CSE 421 Winter 2026: Section 2

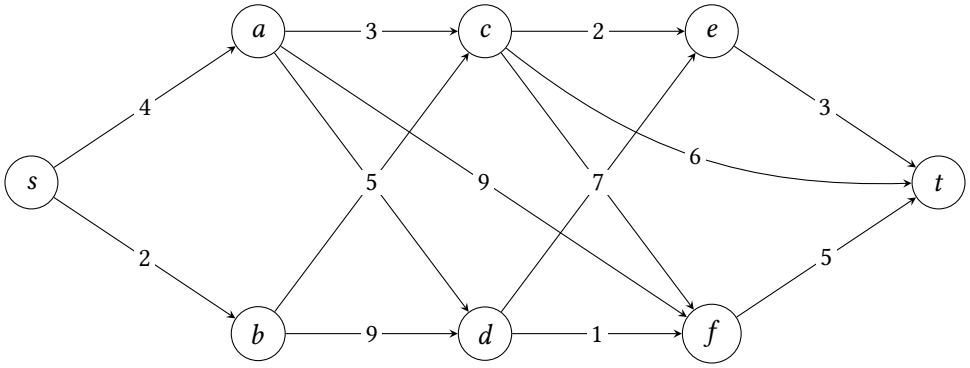
January 15, 2026

Instructions: This section worksheet is designed to assist you in working through some example problems and developing your basics. You are encouraged to collaborate on the problems on this section as well as use the course's generative AI.

1 Dijkstra's algorithm practice

Practice computing distances for this graph, starting from s, by implementing Dijkstra's algorithm by hand. Compute the tree of shortest paths from s. What is the order in which the vertices are processed?

Assume that when the algorithm is deciding which vertex to process when there are multiple at the same distance, it processes alphabetically first.



2 Step-by-step algorithmic walkthrough

A towing company needs to prepare the highway for a snowstorm. The problem input is a *sorted* array A of n integers, representing marked mile markers on the highway where we believe someone will need a tow. Find an efficient algorithm which computes where to park tow-trucks such that at least one tow-truck is ≤ 5 miles from *every* marked location. Compute *a* set of locations for truck placement that is minimal (there may be multiple solutions with the same number of trucks; we just want one solution).

Instructions: You are free to solve this problem however you like — the following "step-by-step" procedure illustrates our general technique for thinking about algorithms. Hopefully, the walkthrough will be helpful for you!

1. **What is the problem asking?** It is hard to solve a problem if you don't know what you are supposed to do. Reread the problem slowly, highlighting key parameters, requirements, and the format of the "return type" that the answer needs to be provided in. Ask yourself the following questions:

   - Are they are technical terms in the problem that you don't understand?
   - What is the input type? (ex., Array, Graph (directed/undirected/weighted), Integer, something else)?
   - What is the output type? Do I need to keep track of decisions I make (ex. assignments, weights, values, etc.) as I run the algorithm?
   - For a word problem, can you extract the core algorithmic question?

2. **What are some good examples?** Come up with some examples in an effort to ensure that you truly understand the problem. Generate a few sample instances and calculate the optimal solution.

   Did you notice any patterns while coming up with the solution? If so, what were they, and can those "patterns" be expressed algorithmically? If so, what properties might we need to prove? At this stage, the goal is to just come up with a few (2-3) examples, but it is important to keep an eye on the goal.

   In terms of examples, our suggestion is to **not** focus on base or edge cases. Rather, we want to understand the general behavior of the algorithm. Come up with examples that are decently large ($n \approx 5 - 10$) and are sufficiently different. For example, if the problem is described by a graph, make sure that at least one example is *non-planar*.

3. **What is a baseline?** In a time-constrained setting (ex., interview or exam), it is good to come up with a "baseline" or "brute-force" algorithm. It may not be optimal, but it can help give you a baseline to compare any future improvement against.

4. **What problems does this remind me of?** Since it is only the second week of the course, our collection of algorithmic techniques is small. By the end of the course, you will know quite a few different methods, so it will be important to garner an intuition about the problems that each method is good for. A great starting point is identifying all the problems you have seen that are similar. Try thinking through the following questions:

   - Does this remind me of any prior problems? What techniques did we use there?
   - Does the problem look like a *standard* algorithm when turned on its head? For example, do I need to create a graph or adjust the data and *then* run a standard algorithm on the new graph/data?

