CS 341, Winter 2019
Trevor Brown, Semih Salihoglu, Doug Stinson

Assignment 5 (due Sunday, April 7th, 6:00pm)

Instructions:

- Hand in your assignment using Crowdmark. Detailed instructions are on the course website.
- Give complete legible solutions to all questions.
- Your answers will be marked for clarity as well as correctness.
- For any algorithm you present, you should justify its correctness (if it is not obvious) and analyze the complexity.

1. [*20 marks*] *Shortest Paths* Consider another variant of single source shortest paths, where you are given a directed graph G (could be undirected as well), a source s, and edge weights that are positive integers from 1 to 10. Describe an $O(n+m)$ time algorithm to solve this version of the problem.

   [Hint: Consider a faster implementation of the main data structure used in Dijkstra's algorithm from class.]

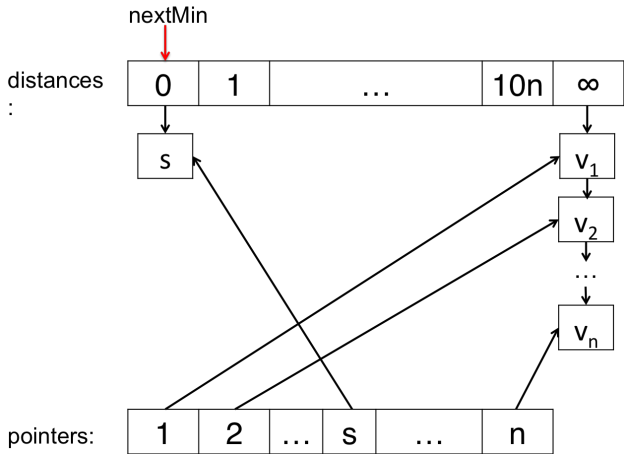
   Solution: Note that if a path from s to a vertex v exists, the shortest $s \rightsquigarrow v$ path distance will be between 1 to $10n$. Based on this observation, we implement the following priority queue data structure that consists of three pieces:

   - $distSoFar$: An array (instead of a heap) of size $10n+1$, where the last cell is used for vertices whose known distances are ∞. Each cell of $distSoFar$ keeps a linked list of vertices whose known distances are the index of the cell. For example, in the beginning of the algorithm, each node, except s will be in the cell $10n+1$, which represents ∞.
   - $nextMin$: A pointer on $distSoFar$ that keeps track of the next possible minimum distance vertex in the queue.
   - $pointers$: An array of size n, whose i'th cell points to the location of vertex i if vertex v_i is still in the $distSoFar$ priority queue or null if Dijkstra's algorithm has already found the correct distance to v_i.

   For the purpose of this solution, we have as reference the pseudocode of Dijkstra's algorithm that can be found on slide 24 of Semih's lecture 17 slides.

   We next describe the two operations that Dijkstra's algorithm executes on the priority queue:

   - DECREMENT(v_i, $newDist$): This is the operation to update the distance of v_i to a new distance $newDist$. We can implement this in $O(1)$ time by locating the position of v_i in the linked list for v_i's old distance, removing v_i from that list and adding v_i to the linked list at cell $newDist$ (and updating v_i's pointer in $pointers$ array).



- EXTRACTMIN: Notice that because the distances are positive, each time the algorithm extracts a min from the priority queue, the min distance across all remaining nodes in $distSoFar$ increases. So we can implement this by incrementing the $nextMin$ pointer to the right until we find a non-null linked list and removing the first node there (and updating that nodes' pointer to null in the *pointer* array). Therefore throughout the execution of the algorithm the EXTRACTMIN operation will take $10n$, so $O(n)$ time.

Since decrementKey is an $O(1)$ operation, the algorithm will take at most $O(n+m)$ time as it does at most $O(m)$ decrement key operations and $O(n)$ extract min operations which we argued takes $O(n)$ time throughout the execution of the algorithm.

