

Graph Modeling and DP

Lecture 17

Part I

Graph Modeling

Graph Algorithms

What have we learnt so far about graphs and graph algorithms?

- Whatever-first search (WFS) for graph reachability; DFS and BFS as special cases that give more structure.
- Basic structure of undirected graphs via connected components.
- Basic structure of directed graphs via strong connected components and meta-graph (which is a DAG)
- DAGs and topological sort.
- Algorithms for connected components, strong connected components, meta-graph, topological sort, cycles.
- Shortest Path Algorithms.

Graphs

Why are graphs and graph algorithms very useful?

- Because they can be used to *model* many problems *directly*.
- Because graphs are flexible, one can also *transform* graphs into other graphs and then use standard algorithms.

Graphs

Why are graphs and graph algorithms very useful?

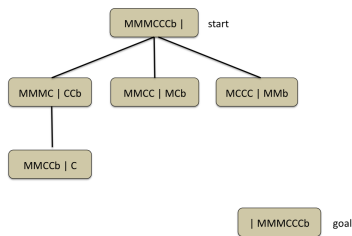
- Because they can be used to *model* many problems *directly*.
- Because graphs are flexible, one can also *transform* graphs into other graphs and then use standard algorithms.

Goal: highlight a few useful modeling tricks.

Graphs to model states and transitions

Recall: we modeled the missionaries and cannibals puzzle as a graph.

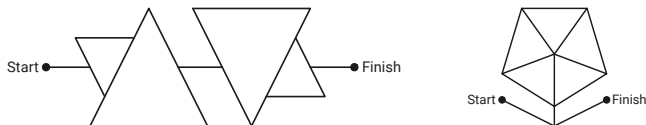
- Vertices: *states* (who is on which bank, where the boat is).
- Edges: legal *moves* from one state to another.
- Solution: a path from the start state to the goal state.



Many puzzles can be modeled similarly. The trick is to choose the right notion of *state*.

States and transitions: acute-angle maze

Trace a path from Start to Finish that has only straight or *acute* turns.



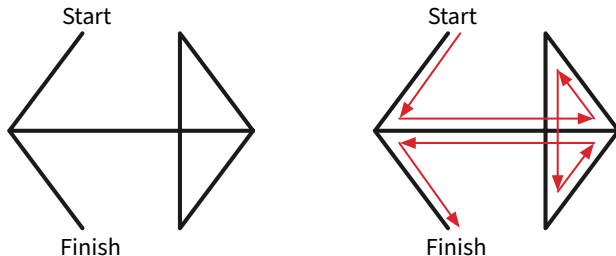
- Input: connected undirected graph G drawn in the plane with straight-line edges, and two vertices Start and Finish.
- A walk is *valid* if for any two consecutive edges $u \rightarrow v \rightarrow w$: $\angle uvw = \pi$ (straight) or $0 < \angle uvw < \pi/2$ (acute).
- Goal: decide whether there is a valid walk from Start to Finish.

Question: why can't we just run WFS in G ?

Puzzles from Batterson and Rogers. *Beast Academy Math: Practice 3A*, 2012.

Acute-angle maze: Why not search G ?

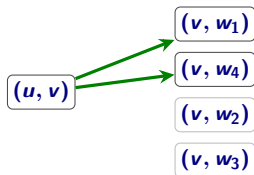
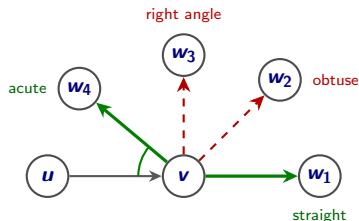
WFS in G visits each vertex once, so it only finds *simple paths*. A valid walk may have to visit the same vertex several times.



A maze (left) and its unique solution (right). The solution passes through some vertices more than once, so no simple path in G describes it.