- Is there some structure to the input? For example, is it 1-dimensional (like this problem!)?
- Is there some structure to the output? If we need to make multiple selections or decisions, can we make them *greedily*? In this case, greedy means that once a decision is made, we never have to go back and revisit it. In greedy algorithms, we only make decisions when we are certain (provably) that they are optimal.

For today, we will be using a greedy algorithm. Come up with a few plausible ideas for deciding where to place the tow-trucks greedily.

5. **Write an algorithm.** Try running your algorithms against your examples. Did something click? If one of your algorithms seems to be working, construct an example problem that bucks your algorithm – meaning make sure every path through your example is explored. If you have an "if" argument in your algorithm, make sure there exist examples that apply to both sides of the "if" statement. If you cannot come up with such an example, perhaps the "if" statement wasn't needed.
6. **Prove that your algorithm is correct.** As we explore more algorithms, we will also come up with more techniques for proving correctness. For this problem, there are a couple of equivalent proofs of correctness. Any will do; come up with the one that feels most natural for you to write. Remember to prove the following two requirements: (1) your algorithm outputs a valid solution; in this case, a set of tow-truck locations covering all marked locations. And (2) that the number of parking locations is minimal.

3 Additional practice algorithms

3.1 Commuting across Seattle

Little Johnny is trying to commute from locations s to t in Seattle. He has an electric scooter and a coupon for one free ride on the 1-Line metro. He has modeled Seattle as a weighted directed graph $G = (V, E)$ with $w : E \to \mathbb{R}_{\geq 0}$ representing the electricity cost on his electric scooter. On the graph, assume $v_1, v_2, \ldots, v_k$ represents the 1-Line metro stations.

Construct an algorithm that finds the *cheapest* path for Little Johnny, keeping in mind he can ride the 1-Line metro at most once.

Hint: Construct a graph G' with $2n + k$ vertices (there could be alternate constructions as well).

3.1.1 Back and cross edges

In lecture, we saw that an undirected connected graph has only back edges and tree edges (no cross edges) with respect to any DFS tree.

For this problem, show that an undirected connected graph has only cross edges and tree edges (no back edges) with respect to any BFS tree.

1. First, come up with a graph and a choice of source s for which running BFS yields both cross and tree edges.
2. Second, come up with a proof that there are no back edges with respect to BFS trees.

3.2 Interval covering

The input is a set of intervals $[a_i, b_i]$ for $i = 1, \ldots, n$ and $a_i < b_i$. The output is a minimal set of points $x_1, \ldots, x_k$ such that for every interval $[a_i, b_i]$, there exists some point $x_j \in [a_i, b_i]$.

1. Before solving this problem, come up with at least two "word problems" that are morally equivalent to this bare-bones mathematical problem. This is a good practice for ensuring that you understand what the mathematical question is asking.
2. Show how this problem is a generalization of the "tow-truck" problem.
3. Briefly explain how to generalize your solution to the "tow-truck" problem to satisfy this problem.

3.3 BFS layers

Let $G = (V, E)$ be an undirected, unweighted graph, and let $s \in V$ be a fixed source vertex. Run the Breadth-First Search (BFS) algorithm starting from s. For each integer $i \geq 0$, define the BFS layer

$$L_i = \{v \in V : \text{dist}(s, v) = i\},$$

where $\text{dist}(s, v)$ denotes the length of a shortest path from s to v.

1. Prove that every path in G from s to a vertex $v \in L_d$ must contain at least one vertex from each layer $L_0, L_1, \ldots, L_d$.
2. Prove that every path in G of length d from s to a vertex $v \in L_d$ contains *exactly one* vertex from each layer $L_0, L_1, \ldots, L_d$.

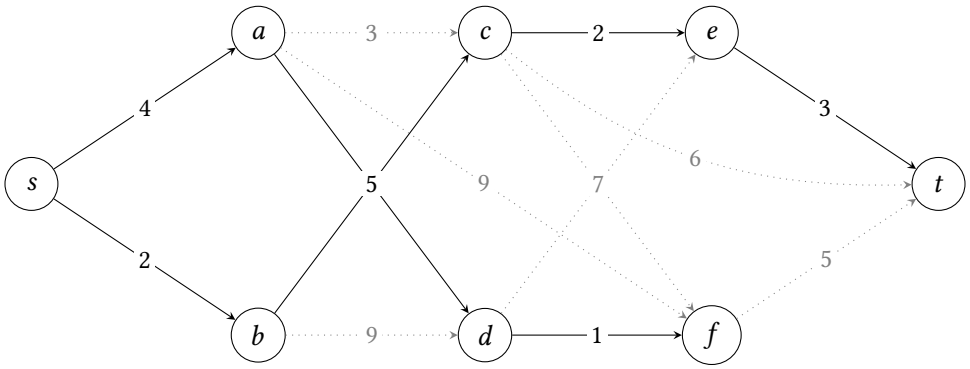
4 Solutions

4.1 Dijkstra's algorithm practice

The vertices are the following distances:

$$s = 0, a = 4, b = 2, c = 7, d = 9, e = 9, f = 10, t = 12.$$

The visit order is s, b, a, c, d, e, f, t.



