Problem Set 4

All parts are due on November 15, 2016 at 11:59PM. Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

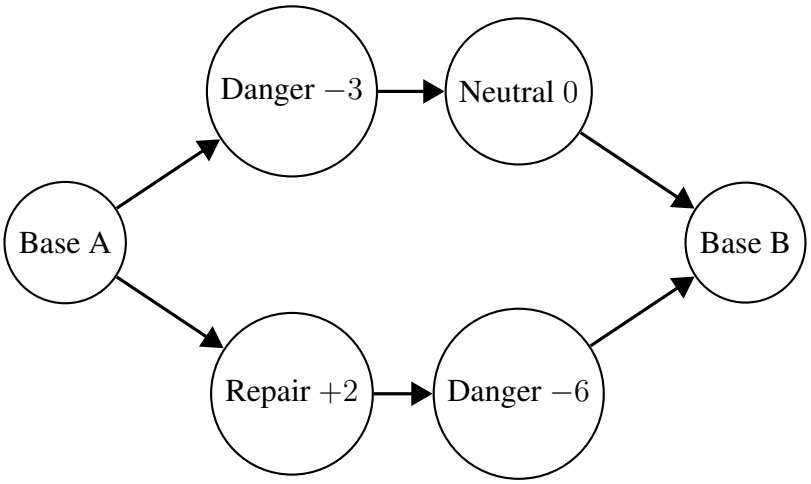
Problem 4-1. [30 points] **Runaround**

Powell and Donovan were sent to Mercury with the latest robot model, SPD-13 (Speedy) to revitalize an abandoned mining station. After several near disasters following the rules of Prof. Asimov, they have learned to be more careful when sending Speedy on missions to obtain selenium from other bases. Speedy has been programmed to balance his need to follow human orders with his need to protect himself.

Speedy can travel through Mercury along a certain network of established, directed roads. At the intersections of these roads, there may be dangerous gases that could hurt him, nothing, or occasional repair outposts. Dangerous intersections reduce Speedy's armor integrity, repair posts increase his integrity, and neutral posts do nothing. The starting base can be assumed to be neutral.

When they need to send Speedy on a mission, Powell and Donovan reset Speedy's armor integrity to a starting integer value C. When Speedy receives an order, he leaves his starting base A immediately. If there exists a route from A to his destination base B with an integrity cost $\leq C$, Speedy will complete the route sucessfully every time. If, however, Speedy's armor integrity falls below 0, he will become unable to move, and will call for emergency repairs.

For example, consider the terrain depicted below. If $C < 3$, Speedy will get stuck at one of the danger nodes and call for emergency repairs, since the total damage he accumulates will have exceeded the integrity points he was willing to sacrifice. If $3 \leq C < 4$, Speedy will take the top route to B. Finally, if $C \geq 4$, Speedy could take either path to B, since the total damage accumulated on either path does not exceed his starting value.



(a) [3 points] View the scenario as a directed graph. Note that so far in this course, we have only discussed graphs with *edge* costs, and yet in this scenario, there appears to be *vertex* costs. How can we modify the graph such that 1) there are only edge costs, and 2) the sum of the edge costs of a path from A to any node v is equivalent to the change in armor integrity Speedy will incur if he uses this path to get to v?

(b) [10 points] Design an $O(VE)$-time algorithm to determine whether Speedy can successfully complete a given mission from Base A to Base B with a given initial armor integrity value C.

(c) [5 points] Yesterday, Powell and Donovan remotely assigned Speedy a mission to return to home base from another outpost, but Speedy has neither made it back sucessfully, nor has he called for repairs. Speedy would have returned to base if he could have. Furthermore, if at any point Speedy's armor integrity dropped below 0, Speedy would have called for emergency repairs. Since neither of these things happened, what happened to Speedy? Give an example of a graph in which this situation could occur.

Hint: Read "Runaround" from Asimov's *I Robot*.

(d) [12 points] Thanks to your answer in part (c), Powell and Donovan have some idea of what could have happened to Speedy. Now they just need to find him. They know Speedy started his trip at base B, with an initial integrity value of C. Design an $O(VE)$-time algorithm that outputs exactly the set of vertices where Speedy could be.

Solution: a. If a vertex v has a cost, c, then for all edges $u \to v$, the cost of the edge becomes c.

b. Modified Bellman-Ford.

1. Init all verticies to $-\infty$. Change the start vertex to be C. These numbers represent the integrity available at each node.

