, 4],
 2: [0, 5],
 3: [4],
 4: [0, 1, 3, 5],
 5: [0, 2, 4]}
```

Rubric: -1 point per vertex error; minimum 0 points

(3) [4 points] Perform **breadth-first search** on G, starting at vertex 2. Visit the neighbors of each vertex in increasing order. What is the BFS tree that you get? Express this as a Python dictionary mapping each vertex `u` to its **parent** (the vertex `v` from which you first visited `u`) or `None` if `u` is the root.

Solution:

```
{0: 2,
 1: 0,
 2: None,
 3: 4,
 4: 0,
 5: 2}
```

Rubric: -1 point per parent error; minimum 0 points

Problem 5-2. [20 points] **Friendship Individual problem 2**

In a social network graph, nodes represent people, and edges between nodes represent friendship between two people (we can assume friendships are symmetric in this case). Suppose that in our graph, nobody has more than 10 friends. We are trying to come up with a way to determine whether two specific people are within a distance d from each other, meaning that the shortest path between them has at most d edges. Assume that there are n people with distinct IDs in the range $\{0, \ldots, n-1\}$ and m friendship relationships.

(1) [10 points] Describe an algorithm to determine whether two people are within distance 6 of each other. You don't need to analyze the runtime/correctness of the algorithm.

Solution: Run a BFS starting from either person. Keep track of how many levels have been traversed from the starting node. If that level exceeds 6, then terminate and return False. Otherwise, if the BFS reaches the other person, return True.

(2) [10 points] In order to implement the algorithm above, which representation would you choose to represent the social network graph, an *adjacency list* or an *adjacency matrix*? Explain why your choice is more efficient.

Solution: Adjacency lists would be the best representation, using a direct access array to represent the vertices and their adjacency lists. The indices of the direct access array correspond to a person's id, and each array slot i points to a list containing the people that person i is friends with. Adjacency matrix does not work as well in this scenario since the graph is sparse, and we never need to check whether an edge exists between two arbitrary people in our BFS algorithm. As a result, the adjacency matrix approach would take order $O(n^2)$ time to create, and $O(n)$ for finding the neighbors of a node, while the adjacency list would take $O(n)$ time to create, and finding the neighbors of a node would take constant time as each node only has up to 10 neighbors.

Problem 5-3. [10 points] **Finding a Needle in a Haystack**

The Binary Max Heap is optimized to perform Priority Queue methods `build(X)`, `insert(x)`, and `delete_max()` at the expense of other Set interface methods.

Suppose each item `x` has a **unique** integer ID `x.id` in addition to its key `x.key`. Design a Priority Queue data structure that supports, in addition to the above three methods, the ability to efficiently delete the item with a given ID. Your running times may be average-case and/or amortized.

- `build(X)`: build the priority queue from the given list of n items in $O(n)$ time.
- `insert(x)`: insert `x` into the queue in $O(\log n)$ time.
- `delete_max()`: delete and return an item with the largest key in $O(\log n)$ time.
- `delete(id)`: delete and return the item with ID `id` from the queue in $O(\log n)$ time.

Solution: Our data structure will consist of a Binary Max Heap `Q` mapping keys to items (using a Dynamic Array) plus a Hash Table `T` mapping IDs to the index of the corresponding item in `Q`.

- `build(X)`: first `Q.build(X)`, then `T.build((Q[i].id, i) for i in range(n))`.
  $O(n)$ worst-case time
- `insert(x)`: perform `Q.insert(x)`, but whenever we swap `Q[i]` with `Q[j]` we also swap `T.find(Q[i].id).index` with `T.find(Q[j].id).index`.
  $O(\log n)$ average-case amortized time, due to resizing and $O(\log n)$ lookups in `T`.
- `delete_max()`: first perform `Q.delete_max()` (swapping indices in `T` as above) to get max item `x`, and do `T.delete(x.id)` before returning `x`.
  $O(\log n)$ average-case amortized time, due to resizing and $O(\log n)$ lookups in `T`.
- `delete(id)`: first call `T.find(id)` to get the index `i` where item `x` is located in `Q`. Next, swap `Q[i]` with `Q[-1]` and do `Q.delete_back()` and `T.delete(x.id)` to remove `x` from both data structures. Lastly, perform either `Q.max_heapify_up(i)` or `Q.max_heapify_down(i)` to fix the possible violations of the Max Heap Property of the moved item.
  $O(\log n)$ average-case amortized time, due to resizing and $O(\log n)$ lookups in `T`.