4.2 Step-by-step algorithmic walkthrough

1. **What is the problem asking?** Here, "satisfying a marked location" means "place a truck at most 5 (miles) from the location." The input type is a list of numbers. The return type is a list of numbers. A core algorithmic question might be: Find a minimal set P such that

   $$\forall\, i \in [n],\ \exists\, p \in P \text{ s.t. } |A[i] - p| \leq 5.$$

2. **Examples.** There are multiple examples and outputs you can pick for the best solution. Here are some we thought of:

   - Input: [0,1,2,3,4,5,6,7,8,9,10,11,12] Output: [3,7]
   - Input: [0,6,7,8,9,13] Output: [0,10]
   - Input: [0,1,2,3, 100,101,102,103] Output: [1.5,101.5]

3. **Baseline.** Place a tow-truck at every marked location. A non-optimal algorithm, but it runs in $O(1)$ time.

4. **Similar problems?** The one-dimensional nature suggests a greedy algorithm might be good. Particularly because the choice of where to place a tow-truck to satisfy the first few locations will probably

have little to no relevance on where to place a tow-truck to satisfy later solutions.

This reminds us of interval scheduling problems.

5. **Algorithm.** Let A be the *sorted* input with $A[1] < A[2] < \ldots < A[n]$. Initialize $P \leftarrow \{\}$, which will be the set of tow-truck locations, and let $j \leftarrow 1$. Add a tow-truck at $P \leftarrow P \cup \{A[j]+5\}$ and then linearly traverse the array A to find the first index j' such that $A[j'] > A[j] + 10$. Then redefine $j \leftarrow j'$ and repeat until the entire array is parsed.

   Runtime. The algorithm parses the array A linearly and makes one pass through the array with decisions at each index taking $O(1)$ time. This yields an $O(n)$ runtime.

6. **Correctness.** We employ a "greedy-stays-ahead" strategy for our proof. Let OPT be the locations selected by any optimal strategy (sorted in increasing order) and let ALG be the locations selected by our greedy strategy (also sorted in increasing order). We prove by induction that the number of marked locations covered by the first i elements of OPT is at most that covered by i elements of ALG.

   For a base case of $i = 1$, notice that by placing the first tow-truck at $A[1] + 5$ we cover location 1 and, since the array is sorted, the maximal number of future locations. For induction, let j be the first index not covered by the first i elements of ALG. By induction, it is not covered by the first i elements of OPT, and every prior location is covered by both strategies. Because ALG includes the $i+1$-th tow-truck at $A[j] + 5$, we cover the j-th location and at least as many locations as OPT, as every novel location covered by the $i+1$-th truck of OPT is also covered by ALG. This proves the minimality of ALG by induction. Observe that this argument also proves feasibility.

4.3 Practice algorithms

4.3.1 Commuting across Seattle

Algorithm. We reduce to a shortest-path computation on a layered graph. Construct a directed graph $G' = (V', E')$ with three layers:

$$V' = \{(v, 0) : v \in V\} \cup \{(v_i, 1/2) : i \in \{1, \ldots, k\}\} \cup \{(v, 1) : v \in V\},$$

where layer 0 means "metro not yet used", layer $1/2$ means "metro in use" and layer 1 means "metro already used". Then, add the following edges.

1. For every original edge $(u, v) \in E$, add edges $(u, 0) \to (v, 0)$ and $(u, 1) \to (v, 1)$, each with weight $w(u, v)$. (Scooter travel is always allowed.)

2. For each $i \in \{1, \dots, k-1\}$, add edges $(v_i, 1/2) \leftrightarrow (v_{i+1}, 1/2)$ (in both directions) of weight 0. (The metro is free).

3. For $i \in \{1, \dots, k\}$, add edges $(v_i, 0) \to (v_i, 1/2)$ and add $(v_i, 1/2) \to (v_i, 1)$ of weight 0. (This represents getting on or getting off the metro.)