2. Run BF loop V-1 times to find the shortest path to all nodes, and relax edges with the following modification: If the relaxation tries to update to a negative number, dont update (you cant get to that node).
3. You can reach B, if the final value there is ≥ 0.
4. If the final value at T is not ≥ 0, relax the edges again. If value at T changes, you have a positive cycle, and you can still get there.

c. Speedy must be zero cost cycle, but doesn't have enough armor integrity to reach his destination.

d. Compute reachable nodes and edges by traversing the graph using the algorithm from part b. Make a copy of this graph that only includes these edges and nodes. Note that there are no positive weight cycles in this reachable set. If there were, Speedy would have reached his destination, and you wouldn't be in this mess.Take all such edges $u \to v$, where $bestDistanceTo(u) + weight(u, v) = bestDistanceTo(v)$, and copy them over to a new graph G'. Use DFS to check G' for cycles. Output nodes involved in cycles in G'.

Problem 4-2. [25 points] **It's Hanna Barbera Time!**

Jerry wants to get to the fridge f from his starting location s, without being caught by Tom. Unfortunately, Jerry is a small mouse, and can only travel along a very specific graph of edges in the kitchen. Each of these edges e costs a certain amount of effort, $w(e) > 0$. A subset $H \subseteq V$ of the vertices in this graph make good hiding places, where Tom cannot see Jerry. Other vertices (in $V \setminus H$) leave Jerry exposed. Tom, meanwhile is patrolling the house, and glances into the kitchen once every three minutes. It takes Jerry a single minute to traverse any edge in the graph. To avoid Tom's watchful eyes, Jerry must be at a hiding spot in H on every third minute. Thankfully, the fridge $f \in H$ is a hiding spot.

In other words, at start time, Jerry can traverse three edges and must wind up in a hiding spot in H after traversing these three edges. He can then traverse three more edges, and again must wind up in a hiding spot in H, and so on, until he reaches the fridge f.

(a) [10 points] First assume that the graph forms an $X \times Y$ grid graph, where $X \cdot Y = |V|$, each vertex has coordinates (x, y) where $0 \leq x \leq X$ and $0 \leq y \leq Y$, and edges connect all horizontally and vertically adjacent vertices. Further assume that the starting point s is in the southwest corner, $(0, 0)$, and fridge f is in the northeast corner of the kitchen, (X, Y). At each minute, Jerry may only go one step north or one step east. Each such traversal requires a positive effort given by edge-weight function w. Jerry may not wait in any location. How can Jerry find the least-effort path from his starting location s to the fridge f, whilst also not getting caught by Tom? Your algorithm should run in $O(V + E)$ time.

(b) [15 points] Now consider the general case of an arbitrary graph $G = (V, E, w)$ and vertices $s, f \in V$. In addition, suppose that Jerry can now also wait at a vertex for as many minutes as he wants. (In other words, Jerry can now traverse at most three edges

between safe hiding places.) Help Jerry find the least-effort path to the fridge f from his starting location s, whilst still not getting caught by Tom. Your algorithm should run in $O(V \log V + E)$ time.

Hint: Formulate the problem as a graph problem, but not with the obvious graph. What is relevant in this problem is not just Jerry's location, but the pair of Jerry's location and how many minutes are left before Tom glances over.

Solution: a. BFS with branch and bound. Traverse three layers into the BFS, keeping only the lowest cost path that lets you arrive at a given hiding spot node in the third layer. Repeat until you reach the fridge.

b. Modify the graph to G1, G2, and H. For every node v, in the original graph G, let there be a node v1 in G1, and v2 in G2. For every good ingind spot node v, in the original graph C, let there be a node vh in H. For every edge u v in the original graph, let there be an edge from u1 to v2. If v is a good hiding spot, let there be an edge from u2 to vh. If u is a good hiding spot. Let there be an edge from uh to v1. Connect f1, f2, and fh, to a new node F with edge costs of 0 each. Compute shortest paths with effort costs as edge weights from s1 to F.

Problem 4-3. [15 points] **Rubinfeld's Cube**

You are given an undirected graph $G = (V, E)$ in which each vertex represents a unique configuration of an $r \times r \times r$ Rubik's cube, and each edge represents a ***move***, that is, a 90° rotation of one slab of the cube. (The graph is given to you, so you don't need to worry about the details of moves in Rubik's cubes.)

(a) [5 points] Given a specific configuration $c \in V$, design an $O(V + E)$-time algorithm that finds a minimum-length sequence of configurations that solves the cube.

