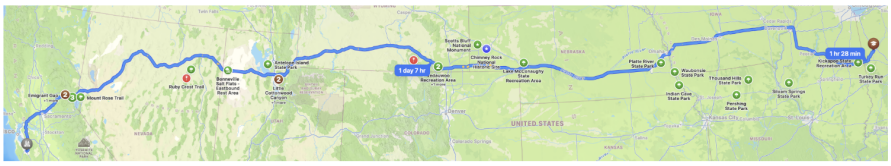
Exercise 5.10. Imagine you are planning a long road trip to San Jose, CA and you've already fixed your route (through Illinois, Iowa, Nebraska, Wyoming, Utah, Nevada, and then California via Tahoe).



You don't mind the drive, but you hate having to stop to get gas. What a pain! And some stations take longer than others. The car can drive at most 500 miles before needing to stop to get gas. The high-level goal is to plan your gas stops to minimize the amount of time spent pumping gas, while making sure you don't run out of gas between them.

Design and analyze an algorithm to compute the minimum amount of time one has to spend at gas stations in going from location 0 to location Z, while ensuring that you never drive more than 500 miles without visiting a gas station. (You should return $+\infty$ if it is impossible to do so.)

Guiding Questions

Our problem is to design an algorithm, that given distances $D[1, n]$, times $T[1, n]$ and destination Z, compute the minimum time to get to Z, stopping for gas every 500 miles or less.

1. Write out your recursive spec. What would be good to keep track of?

   Let DP(_____) =the minimum time it takes to get to _____,
   given that we do not go over 500 miles per station.

2. *Recursive spec implementation.* This is clearly a minimization problem, so we just need to fill in the blanks:

$$\texttt{DP}(\_\_\_) = \min_{i \in [n] \text{ such that } \_\_\_\_\_ < 500} (\texttt{DP}(\_\_\_) + \_\_\_\_\_) .$$

3. How do we use this to solve the original problem?

Solution: We describe the DP algorithm $\texttt{DP}(i)$ as

$\texttt{DP}(i)$:the minimum time it takes to get to gas station i,
given that we do not go over 500 miles per station.

This can be implemented as the recurrence,

$$\texttt{DP}(i) = \min_{\substack{j<i \\ D[i]-D[j] \leq 500}} (T[i] + \texttt{DP}(j)) ,$$

with the final output being computed as,

$$\texttt{Output} = \min_{\substack{i \in [n] \\ Z - D[i] \leq 500}} \left(\texttt{DP}(i) + \underbrace{(Z - D[i])}_{\text{Rem. dist from } i\text{'th station}} \right).$$

If there is no such $i \in [n]$ such that $Z - D[i] \leq 500$ and $\texttt{DP}(i)$ is defined, return $\perp$.

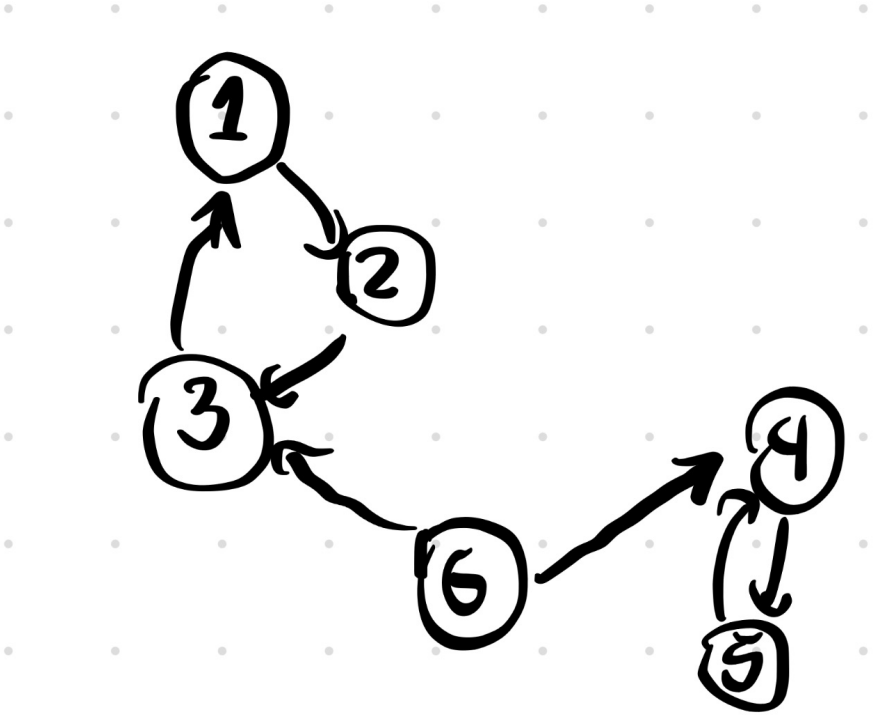
This solution takes $O(n^2)$ time because we may cache each prior $\texttt{DP}(i)$ call, for $i \in [n]$, where each call takes $O(n)$ time to compute the minimum.

