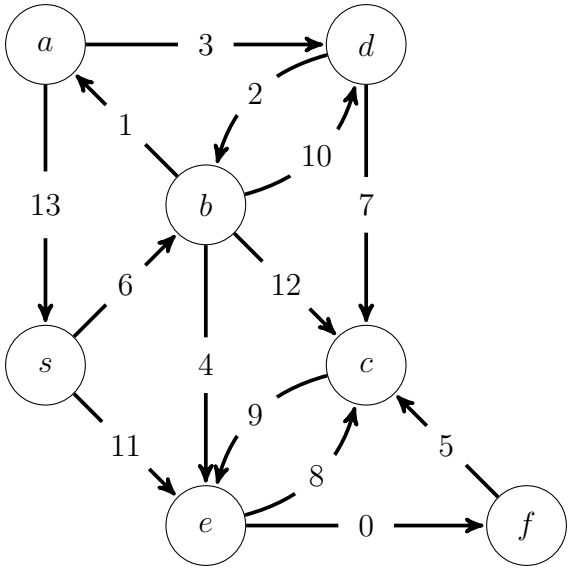
# Problem Set 8

**All parts are due on November 11, 2018 at 11PM**. 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 8-1.** [25 points] **Graph Practice**

**(a)** [8 points] Run Dijkstra on the following graph from vertex $s$ to every vertex $v \in V = \{a, b, c, d, e, f, s\}$. Write down (1) the minimum-weight path weight $\delta(s, v)$ for each vertex $v$, and (2) the order that vertices are removed from your Dijkstra queue.




**Solution:** The $\delta(s, v)$ values are as follows:

| Vertex $v$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $s$ |
|---|---|---|---|---|---|---|---|
| $\delta(s, v)$ | 7 | 6 | 15 | 10 | 10 | 10 | 0 |

These nodes are processed in increasing order of $\delta(s, v)$, but there is some flexibility in the order that $d$, $e$, and $f$ are processed: a tie arises between $d$ and $e$, and if $e$ is processed first then another tie arises between $d$ and $f$. The three possible processing orders are $sba\{def, edf, efd\}c$.

**Rubric:**

- 4 points for minimum-weight path weights
- -1 point per incorrect minimum-weight path weight, minimum zero points

- 4 points for vertex order
- -1 point per vertex order inversion, minimum zero points

**(b)** [8 points] Given an **undirected** weighted[1] graph $G = (V, E, w)$ (not necessarily connected) and a vertex $s \in V$, describe an algorithm to solve the single-source shortest path problem that runs in $O(|E| + |V| \log |V|)$ time, i.e. compute $\delta(s, v)$ for every $v \in V$, including cases where $\delta(s, v) = \pm\infty$.

**Solution:** There is a negative-weight cycle reachable from $s$ if and only if there is a negative-weight **edge** reachable from $s$, because an undirected negative-weight edge can be traversed back-and-forth and therefore **is** a negative weight cycle. (And conversely, any negative-weight cycle must have at least one negative-weight edge.)

So start by running a BFS from $s$ to identify $s$'s connected component, i.e., the subgraph $H$ of all nodes and edges reachable from $s$. Label all nodes $v$ that aren't in $H$ with $\delta(s, v) = \infty$. If some edge in $H$ has negative weight then label all vertices $u$ in $H$ with $\delta(s, u) = -\infty$ and return. Otherwise, all weights in $H$ are non-negative, so finish with Dijkstra in $H$ from $s$. Dijkstra takes $O(E + V \log V)$ time and dominates $O(V + E)$ used for the other steps. Correctness follows from the above observation.

**Rubric:**

- 8 points for a correct algorithm
- Partial credit may be awarded

**(c)** [9 points] As shown in lecture, when running Bellman-Ford on any weighted graph, every negative weight cycle in the graph contains an edge that is relaxed in round $|V|$ of Bellman-Ford. Further, the Bellman-Ford dynamic program can return for each edge relaxed in round $|V|$, a path from the source to the edge containing a negative weight cycle. Given a weighted[1] graph $G = (V, E, w)$ and two vertices $s, t \in V$ for which $\delta(s, t) = -\infty$, describe an $O(|V||E|)$-time algorithm to return a path from $s$ to $t$ that traverses a negative weight cycle.