Rubric:

- points for

Problem 5-4. [30 points] **The Adventures of Mick and Rorty**

The year is 2021 in Dimension C-138, and Mick and Rorty are not happy. They used to be able to instantly travel wherever they wanted in the universe using Mick's trusty portal gun, but this is no longer possible as the Galactic Federation has placed new regulations on portal gun travel. Now, the only portal gun travel allowed is between m specific pairs of planets (known as planet hops). As a result, in order for Mick and Rorty to reach any destination planet, they must travel via a series of planet hops. Each planet hop is denoted by the pair (p_1, p_2) of planets it connects, and they can hop in either direction.

(1) [15 points]

Mick and Rorty have their eyes on Kernel, a faraway planet where everything is made of corn, even to the atomic level. Their portal gun has a limited amount of fuel (known as portal fluid). Each planet hop (p_i, p_j) requires a given positive integer amount of portal fluid units denoted by h_{ij}. Describe an $O(km)$ worst-case algorithm to determine whether there exists a sequence of planet hops that they can take from Earth to Kernel that costs a total of less than k fluid units. Assume that $m \geq n$.

Solution: Construct a graph of all the planets, connecting two planets p_1 and p_2 with an a-edge undirected chain of edges for every planet hop (p_1, p_2) having cost $a < k$ fluid units (and no chain of connections if the hop costs k or more units). This graph has at most $(k-1)m$ edges and at most $n + m(k-1)$ vertices, so constructing this graph takes $O(km)$ time. Then we can run BFS from Earth to find the length ℓ of a shortest path to Kernel in this graph, also in $O(km)$ time. If $\ell < k$, then Mick and Rorty can reach Kernel spending at most k units, and cannot reach the planet within that budget if $\ell \geq k$.

Rubric:

- 3 points for description of graph
- 2 points for description of a correct algorithm
- 2 point for correct analysis of running time
- Partial credit may be awarded

(2) [15 points] After realizing the resource costs of the portal gun, Mick attempts to improve its usage. He adds the hyper-hop feature, which allows them to make any planet hop using only one fluid unit. However, after making a hyper-hop from one planet to another, the portal gun goes on cool-down and cannot hyper-hop again until it has made another normal planet hop. Describe an $O(km)$ worst-case algorithm to determine whether there exists a sequence of planet hops (both normal and hyper-hops) that Mick and Rorty can take to reach Kernel using less than k fluid units.

Solution: To solve this problem, we combine the ideas of Part 1 with some graph duplication ideas. We construct a graph having two vertices per planet p: (1) vertex p^s corresponding to being at planet p and able to travel to another planet using either a normal or hyper hop (i.e. the portal gun is not on cooldown), and (2) vertex p^r

corresponding to being at planet p after using a hyper-hop, with the portal gun on cooldown. Then for each hop (p_1, p_2), add the following edges:

- a single directed edge from vertex p_1^s to p_2^r
- a single directed edge from vertex p_2^s to p_1^r

and if the cost of the hop is $a < k$ units, we also add following additional edges:

- a-edge directed chain of edges from p_1^s to p_2^s
- a-edge directed chain of edges from p_1^r to p_2^s
- a-edge directed chain of edges from p_2^s to p_1^s
- a-edge directed chain of edges from p_2^r to p_1^s

The number of edges and vertices in this graph is still $O(km)$, so constructing this graph takes $O(km)$ time. Then any sequence of hops from Earth to Kernel in this graph corresponds visiting a sequence of planets without making two hyper-hops in a row. So we can run BFS from Earth (starting from the vertex that is allowed to hyper-hop to other planets) to find the length ℓ of a shortest path to Kernel (either vertices) in $O(km)$ time. As in part (b), the duo can reach Kernel by spending at most k units if and only if $\ell < k$.

Rubric:

- 4 points for description of graph
- 4 points for description of a correct algorithm
- 2 point for correct analysis of running time
- Partial credit may be awarded

Problem 5-5. [30 points] **Noobls**

Noolbs is a video game where the goal is to pop all of the mysterious balloons that are raining down from the sky. You play as Earth's last line of defense, a single monkey with an unlimited number of darts. The game is played on a board B which is a 2D array with R rows and C columns. Rows are listed in decreasing altitude order, with the last row representing the ground. As a monkey that is unable to fly, you are restricted to stay in this row on the ground. Balloons can be of either type 1, 2 or 3. There can only be one balloon per cell, but there can be several balloons on the same column.

The monkey starts with 3 lives. Each time you make a move of either moving or throwing a dart, all balloons descend by exactly 1 row. If a balloon ever lands on top of your monkey, the monkey loses all of its remaining lives and you lose the game. If a balloon ever lands on the ground but not on your monkey, your monkey will only lose one life but the balloon will magically re-appear in the same column but on the top row with a type one more than its current type (a 1 would become a 2, a 2 would become a 3, and a 3 would remain a 3). This re-spawn happens instantaneously, before your monkey can make its next move (in particular, there will never be a balloon on the ground with the monkey). Needless to say, if the monkey ever loses its 3 lives, you lose the game.