Remark. This solution can actually be implemented in $O(n)$ time via a sliding window type of approach. Suppose we have access to a queue data structure `Q` that supports `Q.{front,back,isempty,enq,popFront,popBack}`. Since each $D[i]$ is sorted, we can approach $\texttt{DP}(i)$ by keeping a sorted queue `Q` of candidate $\texttt{DP}(j)$ values. We can maintain this queue by enqueuing/dequeueing each $i \in [n]$ at most once, leading to a $O(n)$ solution.

$\texttt{DP}(i)$

1. **(Base case)** If $i = 0$ output 0. ($\texttt{DP}(0) \leftarrow 0$)
2. Set `Q` to be a queue initially with element 0.
3. For $i = 1, \ldots, n$:
   (a) While $D[i] - D[\texttt{Q.front()}] > 500$, run `Q.popFront()`. If `Q` is empty, set $\texttt{DP}(i) \leftarrow \infty$.
   (b) Otherwise, set $j \leftarrow$ `Q.popFront()`. (First j s.t. $D[i] - D[j] \leq 500$)
   (c) Set
   $$\texttt{DP}(i) \leftarrow T[i] + \texttt{DP}[j].$$
   (d) While `DP[Q.back]` $\geq$ `DP[i]`, run `Q.popBack()`.
   (e) Run `Q.enq(i)`

□



Problem (Warmup Questions/Test Your Understanding).

Searching Graphs

1. Count the number of strongly connected components in the following graph.
2. In the same graph, find the topological sort of the condensed graph (where each scc is a single vertex). How many topo sorts exist?
3. (True/False) In a graph $G = (V, E)$, all vertices reachable from vertex $v \in V$ are marked by `DFS`(v).
4. (True/False) In the post DFS-ordering, the first vertex is always the sink.

Shortest Paths

1. (True/False) Dijsktra's algorithm is a *single-source* shortest paths algorthm. That is, Dijkstra only finds the shortest path of a single source.
2. (True/False) Dijkstra with a priority queue takes $O(m \log n)$ time

Exercise 6.4. Let $G = (V, E)$. Design and analyze an algorithm for each of the following problems.

1. Decide if there is a vertex x such that x can reach all other vertices.
2. Decide if there is a vertex x such that x can be reached by all other vertices.
3. Decide if there is a vertex x such that every vertex can either reach x or be reached from x.

Guiding Questions

1. How do you check reachability of a single vertex x? That is, how do you compute the vertices v such that there is a path from x to v? Note that part 1 asks us to decide if such a vertex x exists, so we do not know which vertex x before hand.
2. Recall one can visit all vertices using `DFS-Driver` from the lecture notes (i.e., for each unmarked vertex v remaining, run `DFS(v)`). What can you say about the last unmarked vertex that `DFS-Driver` runs `DFS` on?
3. Hint for part 2 : *change the graph, not the algorithm.*
4. Hint for part 3 : cycles can be annoying when trying to solve this problem. How can we 'condense/contract' them?

Solution:

1. We want an algorithm to decide if there exists a vertex $x \in V$ such that every vertex $v \in V$ is reachable from x within the graph $G = (V, E)$. Consider the cannonical `DFS-Driver`$(G = (V, E))$ for visiting all vertices in the graph using `DFS`(v):

   `DFS-Driver`$(G = (V, E))$

   `DFS`(v)

   If v is unmarked:

   (a) Mark v,

   (b) For each edge $(v, w) \in E$ (i.e., each edge leaving v): run `DFS(w)`

   Append v to the output.

   (a) For $v \in V$: run `DFS`(v)

   Observe that if there exists a such a vertex x, then it will be the last chosen vertex in `DFS-Driver` when all vertices $v \in V$ are marked[1] (Take a second to confirm that the

[1]Suppose not, then there is some other vertex w that is visited last by `DFS-Driver`. This implies that w

converse is not necessarily true – think about how this is incorporated into our solution). Thus, we can modify the `DFS-Driver` to solve our problem as follows:

`DFS-Driver-Mod`$(G = (V, E))$

(a) For $v \in V$:

i. run `DFS`(v)

ii. If all vertices are marked, break out of the loop and let `candidate` $= v$.

Unmark all vertices and run `DFS`(`candidate`). If all vertices are marked, output `YES`. Otherwise, output `NO`.

This takes $O(n + m)$ time, following from the usual analysis of `DFS-Driver` and the fact that `DFS-Driver-Mod` only runs one more additional `DFS` than `DFS-Driver`.

2. Let G_{rev} be the graph where we reverse all edges in G, i.e., $G_{rev} = (V, E_{rev})$ where $E_{rev} = \{(v, u) : (u, v) \in E\}$. Then, output `DFS-Driver`$(G_{rev})$. Since computing G_{rev} takes $O(m)$ time, this algorithm takes $O(n + m)$ time.