**Solution:** Run Bellman-Ford from $s$ to find every edge $(a, b)$ relaxable in round $|V|$, along with a path to $b$ containing a negative weight cycle (by the Bellman-Ford dynamic program). Then use BFS or DFS to search from each such edge $(a, b)$ to find vertices reachable from $b$, being careful not to search vertices more than once. In doing so, we can find an edge $(a, b)$ relaxable in round $|V|$ from which $t$ is reachable (which must exist since $\delta(s, t) = -\infty$). Then we concatenate the path to $b$ containing a negative weight cycle, and the path from $b$ to $t$ that was constructed in the graph search from $b$.

**Rubric:**

- 9 points for a correct algorithm
- Partial credit may be awarded

[1] where the weight $w(e)$ of each edge $e \in E$ may be positive **or negative** (or zero)

**Problem 8-2.** [12 points] **Anti-Aloft Aviation Again**

The Wleft brothers have expanded their airline into outer space! They have a list of the $f$ flights operated between the $n$ spaceports in the solar system, where each flight is defined by a source spaceport, a destination spaceport, and the number of minutes of flying time. Unlike in PS6, flying times are now listed as arbitrary positive numbers, not restricted to bounded integers. But just like PS6, the **flying time** of a sequence of flights is the sum of the flying times of each flight in the sequence ignoring layovers. Describe an efficient algorithm to help the Wleft brothers in each of the following scenarios.

**(a)** [4 points] The Wleft brothers want to plan a route from their headquarters in Nighton, Ohio, to Moon Base Beta for their family vacation! Help the Wleft brothers compute the minimum possible flying time from Nighton to Moon Base Beta.

**Solution:** Construct a weighted, directed graph $G$ with a vertex for each of the $n$ spaceports, and a directed edge for each of $f$ flights connecting from its source spaceport to destination spaceport, weighted by the flight time. Since the graph has non-negative edge weights, run Dijkstra to find shortest paths from the node corresponding to Nighton, and return the length of the shortest path to Nighton. Dijsktra runs in $O(n \log n + f)$.

**(b)** [4 points] The Wleft brothers have discovered wormholes in deep space that take **negative** time to traverse! As a result, some of their new Negative Class flights take negative time to fly, but by the Consistent Timeline Hypothesis, they know they can never travel back in time to the same location. Help the Wleft brothers compute the minimum flying time from Moon Base Beta to the trading post, Deep Space Ten.

**Solution:** Construct the same weighted directed graph as $G$, but this time a flight times may be negative. Since edges can be both positive and negative, we run Bellman-Ford to find the shortest path from Moon Base Beta to Deep Space Ten, which will exist of a path exists, since there can be no negative weight cycles in the graph (by the Consistent Timeline Hypothesis). Bellman-Ford runs in $O(nf)$ time.

**(c)** [4 points] The Wleft brothers are tired of working in Nighton, Ohio, and want to move their headquarters somewhere else in the universe. They would like to choose somewhere that minimizes their average flying time to any other destination (still allowing Negative Class flights). Help the Wleft brothers find the spaceport that minimizes the sum of the minimum flying times to all other spaceports.

**Solution:** Again, construct a weighted, directed graph $G$ with a vertex for each of the $n$ spaceports, and a directed edge for each of $f$ flights connecting from its source spaceport to destination spaceport, weighted by the flight time. Since the graph has both positive and negative weights, we find the shortest path length between each pair of cities using Johnson's Algorithm. For each city, compute the sum of the shortest paths to all the other cities, and return any city which minimizes this quantity. Running Johnson's Algorithm takes $O(n^2 \log n + nf)$ time, and finding the sum of distances

for each city takes $O(n^2)$ time in total, so the algorithm runs in $O(n^2 \log n + fn)$ time.

**Rubric:**

- 3 points for each correct algorithm
- 1 point for each correct running time analysis
- Partial credit may be awarded

**Problem 8-3.** [8 points] **Bottle Breaker**

The hit new videogame *Green Zombie Atonement II* features a new minigame, where you can throw balls at a line of bottles, each labeled with a number (positive, negative, or zero). You can throw as many balls at the bottles as you like, and if a ball hits a bottle, it will fall and shatter on the ground. Each ball will either hit no bottle, exactly one bottle, or two bottles (but only when the two bottles were **adjacent in the original lineup**). If a ball hits two adjacent bottles, you receive a number of points equal to the product of the numbers on the bottles. Otherwise, if a ball hits zero or one bottle, you do not receive any points. For example, if the line of bottle labels is $(5, -3, -5, 1, 2, 9, -4)$, the maximum possible score is 33, by throwing two balls at bottle pairs $(-3, -5)$ and $(2, 9)$. Given a line of bottle labels, describe a efficient dynamic-programming algorithm to maximize your score.