2. [*25 marks*] *NP-Completeness* Consider the following *optimization problem* K-CLOSEST-SUBSET (KCS-OPT):

   Input: n positive integers $S : \{a_1, \ldots, a_n\}$, a number k, and a positive integer W.
   Output: a subset $T \subseteq S$ with exactly k elements minimizing $|\sum_{a_i \in T} a_i - W|$.

   Let KCS-OPTVAL be the *optimal value* version of KCS-OPT, i.e. KCS-OPTVAL is the problem where you don't have to return the optimal set T but only the value $|\sum_{a_i \in T} a_i - W|$.

   (a) [*10 marks*] Turn KCS-OPT into a decision problem KCS-DEC and show that the decision problem is in NP.

   Solution: Define the decision problem KCS-DEC:

   Input: n positive integers $a_1, \ldots, a_n$, a number k, a positive integer W, and a target value V.
   Output: YES iff there is a subset $T \subseteq S$ with exactly k elements where $|\sum_{a_i \in T} a_i - W| \leq V$.

   KCS-DEC is in NP because if there is an instance of I of KCS-DEC whose answer is YES, one can show a short $O(n)$ certificate that consists of a $T \subseteq S$. Then we can verify that T contains exactly k elements and that $|\sum_{a_i \in T} a_i - W| \leq V$, simply by summing all the elements in T in $O(n)$ time.

(b) [*7 marks*] Show that if you can solve KCS-DEC in polynomial time, then you can solve KCS-OPTVAL in polynomial time.

Solution: Let A be an algorithm for KCS-DEC that takes a polynomial time in the size of the input. Then given an instance I to KCS-OPTVAL we can find the answer, i.e., the optimal value, as follows. We know that $\{a_1, ..., a_k\}$ is a feasible (but possibly not the optimal) solution to I and has difference $z = |(a_1 + ... + a_k) - W|$. So the optimal solution has to be between $[0, z]$. So we run $A(S, V = z), A(S, V = z/2), A(S, V = z/4)...$ and through binary-search find exactly the point where the answer switches from YES to NO. We will call this binary-search algorithm B and use in the solution to part (c) of this question. This will take at most $log(z)$ iterations and each execution of A will take poly-time in the input size. The total run time will be poly-time, as $log(z)$ must be a polynomial in terms of the number of bits to represent the input. Note that z is the addition/subtraction of $k + 1$ numbers in the input. Although numeric value of z can be large, e.g., z could be 2^{1000} which would take only 1000 bits to represent, but $log(z)$ will be polynomial in the input.

(c) [*8 marks*] Show that if you can solve KCS-DEC in polynomial time, then you can solve KCS-OPT in poloynomial time.

Solution:

Let A and B be respectively the poly-time algorithms for KCS-DEC and KCS-OPTVAL. Let *opt* be the solution to KCS-OPTVAL, i.e., $B(S, k, W)$. We design a poly-time algorithm C to solve KCS-OPT. C constructs a T one by one by finding an element that changes *opt* by removing an element from S, which we can check by comparing the value of $B(S', k, W)$ to *opt* (or checking if $A(S', k, W, opt)$ is still YES). If a removed element a_j changes the optimal value, it must be in T, otherwise we can safely remove it from S because either it's not in the optimal solution (or there is an alternative optimal solution that does not contain a_j if the optimal solution is not unique). The algorithm keeps removing items this way one by one. Note that this algorithm is guaranteed to return exactly k elements that achieves *opt* (prove this by a simple contradiction as an exercise, i.e., show that if in the final output there was more than $k + 1$ elements C would have removed one of the elements earlier).

```
C(S, k, W):
1. opt = B(S, k, W)
2. while k > 0:
3.     S.remove(a_i)
4.     // if the opt value changes we know a_i must be in T; we add a_i back to S
5.     if (A(S, k, W) > opt):
6.         S.add(a_i)
7. return S
```

3. [*25 marks*] *NP-Completeness* Consider the SINK-SOURCE SUBGRAPH (SSS) problem:

Input: a directed graph $G(V, E)$.
Output: "YES" iff there is a subgraph $H(V, E')$, with $E' \subseteq E$, such that each $v \in V$ satisfies one of two conditions: (i) either, in-degree$(v) = 0$ and out-degree$(v) > 0$; or (ii) out-degree$(v) = 0$ and in-degree$(v) > 0$. The in/out degree conditions are within H.

