*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology — March 22, 2019
Instructors: Jason Ku, Vinod Vaikuntanathan, and Virginia Williams — Problem Set 6

# Problem Set 6

**All parts are due on April 5, 2019 at 6PM**. 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 `alg.mit.edu`.

**Problem 6-1.** [10 points] **Topological Tournament**

Eight soccer teams have qualified for the Global Goblet Soccer Tournament:

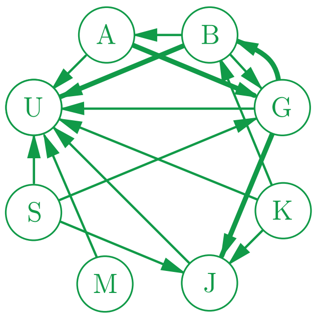
{**A**ustralia, **B**razil, **G**ermany, **K**orea, **J**apan, **M**orocco, **S**outh Africa, **U**SA}

Below is a list of games that were played between them during the season. A game where team $x$ won against team $y$ is represented by the ordered pair $(x, y)$.

```
games = [
    (S, U), (K, J), (B, A), (S, J), (K, B), (B, U), (S, G), (K, U),
    (B, G), (G, J), (A, G), (J, U), (G, U), (A, U), (M, U), (G, B),
]
```

**(a)** [2 points] Draw a directed graph $G$ on the teams where each game $(x, y)$ corresponds to a directed edge from team $x$ to team $y$.

**Solution:**



**Rubric:**

- 2 points for a correct graph
- Partial credit may be awarded

**(b)** [5 points] Run Full DFS on $G$ starting from Australia, and return the list of teams in reverse order of their DFS finishing times. Whenever there is ambiguity over which team to search next, break ties alphabetically. Is the returned list a topological sort?

**Solution:** DFS finishes vertices in order: `[U,B,J,G,A,K,M,S]`, the reverse of which is `[S,M,K,A,G,J,B,U]`. This is not a topological sort because edges `(B,G)` and `(B,A)` both go against this order.

**Rubric:**

- 4 points for a correct finishing time order
- -1 point per inversion from correct, minimum zero points
- 1 point for argument that list is not a topological sort

**(c)** [3 points] A game $(x, y)$ **goes against** an ordering of teams if $y$ appears before $x$ in the ordering. Use Full DFS to find an ordering of the teams which minimizes the number of games that goes against the ordering.

**Solution:** Because the graph contains a cycle, at least one game must go against any ordering. It is possible to find an ordering of the games such that only one edge goes against the ordering. In particular, running Full DFS from vertex `K` achieves the ordering `[S,M,K,B,A,G,J,U]` for which only one edge `(G,B)` goes against the ordering.

**Rubric:**

- 2 points for an ordering where only one edge goes against the ordering
- 1 point for identifying a vertex from which Full DFS provides such an ordering

**Problem 6-2.** [10 points] **Limited Improvement**

A weighted directed graph $G = (V, E, w)$ is ***k*-limited from vertex *s*** $\in V$ if every vertex $v \in V$ has a minimum-weight path from $s$ that traverses at most $k$ edges. Given a weighted directed graph that is $k$-limited from $s$, describe an algorithm that solves weighted single-source shortest-paths from $s$ in $O(|V| + k|E|)$ time.

**Solution:** Perform Bellman-Ford, but stop after $k$ rounds instead of $|V| - 1$. This requires $O(|V| + |E|)$ time for initialization and $O(k|E|)$ time for the $k$ rounds, for a total of $O(|V| + k|E|)$ time.

To prove correctness, first observe that every node *has* a shortest path traversing at most $k$ edges, so $\delta(s, v)$ is finite for every $v$, and there are no negative weight cycles in the graph. Then by Lemma 1 from Recitation 12 notes, at the end of relaxation round $i$, $d(s, v) = \delta(s, v)$ for any vertex $v$ that has a shortest path from $s$ to $v$ which traverses at most $i$ edges. Then at the end of round $k$, $d(s, v) = \delta(s, v)$ for every vertex, so this algorithm is correct.

**Rubric:**

- 6 points for description of a correct $O(|V| + k|E|)$-time algorithm
- 2 points for a correct argument of correctness
- 2 points for a correct running time analysis

**Problem 6-3.** [10 points] **Token Jumping**

Token Jumping is a token game played on a weighted directed graph $G = (V, E, w)$, where each edge weight is either a positive or negative integer. A token is placed on a starting vertex $s$, with an integer **score** that is initially zero. The token may **jump** around the graph along directed edges. Whenever the token jumps along a directed edge, the token adds the edge's weight to its score. Let $k > 0$ be the fewest jumps that the token needs to reach a target vertex $t$ from $s$ (assume that $t$ is reachable from $s$). Describe an efficient algorithm to determine whether the token can reach $t$ in exactly $k$ jumps, arriving at $t$ with a positive score.