**Solution:**

1. **Subproblems**
   - Points are independent of the order we hit the bottles.
   - All that matters is which bottles are paired.
   - Let $l(i)$ denote the label of bottle $1 \leq i \leq n$, where $n$ is number of bottles
   - $x(i)$: maximum score possible by hitting only bottles 1 to $i$
2. **Relate**
   - Either the last bottle is part of a pair, gaining points and leaving $i - 2$ bottles remaining,
   - or the last bottle is not part of a pair, and we can evaluate the remaining $i - 1$ bottles
   - $x(i) = \max\{x(i-2) + l(i) \cdot l(i-1), x(i-1)\}$
   - Subproblems $x(i)$ only depend on strictly smaller $i$, so acyclic
3. **Base**
   - No points possible if only zero or one bottles
   - $x(0) = x(1) = 0$
4. **Solution**
   - Solution is $x(n)$, the maximum considering all the bottles
   - Store parent pointers to reconstruct hit pairs
5. **Time**

- # subproblems: $n + 1$
- work per subproblem $O(1)$
- $O(n)$ running time

**Rubric:**

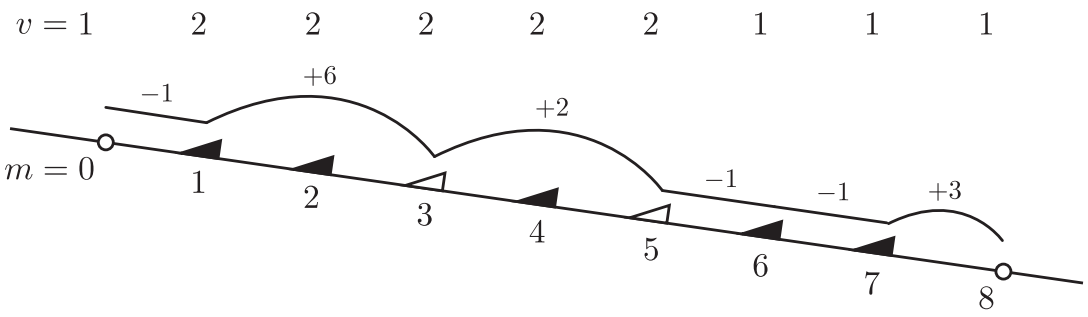
- 2 points for subproblem description
- 2 points for a correct recursive relation
- 1 point each for base cases
- 1 point for solution in terms of subproblem
- 1 point for running time analysis
- 1 point if correct algorithm is $O(n)$
- Partial credit may be awarded

**Problem 8-4.** [10 points] **Sk8er Boi**

In the videogame *Hony Tawk's Adventure: 6.006 or Die*, Hony rides a skateboard down a **linear** race course, with a ramp located every meter from the top of the course down to the last ramp at meter $n$. Each ramp is either white or black. Hony begins at the top with a velocity of 1 meter per second. At each ramp, Hony can either jump the ramp, or skateboard past it to the next ramp. If he jumps the ramp located $m$ meters from the top with velocity $v$, he will fly through the air at constant speed for one second doing gnarly tricks, landing at the ramp located at $m + v$ meters, where he can then make another choice on whether to jump from that ramp. Successfully landing a trick with velocity $v$ will give him $v$ awesome points if he jumped from a white ramp, and $3v$ points if he jumped from a black ramp. If he chooses not to jump, and instead skateboards to the next ramp, he will lose 1 awesome point per meter he skateboards on the ground; but at each meter on the ground where he chose not to jump a ramp, he can choose to either maintain speed, increase his speed by 1 meter per second, or decrease his speed by 1 meter per second, always maintaining a speed between 1 and 40 meters per second inclusive. Hony's score on the course will be the number of awesome points he has when he has reached one meter past the last ramp. If he reaches the end of the course in the air mid-trick, he will not receive any points for that trick. Describe an efficient algorithm to find the maximum score Hony can acheive on a given course. Below is an example course with 7 ramps on which Hony can acheive 16 points, depicting a suboptimal jumping and acceleration strategy that achieves only 8 points.

**Solution:** This problem can be solved by constructing a graph and running DAG relaxation to find the shortest path between a start state and an end state. The graph is related to the dependency graph provided in the following dynamic programming solution.