Now run Dijkstra's algorithm from $(s, 0)$ in G'. The answer is

$$\min\left\{\text{dist}_{G'}((s,0),(t,0)),\ \text{dist}_{G'}((s,0),(t,1))\right\},$$

and we recover a corresponding path in G by projecting away the layer labels with travel between layer $1/2$ representing metro travel.

Runtime. The graph G' has $2n + k$ vertices. It has $2m$ scooter edges and $O(k)$ metro edges, so $|E'| = O(m + k)$. Running Dijkstra's algorithm takes $O(|E'| \log |V'|) = O((m + k) \log n)$ time.

Correctness. We establish a one-to-one correspondence between valid s-to-t routes in the original problem and paths in G' from $(s, 0)$ to either $(t, 0)$ or $(t, 1)$, preserving total cost.

Given any valid route in G in which Johnny rides the metro at most once, map scooter moves $(u \to v)$ to the corresponding edge within the current layer (either $(u, 0) \to (v, 0)$ or $(u, 1) \to (v, 1)$) with the same cost $w(u, v)$. If the route ever uses the metro, map the first metro step to an edge from layer 0 to layer $1/2$ of weight 0, and map all subsequent metro steps with layer-$1/2$ metro edges of weight 0 followed by an edge from layer $1/2$ to layer $1/$ for getting off the metro. This produces a path in G' whose total weight equals the scooter electricity cost of the route (metro contributes 0).

Conversely, any path in G' starting at $(s, 0)$ can switch from layer 0 to layer 1 at most once (since there are no edges deceasing in layer). Therefore, projecting the path to G yields a route that uses the metro at most once. Scooter edges contribute exactly their original weights, and metro edges contribute 0, so the total weight of the path equals the total cost of the route.

Thus, minimizing route cost in the original problem is equivalent to finding a shortest path in G' from $(s, 0)$ to $(t, 0)$ or $(t, 1)$, as it is not required that Johnny takes the metro. Dijkstra's algorithm returns such a shortest path, so the algorithm outputs the cheapest valid route.

4.3.2 Back and cross edges

1. Take the 4-cycle (a square) with vertices $\{s, a, b, c\}$ and edges

   $$\{(s, a), (a, b), (b, c), (c, s)\}.$$

   Run BFS from s. One possible BFS tree chooses tree edges (s, a) and (s, c) (discovering a and c at distance 1), and then discovers b from a using tree edge (a, b). Now the edge (b, c) is a *non-tree*

edge. Neither b is an ancestor of c nor c an ancestor of b in the BFS tree, so (b, c) is a *cross edge*. Thus, BFS can produce both tree edges and cross edges.

2. Let $\text{dist}(s, v)$ denote the shortest-path distance in the unweighted graph. BFS ensures that for every vertex v, its depth in the BFS tree equals $\text{dist}(s, v)$. By triangle inequality, for any edge $\{u, v\}$ in an undirected unweighted graph, $|\text{dist}(s, u) - \text{dist}(s, v)| \leq 1$.

   Suppose for contradiction that $\{u, v\}$ is a back edge, where (wlog) v is a strict ancestor of u in the BFS tree. Let the BFS-tree depth of a vertex x be $\ell(x)$. Because BFS-tree depth equals shortest distance, $\ell(x) = \text{dist}(s, x)$ for all x. If v is a *strict* ancestor of u, then by being a back edge, the tree path from v down to u has length at least 2, so $\ell(u) \geq \ell(v) + 2$. This contradicts the triangle inequality, and so no back edge can exist.

4.3.3 Interval covering

1. Some other ideas could be appointment slots that need to be covered by doctors or security camera placement. The general theme is "covering".

2. This problem is conceptually similar to the tow-truck problem. For each location $A[i]$, construct interval $[a_i, b_i] = [A[i] - 5, A[i] + 5]$. Observe that covering the interval is equivalent to a valid tow-truck placement.

3. Let's sort the intervals by increasing end-time b_i. We now include a new point at the very end of the first uncovered interval. The prior proof can be lightly adjusted to fit this general problem. Observe that the runtime does change, however as sorting requires $\Theta(n \log n)$ time.

4.3.4 BFS Layers

We rely on the fact that $v \in L_i$ iff $\text{dist}(s, v) = i$. On the path s to $v \in L_d$, every distance must be covered since the distance from s of adjacent vertices differs by at most 1 (by the triangle inequality). By the equivalence of distance and layer index, at least one vertex from each layer must be included on the path. For the second direction, since the path has length d, by the pigeonhole principle, exactly one vertex from each layer must exist.