In the board, 'x' represents the monkey, 0s represent empty space, and 1s, 2s, and 3s represent the balloons. Every turn, your monkey must either move left, move right, or throw a single dart straight up, hitting the closest balloon in your current column. When hit, balloons represented by a 3 become a 2, balloons represented by 2 become a 1, and balloons represented by a 1 are destroyed.

To win the game, you must guide your monkey to completely destroy all of the balloons without losing all 3 of the monkey's lives.

(1) [3 points] Argue that the number of possible configurations for a game is $3(C * 4^{(R-1)C})$. Note that 'configuration' refers to the information needed to precisely represent the state of the game at any point in time.

Solution: Each space in the game board (except for the last row) can be any integer in the range $[0, 4]$, thus contributing $4^{(R-1)C}$ possibilities. Additionally, the monkey can be in any of the C columns in the bottom row (and the rest must be 0). Finally, there can be up to 3 lives remaining.

Rubric:

- 1 point for the $4^{(R-1)C}$ factor
- 1 point for considering the position of the monkey
- 1 point for considering the number of lives

(2) [4 points] Argue that maximum number of moves in a game is $3R$.

Solution: Any balloon will hit the ground 3 times after 3R steps. So, if it hasn't been destroyed before, the game will end as a loss to the player.

Suppose you have not won within 3R moves. Then, after making 3R moves, there must still exist at least one balloon on the board. This balloon must have hit the ground 3 times (because it strictly moves down one row per move, and if the balloon starts in the top row then it takes R moves to hit the ground). Hence, you must have run out of lives, and the game must be over.

Rubric:

- 4 point for correct argument for $3R$ term.

(3) [8 points]

Given an input configuration B, describe an algorithm to determine the minimum number of moves to win or return None if the game is not winnable. Your algorithm should run in time $O(RC * 3^{3R})$. *Hint:* Consider traversing through a graph where the nodes represent different configurations of the game board, but without building the entire graph beforehand. Instead, compute a node's neighbors only when you are ready to process that node. Thus the runtime will be determined by the number of configurations you go through and the time it takes to compute the neighbors at each step.

Solution: Let $G = (V, E)$ be the directed graph where the vertices represent all possible board configurations. We define an edge e to exist from vertex v_1 to vertex v_2 if and only if you can transform the configuration in v_1 to the configuration in v_2 in a single move (remember the possible moves are move left, move right, or throw a dart). Given a starting configuration, we additionally store extra variables to store the total number of balloons remaining and the number of lives remaining. We run BFS on this graph, starting from the input configuration, until we find a configuration that has 0 balloons left. We also store past configurations we have already seen and do not process the same configuration twice. We can store/look up configurations in a hash table. Instead of computing the entire game board beforehand, we compute the neighbors of each only when we are ready to process that node. To compute a node's neighbors takes $O(RC)$ time to create a copy of the board configuration, update moving the monkey or throwing a dart, move all balloons down, check for a collision with the monkey, and spawn higher tier balloons and decrement the ”lives remaining” count if necessary. If a balloon is completely destroyed, we decrement our ”balloons remaining” counter. Note that if a neighbor is a loss (the monkey collides with a balloon or we run out of lives) then we do not add this configuration to the frontier.

Correctness: Because we are running BFS on the graph of all possible board configurations, where nodes have directed edges to nodes that can be reached in one move, we are guarenteed to find the shortest path to any configuration that does not have any more balloons, if one exists. The length of this path represents how many moves it took to get to that configuration, thus our algorithm returns the minimum number of moves to win if a path exists. By the previous part, we know that our algorithm must terminate after $3R$ steps since any configuration after $3R$ turns must be a losing configuration.

Runtime: To analyze runtime, we first consider the maximum number of turns we can make. By the previous part, we know that any game can last at most $3R$ turns. Thus our BFS will create at most $O(R)$ level sets. Since every node has an outdegree of 3, we know that after making k turns the upper bound on the number of configurations we are looking at in the current level set is $O(3^k)$. Thus we consider at most $O(3^{3R})$ configurations. Since it takes $O(RC)$ to compute the neighbors of each node, the overall runtime is $O(RC * 3^{3R})$.

Rubric:

- 2 points for describing how to compute neighbors
- 1 point for describing when the algorithm returns None
- 2 points for correct runtime analysis

(4) [15 points] Implement the function `solve_noolbs` that implements your above algorithm. Download a template and submit your code at `https://alg.mit.edu/spring21/PS5`.

Please submit a screenshot of your report that includes your `alg.mit.edu` student ID (top right corner) and the percentage of tests passed to Gradescope.