Let us call a subgraph H satisfying the above property *sink-source* subgraph. Observe that in a sink-source subgraph, there are no paths of length 2, since every vertex either has zero in-degree or zero out-degree.

(a) [*5 marks*] Prove that SSS is in NP.
Solution: We use the abbreviation SSS to refer to both the decision problem defined about and the sink-source subgraph H in an input graph G. Suppose an instance I of SSS has an answer YES, i.e., the input $G(V, E)$ has an SSS H. Then we can verify this by the subgraph H and checking that each vertex either has non-zero out-degree and 0 in-degree or zero out-degree and non-zero in-degree. This can be done in $O(n + m)$ time.

(b) [*20 marks*] Prove that SSS is NP-Complete through a reduction from 3SAT. Remember to have an iff argument in your reduction.
[Hint: Consider having one vertex u_i for each clause C_i in the 3SAT formula. Similarly have one vertex v_j for each variable x_j in the 3SAT formula. If x_j appears in C_i have an edge from v_j to u_i. If $\overline{x_j}$ appears in C_i, add another edge from u_i to v_j. Add another vertex t to the graph and add edges from each v_j to t. Finally, add one more node s and add an edge from s to t.]
Solution: Given an instance ϕ of 3SAT with m clauses $C_1, \ldots, C_m$ and n variables, we construct $G(V, E)$ as suggested in the hint: $V = \{u_1, \ldots, u_m, v_1, \ldots, v_n, s, t\}$. In G, t will always be a node that has out-degree 0 and in-degree > 0 and s a node with in-degree 0 and out-degree > 0. $E = E_T \cup E_F \cup E_{TF} \cup s \to t$ where:

- E_T (for "True literal" edges) are from v_j to u_i if x_j appears in C_i. These will represent in a particular satisfying assignment A to ϕ, x_j takes value True and makes C_i True (so literal x_j is "responsible" for C_i being true).
- E_F (for "False literal" edges) are from u_i to v_j if $\overline{x_j}$ appears in C_i. These represent that when in a particular satisfying assignment A to ϕ, x_j takes value False (so $\overline{x_j}$ is True) and makes C_i True (so literal $\overline{x_j}$ is "responsible" for C_i being true).
- E_{TF} (for "True or False literal" edges) are from each v_j to t. These represent that when in a particular satisfying assignment A to ϕ x_j can take either True or False, we make these point to t (so literals x_j or $\overline{x_j}$ are not "responsible" for any clause being true).
- $s \to t$ is the single edge from s to t to ensure that when each variable is forced to take a T or F value in satisfying assignment at least one other node points to t.

Correctness: We next prove that this reduction is correct, i.e., given an arbitrary ϕ as input to $3SAT$, ϕ is satisfiable iff G (constructed out of ϕ) has an SSS H.
$\to$: Suppose there is a satisfying assignment A (e.g., $(x_1 : T, x_2 : F, x_3 = T..., x_n = F)$) for ϕ. We construct an SSS $H(V, E')$ as follows. First, for each C_i we isolate exactly one

"responsible" literal (an x_j or $\overline{x_j}$) that makes C_i true. If the literal we isolate is x_j, we add the edge $v_j \to u_i$ to E'. Otherwise, if the literal we isolate is $\overline{x_j}$, we add the edge $u_i \to v_j$ to E'. If a particular variable x_j has not been isolated, we add the edge $x_j \to t$ to E'. Finally we add the edge $s \to t$ to E'. Note that H is an SSS because:

- Each u_i has exactly 1 edge (so either the in or out-degree is 0 and the other is > 0).
- Each v_j has exactly 1 edge (either an incoming from a u_i or an outgoing to a u_i or to t).
- t has at least 1 incoming edge (from s) no outgoing edge.
- s has exactly 1 outgoing edge and no incoming edge.