1. **Subproblems**
   - At each location, Hony has a position and a velocity
   - Let $x(m, v)$ be maximum awesome points arriving at ground location $m$ with velocity $v$




2. **Relate**
   - Hony arrives at each location by either landing from a jump or skating on the ground, possibly changing speed
   - Let $c(m) = 1$ if ramp at meter $m$ is white and $c(m) = 3$ if ramp at meter $m$ is black

$$x(m, v) = \max \left\{ \begin{array}{l} x(m - v, v) + c(m - v)v, \\ x(m - 1, v) - 1, \\ x(m - 1, v - 1) - 1, \\ x(m - 1, v + 1) - 1 \end{array} \right\}$$

   - Subproblems $x(m, v)$ only depend on strictly smaller $m$, so acyclic
3. **Base**
   - $x(0, 1) = 0$ (Hony's start)
   - $x(m, v) = -\infty$ for $m \leq 0$, $v < 1$, or $40 < v$ (unreachable states)
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - It is never good to end the course skating on the ground, as that would lose points
   - Solution to original problem checks ramp of last jump

$$\max\{x(n + 1 - j, v) \mid v \in \{1, \ldots, 40\}, j \in \{0, \ldots, v - 1\}\}$$

5. **Time**
   - # subproblems: $n \cdot 40 = O(n)$
   - work per subproblem $4 \cdot O(1) = O(1)$
   - solution computation compares $\leq 40^2$ subproblems in $O(1)$ time
   - $O(n)$ running time

**Rubric:**

DAG Relaxation Approach:

- 5 points for graph description

- 3 points for description of algorithm
- 1 point for running time analysis
- 1 point if correct algorithm is $O(n)$
- Partial credit may be awarded

Dynamic Programming Approach:

- 3 points for subproblem description
- 3 points for a correct recursive relation
- 1 point each for base cases
- 1 point for solution in terms of subproblem
- 1 point for running time analysis
- 1 point if correct algorithm is $O(n)$
- Partial credit may be awarded

**Problem 8-5.** [45 points] **Bidirectional Dijkstra**

Bidirectional Dijkstra finds a shortest path from $s$ to $t$ in a connected undirected weighted graph having only nonnegative edge weights. It trys to look at less of the graph by searching in two directions at once. More precisely, the algorithm runs a forward Dijkstra search from $s$, and a backward Dijkstra search from $t$. Each Dijkstra search maintains minimum-weight path estimates, $d(s, v)$ and $d(t, v)$, and parent pointers, $\pi(s, v)$ and $\pi(t, v)$, for each node $v$; and stores shortest path estimates in priority queues, $Q_s$ and $Q_t$. At each step of the algorithm, Bidirectional Dijkstra removes from a priority queue (and relaxes outgoing edges from) a vertex $v$ having smallest priority queue key (either $d(s, v)$ or $d(t, v)$), from among all vertices in either priority queue. Each time a vertex is processed, we maintain a vertex $v^*$ that minimizes $d(s, v^*) + d(t, v^*)$, i.e. $d(s, v^*) + d(t, v^*) = \min_{v \in V} d(s, v) + d(t, v) = D$, and every time an edge is relaxed, we update $v^*$ as necessary. Stop processing vertices when the smallest minimum-weight path estimate, from among all vertices in either priority queue, is strictly larger than $D/2$. Then we return the path $p^*$ of weight $D$ formed by concatenating parent pointers back to $s$ from $v^*$ and parent pointers to $t$ from $v^*$.

**(a)** [2 points] Prove that $D \geq \delta(s, t)$ at termination.

**Solution:** $D$ is at least $\delta(s, t)$, because $\delta(s, t)$ is the minimum distance between $s$ and $t$, while $D$ is the length of some path from $s$ to $t$ (in particular a path through $v^*$).

**Rubric:**

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

**(b)** [4 points] Consider any shortest path $p$ from $s$ to $t$, where $w(p) = \delta(s,t)$. Let $v_s \in p$ be the farthest vertex from $s$ that is (weakly) closer to $s$ than to $t$, and let $v_t \in p$ be the farthest vertex from $t$ that is (weakly) closer to $t$ than to $s$. Prove that $d(s,v_s) = \delta(s,v_s)$ and $d(t,v_t) = \delta(t,v_t)$ at termination.

**Solution:** Since $\delta(s,t) \leq D$, then $\delta(s,v_s) \leq D/2$, and $v_s$ and all vertices from $s$ to $v_s$ in $p$ were removed from $Q_s$ prior to termination. When Dijkstra pops a vertex $v$, we know $d(s,v) = \delta(s,v)$, thus at termination, $d(s,v_s) = \delta(s,v_s)$. A symmetric argument shows that $d(t,v_t) = \delta(t,v_t)$ at termination.

