Problem Set 5 Solutions

This problem set is due in lecture on Wednesday, November 28th, 2007.

Instructions:

- Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

- You will often be called upon to “give an algorithm” to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

  1. A description of the algorithm in English and, if helpful, pseudo-code.
  2. At least one worked example or diagram to show more precisely how your algorithm works.
  3. A proof (or indication) of the correctness of the algorithm.
  4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which are 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, prevent you from writing convoluted solutions, and also help you conceptualize the key idea of the problem.

- Be sure to know the section on “Guide to writing up homework” in the “Course Information” handout before writing your solutions.

Problem 5-1. Finding the mode of a set without much space.

(a) This problem is a preparation for the next part. Suppose that you are given an array $A$ of $n$ elements. Give an $O(n \log n)$ algorithm to find the most frequent element of $A$. Can you do better if you know that all the elements of $A$ are small positive integers?

Solution: Sort the array using Quicksort, Heapsort or Mergesort in $O(n \log n)$ time. Then go from left to right in the sorted array keeping track of the length of the longest run of items found so far. Return the element achieving that maximum. If the elements are small integers, we can use Radix Sort to sort, which gives $O(n)$ time.
(b) Suppose now that you are given a read-only media containing a HUGE string $A$. The string $A$ has a total length of $L$ characters and consists of $n$ phrases separated by a special character. Your task is to find the most frequent phrase in $A$. (For instance, think of $A$ as the concatenation of the text of all English webpages found in Internet. Your mission is to find the most common English phrase in Internet).

Assume that $L$ is so big that you can’t fit $A$ in memory (and since $A$ is read-only, you can’t use any in-place algorithm). Also assume that $n \ll L$. Devise an algorithm based on hashing that returns the most frequent phrase in $A$ using only $O(n \log n)$ bits of extra space. Can you make the running time of your algorithm linear in $L$?

Solution: Pick a universal random hash function from the set of strings of size at most $L$ to $\{0, \ldots, m - 1\}$, where $m = \Theta(n^d)$, for $d$ a constant bigger than 2. (For instance, we could let $m$ be a prime less than $n^d$ and use the approach of lecture 16).

Apply the hash function to each phrase $P$ (we may assume that the hash function can be evaluated dynamically even if the phrase is too big to fit in memory) to get a small-size fingerprint $h(P)$. Create an array $B$ with $n$ entries, such that $B[i]$ holds both the fingerprint associated to the $i$-th phrase of $A$ and the index $i$.

Consider first the case when $h$ is such that there are no collisions (meaning that for two different phrases in $A$, their image under $h$ is different).

In this case, we can find the most common phrase of $A$ by finding the most common fingerprint in $B$ (using the previous part). Since we are also keeping track of the indices of the phrases from where the fingerprints come from, we can output the index of a phrase mapping to the most common fingerprint. The corresponding phrase is the most common phrase in $A$.

The space needed is just the space used to store the array $B$. Each entry of $B$ holds one fingerprint and one index from 1 to $n$, which can represented with $O(\log m + \log n) = O(d \log n)$ bits. Then $B$ itself can be stored using $O(n + nd \log n)$ bits. Since $d$ is a constant, this is $O(n \log n)$.

The running time used by the algorithm is $O(L)$, to read $A$ and compute each fingerprint, and $O(n \log n)$, to sort $B$ and find the most common fingerprint, as we did in the previous problem (Since all the keys are smaller than $n^d$, we could also sort them using Radix Sort in time $O(n + d)$). Since $n \ll L$, the total running time is $O(L)$.

Note that the previous approach may still work even if the function $h$ has some collisions. As long as no two different phrases maps to the most common fingerprint, the previous algorithm will output the right answer.

We can check if this happens efficiently before outputting the answer: Let $f$ be the most frequent fingerprint. Do a linear scan of $A$ applying $h$ to each phrase until you find the first occurrence of a phrase with fingerprint $f$. Keep a pointer to the first character in such that phrase. Continue scanning $A$ and, for each occurrence of a phrase with the same fingerprint, compare head-to-head for equality with this first occurrence (i.e. check if both phrases are the same).
If we find a collision, we can pick another universal hash function and start all over again. We repeat this until no collisions in the most common fingerprint happens. Note that this checking routine also takes time $O(L)$ since each phrase is checked at most once, and we only need space to keep a constant number of pointers inside $A$ (which we need to assume anyway to be able to access $A$).

