1. After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Champaign-Urbana is currently suffering from a plague of zombies, so even though the bus stops have fences that supposedly keep the zombies out, you’d still like to spend as little time waiting at bus stops as possible. Unfortunately, there isn’t a single bus that visits both your exam building and your home; you must transfer between buses at least once.

Describe and analyze an algorithm to determine a sequence of bus rides from Siebel to your home, that minimizes the total time you spend waiting at bus stops. You can assume that there are \( b \) different bus lines, and each bus stops \( n \) times per day. Assume that the buses run exactly on schedule, that you have an accurate watch, and that walking between bus stops is too dangerous to even contemplate.

**Solution (Dijkstra):** Assume that our input consists of the following information:

- For each bus, an array listing the \( n \) locations and times that bus stops, in chronological order. (These lists are published in a book available on every bus.)
- For each bus stop location, an array listing which buses stop at that location and when, in chronological order. (These lists are posted at every bus stop.)

We need at least one of these sets of arrays to solve the problem at all. Given only one of the two sets of lists, we can construct the other set of arrays in \( O(nb \log nb) \) time essentially by brute force. The extra logarithmic factor comes from either sorting the lists chronologically, or using a priority queue to simultaneously walk through the union of the given lists in chronological order.

We build a directed graph \( G \) with \( nb \) vertices, each labeled by a bus stop and a time when some bus visits that stop, plus two additional nodes: a start node \( s \) representing Siebel Center itself, and a target node \( t \) representing home. \( G \) has four types of edges:

- For every bus, \( G \) contains a sequence of \( n-1 \) ride edges joining its visits in chronological order. Every ride edge has weight 0.
- For every stop, \( G \) contains a sequence of directed wait edges joining its visits by buses in chronological order. The weight of a wait edge is the time between the two visits represented by its endpoints.
- \( G \) has start edges from \( s \) to every node representing the bus stop closest to Siebel; these edges have weight 0.
- Finally, \( G \) has target edges from every node representing the bus stop closest to home to \( t \); these edges also have weight 0.

There are exactly \( b(n-1) \) ride edges. In addition, every node has at most one outgoing wait edge, at most one incoming start edge, and at most one outgoing home edge. Thus, the total number of edges is at most \( 3nb = O(nb) \).

Every ride home is represented by a path from \( s \) to \( t \) in \( G \). The length of this path is equal to the total time spent outside at bus stops. Thus, we want the shortest path in \( G \) form \( s \) to \( t \).

We can compute this path by calling Dijkstra’s algorithm. The distance value \( \text{dist}(t) \) tells us the time spent outside during the best trip home; the predecessor pointers allow us to reconstruct the actual ride path. Because there are no negative-weight edges, Dijkstra’s algorithm runs in \( O(E \log V) = O(nb \log n) \) time. ◼
Solution (dag dynamic programming): We build the same directed graph $G$ as in the previous solution, but we use a different algorithm to compute the shortest path from $s$ to $t$. Specifically, $G$ is a directed acyclic graph, because every edge leads from an earlier time to a later time. Thus, we can compute the shortest path in $G$ from $s$ to $t$ using dynamic programming, as follows.

First, topologically sort the graph in $O(V + E) = O(nb)$ time. Then for any vertex $v$, let $dist(v)$ denote the true shortest-path distance from $s$ to $v$. This function obeys the following recurrence:

$$
dist(v) = \begin{cases} 
0 & \text{if } v = s \\
\min_{u \rightarrow v} \left( dist(u) + w(u \rightarrow v) \right) & \text{otherwise}
\end{cases}
$$

This is a proper recurrence because $G$ is acyclic! Moreover, if $v$ is a source but not $s$, the recurrence correctly gives us $dist(v) = \min \emptyset = \infty$, so we do not need a separate explicit base case.

We can memoize this recurrence by storing each value $dist(v)$ at the corresponding vertex $v$. Since each distance value depends only on values later in topological order, we can compute all distances by considering the vertices in reverse topological order, altogether in $O(V + E) = O(nb)$ time.

However, unless we are given both the schedule for every bus and the schedule for every bus stop in the input, actually building the graph requires an additional $O(nb \log nb)$ time, which dominates the running time of the overall algorithm.

Rubric: $O(nb)$ time via Dijkstra: 10 points; graph-reduction rubric.
$O(nb)$ time via dynamic programming: 12 points = 6 for reduction to shortest path (graph reduction rubric) + 6 from dynamic programming rubric (for shortest path in dag). But $-1$ for ignoring preprocessing time, unless the solution explicitly assumes input structures that would make sorting unnecessary.

These are not the only correct solutions.
2. Kris is a professional rock climber (friends with Alex and the rest of the climbing crew from HW6) who is competing in the U.S. climbing nationals. The competition requires Kris to use as many holds on the climbing wall as possible, using only transitions that have been explicitly allowed by the route-setter.