Acute-angle maze: Solution

Which edge we may take next depends on where we came from. So the state is the current vertex *and* the previous vertex.



part of the new graph G'

New directed graph G' : vertex (u, v) means the token is at v and was previously at u . Edge $(u, v) \rightarrow (v, w)$ exactly when the turn at v is straight or acute. Then run WFS in G' .

Acute-angle maze: Solution (details)

- $V' = \{(u, v) \mid uv \in E\} \cup \{\text{Start}\}$: ordered pairs, so both (u, v) and (v, u) . Start is kept since the token has no previous vertex at first. $2m + 1$ vertices.
- Edges: $\text{Start} \rightarrow (\text{Start}, v)$ for each edge (Start, v) , and $(u, v) \rightarrow (v, w)$ iff $\angle uvw$ is straight or acute.
- The maze is solvable iff some (v, Finish) is reachable from Start in G' .
- WFS from Start in G' , then scan the vertices (v, Finish) for a marked one.
- At most one edge of G' per edge $uv \in E$ and vertex $w \in V$: $O(mn)$ time.
- For a valid walk with the fewest edges: BFS instead of WFS.

Lesson: put just enough of the past into the state.

States and transitions: red/blue walk

Here the state needs more than the current vertex.

Let $G = (V, E)$ be a directed graph. Each vertex is colored red, colored blue, or uncolored; let $R \subseteq V$ be the set of red vertices, $B \subseteq V$ the blue vertices, and $U \subseteq V$ the uncolored vertices.

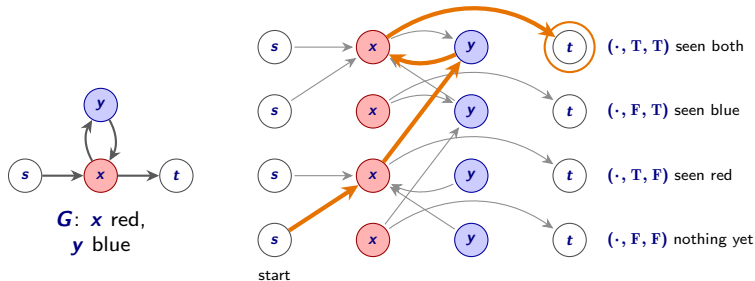
Describe an efficient algorithm that, given distinct vertices s and t , finds whether there is a walk from s to t that contains at least one red vertex and at least one blue vertex. Assume for simplicity that s and t are uncolored.

Red/blue walk: Solution

State (u, r, b) : current vertex u ; r = have we visited a red vertex? b = have we visited a blue vertex? (True/False)

- Start in $(s, \text{False}, \text{False})$. Walking along $u \rightarrow v$ from state (u, r, b) , the new state is (v, True, b) if $v \in R$, (v, r, True) if $v \in B$, and (v, r, b) if $v \in U$.
- $G' = (V', E')$ with $V' = V \times \{\text{True}, \text{False}\} \times \{\text{True}, \text{False}\}$. For each $(u, r, b) \in V'$ and each edge $u \rightarrow v \in E$, E' has the edge $(u, r, b) \rightarrow (v, r \vee (v \in R), b \vee (v \in B))$.
- G has an s - t walk with a red and a blue vertex iff $(t, \text{True}, \text{True})$ is reachable from $(s, \text{False}, \text{False})$ in G' .
- G' has $4n$ vertices and $4m$ edges. Whatever-first search (WFS): $O(n + m)$ time.

Red/blue walk: Solution (example)



$T = \text{True}$, $F = \text{False}$. Highlighted: the walk $s \rightarrow x \rightarrow y \rightarrow x \rightarrow t$ reaches (t, T, T) . The walk $s \rightarrow x \rightarrow t$ only reaches (t, T, F) . Note that a walk may repeat vertices.

Reversing the arcs of a directed graph

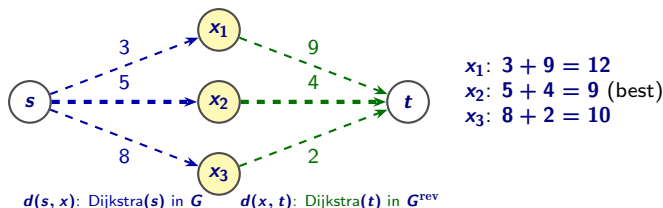
Given directed graph G reverse the graph to find paths *to* a node t rather than *from* a node s .

