

CS2040S

Recitation 9
AY22/23S2

Recitation goals

- Problem casting
- Identify and break down graph problems
- Exploit graph properties and algorithms

 Print Email Link to this page

Search Results My Maps **New!**

4. Take the exit onto I-90 E/Mass Pike/Massachusetts Turnpike toward N.H.-Maine/Boston
Partial toll road 56.0 mi

| | | |
|-----------|--|---------------|
| 5. | Take exit 24 A-B-C on the left toward I-93 N/Concord NH/S Station/I-93 S/Quincy | 0.1 mi |
|-----------|--|---------------|

| | | |
|----|-----------------------------|--------|
| 6. | Take exit 24A for S Station | 486 ft |
|----|-----------------------------|--------|

7. Keep **left** at the fork to go toward **Atlantic Ave** and onto **Atlantic Ave**

8. Turn right at Central St.

| | | |
|-----|--------------------------------|----------|
| 9. | Turn right at Long Wh | |
| 10. | Swim across the Atlantic Ocean | 3,462 mi |

| | | |
|-----|----------------------------|--------|
| 11. | Slight right at E05 | 0.5 mi |
|-----|----------------------------|--------|

| | | |
|-----|---|--------|
| 12. | At the traffic circle, take the 2nd exit onto E05/Pont Vauban | 0.1 mi |
|-----|---|--------|

| | | |
|-----|--------------------------|--------|
| 13. | Turn right at E05 | 5.7 mi |
|-----|--------------------------|--------|

| | | |
|-----|--|---------|
| 14. | Take the exit onto A29/E44 toward Amiens Toll road | 27.8 mi |
|-----|--|---------|

| | | |
|-----|--|--------|
| 15. | Take the exit toward Dieppe/Amiens/Calais/A151/Rouen Toll road | 1.1 mi |
|-----|--|--------|

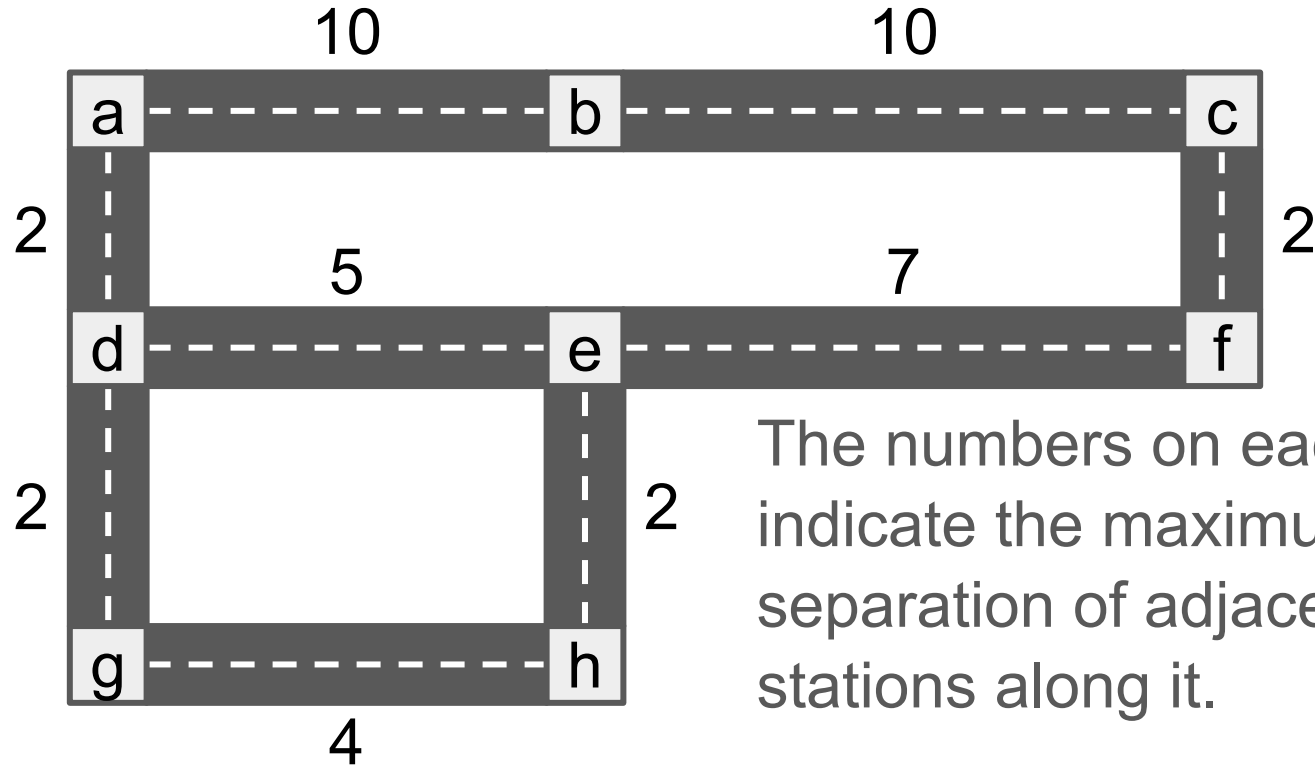
| | | |
|-----|--------------------|---------|
| 16. | Merge onto A29/E44 | 22.6 mi |
|-----|--------------------|---------|



Introduction

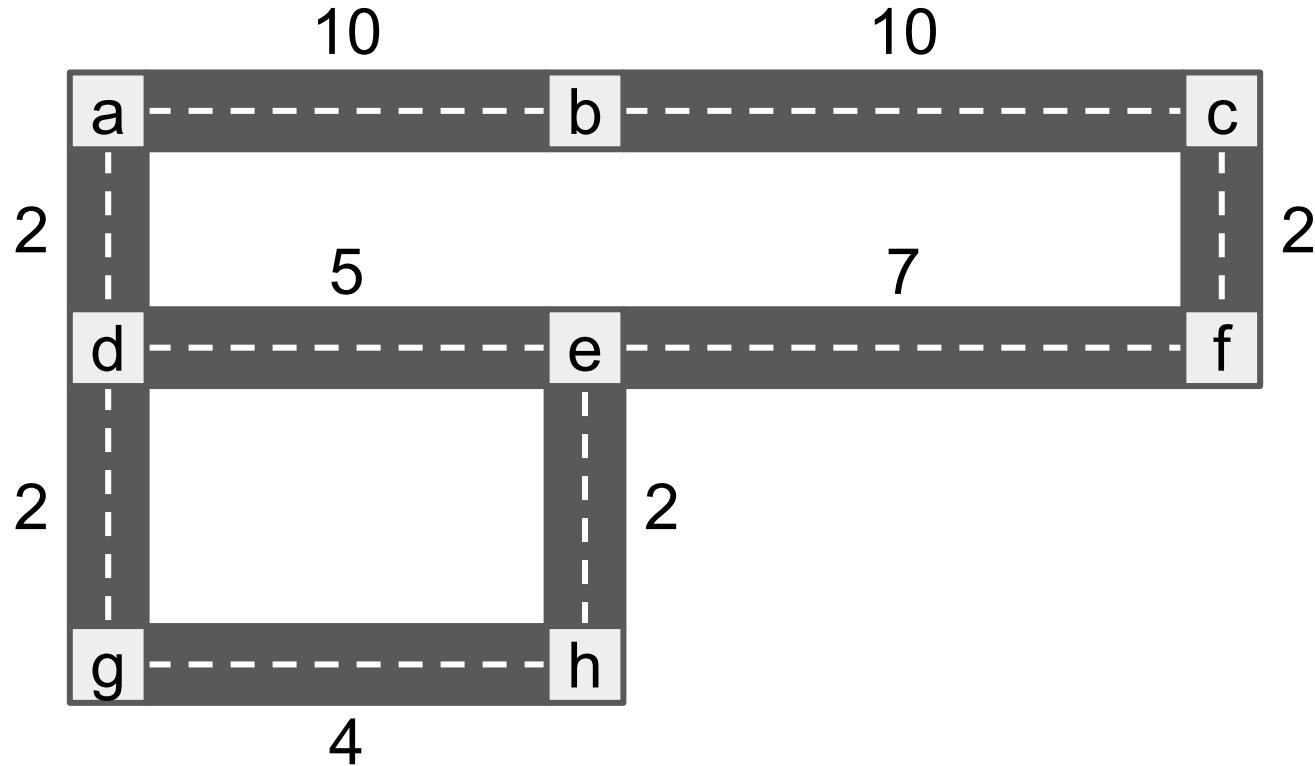
- You are working on a new feature for Google Maps
- Caters for users who wish to have a guarantee over the **maximum distance they will be from the nearest gas station** at any point along their recommended driving route
- Suppose the US mandates adjacent gas stations along the same road to be located apart by a **minimum of 1 mile and a maximum of k miles**
- Suppose further that **distance between gas stations is always an integer number of miles apart**
- You want to determine the *best guarantee* for the furthest gas station from a start to end point

Example: Road network

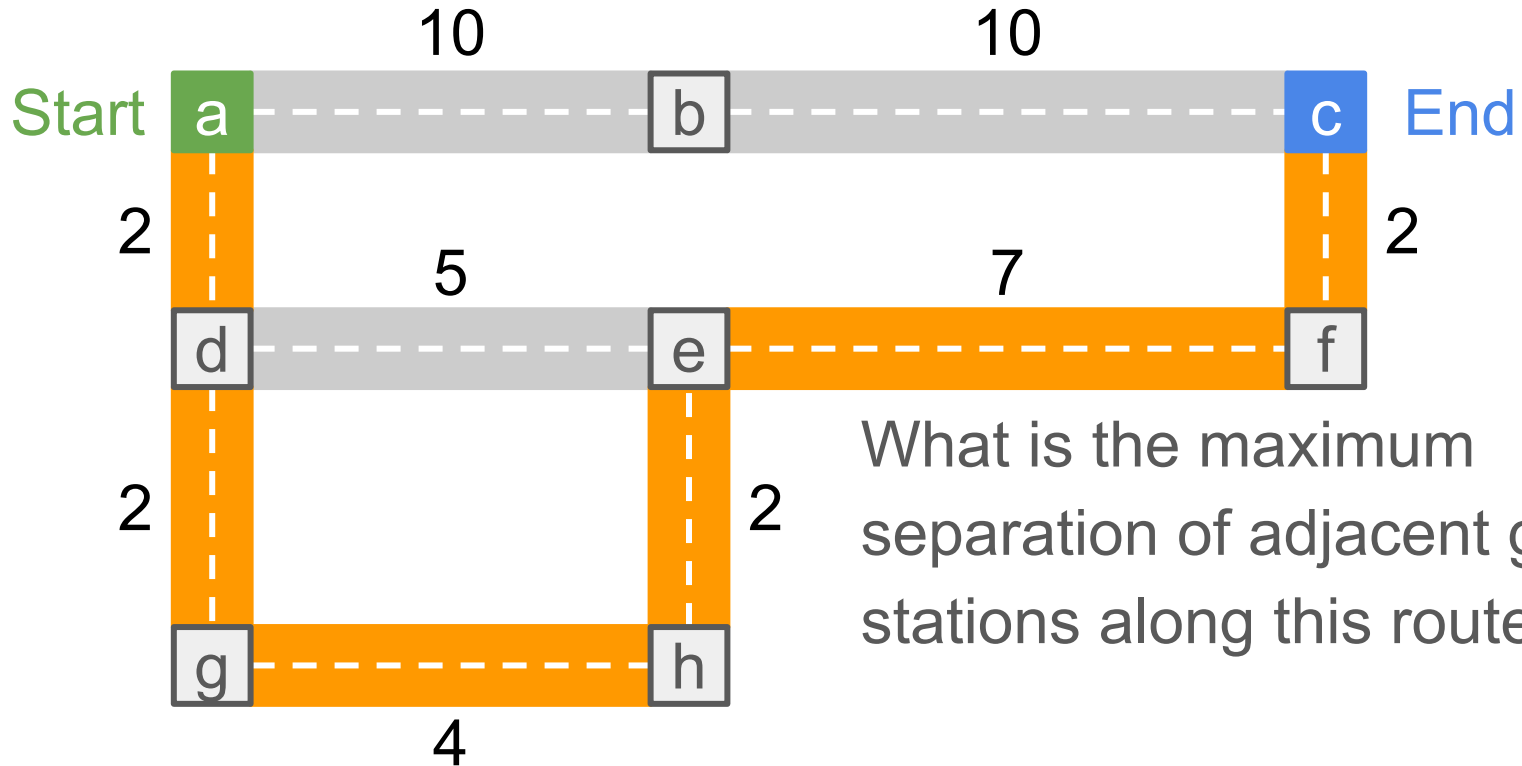


The numbers on each road indicate the maximum separation of adjacent gas stations along it.