How many times we need to perform the algorithm until there is no collision to the most common fingerprint?

To answer this, recall that the hash family from where we are picking the hash function is universal. Then, for a given phrase $P_i$ the probability that a different phrase $P_j$ collides with it is $\Pr[h(P_i) = h(P_j)] = 1/m$. Therefore, using that $m = \Theta(n^d)$, the expected total number of collisions is at most:

$$\mathbb{E}[\# \text{ of collisions}] \leq \sum_{i<j} \Pr[h(P_i) = h(P_j)] \leq n^2/m = \Theta(n^{2-d}) = o(1).$$

The last is true since $d > 2$. Using Markov inequality,

$$\Pr[\text{There is a collision}] = \Pr[\# \text{ of collisions is at least 1}] \leq \mathbb{E}[\# \text{ of collisions}] = o(1).$$

And so, the expected number of repetitions is one over the probability that there is no collision. This is $1/(1 - o(1))$ which is asymptotically close to 1.

(c) (Optional) Assume now that $A$ is given to you as a stream of data (i.e. you can only read it once from left to right). Modify your previous algorithm to output, with high probability, an index $i$ such that the $i$-th phrase of $A$ is the most frequent phrase of $A$.

**Solution:**

Now we can’t really check if the hash function collide (and since there are $n \ll L$ words, some phrases must be really big, therefore we can’t keep a copy of them to verify collisions). In this case, we simply ignore the collision verification step. As we saw before, the probability that there is a collision for $h$ is negligible. Therefore with high probability the algorithm outputs the right answer.

**Problem 5-2. Maximizing skiing fun**

A group of 6,046 students are preparing to go skiing during winter break. They plan to use what they learned in 6.046 to maximize their fun. They have a map, which they naturally regard as an undirected graph $G = (V, E)$; where vertices represent locations and edges represent available trails that can be either climbed up or skied down. Some of the locations are bus stops, denoted by $S \subset V$.

To simplify their problem, they decide to look for a route such that all of the uphill segments come at the beginning and all of the downhill segments come at the end. (The idea is to climb uphill first, and then go downhill skiing). Call such a route a “valid” route.

Let $l(e) \in \mathbb{R}^+$ denote the length of a trail $e \in E$ and $h(v) \in \mathbb{R}$ denote the elevation (height) of a location $v \in V$. Assume that no two locations have the same elevation.
(a) Define the “funniness value” of a route to be the sum of the length of the downhill segments minus the sum of the length of the uphill segments. The students want to find a good valid route with the property that they can reach the start of the route by bus and they can also leave from the end of the route by bus.

Give an efficient algorithm to find the funniest valid route that starts and ends at a bus stop. Analyze the running time of your algorithm as a function of \( n = |V|, m = |E| \) and \( s = |S| \).

**Solution:** Create a directed graph \( G_U \) with the same vertex set as \( G \) such that each arc \((x, y) \in E_U\) represents an uphill segment (i.e. \( h(y) > h(x), xy \in E(G)\)). Since all the elevations are different, \( G_U \) is acyclic.

For each bus stop \( a \in S \) compute both the shortest uphill path and the longest uphill path to all the other vertices. Since \( G_U \) is acyclic we can do this using the modification of Bellman Ford for DAG, using as length function \( l(e) \) in the first case and \(-l(E)\) in the second case.

For each vertex \( v \in V \) compute the closest bus stop \( s_1(v) \) and the farthest bus stop \( s_2(v) \) from which one can reach \( v \) by going uphill (simply cycle over all the bus stops checking the results obtained in the previous step). Let \( P(v) \) be the route that starts in \( s_1(v) \), then goes up to \( v \) using the shortest uphill path available, and then goes down to \( s_2(v) \) using the longest downhill path available (this last segment is the reverse of the longest path from \( s_2(v) \) to \( v \) computed previously). Note that \( P(v) \) is the “funniest” valid route having \( v \) as its high point.

Finally take the maximum over all \( v \) of all the values of \( P(v) \) and output the route that achieves that maximum.