**Solution:** This problem asks us to answer: from among all paths from $s$ to $t$ traversing exactly $k$ edges, does any such path have positive weight/score? It suffices to find determine the maximum weight of any such path and compare that weight to zero. We had originally written this problem to target an $O(k|E|)$ running time via graph duplication and DAG relaxation (or via a modified $k$-round Bellman-Ford), but the constraint that $k$ is the **fewest jumps** needed to reach $t$ leads to a faster linear time algorithm, an approach which would not be possible without that constraint. An $O(k|E|)$-time algorithm is sufficient for full points on this problem, but we present three solutions here for completeness.

***O*(*k*|*E*|) solution (1):** First, breadth-first search from vertex $s$ on $G$ to determine $k$, the minimum number of jumps needed to reach vertex $t$, in $O(|E|)$ time since the number of vertices reachable from $s$ is $O(|E|)$. If the graph is not connected, restrict $G$ to the vertices reachable from $s$ so that it is so that $|V| = O(|E|)$. Also, since $k$ represents the minimum number of jumps in the unweighted graph, $k \leq |V|$. We cannot be assured that the route found by breadth-first search is a path of maximum weight. To find a path of maximum weight using **exactly** $k$ jumps, we use graph duplication to record how many edges were used to traverse to a given vertex. Construct a new graph $G'$ on $(k+1)|V|$ vertices, where each vertex $v \in V$ corresponds to $k-1$ copies, $v_i$ for $i \in \{0, \dots, k\}$ ($v_i$ is the copy of $v$ in the $i$th level of $G'$). Then for each directed edge $(u, v) \in E$ with weight $w$, add $k$ edges to $G'$ each with weight $-w$, specifically edge $(u_i, v_{i+1})$ for each $i \in \{0, \dots, k-1\}$. This graph has $k|V| = O(k|E|)$ vertices and $O(k|E|)$ edges, so has size $O(k(|E|))$, and is acyclic because every directed edge connects a vertex of a lower level to a higher level. Any path in $G'$ from $s_0$ to $t_k$ corresponds to a path in $G$ that performs exactly $k$ jumps, and every path in $G$ from $s$ to $t$ traversing exactly $k$ edges corresponds to a path in $G'$ from $s_0$ to $t_k$ (this statement is somewhat obvious, but in a more formal setting would need to be proven). Then to determine the **maximum** weight of any path from $s$ to $t$ in $G$ which traverses exactly $k$ edges is equivalent to finding the **minimum** weight path from $s_0$ to $t_k$ in $G'$ (negating edge weights here is valid to find a maximum length path as there are no cycles in $G'$). So run DAG relaxation from $s_0$ and return whether the negative of the weight of a shortest path to $t_k$ is greater than zero. Since DAG relaxation runs in time linear in the size of the graph, this algorithm takes $O(k|E|)$ time. Note that this algorithm would still work even if $k$ was part of the input, and not the fewest number of jumps to reach $t$. We will now exploit this property to get a faster algorithm.

***O*(*k*|*E*|) solution (2):** We can also solve this problem via Bellman-Ford. An intuitive but **incorrect** approach would be to negate all edge weights and run $k$ rounds of Bellman-Ford from $s$. While Bellman-Ford does ensure that at the end of round $i$, $d[v]$ is at most the weight of a shortest path to $v$ using at most $k$ edges for each $v \in V$, it is possible that $d[v]$ corresponds to the weight of a shortest path to $v$ using **more edges**, as one round of Bellman-Ford may relax many edges along a path. Instead, we construct a new graph $G'$ identical to $G$ but with modified weights to ensure that $d[t]$ will only correspond to smallest weight of any paths using exactly $k$ edges in $G$ after round $k$. Specifically, let $W = 3\sum_{e \in E} |w(e)|$ (think of this as a very large number). Then for edge $e \in E$ with weight $w(e)$, let its weight in $G'$ be $W - w$. Then given a path $(v_0, \dots, v_j)$ in $G$ with path weight $X_j$, its corresponding weight in $G'$ will be $jW - X - j$, where $jW - 2X_j < (j+1)W$, i.e., every path traversing $j$ edges will have lower weight than any path traversing $j+1$ edges (we chose $W$ specifically to ensure $|2X_j| < W$ for $j \leq k$). Now, we run $k$ rounds of Bellman-Ford on $G'$ from $s$ and return whether $0 < -(d[t] - kW)$. This algorithm is correct because: (1) every path to $t$ in $G'$ traversing more than $k$ edges will have larger weight than any path traversing exactly $k$ edges, (2) $k$ is the minimum number of edges of any path from $s$ to $t$, (3) round $k$ of Bellman-Ford finds the minimum weight of any path from $s$ to $t$ using at most $k$ edges. We assume that each edge weight and $|E|$ is storable in a constant number of machine words, thus so is $W$ (since $\log W < \log(3|E| \max_{e \in E} |w(e)|) = \log 3 + \log |E| + \log(\max_{e \in E} |w(e)|)$), so $G'$ has linear size with respect to $G$, so this algorithm runs in $O(k|E|)$ time.