Example: Road network

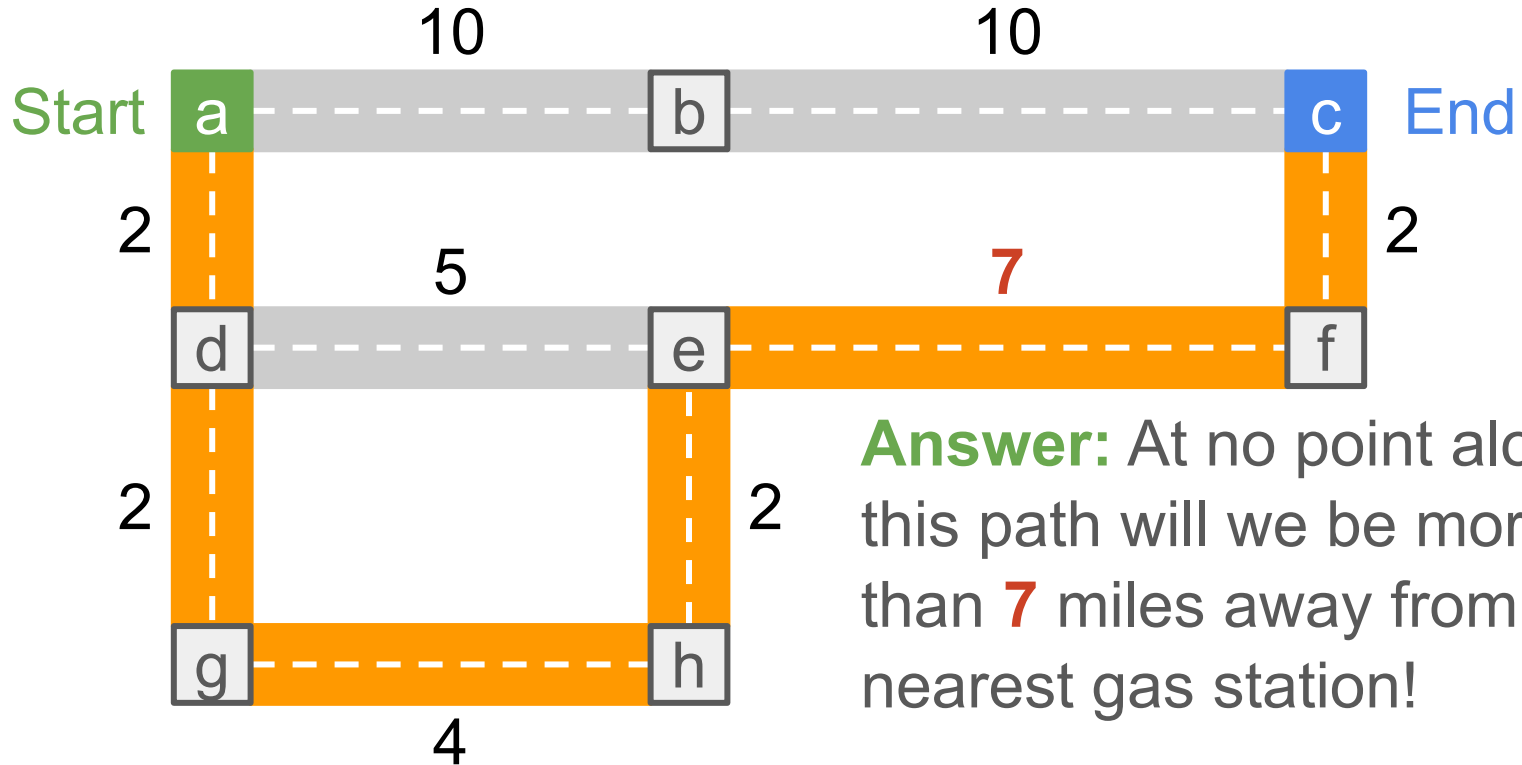


Test yourself!



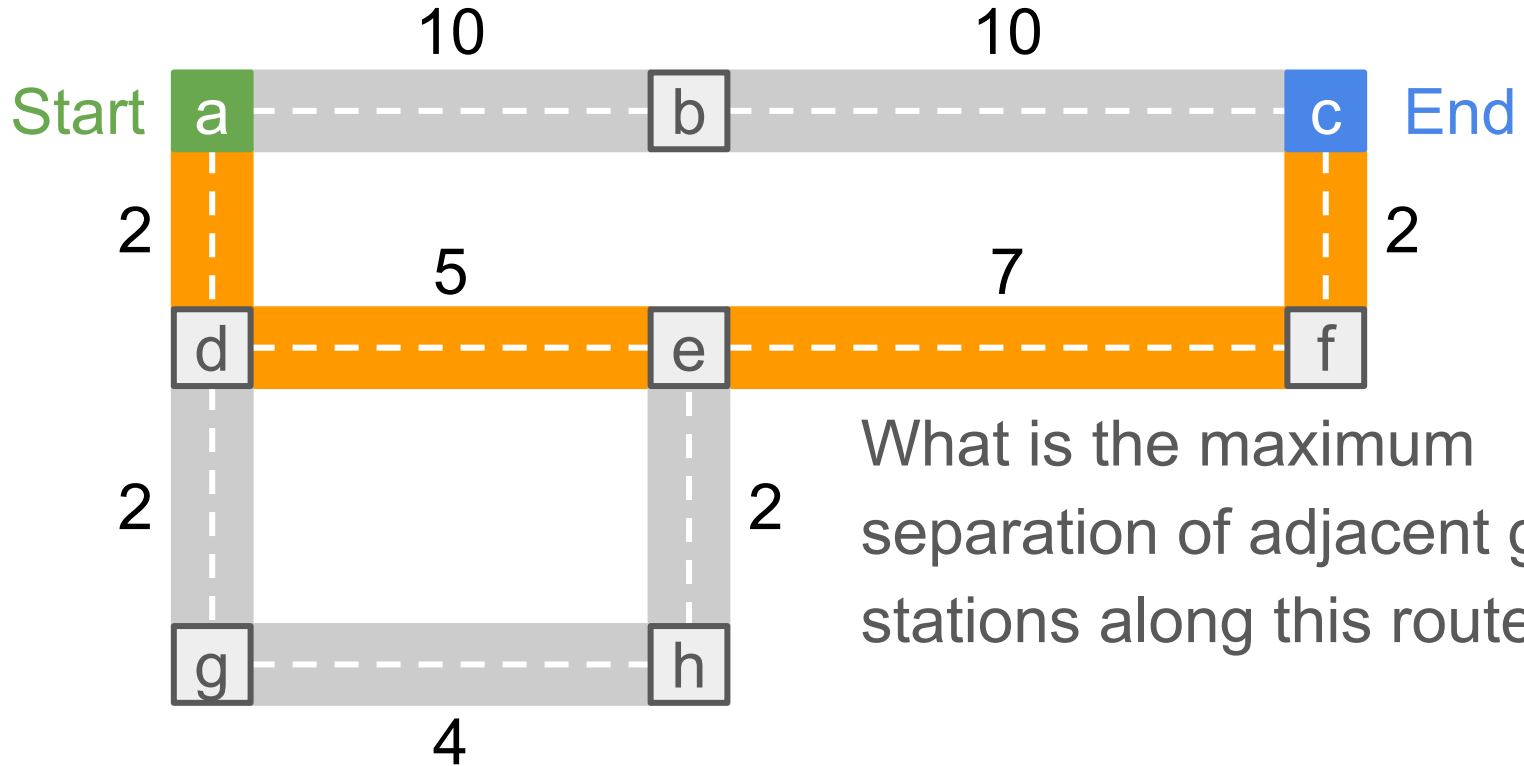
What is the maximum separation of adjacent gas stations along this route?

Test yourself!



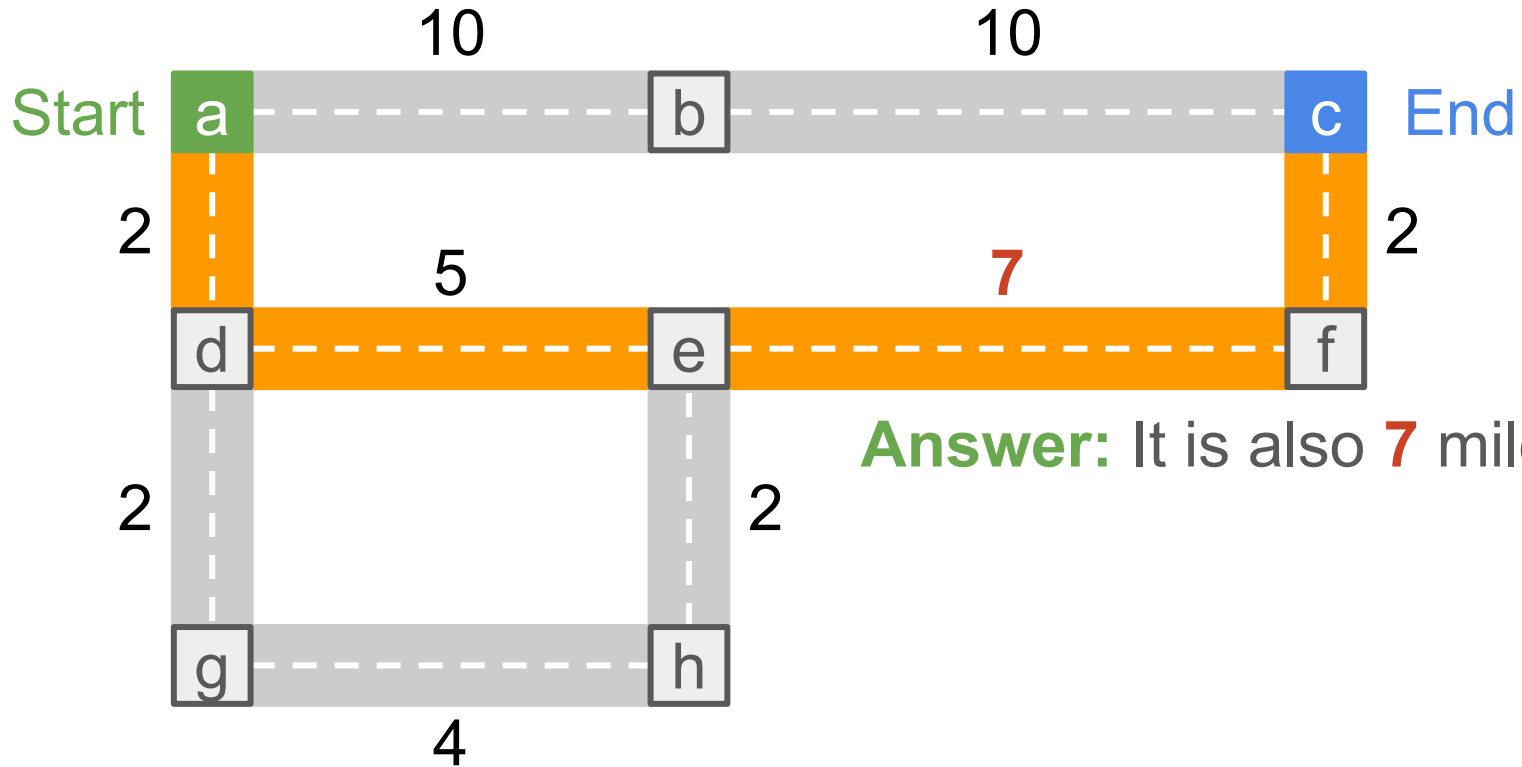
Answer: At no point along this path will we be more than **7** miles away from the nearest gas station!

Test yourself!



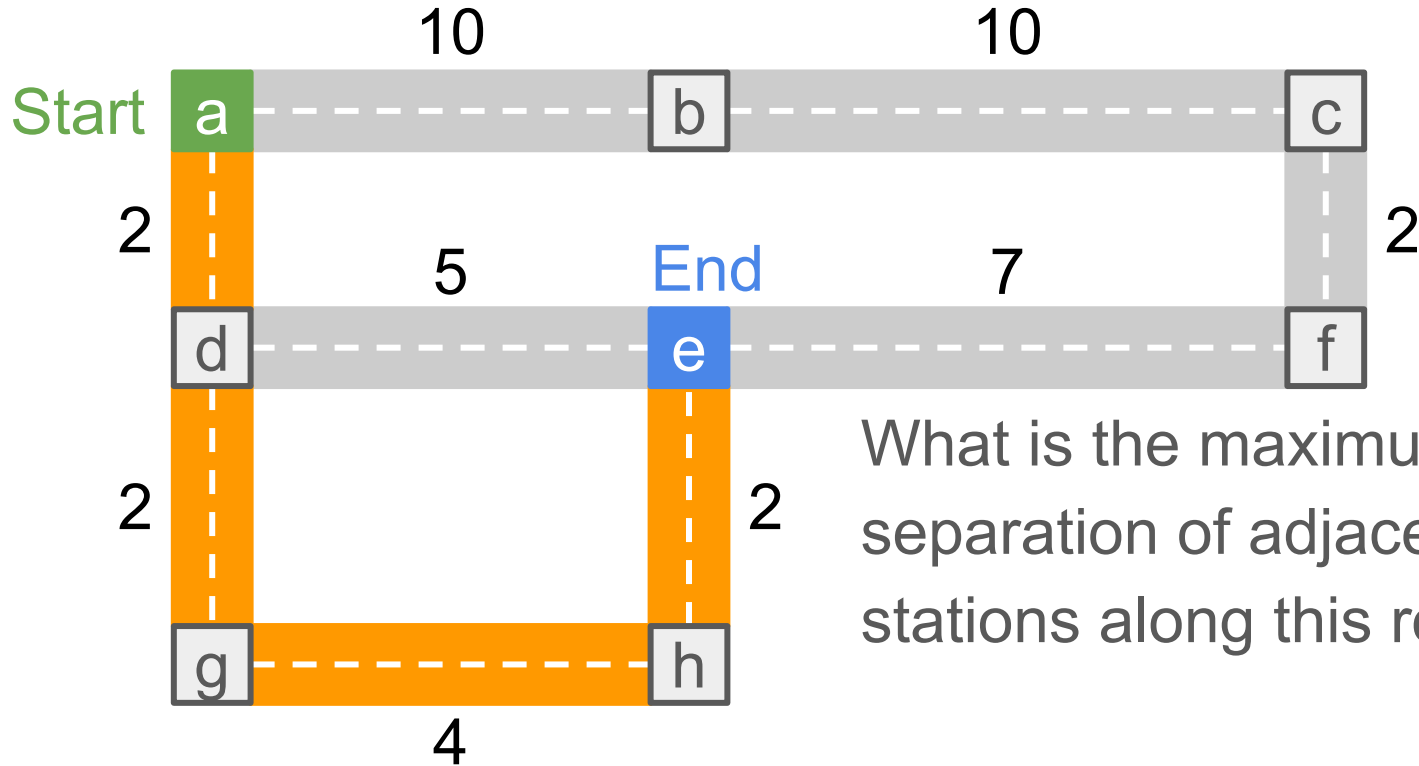
What is the maximum separation of adjacent gas stations along this route?