Example problem: You want to go from your house to a friend's house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the “shortest” trip if you include this stop?

Given $G = (V, E)$ with edge lengths $\ell(e) \geq 0$, home s , friend t , and stores $X \subseteq V$, find $\min_{x \in X} (d(s, x) + d(x, t))$.

Dessert stop: Solution

- Dijkstra from s in G gives $d(s, x)$ for all x .
- Dijkstra from t in G^{rev} gives $d(x, t)$ for all x : a path from x to t in G is a path from t to x in G^{rev} .
- Answer: $\min_{x \in X} (d(s, x) + d(x, t))$. Two runs of Dijkstra: $O(m + n \log n)$ instead of $2|X|$ runs.

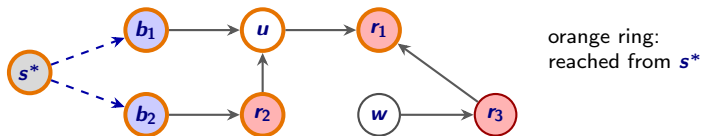


Adding a super source or super sink or both

- Given *directed* graph $G = (V, E)$.
- Two subsets of nodes $R \subset V$ (red nodes) and $B \subset V$ (blue nodes).
- Describe linear-time algorithm to decide whether *every* red node can be reached by *some* blue node. That is, for every red node $r \in R$ there must be some blue node $b \in B$ that can reach r . Note that the same blue node does not need to reach two different red nodes.

Super source: Solution

- Add a new node s^* and arcs (s^*, b) for every $b \in B$.
- One WFS from s^* : $O(n + m)$ time. Answer is YES iff every $r \in R$ is reached.
- Correct because s^* reaches r iff some blue node reaches r .
- Variant: can every red node reach *some* blue node? Add a super sink t^* with arcs (b, t^*) , and run WFS from t^* in G^{rev} .



Here r_3 is not reached, so the answer is NO.

Distance between sets of nodes

Given $G = (V, E)$ with edge lengths $\ell(e) \geq 0$ and two sets $S, T \subseteq V$, find

$$\text{dist}(S, T) = \min_{s \in S, t \in T} \text{dist}(s, t).$$

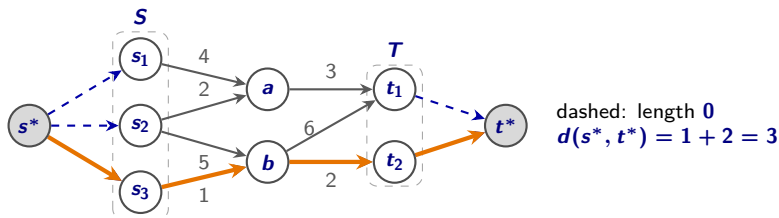
Running Dijkstra from every $s \in S$ takes $O(|S| (m + n \log n))$ time.

Can we do it with a single run of Dijkstra?

We have seen a super source before: Bellman-Ford from a new node with 0-length arcs to every node detects any negative cycle.

Distance between sets: Solution

- Add s^* with an arc (s^*, s) of length 0 for every $s \in S$.
- Add t^* with an arc (t, t^*) of length 0 for every $t \in T$.
- One Dijkstra from s^* : $\text{dist}(S, T) = d(s^*, t^*)$, in $O(m + n \log n)$ time.



Here t^* is only a convenience: we could take $\min_{t \in T} d(s^*, t)$. Both are essential when an algorithm takes a single source and a single sink, e.g. maximum flow with many sources and sinks. In undirected graphs keep the new edges directed: otherwise s^* creates 0 -length shortcuts between nodes of S .