The running time of this algorithm can be analyzed as follows. It takes \( O(n + m) \) to create the auxiliary directed graph. \( O(s(n + m)) \) to find all the shortest and longest uphill paths from \( a \in S \) to every \( v \). \( O(sm) \) to find \( s_1(\cdot) \) and \( s_2(\cdot) \) and \( O(n) \) to find the maximum. Therefore the total running time is \( O(s(n + m)) \) which is \( O(sm) \) if the graph is connected.

(b) For a fixed \( k \geq 1 \), we say that a route \( P \) is a \( k \)-valid route if \( P \) is a (possible self-intersecting) walk that can be decomposed as \( P = U_1D_1U_2D_2\ldots U_kD_k \) where for every \( i, U_i \) is a (possibly empty) path with only uphill segments and \( D_i \) is a (possibly empty) path with only downhill segments. Under this definition, a 1-valid route is simply what we denoted before as a valid route. Give an efficient algorithm to find the funniest 2-valid route that starts and ends at a bus stop.

**Solution:** For every vertex \( u \) use the modification of Bellman Ford for DAG in \( G_U \) to compute the shortest (and the longest) uphill path from \( u \) to every other \( v \). This takes time \( O(n(n + m)) = O(n^2 + nm) \). Denote the length of the shortest uphill path from \( u \) to \( v \) as \( S(u, v) \) (\( \infty \) if none exists) and the length of the longest downhill path from \( u \) to \( v \) as \( L(u, v) \) (\(-\infty \) if none exists; here we note that the longest downhill path from \( u \)
to \( v \) is the reverse of the longest uphill path from \( v \) to \( u \) computed by Bellman Ford). As in the previous part, for each vertex \( v \in V \) also compute the closest bus stop \( s_1(v) \) and the farthest bus stop \( s_2(v) \) from which one can reach \( v \) by going uphill.

Before continuing, let us explore what are the properties of the route we are looking for. Let \( P \) be one such funniest 2-valid route. In particular, \( P \) can be decomposed as \( sU_1uD_1vU_2wD_2t \), meaning that \( P \) is a route that starts in some bus stop \( s \) then it goes uphill through a path \( U_1 \) until some location \( u \), then it goes downhill through a path \( D_1 \) until it hits some location \( v \), then goes uphill again through \( U_2 \) until \( w \) and then goes downhill again until some bus stop \( t \).

We note here that \( s \) must be the closest bus stop to \( u \) from which one can reach \( u \) uphill. If this was not the case then we could replace in \( P \) the subroute \( sU_1u \) by some other shorter route that starts in a bus stop and goes uphill to \( u \). So, without loss of generality we may assume that \( s = s_1(u) \). Similarly we can argue that \( t = s_2(v) \).

Note that if we were given \( u, v \) and \( w \), then we must have that \( U_1 \) is the shortest uphill path from \( s_1(u) \) to \( u \), \( D_1 \) is the longest downhill path from \( u \) to \( v \), \( U_2 \) is the shortest uphill path from \( v \) to \( w \) and \( D_2 \) is the longest downhill path from \( w \) to \( s_2(w) \). It follows that \( P \) is completely determined by the triplet \( (u, v, w) \).

Then, to find \( P \) we just compute for every triple \( (u, v, w) \), the value of the path defined by \( P(u, v, w) \), namely \(-S(s_1(u), u) + L(u, v) - S(v, w) + L(w, s_2(w))\), and we output the path achieving the maximum value.

The running time for the algorithm is \( O(n + m) \) to create \( G_U \), \( O(n^2 + nm) \) time for the computation of all pairs shortest and longest paths, and \( O(n^3) \) to compute the values for \( P(u, v, w) \) for every triple \( (u, v, w) \in V^3 \), and output the maximum. Since \( m = O(n^2) \), the total running time for the algorithm is \( O(n^3) \). A clever twist (which won’t be explained here) of the previous algorithm allows us to decrease the total time to \( O(n^2 + nm) \).

\textbf{(c)} (Optional). Give an efficient algorithm to find the funniest \( k \)-valid route that starts and ends at a bus stop.

\textbf{Solution:} In the previous solution we optimized over all the \( n^3 \) possible breakpoints of a 2-valid path that starts and ends in a bus stop to obtain the best possible path, giving this a running time of \( O(n^3) \).