Test yourself!



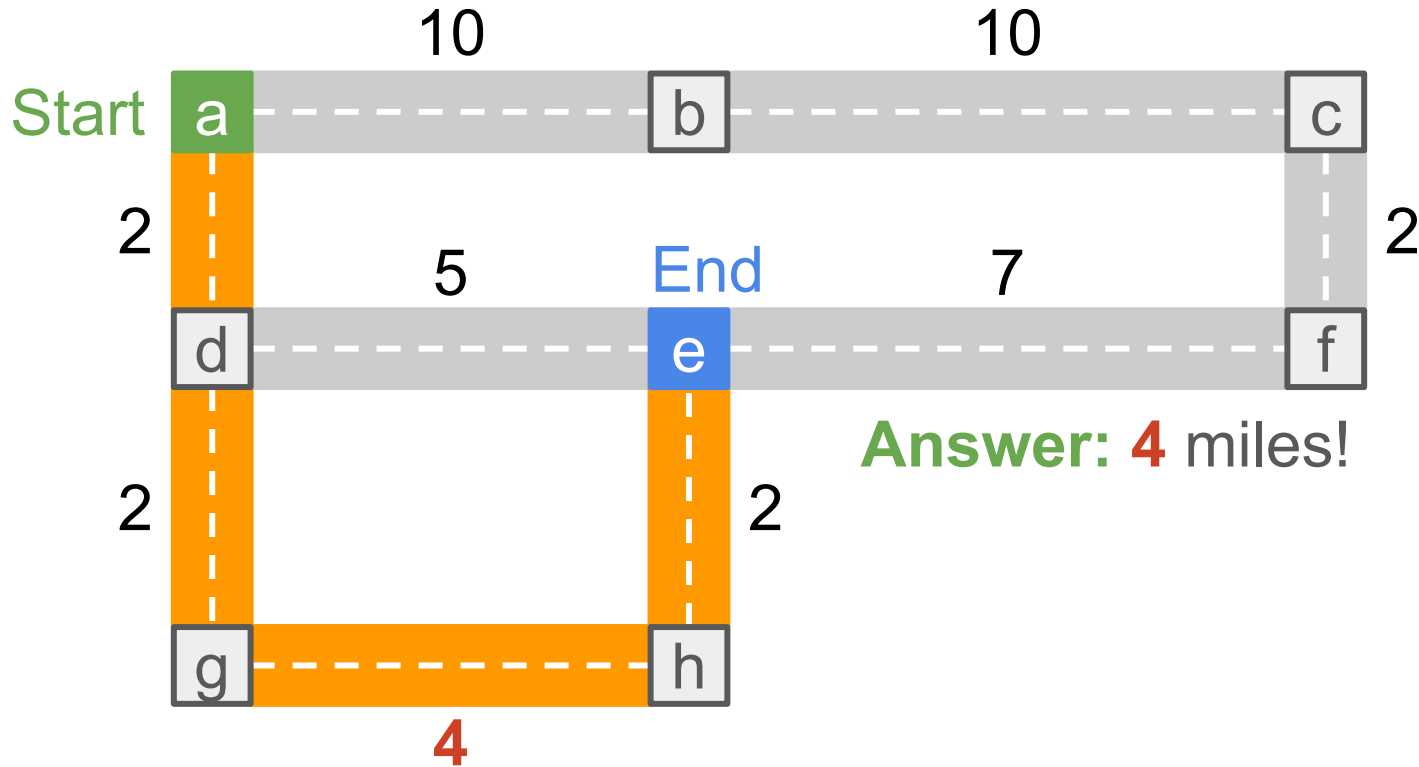
Answer: It is also 7 miles!

Test yourself!

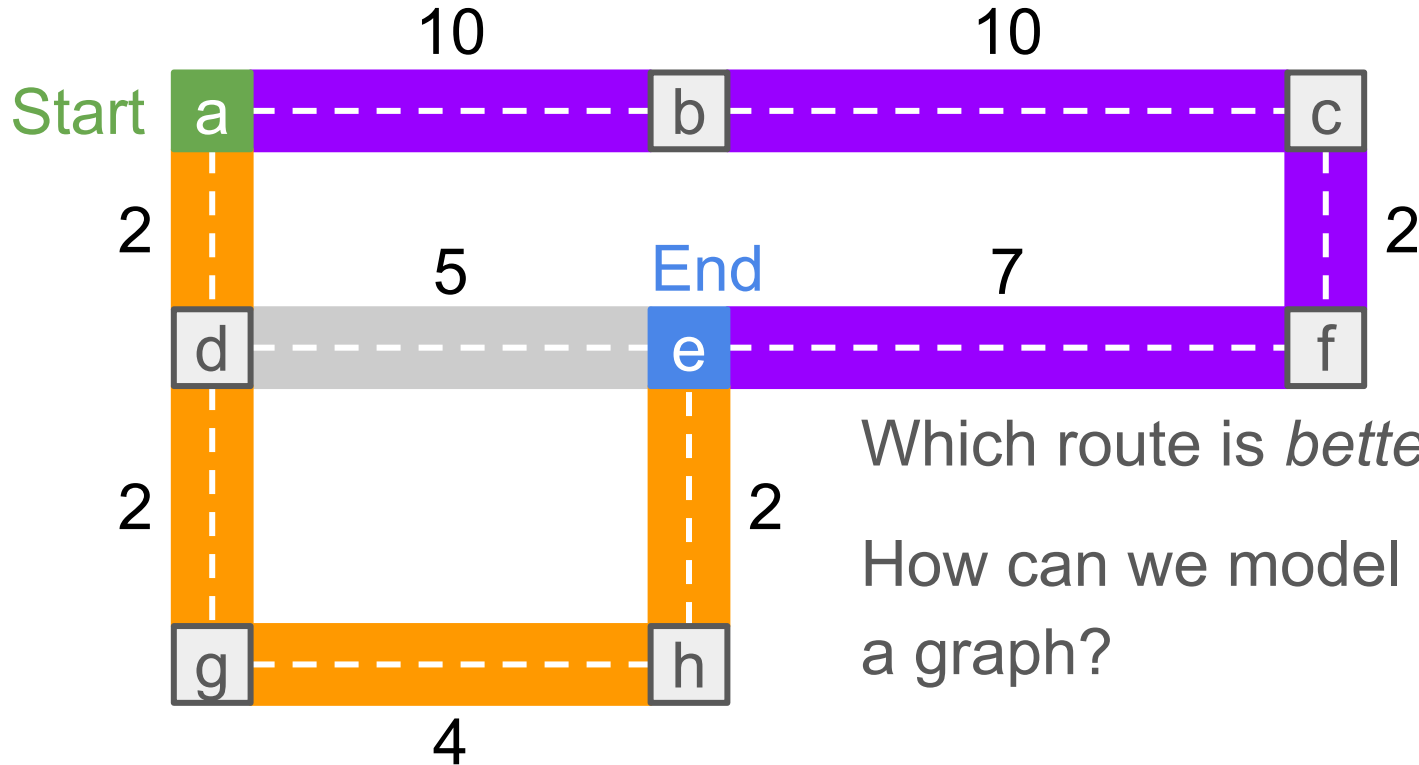


What is the maximum separation of adjacent gas stations along this route?

Test yourself!



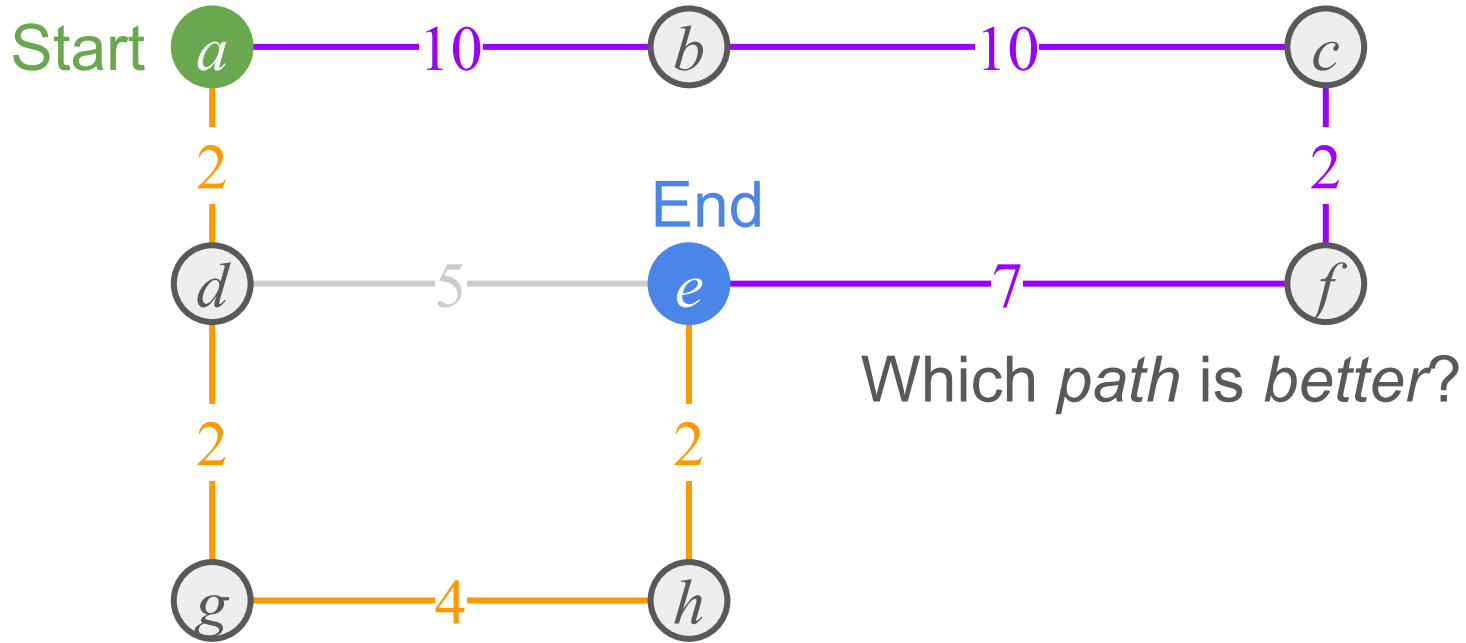
Best route



Which route is *better*?

How can we model this as a graph?

Best route: Graph modelling



Working vocabulary

For the sake of clarity in this problem, let us use *width* to refer to the **heaviest weight** along a path.

You can think of it this way: the length of a path (summation of edge weights) measures it along one dimension, the width of a path (maximum of edge weights) measures it along the other (perpendicular) dimension.

Note: We are actually hijacking the definition of [pathwidth](#) in graph theory for our convenience here.

Best path

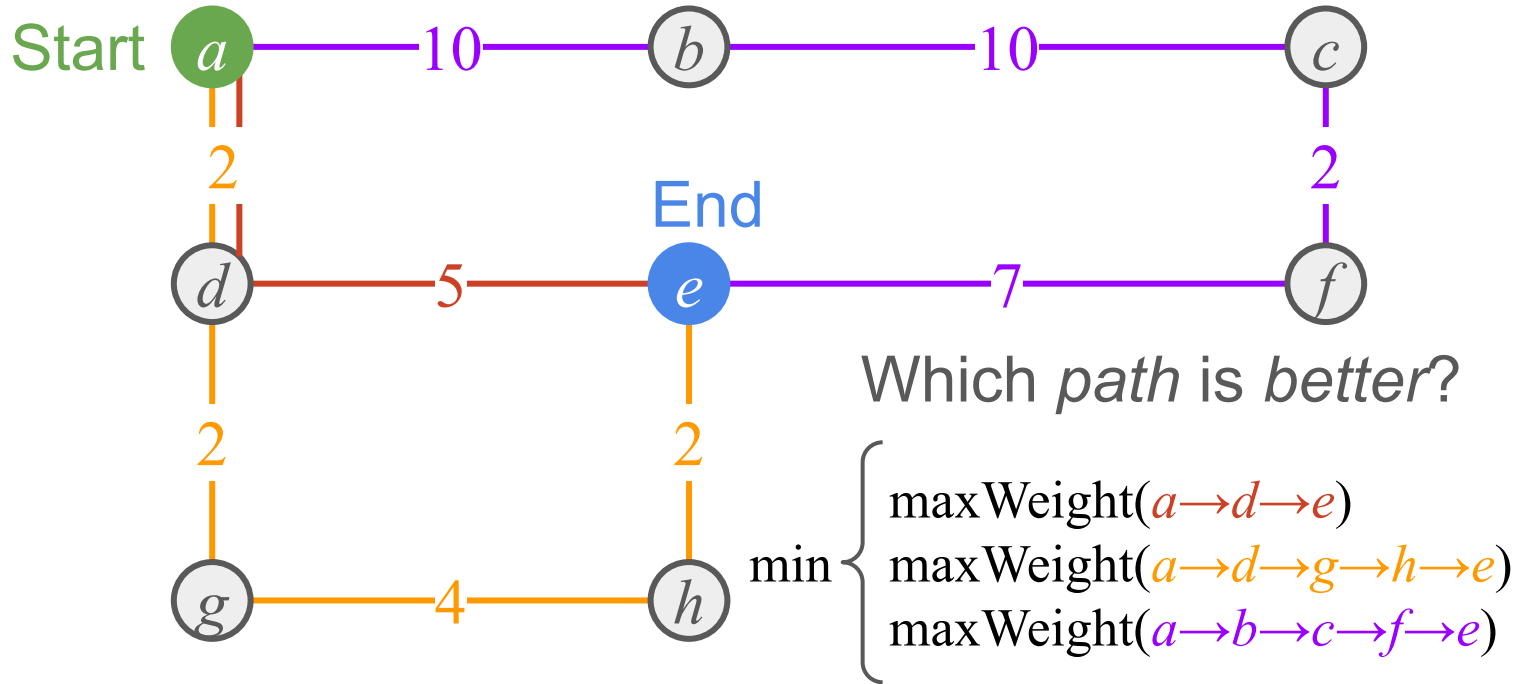
Definition:

Minimax distance: The *minimum possible* width of a path connecting two vertices.