Reducing to DAGs via the meta-graph

A template for directed graphs: solve the problem when G is strongly connected, then when G is a DAG, then lift the solution to a general G via the meta-graph G^{SCC} (which is a DAG). For example: is G semi-connected? Can adding one edge make G strongly connected?

Problem: A node u of a directed graph G is *good* if u can reach every node of G . Describe a linear-time algorithm that checks whether G has a good node, and outputs one if it does.

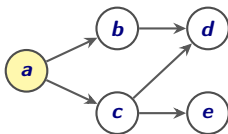
Checking one node is easy: one WFS. Checking every node this way takes $O(n(n + m))$ time.

Warm-up: draw a DAG with a good node and a DAG without one.

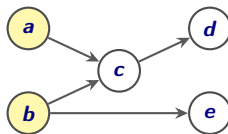
Good node: Solution for DAGs

Lemma. A DAG has a good node iff it has exactly one source; then that source is the good node.

- Two sources $u \neq v$: a source is reached only by itself, so no node reaches both u and v .
- One source u : suppose u misses some node, and let v be the first missed node in a topological order. v is not a source, so it has an in-arc (w, v) with w earlier. u reaches w , hence v . Contradiction.
- Algorithm: count the nodes with in-degree 0. $O(n + m)$ time.



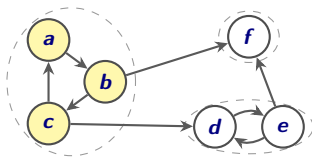
one source: a is good



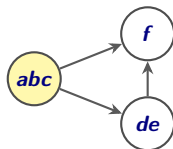
two sources: no good node

Good node: Solution for general graphs

- Nodes in the same SCC reach exactly the same nodes. So u is good in G iff its SCC is good in G^{SCC} .
- G has a good node iff G^{SCC} has exactly one source. Then every node of that source SCC is good.
- Compute G^{SCC} in $O(n + m)$ time and count its sources: $O(n + m)$ total.



G with its SCCs: a, b, c are good



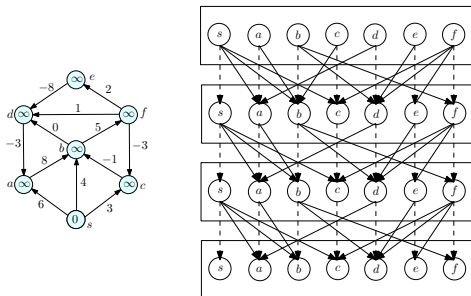
G^{SCC} : one source

Alternative: the node with the largest post number in DFS(G) is in a source SCC; one WFS from it checks whether it is good.

Layering

Build a new graph from “copies” of G (or of V) and connect the copies to enforce constraints. Recall Bellman-Ford:

- $V' = V \times \{0, 1, \dots, k\}$; arc $((u, i), (v, i + 1))$ of length $\ell(u, v)$ for each $(u, v) \in E$. G' is a DAG, and $d'((u, 0), (v, k))$ is the shortest walk with *exactly* k edges.
- The dashed 0 -length arcs $((u, i), (u, i + 1))$ allow *at most* k edges.



Edge lengths in DAG are same as in original graph except dashed edges which have 0 length

Yet another example of layering

A variant with $G = (V, E)$ having non-negative edge lengths. Some edges are labeled risky. Let $R \subseteq E$ be set of risky edges. Want to find s - t shortest walk with at most k risky edges.

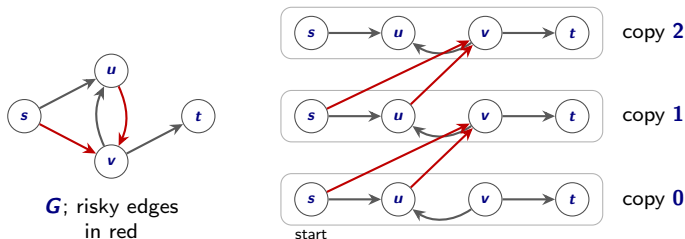
How is this different from what we saw in Bellman-Ford?