$\leftarrow$. Suppose there is an SSS $H(V, E')$ in G. We construct a satisfying assignment A as follows. If a v_j points to a u_i (representing a clause C_i), we set $x_j = T$. If instead a u_i points to v_j we set $x_j = F$. If v_j points to t we set it to T or F arbitrarily. Note that A must satisfy ϕ because consider a clause C_i. C_i is represented by a vertex u_i and there are 2 cases:

- u_i has > 0 incoming edges (not necessarily exactly 1) in E'. These incoming edges are coming from some nodes v_j and each of these edges existed in G because the literal x_j existed in C_i. Note that we set x_j to T in this case (recall if v_j points to a u_i in H we gave set it to T in the assignment A). So C_i is satisfied.
- Or u_i has > 0 outgoing edges in E', and those edges must point to some v_j. Similarly these edges existed in G because the literal $\overline{x_j}$ existed in C_i and we set x_j to F in A, so C_i is again satisfied.

4. [*30 marks*] *Programming Question* Implement Bellman Ford's single source shortest paths algorithm with several optimizations (described below). Given a directed graph $G(V, E)$ with arbitrary edge weights, and a source s (which will be vertex 0), you should return one of the two possible outputs:

   (1) If the graph contains a negative weight cycle, you should output the string: "NEGATIVE WEIGHT CYCLE".

   (2) If there are no negative weight cycles in G, you should output the distances (not the paths) from s to all other nodes (including s) in increasing order of IDs in separate lines in the "ID distance" format, where you output "inf" if the source does not have a path to a particular node v:

   0 0
   1 -3
   2 inf
   ...

   The input format will be as follows. The first line will contain 3 integers "n m s", where n represents the number of nodes in the graph, m the number of edges, and s is the source node and is always 0. Then, m lines will follow with 3 integers "u v w" which represent one edge from node u to v with weight w. For simplicity the weights will be integers, but they can be negative, zero, or positive. The nodes will be in the range $[0, n-1]$ and some nodes

may not have any incoming or outgoing edges, so may not appear in the input file (so your output should be "inf" for them).

In order to get your implementations working correctly and efficiently you will need to implement three optimizations:

(a) *Early stopping:* Recall that if the BF algorithm converges at iteration $i < n$, i.e., the shortest distance of each vertex v at iteration $i-1$ is the same as i, then you can stop and conclude that the distances at iteration i (and $i-1$) are the correct shortest path distances and there are no negative weight cycles in G.

(b) *Space optimization:* Recall that in each iteration i, the algorithm requires the distances of vertices only from iteration $i-1$. So you only need $2n$ space to keep the "current" iteration's distances and "previous" iteration's distances.

(c) *Negative weight cycle checking:* Recall that if there is a negative weight cycle in G, the BF algorithm will not converge. You can check if there is a negative weight cycle simply by checking that the distances in the n'th iteration is not the same as the distances at the $n-1$'th iteration, i.e., there is at least one vertex v, whose distance decreases from iteration $n-1$ to n. If you find such a vertex you should output "NEGATIVE WEIGHT CYCLE". (Note: because this check requires running the algorithm n iterations, we will give only some small input graphs that contain negative weight cycles in the tests.)

Make sure your code runs in a reasonable amount of time. Each test case will have a generous timeout deadline.

The submission guidelines follow from the previous programming question. Submit one zip file through Marmoset: https://marmoset.student.cs.uwaterloo.ca/ which must contain your code file named a5q4.py/a5q4.cpp. You can use only C++ or Python as Programming Languages and your program should read from standard input and write to standard output.

Solution: Omitted