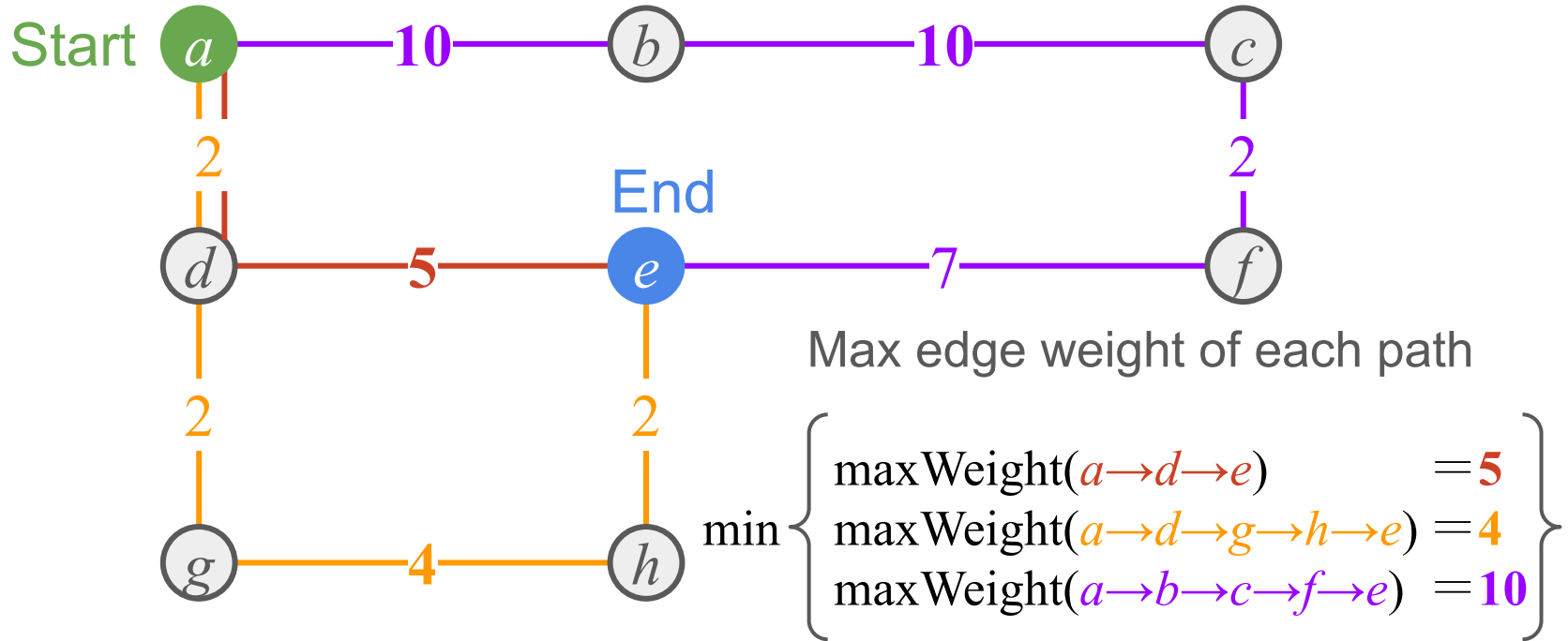
Best path: The path whose width is the minimax distance. I.e. the minimax path itself.

Note: If you found the minimax definition in words to be confusing, just refer to previous illustrated examples.

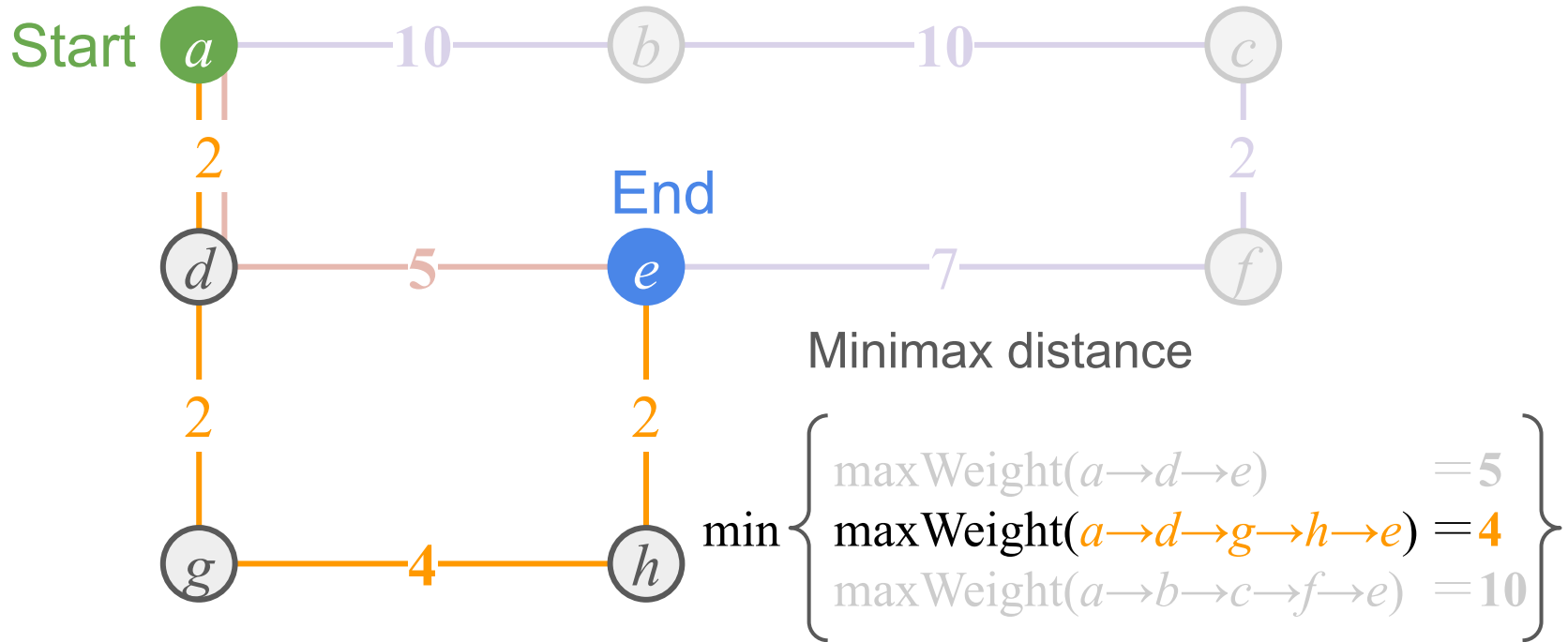
Example: Best path



Example: Best path



Example: Best path



Problem input

Your input to the problem is a road map (like the example)

- Each city is identified by an *integer*
- The map is provided as an *adjacency list*:
 - Each entry in the AL (i.e. a city): A list of roads running out of the city
 - Each road entry: A pair (the city to which the road connect to, the maximum distance between adjacent gas stations along that road)
- Roads are two-way

Problem requirement

Solution comprises 2 stages:

Preprocessing stage: Initialize the service by *preprocessing* the given road map and storing it in a data structure (DS) for querying.

Query stage: Service user requests by *querying* the DS for the maximum distance between adjacent gas stations along the *best* routes recommended to them (i.e the minimax distances).

Problem requirement—Preprocessing stage

```
void preprocessMap(Vector<Vector<ii>> map)
```

Reads in the map and initialize the chosen DS.

Problem requirement—Query stage

```
int queryMinimax(int start, int destination)
```

Return the minimax *distance* corresponding to the best route for the given start-to-destination request.

Note that your algorithm *does not* have to actually track and return the best route itself!

Note

There are many possible solutions to this problem. Different solutions have different trade-offs:

- Some solutions may have a very fast preprocessing step but is very slow at answering queries
- Other solutions may have a very slow preprocessing step but is very fast at answering queries

We would prefer queries to be relatively fast. However we also want to minimize the time spent on preprocessing the map which may be very big.

Problem

Come up with at least two solutions and present them in pseudocode.

Explain why your solutions work and provide their time and space complexities.

What are the tradeoffs between your solutions?

Notational convenience

For convenience sake, I shall use the following notations:

S : the start vertex number

D : Destination vertex number

V : Number of vertices

E : Number of edges

Complexity analysis

Note that there are different aspects of complexities.

Time:

- Due to preprocess
- Due to each query

Space:

- Due to preprocess
- Due to DS (persistent after preprocess)
- Due to each query



Shortest Paths

Solution 1

Test yourself!

How can we cast this problem as a shortest path problem?

Test yourself!

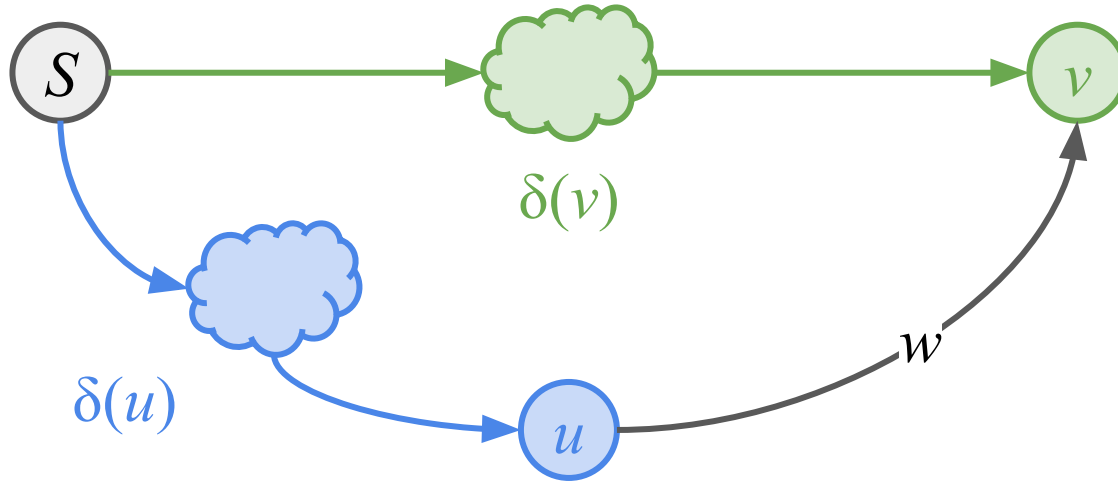
How can we cast this problem as a shortest path problem?

Answer:

We just need to define our “path length” to be the *maximum weight* of an edge along the path.

Then our shortest path naturally turns into the minimax path!

New triangle inequality



$$\delta(v) \leq \max(\delta(u), w)$$

New relax criteria

ModifiedRelax(u, v, w)

```
check  $\leftarrow \max(\text{est}[u], w)$   
if (check < est[v]) then  
    est[v]  $\leftarrow$  check  
end
```

We can treat the weight of an edge as the degree of *badness* for including it in a path.

When edge (u, v) is relaxed, we are comparing the previous best path to v to a path that crosses edge (u, v) .

If the badness for the new path is lower than the badness of the old path, then we update our estimate of v .

Test yourself!

Which SSSP algorithm should we use here?

Test yourself!

Which SSSP algorithm should we use here?

Answer: Since extending a path with an edge in this graph **can only increase** its maximum weight, we can safely run $O((V+E) \log V)$ *Dijkstra's* with the modifiedRelax subroutine.

Test yourself!

What is the space complexity of a query?

Test yourself!

What is the space complexity of a query?

Answer: Since we are running a Dijkstra's on-the-fly for every query that comes in, each query entails a distance estimates table of size V as well as a priority queue containing V vertices. Therefore total auxiliary space incurred is $O(V)$.

Outline: Solution 1

Modified relax SSSP with Dijkstra's

Preprocessing stage

- None

Query stage (given S and D)

- Run SSSP on source vertex S using Dijkstra's with modified relax operation based on choosing lower path width
- Minimax distance to D is $\delta(D)$

Analysis: Solution 1

Modified relax SSSP with Dijkstra's

| | | |
|-------------------------|------------|--|
| Time complexity | Preprocess | |
| | Query | |
| Space complexity | Preprocess | |
| | Query | |
| | DS | |

Analysis: Solution 1

Modified relax SSSP with Dijkstra's

| | | |
|-------------------------|------------|-------------------|
| Time complexity | Preprocess | 0 |
| | Query | $O((V+E) \log V)$ |
| Space complexity | Preprocess | 0 |
| | Query | $O(V)$ |
| | DS | N/A |

Solution 2

Test yourself!

How can you improve on the SSSP solution to achieve faster query time?

Test yourself!

How can you improve on the SSSP solution to achieve faster query time?

Answer: We can preprocess the shortest paths and cache their results. Realize that since SSSP pertain to specific source vertices and we don't know the query vertices ahead of time, we will have to compute APSP to obtain shortest distances for *all possible source vertices*! This would support $O(1)$ query since we just have to look up the precomputed table.

Test yourself!