A first solution to this part of the problem would be to optimize over all the \( n^{2k-1} \) possible breakpoints of a \( k \)-valid path (there are \( 2k - 1 \) breakpoints, each of them can take \( n \) possible values). It is easy to see that this yields an algorithm with running time \( O(n^{2k-1}) \). This is not a good algorithm since it is not polynomial (the exponent in the running time depends on \( k \), which is part of the input).

We can do much better than that using dynamic programming. Let us forget (for a moment) the condition that the path starts and ends at a bus station, and assume for now that \( k \) is a power of two, \( k = 2^l \).
Define for every \( j \) from 1 to \( l \), and for every pair of vertices \((u, v) \in V^2\), \( P^{(j)}(u, v) \) to be the funniest \( 2^j \)-valid path starting in \( u \) and ending in \( v \), and let \( A^{(j)}(u, v) \) be its funniness value (Think of \( A^{(j)} \) as a matrix of \(|V \times V| = n^2 \) entries).

Observe that for every \((u, v)\), there exist a vertex \( w \) (the middle breakpoint of \( P^{(j)}(u, v) \)) such that \( P = P^{(j)}(u, v) \) can be decomposed as \( uR_1wR_2v \), with \( uR_1w \) and \( wR_2v \) two \( 2^{l-1} \)-valid path between \( u \) and \( w \) and between \( w \) and \( v \) respectively. Furthermore, \( R_1 \) must be the funniest \( 2^{j-1} \)-valid path between \( u \) and \( w \) (otherwise, we could improve the value of \( P \) by using a better path) and similarly \( R_2 \) must be the funniest \( 2^{j-1} \)-valid path between \( w \) and \( v \). It follows that if we are able to find the best possible \( w \) efficiently, we can determine \( P \) from paths in \( P^{(j-1)} \). In fact, to determine \( P \) we don’t need to store all the paths, it is enough to store the middle breakpoint \( w \) associated to it (we can store them in a collection of \( n \times n \) matrices \( W^{(j)} \)). This yields the following recurrence.

For any \( j \geq 1 \),

\[
A^{(j)}(u, v) = \max \{ w \in V : A^{(j-1)}(u, w) + A^{(j-1)}(u, w) \}; \quad \forall (u, v) \in V^2,
\]

and

\[
W^{(j)}(u, v) = \arg \max \{ w \in V : A^{(j-1)}(u, w) + A^{(j-1)}(u, w) \}; \quad \forall (u, v) \in V^2,
\]

i.e. \( W^{(j)}(u, v) \) is the \( w \) that achieves the maximum in the definition of \( A^{(j)}(u, v) \).

We also need to show how to compute the base case matrices \( A^{(0)} \) and \( W^{(0)} \) (corresponding to \( 2^0 = 1 \)-valid paths). To do that we proceed as follows. Compute, as in the part (b) of this problem, for every pair \((u, v)\) the shortest uphill path from \( u \) to \( v \) and the longest downhill path from \( u \) to \( v \). As then, denote \( S(u, v) \) and \( L(u, v) \) to the value of those paths. Then

\[
A^{(0)}(u, v) = \max \{ w \in V : -S(u, w) + L(w, v) \}; \quad \forall (u, v) \in V^2,
\]

and \( W^{(0)}(u, v) \) is the vertex \( w \) that achieves the maximum in the definition of \( A^{(0)}(u, v) \).

The funniest \( k \)-valid path that starts and ends in a bus station can be finally computed by finding the maximum of \( A^{(0)}(s, t) \) over all pairs \((s, t)\) of bus terminals and outputting the corresponding \( P^{(0)}(s, t) \) (we can construct that path efficiently from the middle breakpoints and the longest and shortest path constructed using Bellman Ford).

If \( k \) is not a power of two, then we can modify the algorithm a little bit by using the binary expansion of \( k \) and the matrices defined so far. We omit the analysis for that case.