***O*(|*E*|) solution:** Because $k$ is the fewest jumps needed to reach $t$ from $s$, any path of $k$ jumps from $s$ to $t$ visits vertices in increasing level order with respect to a breadth-first search from $s$, so such a path cannot traverse edges that go against this order. Breadth-first search from $s$ and label each vertex with its level number, and then remove all edges in $G$ which do not increase level, i.e., from some level $i$ to $i+1$. This search and filter takes $O(|E|)$ time. The resulting sub graph $G'$ is a directed acyclic graph, so negate edge weights and run DAG relaxation on $G'$ in $O(|E|)$ time to find a minimum weight path in $G$ to $t$, which corresponds to a longest weighted path in $G$ from $s$ to $t$ which traverses exactly $k$ edges as desired.

**Rubric:**

- 6 points for description of a correct algorithm
- 2 points for a correct argument of correctness
- 2 points for a correct running time analysis
- At most $O(k(|V| + |E|))$ running time is needed for full points on this problem

**Problem 6-4.** [15 points] **Acute Fruit Pursuit**

MIT student Neil Millen is an avid avocado aficionado who already ate all his avocados! All he has in his dorm room is an apple leftover from a care package from his parents. Neil badly wants some avocado, but he is too lazy to leave his dorm. He emails his dorm mailing list asking if anyone has any fruit they would be willing to trade for another.[1] Neil receives many responses, where each response is a tuple of four integers $(f_a, a, f_b, b)$ representing a student who is willing to trade $2^a$ fruit of type $f_a$ for $2^b$ fruit of type $f_b$. For example, $(\text{orange}, 2, \text{pear}, 0)$ represents a student willing to give four oranges for each pear received (though they would not necessarily be willing to give

[1]guacamole green for bc-talk

one pear for four oranges). Assume that a student is willing (and able) to trade any amount of fruit at their posted rate, including partial fruits; so the aforementioned student would also be willing to give half an orange in exchange for an eighth of a pear. Given the list of $n$ responses he received, describe an efficient algorithm to determine the maximum amount of avocado that Neil can obtain by trading his apple on the dorm fruit market.

**Solution:** Loop through the $n$ responses to determine the names of all $F$ fruits that are available to trade on the market, i.e., by using a hash table. Construct a weighted graph with a vertex for each fruit in the input, and for every response $(f_a, a, f_b, b)$, add a directed edge from fruit $f_a$ to $f_b$ with weight $a - b$. This graph has $F \leq 2n$ vertices and $n$ edges, so has size $O(n)$. Then a path in this graph from the apple vertex to the avocado vertex having weight $w$ will correspond to a sequence of trades that Neil can perform from his starting apple to achieve $2^{-w}$ avocados (or fractions of an avocado). We are asked to maximize output avocado, i.e., find the path from the apple vertex to the avocado vertex having minimum weight. If there is a negative weight cycle reachable from apple to avocado, than Neil can trade for infinite avocado! Since this graph could contain both positive and negative weights, we can use Bellman-Ford to find the weight of the shortest path from apple to guacamole in $O(n^2)$ time. As discussed in Lecture 12, Bellman-Ford can also assign $d(s,t)$ correctly to $-\infty$ if it is reachable from a negative weight cycle. So, if $d(s,t) = -\infty$, we return that Neil can obtain an unlimited amount of avocado by trading on the market. Otherwise, return $2^{-d(s,t)}$ avocados.