3. Recall that strongly connected components can be found in $O(n + m)$ time. Consider the *condensed* graph $G' = (V', E')$ formed by condensing each strongly connected component $C_1, \ldots, C_k \subseteq V$ into a single vertex $c_1, \ldots, c_k \in V'$. Then, such a vertex x exists iff the DAG formed by the condense graph is a single connected component, which can be checked in $O(n + m)$ time with `DFS`.

   Remark. You can find such a vertex in $O(m + n)$ time as well. Observe that there is a vertex x that every vertex can reach or be reached from iff (1) x is reachable from every source of G' ($v \in V'$ with $\text{indeg}(v) = 0$), and (2) every sink of G' ($v \in V'$ with $\text{outdeg}(v) = 0$) is reachable from x.

   Let C_{in} be all the sources of G' and let C_{out} be all the sinks of G'. Then, for each $c_i \in C_{in}$ let r_i be the nodes reachable from c_i in G'. Symmetrically, for each $d_j \in C_{out}$, let q_j be the nodes that go to sink d_j. Then, output `YES` iff

$$\bigcap_{\substack{c_i \in C_{in} \\ d_j \in C_{out}}} (r_i \cap q_j) \neq \emptyset.$$

   Each r_i can be obtained by running `DFS` at each source in G'. Similarly, each q_j can be obtained by running `DFS` on the reversed graph of G'. Importantly, we only visit each vertex and edge a constant number of times (by marking them), so this takes $O(n+m)$ time.

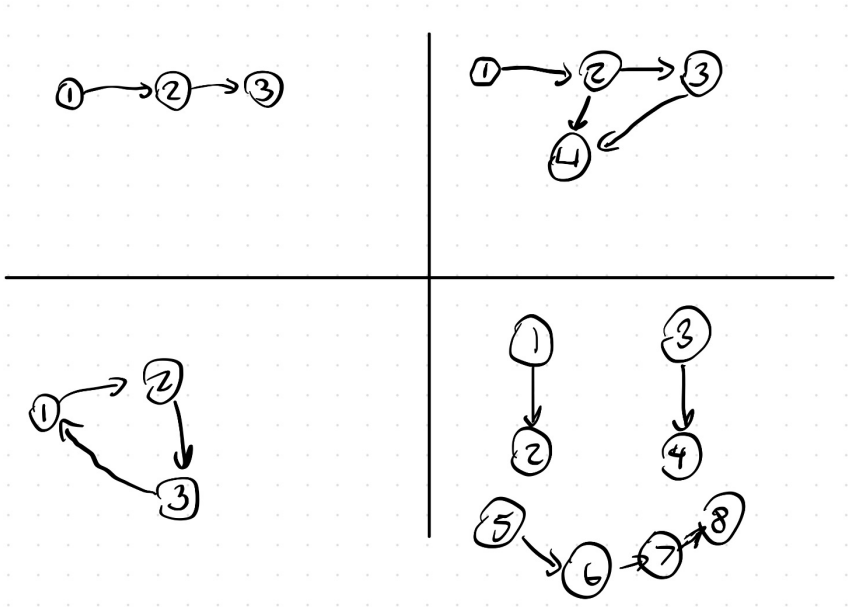
□

visits a vertex not reachable from v, a contradiction.

Exercise 7.1. Each of the following problems describes a graph problem over an unweighted directed graph $G = (V, E)$ with m edges and n vertices. Each problem can be solved by constructing an auxiliary graph and calling a shortest path algorithm from this chapter as a black box. For each problem, (a) design an auxiliary graph and shortest path problem, (b) indicate which algorithm to apply and how, and (c) analyze the running time. No proof of correctness is required.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[3]
2. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is a multiple of 5.

Hint: Don't change the algorithm. Change the graph. I drew a few graphs for you to play around with. Solution:



1. Let $G = (V, E)$ be our graph. Then, we create a new graph $G' = (V', E')$ that encodes the parity of edges. Create two vertex copies $V_{even} = V$ and $V_{odd} = V$ that are disjointly labeled. [2] Then, for any edge $(u, v) \in E$ of the original graph, we map each edge such that they only go from V_{even} to V_{odd} and vice versa. Formally, define

$$E' = \{(u_{even}, v_{odd}, wt), (u_{odd}, v_{even}, wt) \mid (u, v, wt) \in E\},$$

and set $V' = V_{even} \cup V_{odd}$.

Then, to solve the problem, run Dijstra on G' on input source vertex s_{even} and output the shortest path to t_{even}. To prove correctness, we show that any shortest path from

[2] I.e. if $V = \{v_1, \ldots, v_n\}$, then $V_{even} = \{v_{1,even}, \ldots, v_{n,even}\}$ and $V_{odd} = \{v_{1,odd}, \ldots, v_{n,odd}\}$ such that $V_{even} \cap V_{odd} = \emptyset$.

s to t of even length in the original graph G has a one-to-one correspondence to the shortest path from s_{even} to t_{even} in G' (a bijective argument). One can do this by (1) providing a forward mapping of shortest path from s to t of even length in the original graph G to shortest path from s_{even} to t_{even} in G', and (2) provide an inverse mapping.

2. Similar solution as the previous part, except make 5 copies of the graph

□