How can you solve this problem also via graph layering?

How can you adapt the Bellman-Ford style DP solution for this problem?

Risky edges: Solution (layering)

Keep a counter in the state: how many risky edges used so far.
Normal edges stay in a copy; risky edges go up one copy.



Answer: $\min\{d'((s, 0), (t, 0)), d'((s, 0), (t, 1)), d'((s, 0), (t, 2))\}$ for $k = 2$.

Risky edges: Solution (details)

- $V' = V \times \{0, 1, \dots, k\}$. Normal edge (u, v) : arcs $(u, i) \rightarrow (v, i)$ for all i . Risky edge (u, v) : arcs $(u, i) \rightarrow (v, i + 1)$ for $i < k$. Lengths as in G .
- Walks from s to v in G with exactly i risky edges correspond to walks from $(s, 0)$ to (v, i) in G' .
- Dijkstra from $(s, 0)$; answer $\min_{0 \leq i \leq k} d'((s, 0), (t, i))$.
- $(k + 1)n$ nodes and $(k + 1)m$ arcs: $O(k(m + n \log n))$ time.

Difference from Bellman-Ford: only risky edges move up a layer, so each copy keeps the cycles of G and G' is *not* a DAG.

Bellman-Ford style DP: $d(v, i, j)$ = length of a shortest walk to v with at most i edges, at most j of them risky. This is Bellman-Ford on G' : $O(nkm)$ time, slower, but it allows negative lengths.

Splitting vertices: airport taxes

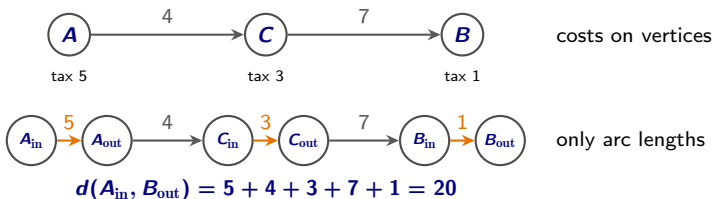
Cities are connected by flights. Flight (u, v) costs $c(u, v) \geq 0$. The airport in city v charges a tax $w(v) \geq 0$ every time you pass through it, including where you start and end.

Find the cheapest trip from city A to city B .

Shortest path algorithms handle lengths on *edges*. How do we handle costs on *vertices*?

Airport taxes: Solution

- Split each city v into v_{in} and v_{out} , with an arc $(v_{\text{in}}, v_{\text{out}})$ of length $w(v)$.
- Flight (u, v) becomes arc $(u_{\text{out}}, v_{\text{in}})$ of length $c(u, v)$.
- Answer: $d(A_{\text{in}}, B_{\text{out}})$ by Dijkstra. $2n$ nodes, $n + m$ arcs.



For shortest paths, adding $w(v)$ to every arc into v also works. Splitting is essential for vertex *capacities*, e.g. in flow problems. A related trick: subdividing each edge of length ℓ into ℓ unit edges turns Dijkstra into BFS.

Part II

Dynamic Programming and Graphs

Connecting DP and Graphs

Two powerful techniques in algorithm design.

- Dynamic Programming
- Graph modeling and graph algorithms

Question: Can we relate them? Can we combine them?

Applying DP to graph problems

To apply DP we need a way to *recursively* decompose a problem. It is natural/easier for problems on structures that already have a simple recursive definition such as on sequences, trees, etc.

Applying DP to graph problems

To apply DP we need a way to *recursively* decompose a problem. It is natural/easier for problems on structures that already have a simple recursive definition such as on sequences, trees, etc.

Graphs do not seem to have such a natural recursive structure.

Applying DP to graph problems

To apply DP we need a way to *recursively* decompose a problem. It is natural/easier for problems on structures that already have a simple recursive definition such as on sequences, trees, etc.

Graphs do not seem to have such a natural recursive structure.