**Rubric:**

- 9 points for description of a correct algorithm
- 3 points for a correct argument of correctness
- 3 points for a correct running time analysis
- At most $O(n^2)$ running time is needed for full points on this problem

**Problem 6-5.** [15 points] **Cross Country Criminal**

Tim the Beaver has built a fuel efficient vehicle called the WoodWagon for his engineering capstone project. Tim decides to drive it across country to test it out. The WoodWagon can travel up to $m$ miles on a full tank of fuel before needing to refill. Not wanting to end up stranded, Tim checks a map to plan his route. Tim's map depicts all the roads and road intersections in the United States. Each of the $n$ road intersections on the map is marked with its position $p = (x, y)$; assume the US is roughly flat so that the **straight-line distance** between two road intersections $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is $d(p_2, p_1) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Each road directly connects two road intersections and is marked on the map with its length; since roads are not always straight, the marked length of a road between $p_1$ and $p_2$ may be much longer than the straight-line distance $d(p_1, p_2)$ between its endpoints. Assume each road intersection connects at most five roads. The map marks $k$ of the road intersections as having gas stations where Tim can refill. Tim is broke, so after filling up his gas tank at a station, he will leave without paying[2]! So whenever he leaves

[2]Tim is a very bad beaver. Please do not follow his example.

a gas station, Tim will only drive through road intersections that strictly increase his straight-line distance from the gas station he just left (until he fills up at the next gas station). Given Tim's map, describe an $O(nk)$-time algorithm to determine whether Tim can reach a destination intersection in California, starting with no fuel from a starting gas station in Massachusetts.

**Solution:** The map in this problem naturally defines an undirected weighted graph on the road intersections, where a road directly connecting two intersections with length $w$ corresponds to an undirected edge between them with weight $w$. However, we will not actually use this graph to solve this problem! In order for Tim to be able to reach his destination $t$ from his starting location $s$, Tim must be able to get there by traversing a sequence of gas stations, where gas station $g_{i+1}$ is reachable from gas station $g_i$ via a **valid path from $g_i$**: a path of length no more than $m$ which strictly increases straight-line distance from gas station $g_i$. So, we would like to construct an unweighted directed graph $G$ with the gas stations and target location $t$ as vertices, with a directed edge from gas station $g$ to gas station or target location $v$ if there is a valid path from $g$ to $v$. Then, we could run a breadth-first search in $G$ starting from gas station $s$, to see whether location $t$ is reachable from $s$, implying that Tim can make it to his destination without running out of gas. Graph $G$ has $k + 1$ vertices and at most $O(k^2)$ edges, so this breadth-first search would take at most $O(k^2)$ time. It remains to show how to identify the edges of $G$, i.e., identify all valid paths from $g_i$ for each gas station $g_i$. To identify the valid paths from a gas station $g_i$, construct a directed weighted graph $G_i$ on the $n$ road intersections, where a road connecting intersections $a$ and $b$ with length $w$ will correspond to either:

- a directed edge from $a$ to $b$ with weight $w$ if $d(g_i, a) < d(g_i, b)$,
- a directed edge from $b$ to $a$ with weight $w$ if $d(g_i, a) > d(g_i, b)$,
- or no edge if $d(g_i, a) = d(g_i, b)$.

Graph $G_i$ contains $n$ vertices and at most $5n = O(n)$ edges. By construction, this graph is a DAG, as no intersection may be visited twice while moving strictly away from $g_i$. So, we can use DAG relaxation to identify all gas stations reachable from $g_i$ via a valid path in this DAG with weight at most $m$ in $O(n)$ time. Repeating this process for each of the $k$ gas stations will identify the edges to include in $G$ in $O(nk)$ time, meaning the entire algorithm runs in $O(nk)$ time.

**Rubric:**

- 9 points for description of a correct $O(nk)$-time algorithm
- 3 points for a correct argument of correctness
- 3 points for a correct running time analysis

**Problem 6-6.** [40 points] **Awesome Alps**

Talent scouts are visiting the Hackson Jole ski resort, looking for new recruits! Vindsey Lonn is a burgeoning downhill skier who wants to show off her skills. She regularly trains at Hackson Jole so she knows the mountain well. On the mountain are many checkpoints. A **checkpoint** is represented by a pair $(c, h)$ corresponding to its name $c$ and its integer height $h$ above sea-level. A **ski route** connects a pair of checkpoints having **different** heights, where Vindsey can ski **downhill** from the higher checkpoint to the lower one, but cannot ski the route in the other direction. Vindsey thinks some ski routes are more awesome than others, so she has assigned an integer **awesomeness** to each ski route: specifically triple $(c_1, c_2, a)$ represents the awesomeness $a$ assigned to a ski route connecting checkpoints with names $c_1$ and $c_2$ (when Vindsey skies that route downhill). A **downhill course** is any sequence of checkpoints of decreasing height, where each adjacent pair of checkpoints is connected by a ski route along the course. The **awesomeness** of any downhill course is the sum total of awesomeness of routes along the course.