What is the space and time complexity of computing APSP by running $O((V+E) \log V)$ Dijkstra's on all V vertices?

Test yourself!

What is the space and time complexity of computing APSP by running $O((V+E) \log V)$ Dijkstra's on all V vertices?

Answer: Doing that would incur $O(V(V+E) \log V)$ preprocessing time complexity and storing shortest distance tables for every of the V vertices would incur $O(V^2)$ space complexity.

Outline: Solution 2

Modified relax APSP with Dijkstra's

Preprocessing stage

- Exhaustively run Dijkstra's with modified relax on all vertices u in the graph as source vertices to obtain shortest distance tables δ_u for all of them

Query stage (given S and D)

- Lookup $\delta_S(D)$

Analysis: Solution 2

Modified relax APSP with Dijkstra's

| | | |
|-------------------------|------------|--|
| Time complexity | Preprocess | |
| | Query | |
| Space complexity | Preprocess | |
| | Query | |
| | DS | |

Analysis: Solution 2

Modified relax APSP with Dijkstra's

| | | |
|-------------------------|------------|--------------------|
| Time complexity | Preprocess | $O(V(V+E) \log V)$ |
| | Query | $O(1)$ |
| Space complexity | Preprocess | $O(V^2)$ |
| | Query | $O(1)$ |
| | DS | $O(V^2)$ |

Solution 3

Test yourself!

Is running $O((V+E) \log V)$ Dijkstra's on all V vertices the only way to compute APSP?

Test yourself!

Is running $O((V+E) \log V)$ Dijkstra's on all V vertices the only way to compute APSP?

Answer: No! There is Floyd-Warshall! We can pre-compute all shortest paths using $O(V^3)$ Floyd-Warshall which also takes $O(V^2)$ space.

Outline: Solution 3

Modified relax APSP with Floyd-Warshall's

Preprocessing stage

- Compute APSP by running Floyd-Warshall's to obtain shortest distance tables δ_u for all vertices u in the graph

Query stage (given S and D)

- Lookup $\delta_S(D)$

Analysis: Solution 3

Modified relax APSP with Floyd-Warshall's

| | | |
|-------------------------|------------|--|
| Time complexity | Preprocess | |
| | Query | |
| Space complexity | Preprocess | |
| | Query | |
| | DS | |

Analysis: Solution 3

Modified relax APSP with Floyd-Warshall's

| | | |
|-------------------------|------------|----------|
| Time complexity | Preprocess | $O(V^3)$ |
| | Query | $O(1)$ |
| Space complexity | Preprocess | $O(V^2)$ |
| | Query | $O(1)$ |
| | DS | $O(V^2)$ |

Test yourself!

So should we use APSP using exhaustive Dijkstra's or Floyd-Warshall?

Test yourself!

So should we use APSP using exhaustive Dijkstra's or Floyd-Warshall?

Answer: It depends! Since they both incur the same space complexity, it all boils down to the difference in their time complexities: $O(V(V+E) \log V)$ vs $O(V^3)$. If our graph is mostly tree-like (i.e. $E=O(V)$), then exhaustive Dijkstra takes $O(V^2 \log V)$ time, which is better than Floyd-Warshall's $O(V^3)$. However, if we are dealing with a dense/complete graph (i.e. $E=O(V^2)$), then exhaustive Dijkstra takes $O(V^3 \log V)$ time, which is inferior to Floyd-Warshall's $O(V^3)$.

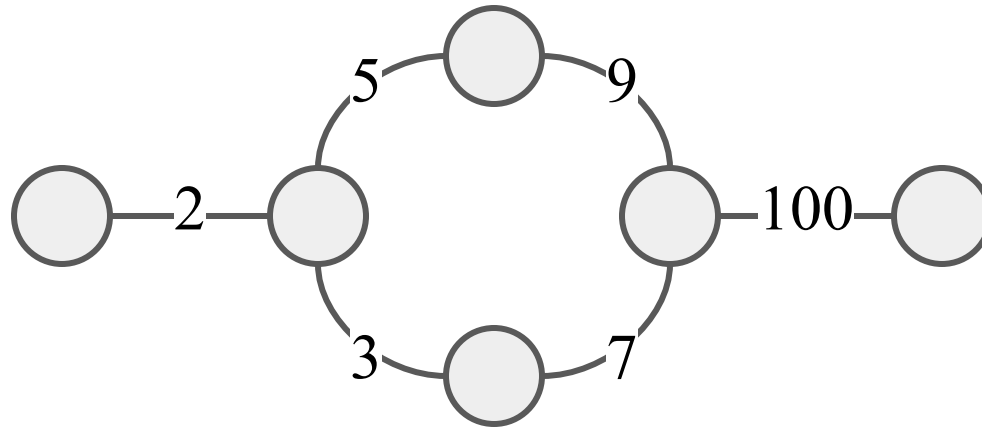
The background of the slide features a complex network graph. It consists of numerous small, light-gray circular nodes connected by thin, dark-gray lines representing edges. The connections are somewhat random, creating a web-like structure that fills the entire frame. The density of the connections is higher in some areas and lower in others, with some nodes having multiple neighbors and others being isolated or part of small clusters.

Spanning Trees

Solution 4

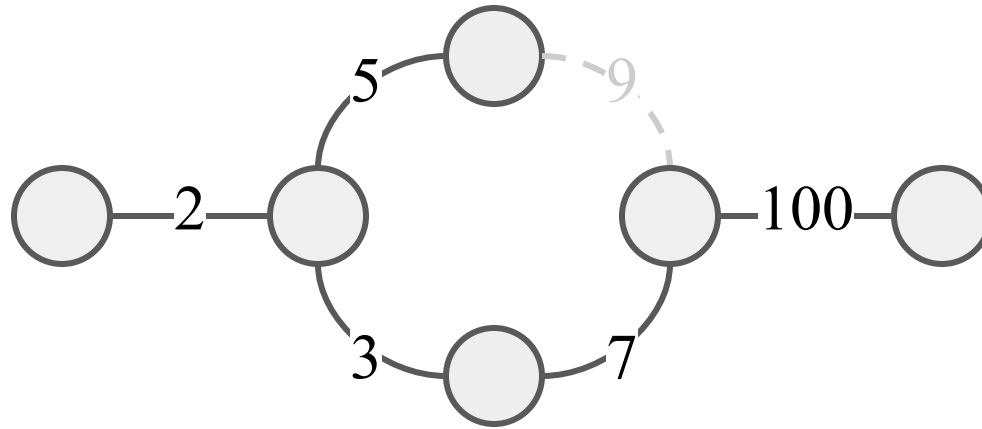
Test yourself!

How do you obtain the MST of the following graph?



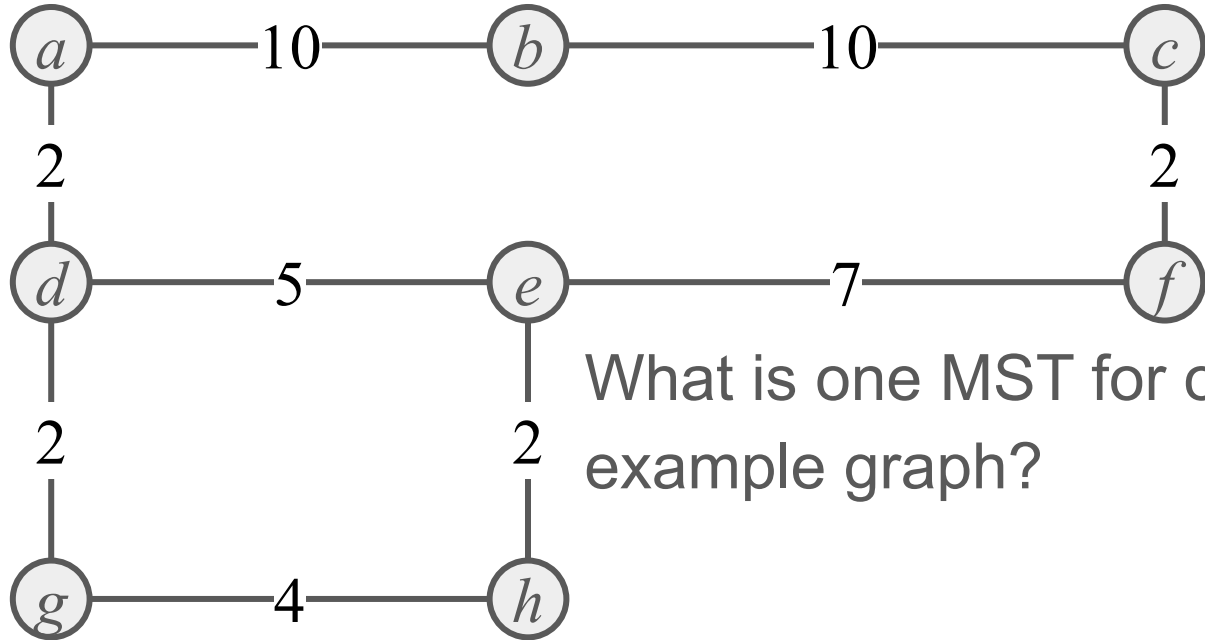
Test yourself!

How do you obtain the MST of the following graph?



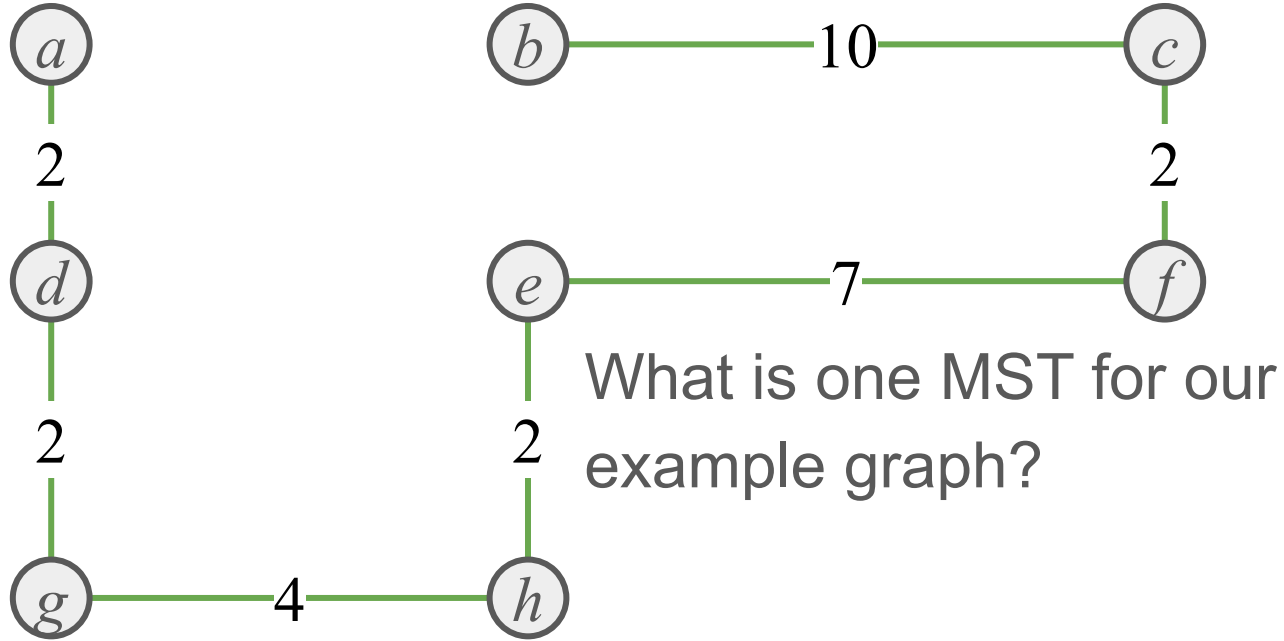
Answer: Remove the *maximum weighted edge 9* in the cycle.

Test yourself!



What is one MST for our example graph?

Test yourself!



Lemma 1

Let T be the minimum spanning tree (MST) of the graph G .

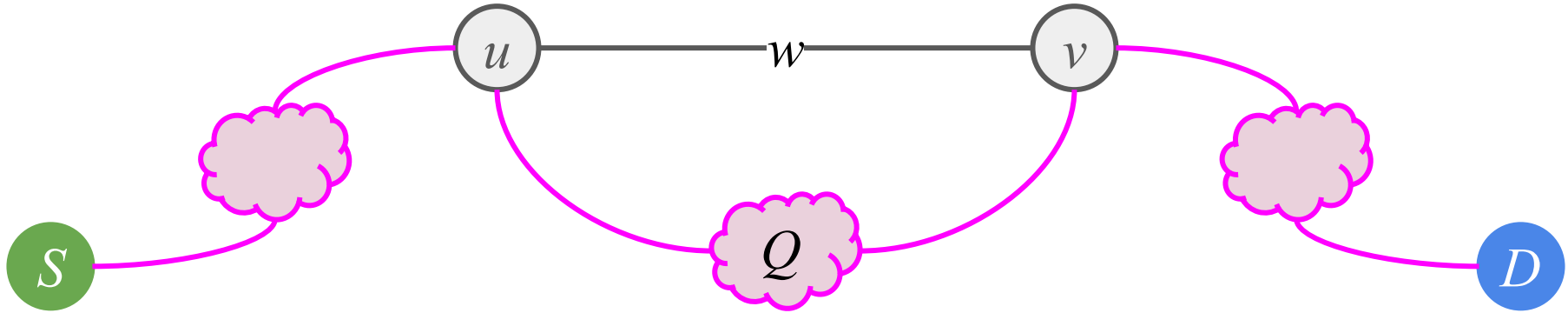
Let $P(S, D)$ be the minimax path from S to D .