Nevertheless we saw examples such as Bellman-Ford and Floyd-Warshall that introduce additional parameters to define recursive subproblems. Floyd-Warshall, in particular, is quite clever.

Applying DP to graph problems

To apply DP we need a way to *recursively* decompose a problem. It is natural/easier for problems on structures that already have a simple recursive definition such as on sequences, trees, etc.

Graphs do not seem to have such a natural recursive structure.

Nevertheless we saw examples such as Bellman-Ford and Floyd-Warshall that introduce additional parameters to define recursive subproblems. Floyd-Warshall, in particular, is quite clever.

Is there an easy/natural template for applying DP to graphs?

DP on DAGs

Suppose we have a DAG $G = (V, E)$.

G has a topological sort which gives an *ordering* of V as $v_1, v_2, \dots, v_n$. This makes G “look” like a *sequence* and we can try to apply sequence based DP ideas for a problem on DAGs.

One can then leverage the solution on DAGs to general graphs via the meta-graph reduction or via layering or other tricks.

Examples that we already saw in an implicit manner:

- Shortest/longest path problems on DAGs
- Bellman-Ford can be viewed as solving shortest paths in the layered DAG even though original graph is not a DAG.

DP on DAGs: Example

Problem: Given a directed graph $G = (V, E)$ and distinct nodes $s, t \in V$, how many simple s - t paths are there in G ?

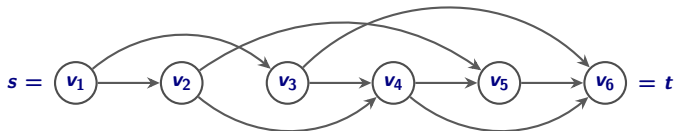
It is a hard problem in digraphs but can be solved in DAGs efficiently. How?

- Let $v_1, v_2, \dots, v_n$ be a topological sort of V and without loss of generality $s = v_1$ and $t = v_n$.
- Let $\text{Count}(i)$ denote the number of simple paths from s to v_i .
- Write a recursive equation for computing $\text{Count}(i)$ from $\text{Count}(1)$ to $\text{Count}(i - 1)$.

Exercise: Suppose we want to count the number of s - t walks in G (not necessarily a DAG) with at most L edges in the walk? Show how you can reduce this to the DAG problem via layering.

Counting paths in a DAG: Solution

- $Count(1) = 1$, and for $i > 1$:
 $Count(i) = \sum_{j: (v_j, v_i) \in E} Count(j)$.
- Every such j has $j < i$ (topological order), so compute $Count(1), Count(2), \dots, Count(n)$ in this order.
- $O(n + m)$ time after a topological sort. Answer: $Count(n)$.



Count: 1 1 1 2 3 6

Each path to v_i arrives by exactly one last arc (v_j, v_i) , so every path is counted exactly once.

DP from the lens of DAGs

DP can be used to solve problems on DAGs and also on graphs.

Are insights from graphs useful or relevant for DP?

DP from the lens of DAGs

DP can be used to solve problems on DAGs and also on graphs.

Are insights from graphs useful or relevant for DP?

Yes!

Recursion and DAGs

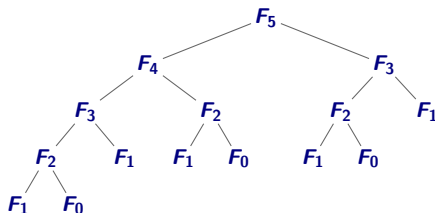
Suppose we have a recursive program $foo(x)$ that takes an input x

- $foo(x)$ generates a recursion *tree* where a subproblem z is a child of subproblem y if $foo(y)$ calls $foo(z)$ during its execution. Note that the same subproblem/instance can occur many times in the tree.
- In DP we are interested in the *distinct* subproblems generated by $foo(x)$. We can create a natural DAG with the recursion
 - Each distinct subproblem corresponds to a node
 - If $foo(y)$ calls $foo(z)$ we add an arc from z to y (y depends on z).