(b) [10 points] Design an $O(VE)$-time algorithm to preprocess G that will then allow you, given two configurations c_1 and c_2, to output in $O(k)$ time a minimum-length sequence of configurations that transforms c_1 into c_2, where k is the length of this shortest sequence.

Solution: a) Because the configuration graph is unweighted, BFS will calculate all shortest paths from a given configuration node c to all other configuration nodes. Then you can backtrack the path from solved configuration node to the root node and output all the encountered nodes on the path. Worst case running time of BFS is $O(V + E)$ and worst case running time of backtracking is $O(V)$. Therefore total running time of the algorithm is $O(V + E)$.

b) To be able to obtain the data you need to answer a query, you need to compute the shortest paths between each pair of configurations and just output that path when the query arrives.

To be able to do that, you will maintain a 2D array A where $A[i][j]$ contains the index of the parent of node c_j in the shortest path tree rooted at node c_i. Shortest path trees

can be obtained with BFS using adjacency list representation in $O(E)$ time per run, running each node as source once. During BFS, you will fill each cell in your table once and not change it afterwards, therefore you will make at most $O(E)$ extra work in each BFS run. Therefore, total running time of the pre-processing algorithm is $O(VE)$.

When query (c_i, c_j) arrives, you will go to $A[i][j]$, output its content, and jump to the $A[i][A[i][j]]$. You will repeat this until you reach to $A[i][i]$. This will take $O(k)$ time because at each jump, you are progressing to the root one node at a time and distance to the root is k so you will run in $O(k)$ time. If you want to output the path in order, you may first extract the path to an array and then print the array in reverse order.

Part B

Problem 4-4. [30 points] **kth shortest paths**

You would like to estimate the latency of search queries to the new 6006LE search engine that is running over a distributed network of n servers. Each server has a direct connection to some subset of servers in the network; these connections are represented by the edges E of a given graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$ is the set of n servers. You are also given the latency $L(u, v)$ for each server connection $(u, v) \in E$. Note that the links in our network may be asymmetric, so $L(u, v) \neq L(v, u)$. Computing the latency from a server u to a server v corresponds to computing the shortest path from u to v in G with respect to the latency function L.

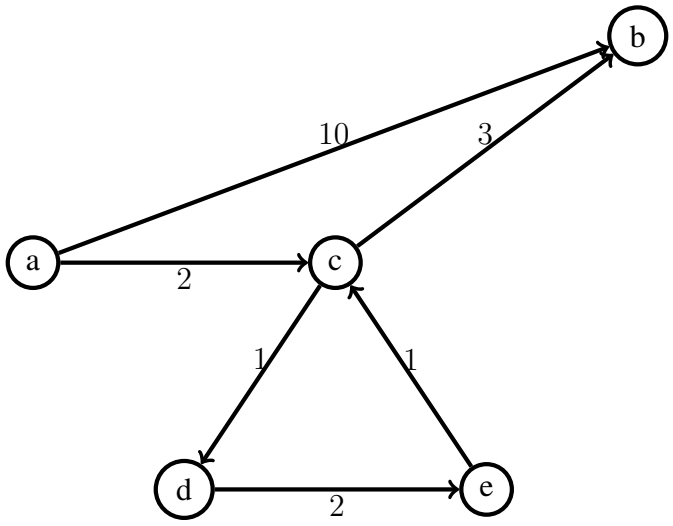
(a) [10 points] Implement the Floyd-Warshall algorithm to solve this all-pairs shortest-paths problem in the `latencies(N, L)` method. Your inputs will be N, the number of servers (or vertices) in the network, and the latency function L, which takes two servers u, v and returns the real-valued, positive latency of the connection from server u to server v. If there is no direct connection from u to v (i.e., $(u, v) \notin E$), then $L(u, v)$ will return $+\infty$. You should return an $N \times N$ matrix A such that A_{ij} is the latency from server i to server j.

Solution: The following function takes as input the number of nodes N and the latency function L. It returns an $N \times N$ matrix A containing the smallest possible latencies for any pair of servers. We use the standard Floyd-Warshall algorithm.

```
1  def floyd_warshall(N, L, A):
2      for i in range(N):
3          for j in range(N):
4              A[i][j] = L(i,j)
5
6      for k in range(N):
7          # can only use first k vertices
8          for i in range(N):
9              for j in range(N):
10                 A[i][j] = min(A[i][k] + A[k][j], A[i][j])
11     return A
```

Unfortunately, you realize that the 6006LE network can be unreliable, and that the routing algorithm that decides where to forward each packet can sometimes make mistakes. You decide to make a more conservative estimate of the latencies between servers, by computing not the shortest path, but the *second-shortest* path between each pair of servers u and v. We define the ***second-shortest path*** from u to v to be the path of minimum latency among all paths from u to v that are distinct from the shortest path. This definition implies that, if there are two paths from u to v of minimal latency, then both the shortest path and the second shortest path have the same latency. Also recall that paths can have repeated vertices, so the second shortest path may contain a cycle.