Then *every edge* in $P(S, D)$ must be in T . (Assume all edge weights are unique)

Proof by contradiction

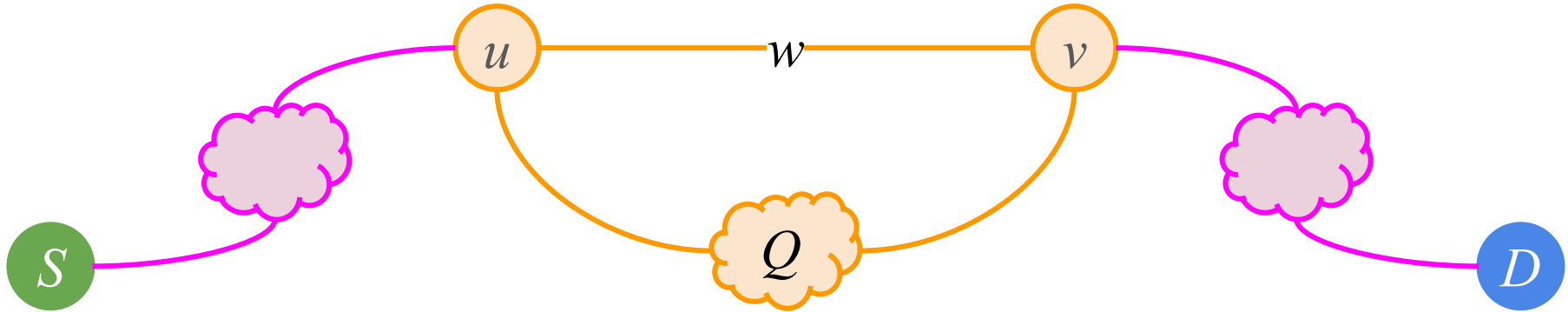
Suppose there is an edge (u, v) that is in $P(S, D)$ but not in T .

Let Q be the path in T from u to v .



Proof by contradiction

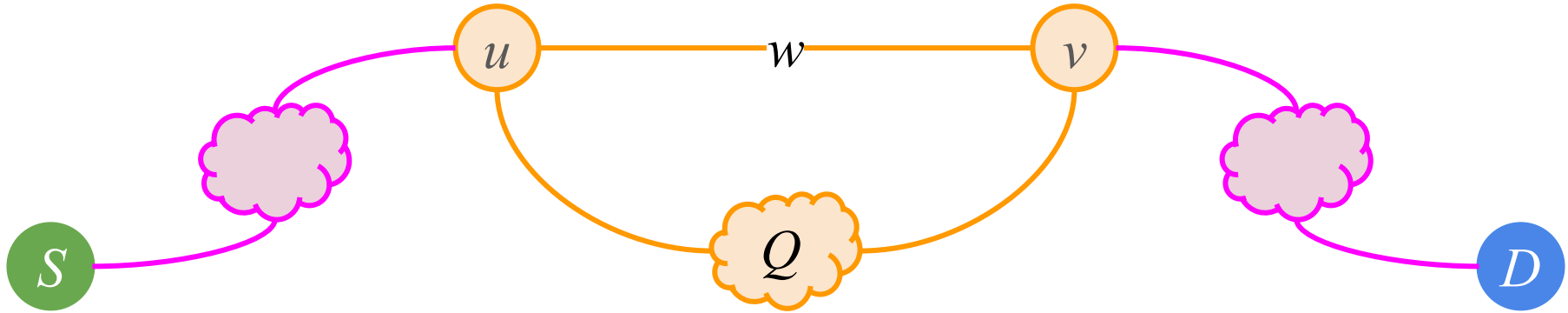
Then (u, v) with Q forms a cycle.



Proof by contradiction

By the cycle property of an MST, we know that the maximum weighted edge in a cycle is not part of the MST.

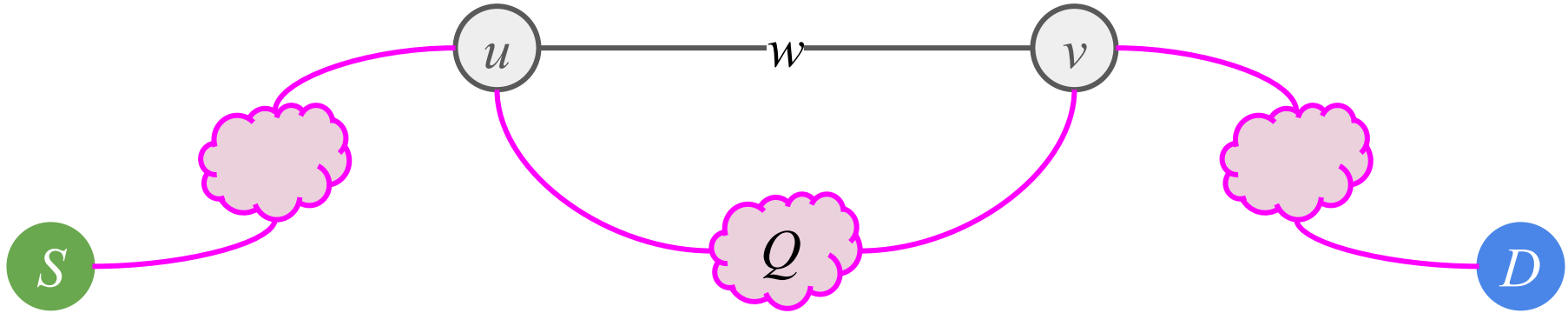
Therefore edge (u, v) must be heavier than *every edge* in Q .



Proof by contradiction

Hence if we replace edge (u, v) in $P(S, D)$ with Q , we obtain a path with a lower maximum edge weight than $P(S, D)$.

This is a contradiction!



Remark

Note that this is a slightly simpler version of the proof where we assumed that there is only one unique MST in the graph (i.e. all edge weights are unique).

This simplifies the discussion and should suffice to convince you that the claim is true.

Test yourself!

What's the time and space complexity of Prim's/Kruskal's?

Test yourself!

What's the time and space complexity of Prim's/Kruskal's?

Answer:

Time (both): $O(E \log V)$

Space:

- Prim's (with binary heap): $O(V)$
- Kruskal's (union find with path compression): $O(E)$

Test yourself!

What's the time complexity of BFS/DFS on a tree with V vertices?

Test yourself!

What's the time complexity of BFS/DFS on a tree with V vertices?

Answer:

$$O(V + E)$$

$$= O(V + V - 1)$$

$$= O(V)$$

Outline: Solution 4

Query-time MST

Preprocessing stage

- None

Query stage (given S and D)

- Obtain MST using Prim's or Kruskal's
- Run BFS/DFS on MST to obtain minimax distance from S to D by keeping track of the maximum weighted edge along the way

Analysis: Solution 4

Query-time MST

| | | |
|-------------------------|------------|--|
| Time complexity | Preprocess | |
| | Query | |
| Space complexity | Preprocess | |
| | Query | |
| | DS | |

Analysis: Solution 4

Query-time MST

| | | |
|-------------------------|------------|---------------|
| Time complexity | Preprocess | 0 |
| | Query | $O(E \log V)$ |
| Space complexity | Preprocess | 0 |
| | Query | $O(V)$ |
| | DS | N/A |

Solution 5

Test yourself!

How can you improve on the MST solution to achieve faster query time?

Test yourself!

How can you improve on the MST solution to achieve faster query time?

Answer: Just preprocess the MST and run the BFS/DFS at query time.

Outline: Solution 5

Preprocessed MST

Preprocessing stage

- Obtain MST using Prim's/Kruskal's

Query stage (given S and D)

- Run BFS/DFS on MST to obtain path width from S to D , keeping track of the maximum weighted edge along the way

Analysis: Solution 5

Preprocessed MST

| | | |
|-------------------------|------------|--|
| Time complexity | Preprocess | |
| | Query | |
| Space complexity | Preprocess | |
| | Query | |
| | DS | |

Analysis: Solution 5

Preprocessed MST

| | | |
|-------------------------|------------|---------------|
| Time complexity | Preprocess | $O(E \log V)$ |
| | Query | $O(V)$ |
| Space complexity | Preprocess | $O(V)$ |
| | Query | $O(V)$ |
| | DS | $O(V)$ |



When you have a hammer,
Everything looks like a nail.

Not every path optimization problem has to be solved by shortest paths or minimum spanning trees.

Search

Solution 6

Test yourself!

How can we cast this as a search problem?

What is our search space?

Test yourself!

How can we cast this as a search problem?

What is our search space?

Answer: Search for the *lowest* $w \in [1, k]$ such that D is reachable from S in the *trimmed* graph where all edges with weight $> w$ are *removed*.

Test yourself!

What is the property of the search procedure and how might we exploit it?

Test yourself!

What is the property of the search procedure and how might we exploit it?

Answer: We can linear search from 1 up to k and determine *the first* trimmed graph with a path from S to D .

Since linear search is monotonic, we can exploit it using binary search!

Algorithm: Query

Query(S, D)

$lo \leftarrow 1$

$hi \leftarrow k$

while $lo < hi$ **do**

$w \leftarrow \lfloor (lo + hi) / 2 \rfloor$

if `reachableInTrimmedGraph`(S, D, w) **then**

$hi \leftarrow w$ // Search LHS

else

$lo \leftarrow w + 1$ // Search RHS

end

end

return hi

Test yourself!

When implementing `reachableInTrimmedGraph(S, D, w)`, do we have to literally construct a new graph with edges of weight greater than w removed?

Test yourself!

When implementing `reachableInTrimmedGraph(S, D, w)`, do we have to literally construct a new graph with edges of weight greater than w removed?

Answer: No, we just need to run $O(V+E)$ *modified* DFS/BFS from S which ignores all edges with weight $> w$. We return `true` once we encounter D , else at the end of DFS/BFS, we return `false`. Therefore we are testing reachability on an *implicitly* trimmed graph.

Test yourself!

What is the time and space complexities for a query?

Test yourself!

What is the time and space complexities for a query?

Answer: Since we there are $O(\log k)$ steps in the binary search routine, and each step entails $O(V+E)$, total time complexity is $O((V+E) \log k)$.

Space complexity of query is $O(V)$ since DFS/BFS requires recursion-stack/queue and a visitation table.

Outline: Solution 6

Binary search on query-time trimmed graph

Preprocessing stage

- None

Query stage (given S and D)

- Binary search $i \in [1, k]$ where each step use DFS/BFS to test for reachability from S to D in the graph implicitly trimmed of weights $> i$

Analysis: Solution 6

Binary search on query-time trimmed graph

| | | |
|-------------------------|------------|--|
| Time complexity | Preprocess | |
| | Query | |
| Space complexity | Preprocess | |
| | Query | |
| | DS | |

Analysis: Solution 6

Binary search on query-time trimmed graph

| | | |
|-------------------------|------------|-------------------|
| Time complexity | Preprocess | 0 |
| | Query | $O((V+E) \log k)$ |
| Space complexity | Preprocess | 0 |
| | Query | $O(V)$ |
| | DS | N/A |

Solution 7

Test yourself!

Pertaining to our previous solution, how can we tweak it slightly to optimize for queries? Which property of the graph allows us to do this?

Test yourself!

Pertaining to our previous solution, how can we tweak it slightly to optimize for queries? Which property of the graph allows us to do this?

Answer: Preprocess all the trimmed graphs!

One trimmed graph for each $i \in [1, k]$.

We can do this because edge weights are *integers*.

Test yourself!

How can we check if any two vertices in an undirected graph are mutually reachable by some path without having to run a full DFS/BFS each time?

Test yourself!

How can we check if any two vertices in an undirected graph are mutually reachable by some path without having to run a full DFS/BFS each time?

Answer: We just need to check if the two vertices reside within the same graph component! Thus we can simply use $O(V+E)$ BFS/DFS to first obtain a component table and then check if the two vertices correspond to the same component id.

Test yourself!

Now in preprocessing step, do we have to literally construct a trimmed graph for every value of $i \in [1, k]$?

Note: we don't know S and D ahead of time here!

Test yourself!

Now in preprocessing step, do we have to literally construct a trimmed graph for every value of $i \in [1, k]$?

Note: we don't know S and D ahead of time here!

Answer: No, we can still just run $O(V+E)$ modified DFS on each implicitly trimmed graph but this time to obtain their *vertex component tables* C_i .

Test yourself!

What is our query time now?

Test yourself!

What is our query time now?

Answer: Just $O(\log k)$, since subroutine `reachableInTrimmedGraph(S, D, w)` can be achieved in $O(1)$ now by comparing $C_w[S]$ with $C_w[D]$ where C_w is the vertex component table obtained from preprocessing step.