The running time for the algorithm can be bounded as follows. As in the previous part, we need \( O(n^2 + nm) \) to compute all pairs shortest and longest path. Also, note that for every \( j \), each entry \( A^{(j)}(u, v) \) and \( W^{(j)}(u, v) \) can be computed in time \( O(n) \) from previous information. It follows that every matrix can be filled in \( O(n^3) \) time. Since we need to fill \( 2l = O(\log k) \) matrices, the final running time of the algorithm is \( O(n^2 + nm + n^3 \log k) = O(n^3 \log k) \).
Problem 5-3. Odd couple roadtrips.

Alice and Bob are a quite odd couple. They both like driving a lot, so everytime they go out together they argue about who should be the driver.

For every trip they do, they use the same route map which they can represent as an undirected graph $G = (V, E)$, where vertices represent locations and edges represent routes. Also, for every $e \in E$, $l(e) \in \mathbb{R}^+$ denotes the length of that route. (Assume that $G$ is connected and has no loops).

To avoid conflicts they have one golden rule: Everytime they stop at (or pass through) a location, they swap driver; and these are the only times when they swap. Due to some really awkward situation in the past they never break this rule.

(a) Next weekend, Alice and Bob need to make a trip from Somerville ($s$) to Toronto ($t$). Since Alice knows both places better, they decide that Alice should drive in the first and last segment of the route.

Devise an efficient algorithm to find a route of minimum distance between both places, such that their trip start with Alice driving and, according to their golden rule, also ends with Alice driving. Your algorithm should also return a message saying “No path” if there is no such path.

Solution: Through this problem $n$ will be the number of vertices in the graph $G$ and $m$ will be the number of edges in $G$.

Any path that starts and ends with Alice driving will have an odd number of edges, thus, in this problem we want to design an algorithm to find the shortest $s$-$t$ path using an odd number of edges. To solve that problem, it will be useful first to find the shortest even paths from $s$ to all the vertices of the graph.

Construct a new graph $G^2 = (V, F)$, with $F = \{ij | \exists k : ik, kj \in E\}$. In other words, every edge in this new graph represents a path of length 2 in $G$. Note that for every $u$, the shortest even $s$-$u$ path in $G$ corresponds to a shortest $s$-$u$ path in $G^2$.

We can find the shortest even paths from $s$ to all the other vertices by using one application of Dijkstra’s algorithm on $G^2$. (To recover the original paths in $G$, we may augment each edge $ij$ of $G^2$ with the label of some $k$ for which $ik$ and $kj$ are in $E$). Denote $L^2(s, u)$ to the length of the shortest even path from $s$ to $u$ (if no path from $s$ to $u$ exists, $L^2(s, u) = \infty$).

To find the shortest odd path from $s$ to $t$ we can compute for all $u$, neighbors of $t$, the quantity $L^2(s, u) + l(u, t)$. This quantity represents the length of the path obtained by concatenating the shortest path from $s$ to $u$ using an even number of edges with the edge $ut$. Compute the minimum over all the $u$’s of the previous quantity. That will be the length of the shortest path from $s$ to $t$ using an odd number of edges. After that, simply output the corresponding path.

It is worth noting here that if there is no solution to the original problem, then Dijkstra’s algorithm will report that no $s$-$u$ path exists in $G^2$ for every $u$ neighbor of $t$. If this happens our algorithm should return a message saying “No path”.

The running time for this algorithm is the time to construct the auxiliary graph, the
time of running one Dijkstra application and the time to compute the minimum of the
quantities defined before. Since the last minimum is taken over at most \( n \) vertices, the
running time for this last step is \( O(n) \). The running time for Dijkstra is \( O(m' \log n) \)
using a binary heap implementation or \( O(m' + n \log n) \) if we use a Fibonacci Heap
implementation.

To construct \( G^2 \), all we need to do is to find, for every \( v \), all the vertices of distance 2
to \( v \). This takes \( O(m) \) per vertex (using BFS), for a running time of \( O(nm) \).

The total running time is then \( O(nm + m' + n \log n + n) \), which, since the graph is
connected (and \( m' = O(n^2) \)), is dominated by \( O(nm) \).

**Comment:** There are alternative solutions obtained by modifying Dijkstra or Bellman
Ford and running them directly on \( G \). These solutions, in general, are not simple to
analyze, since the modified algorithm need to be able to handle walks that visit a vertex
more than once. As an example, consider the graph formed by a triangle \( u, v, w \) and
two extra edges \( su \) and \( ut \). Any odd path from \( s \) to \( t \) will have to visit \( u \) twice.