The *dependency graph* for recursion is naturally acyclic

DP and DAGs: Fibonacci

$F(n) = F(n-1) + F(n-2)$. The recursion tree for $F(5)$, next to the DAG of distinct subproblems.



recursion tree



DAG

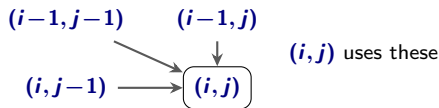
- Tree: $O(\phi^n)$ nodes. DAG: $n + 1$ nodes and $2n - 2$ arcs, so memoization takes $O(n)$ time.
- Each tree node is a path in the DAG that ends at F_5 . The F_1 leaves are the paths from F_1 to F_5 : there are $F_5 = 5$ of them.

DP and DAGs: Edit distance

Recall: $Edit(i, j)$ is the edit distance between $A[1..i]$ and $B[1..j]$.

$$Edit(i, j) = \min \begin{cases} Edit(i, j-1) + 1 & \text{insert} \\ Edit(i-1, j) + 1 & \text{delete} \\ Edit(i-1, j-1) + [A[i] \neq B[j]] & \text{replace} \end{cases}$$

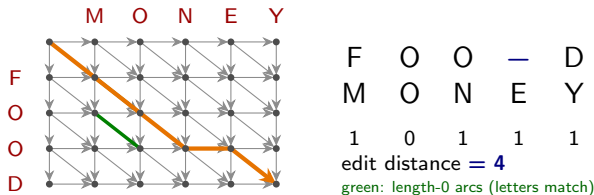
with $Edit(i, 0) = i$ and $Edit(0, j) = j$.



Question: Which *graph problem* is this DP solving?

Edit distance: Solution

- Grid DAG on nodes (i, j) . Arcs into (i, j) : from $(i, j - 1)$ and $(i - 1, j)$ with length 1; from $(i - 1, j - 1)$ with length $[A[i] \neq B[j]]$.
- $Edit(i, j) = d((0, 0), (i, j))$. Answer: shortest path from $(0, 0)$ to (m, n) .
- The path is the alignment: $\rightarrow$ insert, $\downarrow$ delete, $\searrow$ replace or match.



Note: these are exactly the arcs of the dependency DAG.

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

Question: what is the right order? Can we automate it?

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

Question: what is the right order? Can we automate it?

Yes. Compute the DAG. Evaluation order is a topological sort of the DAG

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

Question: what is the right order? Can we automate it?

Yes. Compute the DAG. Evaluation order is a topological sort of the DAG

Question: Why not automate evaluation order?

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

Question: what is the right order? Can we automate it?

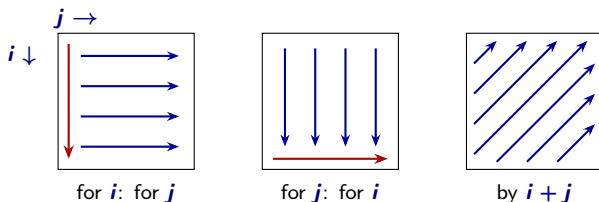
Yes. Compute the DAG. Evaluation order is a topological sort of the DAG

Question: Why not automate evaluation order?

- Creating the DAG explicitly is cumbersome, and wasteful in space/time.
- For many DP problems the subproblem and DAG structure is simple and can be exploited for efficiency

Evaluation order: many valid choices

DP recipe: identify the dependencies, then identify an evaluation order. For edit distance, (i, j) needs $(i - 1, j)$, $(i, j - 1)$ and $(i - 1, j - 1)$ first. All of these orders work:



Each order is a topological order of the dependency DAG. The same holds for any DP.

Avoiding DP and using DAGs directly

In some cases one can reduce a problem to a reachability or shortest/longest path problem on a DAG by *implicitly* doing the DP.