Outline: Solution 7

Binary search on preprocessed trimmed graphs

Preprocessing stage

- Preprocess all component tables C_i , $i \in [1, k]$ using DFS/BFS where C_i corresponds to the component table in the graph trimmed of weights $> i$

Query stage (given S and D)

- Binary search for the lowest $i \in [1, k]$ such that $C_i[S] = C_i[D]$

Analysis: Solution 7

Binary search on preprocessed trimmed graphs

| Time complexity | Preprocess | $O(k(V + E))$ |
|------------------|------------|---------------|
| | Query | $O(\log k)$ |
| Space complexity | Preprocess | $O(kV)$ |
| | Query | $O(1)$ |
| | DS | $O(kV)$ |



Putting it all together

Summary—Trade offs

| Solution | Time Complexity | | Space Complexity | | |
|---|---------------------|-------------------|------------------|--------|---------------|
| | Preprocess | Query | Preprocess | Query | DS |
| Modified relax SSSP with Dijkstra's | 0 | $O((V+E) \log V)$ | 0 | $O(V)$ | N/A |
| Modified relax APSP with Dijkstra's | $O(V(V+E) \log V)$ | $O(1)$ | $O(V^2)$ | $O(1)$ | $O(V^2)$ |
| Modified relax APSP with Floyd-Warshall's | $O(V^3)$ | $O(1)$ | $O(V^2)$ | $O(1)$ | $O(V^2)$ |
| Query-time MST [†] | 0 | $O(E \log V)$ | 0 | $O(V)$ | N/A |
| Preprocessed MST [†] | $O(E \log V)$ | $O(V)$ | $O(V)$ | $O(V)$ | $O(V)$ |
| Binary search on query-time trimmed graph | 0 | $O((V+E) \log k)$ | 0 | $O(V)$ | N/A |
| Binary search on preprocessed trimmed graphs | $O(k(V+E))$ | $O(\log k)$ | $O(kV)$ | $O(1)$ | $O(kV)$ |
| Preprocessed MST [†] + Binary search + LCA | $O(E \log V + V^2)$ | $O(\log^2 V)$ | $O(V \log V)$ | $O(1)$ | $O(V \log V)$ |

[†]: Implemented using Prim's with binary heap

Lowest Common Ancestor

Definition

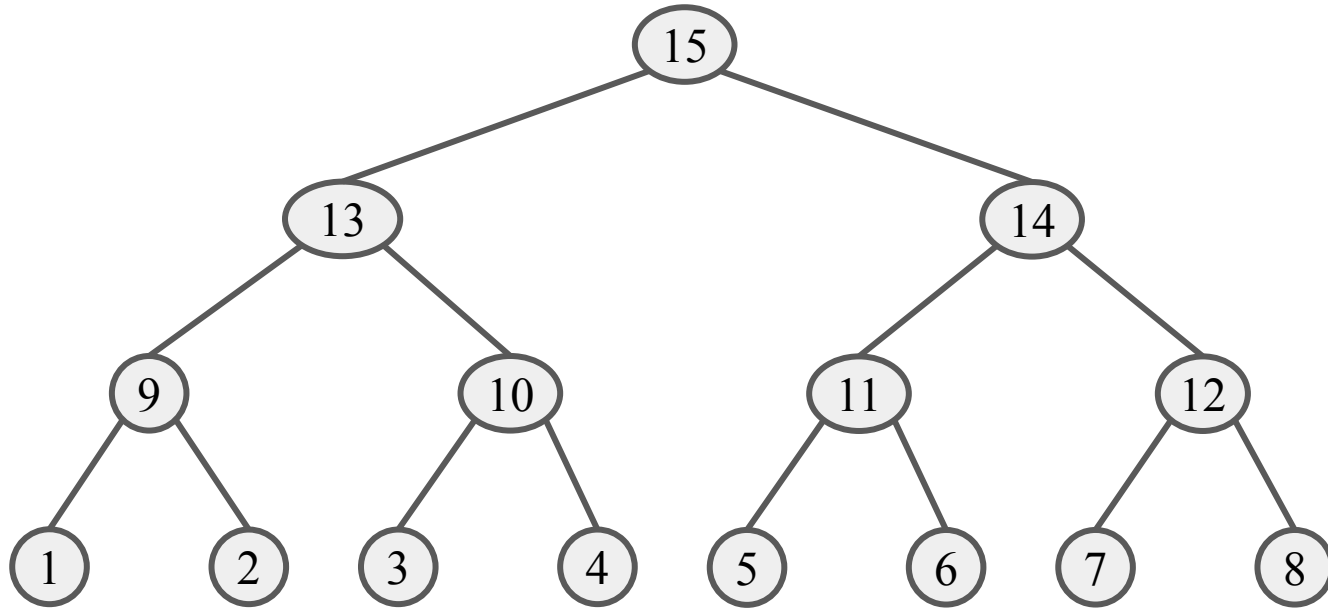
The *Lowest Common Ancestor* (LCA) of two nodes u and v in a tree is the deepest node in the tree with both u and v as its descendants.

You should convince yourself that LCA is unique and it exist for *any two* nodes in a tree.

We shall hereby denote $LCA(u, v)$ as the LCA of u and v .

Test yourself!

For the following binary tree, what are the following LCA:



LCA(1, 2)

LCA(1, 9)

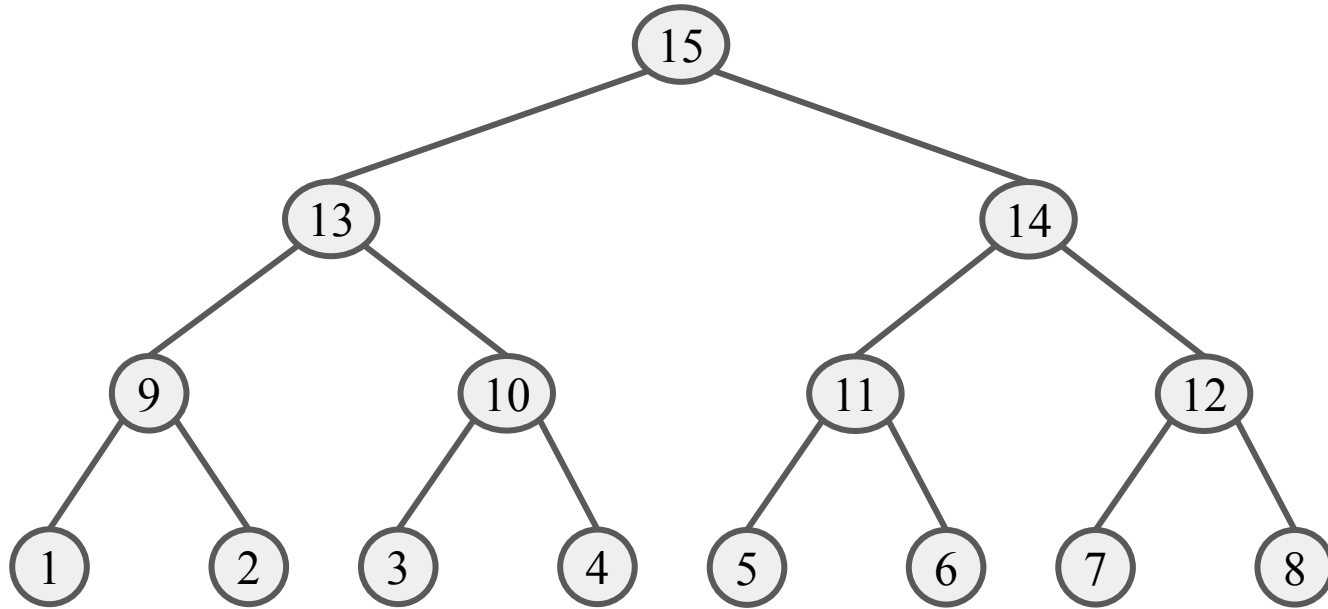
LCA(2, 10)

LCA(9, 5)

LCA(1, 15)

Test yourself!

For the following binary tree, what are the following LCA:



$$\text{LCA}(1, 2) = 9$$

$$\text{LCA}(1, 9) = 9$$

$$\text{LCA}(2, 10) = 13$$

$$\text{LCA}(9, 5) = 15$$

$$\text{LCA}(1, 15) = 15$$

Test yourself!

What is the *highest* possible LCA between two nodes in a tree?

Test yourself!

What is the *highest* possible LCA between two nodes in a tree?

Answer: The root node!

Solution 8

Test yourself!

If we were to improve on the query time of the previous preprocessed MST solution, what does it mean?

Test yourself!

If we were to improve on the query time of the previous preprocessed MST solution, what does it mean?

Answer: It means we cannot afford to do $O(V)$ BFS/DFS on the MST to obtain the path width between 2 vertices!

In other words, we have to do better than traversing the entire path between the 2 vertices.

Test yourself!

What is one important property that trees have over general undirected graphs which we can take advantage of?

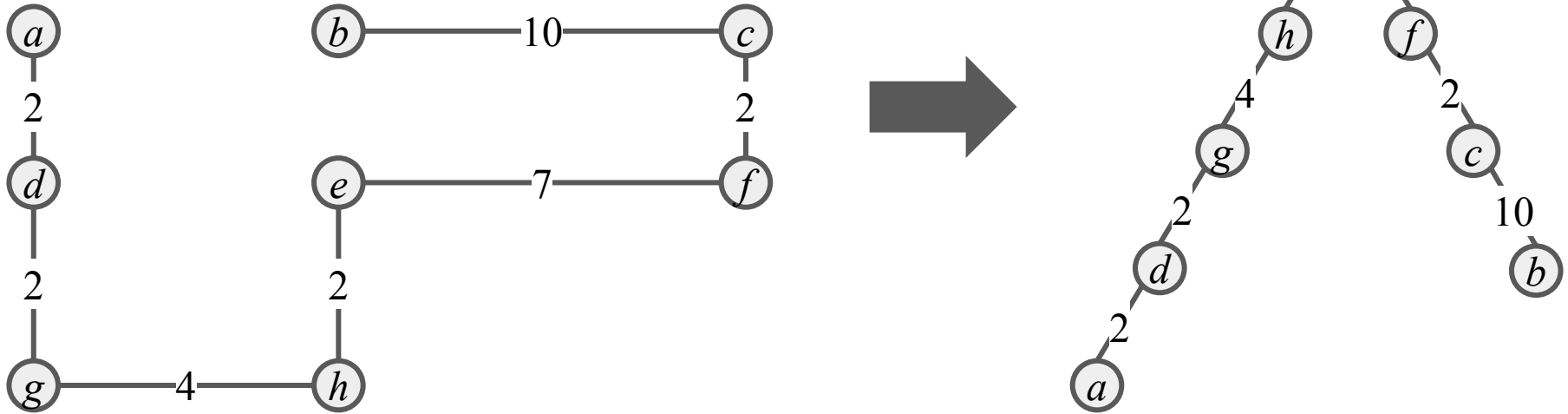
Test yourself!

What is one important property that trees have over general undirected graphs which we can take advantage of?

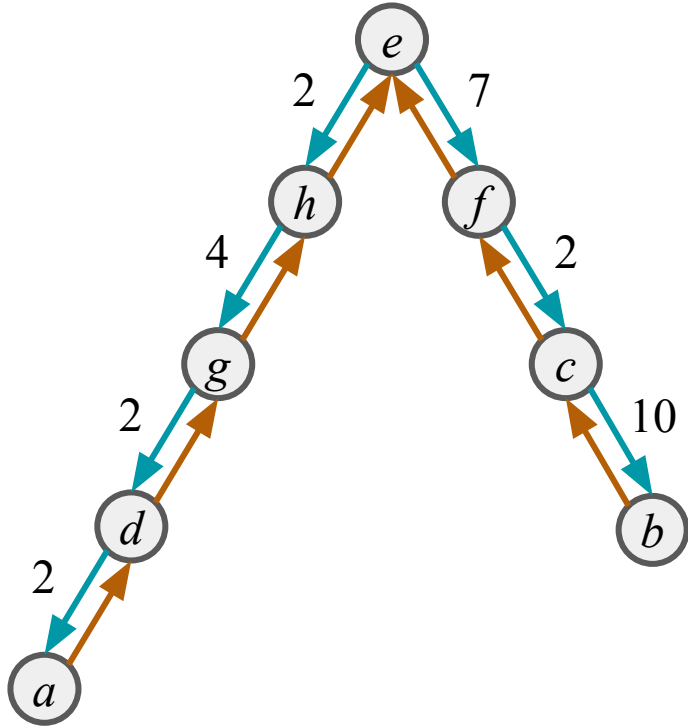
Answer: Every node in the tree has a unique path to every other node. We can therefore preprocess the width along query paths!

Example: Our MST

We can draw the MST as a rooted tree by choosing any vertex as the root.



Building intuitions



Note that trees can have undirected edges but here we need to differentiate between **parent-to-child** links and **child-to-parent** links because we are making use of the hierarchical property of rooted trees.

Query problem — Definition

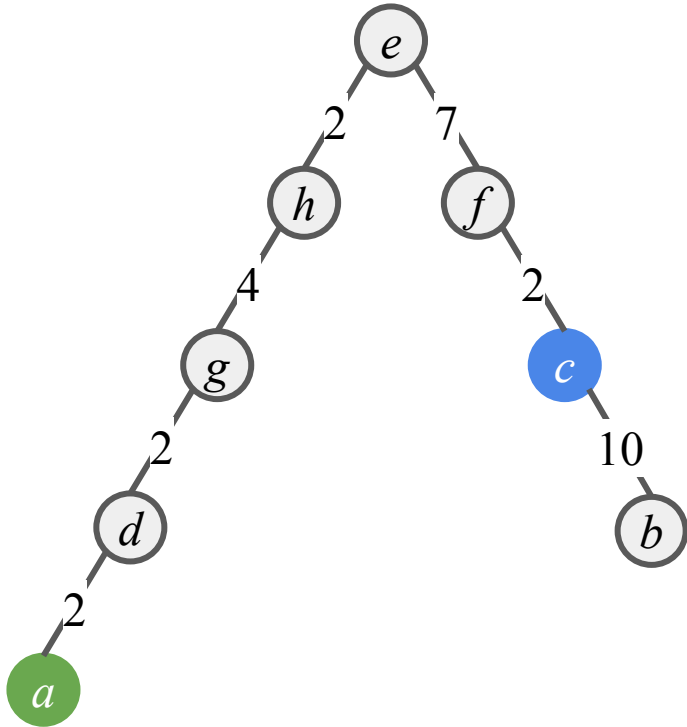
Let us define $\text{width}(u, v)$ as the width of the path from *descendant* u upwards to *ancestor* v in the tree.