The climbing wall has $n$ holds. Kris is given a list of $m$ pairs $(x, y)$ of holds, each indicating that moving directly from hold $x$ to hold $y$ is allowed; however, moving directly from $y$ to $x$ is not allowed unless the list also includes the pair $(y, x)$. Kris needs to figure out a sequence of allowed transitions that uses as many holds as possible, since each new hold increases his score by one point. The rules allow Kris to choose the first and last hold in his climbing route. The rules also allow him to use each hold as many times as he likes; however, only the first use of each hold increases Kris’s score.

(a) Define the natural graph representing the input.

**Solution:** The graph has one vertex for each hold and one directed edge for each allowed transition. There are $n$ vertices and $m$ edges altogether.

**Rubric:** 2 points = 1 for vertices + 1 for edges

(b) Describe and analyze an algorithm to solve Kris’s climbing problem if you are guaranteed that the input graph is a dag.

**Solution:** The solution to Kris’s problem is the longest path in the dag. This path can be found in $O(m + n)$ time using the dynamic programming algorithm described in class.

**Rubric:** 3 points = 1 for problem + 1 for algorithm + 1 for time. Yes, this is enough for full credit.

(c) Describe and analyze an algorithm to solve Kris’s climbing problem with no restrictions on the input graph.

**Solution:** The key observation is that whenever Kris touches a hold in any strong component of $G$, he can then touch every other hold in the same strong component, before moving on to the next strong component. So to find the best walk through the graph, we need to find a kind of weighted longest path in the strong-component graph of $G$.

Let $G$ be the input graph, and let $H$ be the strong-component graph of $G$. Let $S$ and $T$ denote be the strong components of $G$ that contain $s$ and $t$, respectively. For any strong component $C$ of $G$, let $(C)$ denote the number of vertices in $C$; we interpret this value as the weight of the corresponding vertex in $H$. Finally, define the weight of any path $C_1 \rightarrow C_2 \rightarrow \cdots \rightarrow C_t$ in $H$ to be the sum of the weights of its vertices: $\sum_{i=1}^{t} (C_i)$. We need to compute the maximum-weight path in $H$ from $S$ to $T$.

Our algorithm first computes the strong component graph $H$ and the weights $(C)$ for each of its nodes, using the Kosaraju-Sharir algorithm sketched in the notes. We then find the heaviest path from $S$ to $T$ via dynamic programming as follows. For any strong component $C$ of $G$, let $WHP(C)$ denote the weight of the heaviest path in $H$ from $C$ to $T$. This function obeys the following recurrence:

$$WHP(C) = \begin{cases} (T) & \text{if } C = T \\ (C) + \max\{WHP(C') | C \rightarrow C' \in E(H)\} & \text{otherwise} \end{cases}$$
We need to compute \( WHP(S) \). We memoize this function into the nodes of \( H \) and evaluate it in reverse topological order (via depth-first search). Altogether our algorithm runs in \( O(n + m) \) time.

For the sake of completeness, here is a more formal proof that this algorithm is correct.

**Claim 1.** For any path \( C_1 \rightarrow C_2 \rightarrow \cdots \rightarrow C_\ell \) in \( H \), there is a walk in \( G \) that visits at least \( \sum_{i=1}^{\ell} \#(C_i) \) vertices.

**Proof:** For each index \( 1 \leq i < \ell \), fix two vertices \( t_i \in C_i \) and \( s_{i+1} \in C_{i+1} \) such that \( t_i \rightarrow s_{i+1} \) is an edge in \( G \). Also fix two arbitrary vertices \( s_1 \in C_1 \) and \( t_\ell \in C_\ell \).

By definition, for any two vertices \( u, v \in C_i \), the graph \( G \) contains both a path from \( u \) to \( v \) and a path from \( v \) to \( u \). It follows inductively that there is a walk in \( G \) from \( s_i \) to \( t_i \) that visits every vertex in \( C_i \). Let \( s_i \rightarrow t_i \) denote this walk.

By stringing these walks together as

\[ s_1 \rightarrow t_1 \rightarrow s_2 \rightarrow t_2 \rightarrow s_3 \rightarrow \cdots \rightarrow t_{\ell-1} \rightarrow s_\ell \rightarrow t_\ell, \]

we obtain a walk in \( G \) that visits every vertex in every component \( C_i \), and therefore visits at least \( \sum_{i=1}^{\ell} \#(C_i) \) vertices in \( G \). \( \square \)

Call a walk in \( G \) thorough if, for every strong component \( C \) of \( G \), the walk either visits every vertex of \( C \) or none.

**Claim 2.** Any walk in \( G \) that visits the largest possible number of vertices is thorough.

**Proof:** Let \( W \) be an arbitrary walk, and let \( C \) be any strong component of \( G \). Suppose \( W \) visits some vertex \( u \in C \) but does not visit some other vertex \( v \in C \). By definition of “strong component”, \( G \) contains a walk from \( u \) to \( v \) and a walk from \( v \) to \( u \). Splicing these walks into \( W \) yields a new walk that visits more vertices than \( W \).