**Update:** For a really nice solution for this problem, see the end of this handout.

(b) Although Bob does not like to admit it, he has a problem getting the car in and out the
garage. For that reason, everytime they need to go out for a roundtrip ride, they prefer
that Alice drives both the first and last segment of the trip.

They discovered that the place where they live now is such that no matter what closed
route they take that starts and ends with Alice driving, the trip takes too long (remember
that \( G \) has no loops). For that reason they are planing to move to a place where
this does not happen.

Devise an algorithm for finding one location \( v \) that minimizes the shortest closed route
that starts and ends at \( v \) with Alice driving in the first and last segment.

**Solution:** To find such a \( v \), we need to find the smallest odd cycle of \( G \) (we can
output any vertex in one such cycle). We can do this by checking, for every \( v \in V \) the
shortest odd \( v-v \) path, and then outputting any \( v \) that minimizes the length of that path.

A direct way to do this is to perform \( n \) times the algorithm of part (a), for a running
time of \( O(n^2 m) \).

We can do better than that. Note that we only need to construct \( G^2 \) once. This can be
done in time \( O(nm) \). Then we can run any All pairs shortest paths algorithm in \( G^2 \) to
obtain all the shortest even paths in \( G \). Denote \( L^2(u, v) \) to the length of the shortest
even path between \( u \) and \( v \).

Next, for every \( v \) we compute the minimum among all its neighbors \( w \) of \( l(v, w) +
L^2(w, v) \). This will give us the length of the minimum odd walk that starts and ends in
\( v \). Since for each \( v \) this takes time \( O(n) \) to calculate, the running time for this part is
\( O(n^2) \). Finally we output the \( v \) that minimizes the length of such a closed walk. This
takes \( O(n) \) time to determine.
Thus, the running time for this algorithm is \( O(nm + n^2 + n) + T_{\text{APSP}} = O(nm) + T_{\text{APSP}} \), where \( T_{\text{APSP}} \) is the time needed to compute all pairs shortest paths in \( G^2 \). Let \( m' \) be the number of edges of that graph. Depending on which algorithm we use to compute all pairs shortest paths, the total running time will be:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>( O(nm + n^3 \log n) )</td>
</tr>
<tr>
<td>Floyd Warshall (see CLRS)</td>
<td>( O(nm + n^3) )</td>
</tr>
<tr>
<td>Bellman Ford</td>
<td>( O(nm + n^2m') )</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>( O(nm + nm' + n^2 \log n) )</td>
</tr>
</tbody>
</table>

The running time of these algorithms depend on how “sparse” are \( G \) and \( G^2 \) (i.e. how big are \( m \) and \( m' \) compared to \( n \)). If both \( m \) and \( m' \) are \( o(n^2) \), then the approach using Dijkstra is the most efficient.

**Update:** Alternative simple solution for both parts. Consider the graph \( G' = (V', E') \) where, for every vertex \( v \) in \( V \), \( V' \) contains two copies \( v_{\text{even}} \) and \( v_{\text{odd}} \) of \( V \), and for every edge \( e = uv \) in \( E \), there are two edges \( u_{\text{odd}}v_{\text{even}} \) and \( u_{\text{even}}v_{\text{odd}} \). Note that there is a one to one correspondence between paths in \( G \) and paths in \( G' \). Furthermore, any path from \( s \) to \( t \) with an odd number of edges in \( G \) will correspond to a path from \( s_{\text{even}} \) to \( t_{\text{odd}} \) in \( G' \) (at every step, the path jumps from an odd vertex to an even vertex and vice versa). Note that \( |V'| = 2|V| = O(n) \) and \( |E'| = 2|E| = O(m) \).

To solve part (a), we just find using Dijkstra an \( s_{\text{even}} - t_{\text{odd}} \) shortest path in \( G' \) in \( O(m + n \log n) \), and to solve part (b) we just use Dijkstra \( n \) times (finding shortest paths from \( v_{\text{even}} \) to \( v_{\text{odd}} \) for every \( v \)), and then output any \( v \) minimizing that, this takes time \( O(nm + n^2 \log n) \).