For example, consider the network of servers shown below.



In this case, the shortest path from a to b is $a \to c \to b$, with latency 5, and the second shortest path from a to b is $a \to c \to d \to e \to c \to b$, with latency 9. Furthermore, the shortest path from a to c is $a \to c$, with latency 2, and the second shortest path from a to c is $a \to c \to d \to e \to c$, with latency 6.

(b) [20 points] Implement an algorithm to solve this all-pairs second-shortest-paths problem in the `conservative_latencies(N,L)` method, with the same inputs as the method from the previous part. This time, you should return an $N \times N$ matrix B, such that B_{ij} is the latency of the second shortest path from server i to server j. Your algorithm should run in $O(N^3)$ time.

Hint: This is a pretty tricky algorithm to get right, so we'll walk you through the general steps. First modify your Floyd-Warshall algorithm from the previous part to maintain two matrices, A and B, with A storing the solutions to all of the subproblems of shortest paths, and B storing the solutions to all of the subproblems of second-shortest paths. In each iteration of the algorithm, for each pair of servers i and j, you should now update both the shortest and the second shortest path from i to j among all paths which use only the first k vertices. What is the correct relaxation recurrence when calculating $B(i, j, k)$ in general? What about when either $k = i$ or $k = j$? How will this algorithm handle the case of multiple shortest paths?

Hint: Just modifying the relaxation recurrence might not detect whether there exists a second-shortest path which is simply the shortest path plus a cycle around one of the vertices on this path. First, modify your Floyd-Warshall implementation to store the computed shortest path from server i to server j, for each i and j, in addition to the latencies. Then, after you have completed your Floyd-Warshall pass, you will need to make sure that you've actually found the second shortest path. Specifically, for each pair (i, j), you will need to iterate through each vertex v on the computed shortest path from i to j, and check whether the shortest-path length plus the shortest

nontrivial cycle around v is less than the computed second shortest path from i to j. How will you know the length of the shortest nontrivial cycle around v?

Solution: The following function takes as input the number of nodes N and the latency function L. It returns an $N \times N$ matrix A containing the smallest possible latencies for any pair of servers, and a matrix B containing the smallest possible latencies using a different that the shortest path. If there are more than one shortest path of equal length from i to j, then $A[i][j] = B[i][j]$.

Algorithm idea:

For this part, we are using a modification of Floyd-Warshall's algorithm. The main idea is the following: when a new node k is considered as an intermediate node for the paths, for each pair of nodes (i, j) we first consider the concatenation of the paths $i \to k$ and $k \to j$ using nodes with index up to $k - 1$ as in the Floyd-Warshall's algorithm. If the shortest path gets updated, we are trying to update the second shortest path which either takes the value of the previous shortest path or, the sum of the previous second shortest $i \to k$ path with the shortest $k \to j$ or vice versa. In the case where $A[i][k] + A[k][j] = A[i][j]$ since we have now found 2 distinct shortest $i \to j$ paths. So, the value of the second shortest should now equal the value of the shortest. Finally, when $A[i][k] + A[k][j] > A[i][j]$ we still need to compare $A[i][k] + A[k][j]$ to $B[i][j]$ and potentially update that value.

Remark 1:

One edge case we need to be careful about is when $k == iork == j$ in some iteration of the 3 nested for loops. In that case, it will always be true that $A[i][k] + A[k][j] = A[i][j]$, but we do not want that to count as a distinct shortest path. The only distinct shortest path we may find is one that includes a 0-weight cycle that contains either i or j (whichever of the two is equal to k and is now allowed as an intermediate node of the path). Note that the shortest path from some node to itself is always the empty path of length 0 since there are no negative weight cycles. The second shortest path for node v should be stored in $B[v][v]$ and is equal to the shortest non-empty cycle containing v.