**(a)** [5 points] Below is a list $C$ of mountain checkpoints and a list $R$ of awesomeness-assigned ski routes. There are nine downhill courses on the mountain. List the three most awesome downhill courses, and indicate how awesome they are.

```
C = [                              R = [
    ("north_summit", 34),              ("alpine_cliff", "north_summit", 22),
    (  "west_bluff", 23),              ( "eagle_ridge", "north_summit", 43),
    ( "eagle_ridge", 17),              ("alpine_cliff",   "west_bluff", 12),
    ("alpine_cliff", 24),              (  "west_bluff", "north_summit", 37),
]                                      ( "eagle_ridge",   "west_bluff", 38),
                                   ]
```

**Solution:** Here are the nine downhill courses on the mountain, ordered by awesomeness:

```
12 : (a, w)
22 : (n, a)
34 : (n, a, w)
37 : (n, w)
38 : (w, e)
43 : (n, e)
50 : (a, w, e)       * three
72 : (n, a, w, e)    * most
75 : (n, w, e)       * awesome
```

**Rubric:**

- 5 points for course and awesomeness of the three most awesome
- -1 for each incorrect course or awesomeness (min zero points)

**(b)** [10 points] Given a list $C$ of checkpoints on the mountain and a list $R$ of awesomeness-assigned ski routes, describe an efficient algorithm to return any most awesome downhill course, so Vindsey can impress the talent scouts.

**Solution:** Construct a graph $G$ on checkpoint names, where for each awesomeness assignment $(c_1, c_2, a)$, add directed edge $(c_1, c_2)$ with weight $a$ if checkpoint $c_1$ is strictly higher than $c_2$, and directed edge $(c_2, c_1)$ with weight $a$ if $c_2$ is strictly higher than $c_1$. We do this by first constructing a hash table from checkpoint names to their heights, and then adding each edge by comparing two lookups into this hash table in expected constant time, so this graph can be constructed in (expected) $O(C + R)$ time. Graph $G$ is acyclic because downhill ski routes only go from higher checkpoints to lower checkpoints. The problem asks for the most awesome downhill course from among all downhill paths, so we can reduce this to a single source shortest paths problem by constructing a new graph $G'$ from $G$ by negating all edge weights and then adding two new vertices $s$ and $t$, where $s$ has a directed edge to each vertex in $G$ of weight zero, and $t$ has a directed edge from each vertex in $G$ of weight zero. Then every downhill course corresponds to a path in $G'$ from $s$ to $t$, so we can use DAG relaxation to find a shortest path in $G'$, which will correspond to a course in $G$ that is maximally awesome. $G'$ also has size $O(C + R)$, so this algorithm runs in $O(C + R)$ time. Note that heights may not be in a polynomially bounded range, so sorting the heights directly may take $O(C \log C)$ time, which is not efficient. Topological sort on $G'$ will sort the checkpoints in linear time which is more efficient.

**Rubric:**

- 6 points for description of a correct algorithm
- 2 points for a correct argument of correctness
- 2 points for a correct running time analysis
- At most $O(C + R)$ running time is needed for full points on this problem

**(c)** [25 points] Write a Python function `most_awesome(C, R)` that implements your algorithm. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

**Solution:**

```
def dfs(Adj, s, parent = None, order = None):
    if parent is None:
        order, parent = [], {}
        parent[s] = s
    for v in Adj[s]:
        if v not in parent:
            parent[v] = s
            dfs(Adj, v, parent, order)
    order.append(s)
    return parent, order

def DAG_shortest_paths(Adj, s):
    _, order = dfs(Adj, s)
    order.reverse()
    d, parent = {}, {}
    for v in order:
        d[v] = float('inf')
    d[s], parent[s] = 0, s
    for u in order:
        for v in Adj[u]:
            a = Adj[u][v]
            if d[v] > d[u] + a:
                d[v] = d[u] + a
                parent[v] = u
    return d, parent

def most_awesome(C, R):
    s, t = 's_', 't_'
    H, Adj = {}, {}
    Adj[s], Adj[t] = {}, {}
    for c, h in C:
        H[c] = h
        Adj[s][c] = 0
        Adj[c] = {t: 0}
    for c1, c2, a in R:
        h1, h2 = [H[c] for c in (c1, c2)]
        if h1 < h2:
            c1, c2 = c2, c1
        Adj[c1][c2] = -a
    _, parent = DAG_shortest_paths(Adj, s)
    course = []
    while parent[t] != s:
        t = parent[t]
        course.append(t)
    course.reverse()
    return course
```