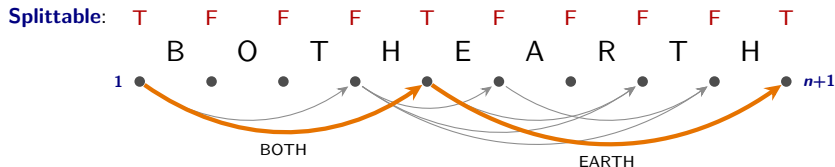
Example: text segmentation. Given a string $A[1..n]$ and a subroutine $\text{IsWord}(i, j)$ that says whether $A[i..j]$ is a word, can A be split into words? $\text{Splittable}(n + 1) = \text{True}$, and for $i \leq n$:

$$\text{Splittable}(i) = \bigvee_{j=i}^n (\text{IsWord}(i, j) \wedge \text{Splittable}(j + 1))$$

Question: which graph problem is this?

Text segmentation: Solution

- DAG on nodes $1, 2, \dots, n+1$: arc $i \rightarrow j+1$ whenever $\text{IsWord}(i, j)$. Arcs only go forward, so it is a DAG.
- **Splittable**(i) is True iff $n+1$ is reachable from i . So **A** can be split iff $n+1$ is reachable from **1**: one WFS.
- Each path from **1** to $n+1$ is a segmentation. Building the DAG takes $O(n^2)$ calls to **IsWord**, as in the DP.



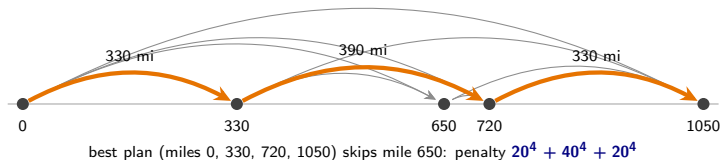
Dictionary: BOT, BOTH, HE, HEAR, HEART, EAR, EARTH, ART.

Another example: road trip

You are planning a long road trip. There are n hotels along your route; you are given a *sorted* array $\text{Dist}[1..n]$, where $\text{Dist}[i]$ is the positive distance in miles from the start of the trip to the i th hotel. These hotels are the only places you are allowed to stop, but you can choose which hotels you stop at. Your final destination is the final hotel (at distance $\text{Dist}[n]$), so you must stop there. Ideally, you would like to travel exactly 350 miles per day, but this may not be possible, because the hotels are not spaced exactly 350 miles apart. If you travel x miles between two hotel stops, you must pay a penalty $(350 - x)^4$ for that day. (Notice that x could be either larger or smaller than 350.) You want to plan your trip so that the sum of all your daily penalties is as small as possible. For psychological reasons, you must drive directly from each hotel to the next, and each hotel must be further along the road than the previous hotel, even if driving backward for a day would decrease your total penalty. Describe and analyze an algorithm that computes the smallest possible total penalty.

Road trip: Solution via a DAG

- Add the start of the trip as v_0 , with $\text{Dist}[0] = 0$.
- For each $i < j$, arc (v_i, v_j) of length $(350 - (\text{Dist}[j] - \text{Dist}[i]))^4$: the penalty for driving from stop i to stop j in one day.
- Answer: shortest path from v_0 to v_n . $v_0, \dots, v_n$ is a topological order and there are $\Theta(n^2)$ arcs: $O(n^2)$ time.
- Same as the DP
$$P(j) = \min_{i < j} \{P(i) + (350 - (\text{Dist}[j] - \text{Dist}[i]))^4\}.$$



Summary

Graph modeling tricks

- States: put just enough history into a node (last edge used, colors seen so far).
- Reverse the graph; add a super source/sink for sets of nodes.
- Meta-graph: solve on DAGs first, then lift via G^{SCC} .
- Layering: copies of G indexed by a counter (hops, risky edges).
- Vertex splitting: move vertex costs onto edges.

DP and DAGs

- Subproblems are nodes, with an arc from z to y if y uses z : a DAG.
- Evaluation order is a topological order.
- Many DPs are reachability, shortest/longest paths or path counts in a DAG: Fibonacci, edit distance, text segmentation, road trip.