We conclude that \( W \) does not visit the largest possible number of vertices in \( G \). \( \square \)

**Claim 3.** For any thorough walk \( W \) in \( G \), there is a path in \( H \) whose weight is the number of vertices visited by \( W \).

**Proof:** Let \( W \) be an arbitrary thorough walk in \( G \), and let \( C_1, C_2, \ldots, C_\ell \) be the sequence of strong components of \( G \) that are visited by \( W \). Then \( W \) visits exactly \( \sum_{i=1}^{\ell} \#(C_i) \) vertices, which is the weight of the corresponding path \( C_1 \rightarrow C_2 \rightarrow \cdots \rightarrow C_\ell \) in \( H \). \( \square \)

Claim 1 implies that the best walk in \( G \) visits at least \( WHP(S) \) vertices. Claim 2 imply that the best walk in \( G \) visits at most \( WHP(S) \) vertices.

**Rubric:** 5 points = 2 for clearly describing the underlying problem (“heaviest path in \( SCC(G) \)”) + 2 for describing the algorithm (“Kosaraju-Sharir + dynamic programming”) + 1 for brief correctness argument (as in the first paragraph of this solution).
3. Describe and analyze an algorithm to solve the following problem. Your input consists of the following information:

- A directed graph $G = (V, E)$.
- Two vertices $s, t \in V$.
- A set of $k$ new edges $E'$, such that $E \cap E' = \emptyset$
- A length $\ell(e) \geq 0$ for every edge $e \in E \cup E'$.

Your algorithm should return the edge $e \in E'$ whose addition to the graph yields the smallest shortest-path distance from $s$ to $t$.

For full credit, your algorithm should run in $O(m \log n + k)$ time, but as always, a slower correct algorithm is worth more than a faster incorrect algorithm.

**Solution:** The algorithm has four steps:

1. Compute the shortest-path distance in $G$ from $s$ to every other vertex using Dijkstra’s algorithm, in $O(E \log V) = O(m \log n)$ time.
2. Compute the shortest path distance in the reversal of $G$ from $t$ to every other vertex using Dijkstra’s algorithm, again in $O(m \log n)$ time.
3. Find the edge $u \rightarrow v \in E'$ that minimizes the distance $\text{dist}(s, u) + \ell(u \rightarrow v) + \text{dist}(v, t)$, by brute force, in $O(k)$ time.
4. Finally, if $\text{dist}(s, u) + \ell(u \rightarrow v) + \text{dist}(v, t) < \text{dist}(s, t)$, return an error condition NONE; otherwise, return the edge $u \rightarrow v$.

The algorithm runs in $O(m \log n + k)$ time, as required. ■

**Rubric:** 10 points = 3 for the formula $\text{dist}(s, u) + \ell(u \rightarrow v) + \text{dist}(v, t)$ + 5 for algorithm details + 2 for time analysis. Max 5 points for the obvious $O(km \log n)$-time algorithm.
Solved Problem

4. Although we typically speak of “the” shortest path between two nodes, a single graph could contain several minimum-length paths with the same endpoints.

![Diagram of a graph with multiple paths]

Describe and analyze an algorithm to determine the number of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that all necessary arithmetic operations can be performed in O(1) time.

[Hint: Compute shortest path distances from s to every other vertex. Throw away all edges that cannot be part of a shortest path from s to another vertex. What’s left?]

Solution: We start by computing shortest-path distances dist(v) from s to v, for every vertex v, using Dijkstra’s algorithm. Call an edge u→v tight if dist(u) + w(u→v) = dist(v). Every edge in a shortest path from s to t must be tight. Conversely, every path from s to t that uses only tight edges has total length dist(t) and is therefore a shortest path!

Let H be the subgraph of all tight edges in G. We can easily construct H in O(V + E) time. Because all edge weights are positive, H is a directed acyclic graph. It remains only to count the number of paths from s to t in H.

For any vertex v, let PathsToT(v) denote the number of paths in H from v to t; we need to compute PathsToT(s). This function satisfies the following simple recurrence:

\[
\text{PathsToT}(v) = \begin{cases} 
1 & \text{if } v = t \\
\sum_{w \leftarrow v} \text{PathsToT}(w) & \text{otherwise}
\end{cases}
\]

In particular, if v is a sink but v ≠ t (and thus there are no paths from v to t), this recurrence correctly gives us PathsToT(v) = ∑ ∅ = 0.

We can memoize this function into the graph itself, storing each value PathsToT(v) at the corresponding vertex v. Since each subproblem depends only on its successors in H, we can compute PathsToT(v) for all vertices v by considering the vertices in reverse topological order, or equivalently, by performing a depth-first search of H starting at s. The resulting algorithm runs in O(V + E) time.

The overall running time of the algorithm is dominated by Dijkstra’s algorithm in the preprocessing phase, which runs in O(E log V) time.

Rubric: 10 points = 5 points for reduction to counting paths in a dag (standard graph reduction rubric) + 5 points for the path-counting algorithm (standard dynamic programming rubric)