Remark 2:

One more case we need to take care of is when the second shortest $i \to j$ path is the same as the shortest with a loop in one of the intermediate nodes. Note that the algorithm we have described above will find this path if and only if the highest index node in that loop does not belong to the shortest $i \to j$ path. If it does, then the first and second shortest path cannot be found until this node is considered, and even when this node is considered, the second shortest path is a concatenation of 3 previous paths ($i \to j$,$k \to k$(cycle) and $k \to j$). So, it cannot be found by our algorithm, which concatenates only pairs of paths. The solution to this, is to either run every iteration in the 3 nested loops twice, or do an extra pass at the end to see if a cycle can be added in any of the intermediate nodes of the any $i \to j$ shortest path to improve the second

shortest $i \to j$ path.

The code follows below:

```
    # YOUR CODE HERE
    A = [[0 for x in range(N)] for y in range(N)]
    B = [[0 for x in range(N)] for y in range(N)]
    next = [[0 for x in range(N)] for y in range(N)]
    A,B = mod_floyd_warshall(N,L,A,B,next)
    return B


def mod_floyd_warshall(N,L,A,B,next):
    for i in range(N):
        for j in range(N):
            A[i][j] = L(i,j)
            if L(i,j)!=float('inf'):
              next[i][j]=j
            else:
              next[i][j]=-1
            B[i][j] = float('inf')

    for k in range(N):
        # can only use first k vertices
        for i in range(N):
            for j in range(N):
              if k==i or k==j:
                B[i][j]=min(B[i][j],B[i][i]+A[i][j],B[j][j]+A[i][j])
              else:
                if A[i][k] + A[k][j] < A[i][j]:
                    B[i][j] = min(A[i][j],B[i][k] + A[k][j], A[i][k]
                        + B[k][j])
                    A[i][j] = A[i][k] + A[k][j]
                    next[i][j]=next[i][k]
                else:
                    B[i][j] = min(B[i][j],A[i][k] + A[k][j])
    for i in range(N):
       for j in range(N):
          if next[i][j]==-1:
            continue
          else:
            curr=i
            while curr!=j:
              if A[i][j]+B[curr][curr]<B[i][j]:
                 B[i][j]=A[i][j]+B[curr][curr]
              curr=next[curr][j]
            if A[i][j]+B[j][j]<B[i][j]:
                 B[i][j]=A[i][j]+B[j][j]
    return A,B
```

(c) [10 points] **(Extra credit, no collaboration)** Design and analyze an efficient algo-

rithm for solving the all-pairs kth-shortest-path problem for general k.

Note that no collaboration is allowed for this extra credit part.

Solution: We will use a modification of Dijksta's algorithm to find the k-th single source shortest paths from every vertex of the graph and combine those solutions to get the required answers for the k-th all pairs shortest path.

For the modified version of Dijkstra, each node will be allowed to enter the priority queue (i.e a min-heap) at most k times (this is consistent to the regular Dijkstra, where it can only enter once). We will keep a counter c_v fora each node $v \in V$ which we will increment by 1 each time a node is added to the heap. The counters are initialized to 0 for every node except the source for which we initialize it to 1. For every node i, we store k distances: $d_{i,c}, c \in \{1, .., k\}$ which are all initialized to $+\infty$, except for $d_{s,1}$, which is initialized to 0 corresponding trivially to the (empty) shortest path from the source to itself. Each time we pop a node u from the queue using the EXTRACT-MIN operation, we relax the edges in $\{\{u, v\} : \{u, v\} \in E\}$. We add a new instance of node v to the heap if and only if $d_{v,k} = +\infty$ (equivalently, $c_v \leq k$) in which case we set d_{v,c_v} to the appropriate value and increment c_v by 1. Otherwise, we have to decrease the maximum key corresponding to node v in the min heap to the appropriate value. Finding this maximum can take $O(k)$ time

Note that the value $d_{v,i}$ does not correspond to the i-th shortest path from s to v when the algorithm terminates. The value of the i-th shortest path is the key that was popped from the queue along with node v, when v was popped for the i-th time.

Runtime analysis: Each single source k-th shortest path algorithm that we are running takes time $O(k|V| \log(k|V|) + |E|k^2)$ since the algorithm requires $O(k|V|)$ EXTRACT-MIN operations in a heap of size $O(k|V|)$ and $O(k|E|)$ DECREASE-KEY operations for each of which we spend $O(k)$ time to find the maximum. The last part can be improved to $O(\log k)$ using a max heap bringing the overall running time down to $O(k|V| \log(k|V|) + |E|k \log k)$. Since we are running the algorithm $|V|$ times, we get a running time of $O(k|V|^2 \log(k|V|) + |V| \cdot |E|k \log k)$ for all pairs k-th shortest paths.