**Rubric:**

- 4 points for a correct proof
- Partial credit may be awarded

**(c)** [4 points] Prove that $\delta(s,t) = d(s,v_t) + d(t,v_t)$ at termination. Hint: use (b).

**Solution:** Either $v_s$ and $v_t$ are the same vertex, or they are adjacent in $p$. If $v_s$ and $v_t$ are the same vertex then $\delta(s,t) = d(s,v_t) + d(t,v_t)$ by (b). Alternatively, $v_s$ and $v_t$ are adjacent vertices in $p$, and when $v_s$ was popped from $Q_s$, edge $(v_s,v_t)$ was relaxed, setting $d(s,v_t)$ to $\delta(s,v_t)$, proving the claim.

**Rubric:**

- 4 points for a correct proof
- Partial credit may be awarded

**(d)** [2 points] Prove that $D \leq \delta(s,t)$ at termination. Hint: use (c).

**Solution:** Bidirectional Dijkstra maintains that $D = \min_{v \in V} d(s,v) + d(t,v)$, so $D \leq d(s,v_t) + d(t,v_t)$, and by (c), $D \leq \delta(s,t)$.

**Rubric:**

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

Thus by (a) and (d), $w(p^*) = D = \delta(s,t)$ at termination, and $p^*$ is a shortest path from $s$ to $t$. Bidirectional Dijkstra is not asymptotically faster than one-way Dijkstra for all graphs. For example, if the graph is a chain from $s$ to $t$, both algorithms have the same asymptotic running time. However, Bidirection Dijkstra is asymptotically faster for many common graphs.

**(e)** [8 points] Let $f(s,r)$ be the number of vertices $v \in V$ having shortest path weight no greater than $r$, i.e. $\delta(s,v) \leq r$. Define a ***k*-dense graph** to be a graph where each vertex has degree at most $O(k)$ and $f(s,r) = \Theta(r^k)$ for some fixed $k$. Examples of graphs satisfying this property are the infinite two dimensional square grid for $k = 2$, or more generally any infinite $k$-dimensional lattice[2]. Show that Bidirectional Dijkstra from $s$ to $t$ in a $k$-dense graph is faster than one-way Dijkstra by a factor of $\Theta(2^k)$.

[2]Exercise: Can you prove that a square grid is 2-dense?

**Solution:** Let $D$ be the minimum-weight of any path from $s$ to $t$. One-way Dijkstra will explore $\Theta(D^k)$ vertices in the worst case, so will run in $\Theta(D^k \log D^k + D^k k) = \Theta(kD^k \log D)$ time. Alternatively, Bidirectional Dijkstra will explore $\Theta(2 \cdot (D/2)^k) = \Theta(D^k/2^k)$ vertices in the worst case, so will run in $\Theta(D^k/2^k \log(D^k/2^k) + D^k/2^k k \log D) = \Theta(kD^k/2^k \log D)$, i.e. $\Theta(2^k)$ times faster than one-way.

**Rubric:**

- 8 points for a correct proof
- Partial credit may be awarded

**(f)** [25 points] Implement the Bidirectional Dijkstra algorithm described above in a Python function `bidirectional_dijkstra(Adj, w, s, t)`. You can download a code template containing some test cases from the website. You may adapt any code presented in lecture or recitation, but for this problem, **you may NOT import external packages** and **you may NOT use Python's built-in sort functionality** (the code checker will remove `List.sort` and `sorted` from Python prior to running your code). Submit your code online at `alg.mit.edu`.

**Solution:**

```
def bidirectional_dijkstra(Adj, w, s, t):
    inf = float('inf')
    ds, dt = [inf  for _ in Adj], [inf  for _ in Adj]
    ps, pt = [None for _ in Adj], [None for _ in Adj]
    ds[s], dt[t] = 0, 0
    Qs, Qt = PriorityQueue(), PriorityQueue()
    for v in range(len(Adj)):
        Qs.insert(v, ds[v])
        Qt.insert(v, dt[v])
    D, v_star = inf, None
    while min(Qs.find_min(), Qt.find_min()) <= (D / 2):
        if Qs.find_min() <= Qt.find_min():
            Q, d, p = Qs, ds, ps
        else:
            Q, d, p = Qt, dt, pt
        u = Q.extract_min()
        for v in Adj[u]:
            relax(Adj, w, d, p, u, v)
            Q.decrease_key(v, d[v])
            d_v = ds[v] + dt[v]
            if d_v < D:
                D, v_star = d_v, v
    v = v_star
    path = [v]
    while v != s:
        v = ps[v]
        path.append(v)
    path.reverse()
    v = v_star
    while v != t:
        v = pt[v]
        path.append(v)
    return path
```