For a given start node S and target node D , the minimax distance between them is

$$\max(\text{width}(S, L), \text{width}(D, L))$$

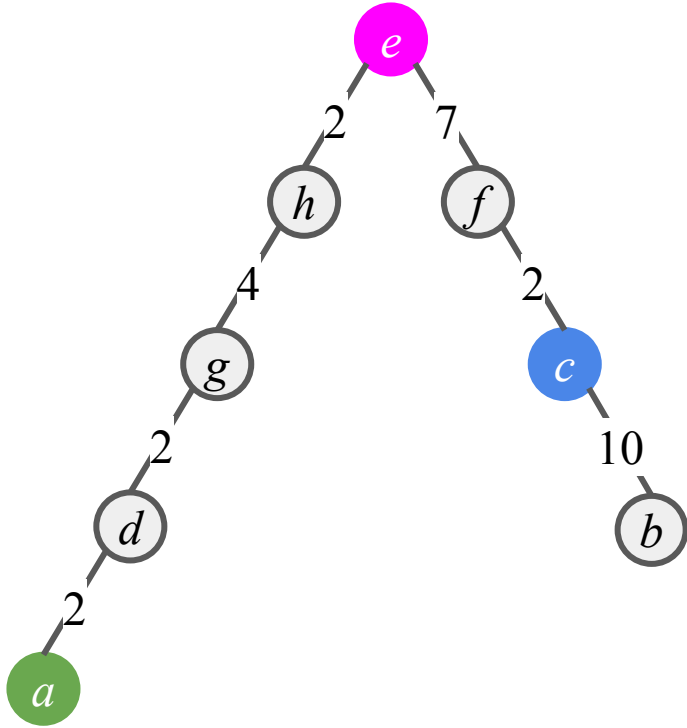
Where L is the $\text{LCA}(S, D)$.

Test yourself!



What is the LCA of a and c in this tree?

Test yourself!

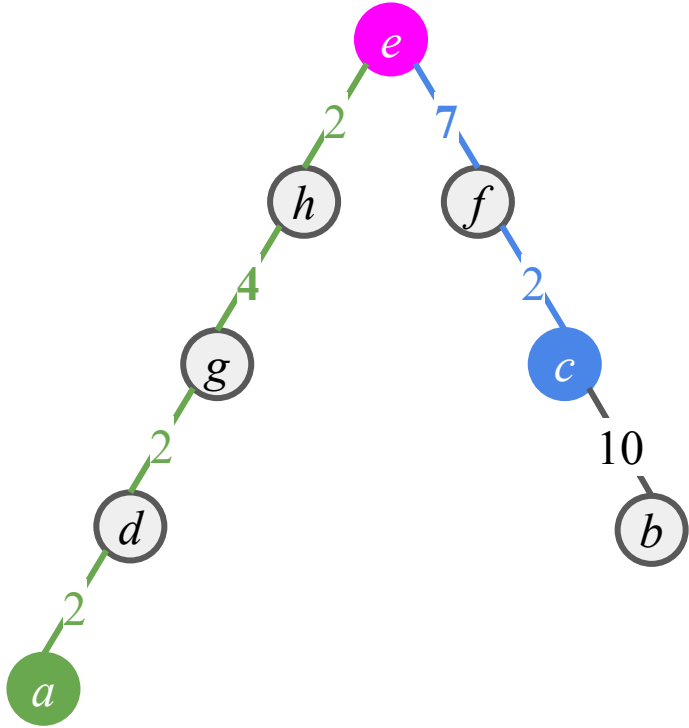


What is the LCA of a and c in this tree?

Answer: Node e .

Note that it just so happens to be the root in this simple example.

Example: Query problem

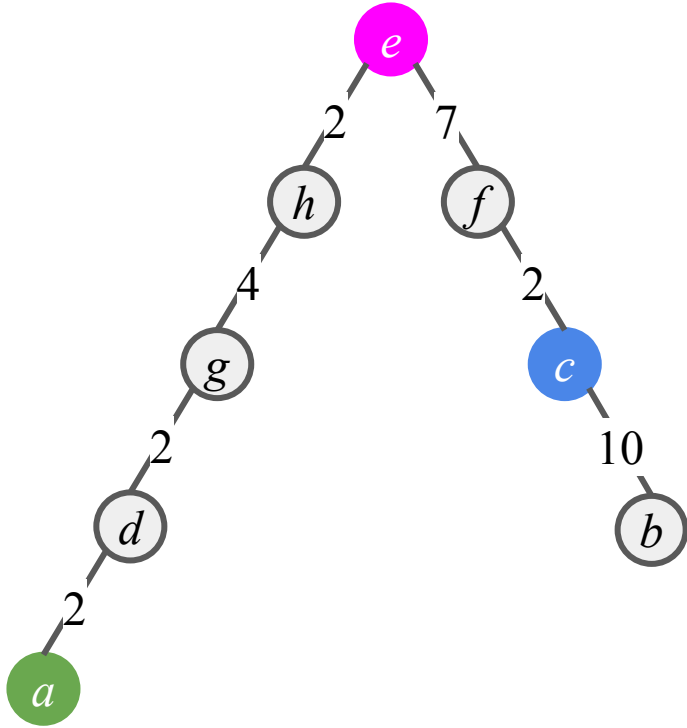


So if we were to query for the maximum edge between *a* and *c*, the query problem now becomes finding:

$$\max(\text{width}(a, e), \text{width}(c, e))$$

$$\max(4, 7) = 4$$

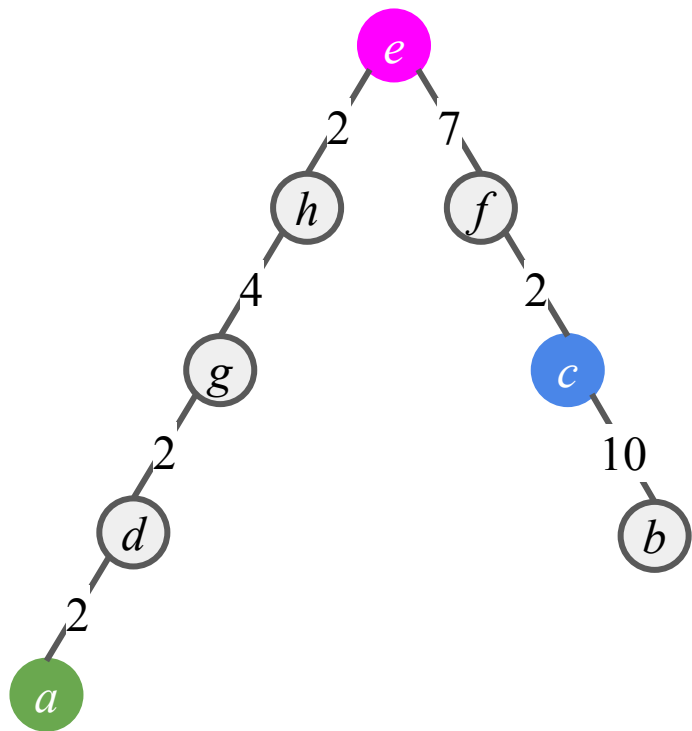
Example: Query problem



To compute that, you might be tempted to: start from *a* and *c* and traverse up the tree till we reach *e*, tracking the heaviest edge weight along the way.

However that is no different than doing a DFS on the MST, which is exactly what we **don't want** to do!

Preprocess — Key idea



Realize every node in the tree has a unique path to every of its ancestors.

In order to avoid full path traversals along vertex-to-LCA paths at query time, we can precompute the widths of *vertex-to-ancestor* paths.

Working vocabulary

Before we proceed, let's use *lineage* to refer to a *vertex-to-ancestor* path (upwards the tree).

Again, the usage of this term is not part of the standard graph theory nomenclature! It's just for my own convenience in words.

Strategy

Preprocessing stage

- Precompute lineage widths

Query stage (given S and D)

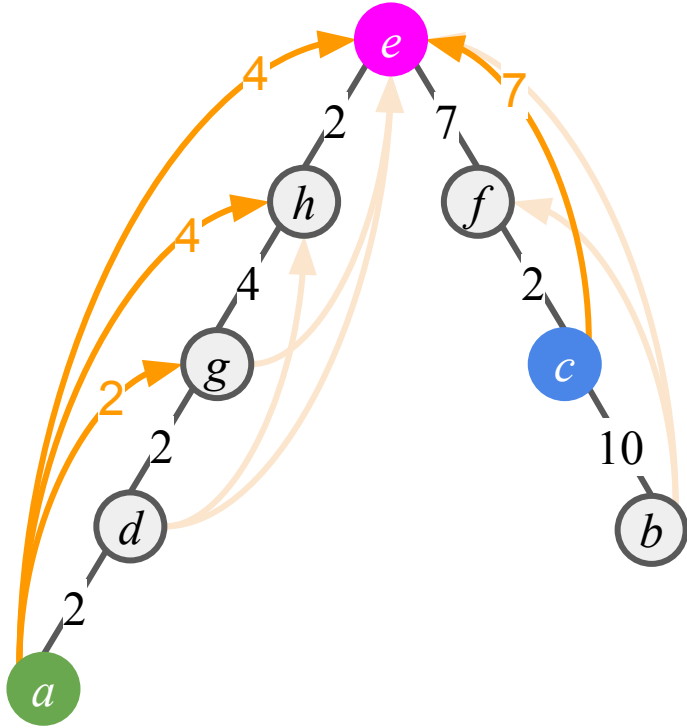
1. Find $LCA\ L \leftarrow LCA(S, D)$ *efficiently*
2. Use precomputed lineage widths to obtain $width(S, L)$ and $width(D, L)$ *efficiently*
3. Obtain $\max(width(S, L), width(D, L))$

Keys to success

At this point it should be clear that the keys to success lies in how we tackle these 2 challenges:

1. Which path widths to precompute?
2. What is the fastest way to find LCA?

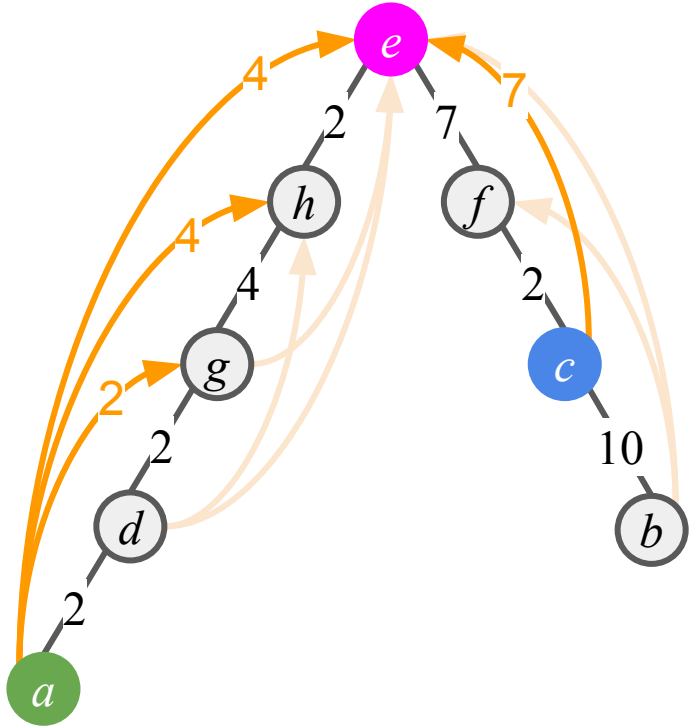
Precomputing widths—Exhaustive



Suppose for every node, we store the heaviest weights along every of its *lineages* as depicted in the diagram:

- Orange arrows denote *pointers* to ancestors and are not edges
- Numbers on each pointer denote width along the path up to that ancestor

Precomputing widths—Exhaustive



With the widths along all lineages precomputed, we can quickly determine the following:

$$\max(\text{width}(a, e), \text{width}(c, e))$$

$$\max(4, 7) = 7$$

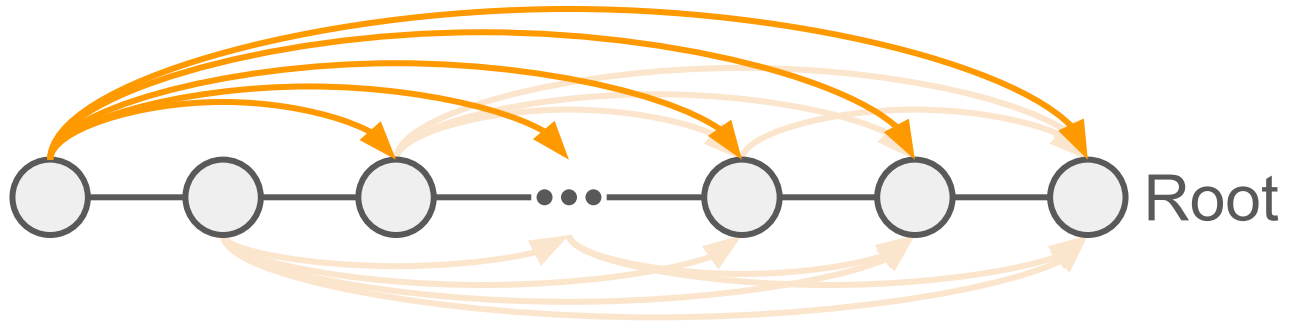
Test yourself!

How many lineages are there in a tree?

Test yourself!

How many lineages are there in a tree?

Answer: In the worst case, the tree is a line of vertices with the root on one end. Therefore the number of vertex-to-ancestor *paths* is $V-2 + V-3 + \dots + 1 = O(V^2)$.



Test yourself!

What is the space and time complexity needed for preprocessing each node and memoizing the widths along all of its lineages?

Test yourself!

What is the space and time complexity needed for preprocessing each node and memoizing the widths along all of its lineages?

Answer: Since there are a total of $O(V^2)$ such paths, caching the width along every path requires an additional $O(V^2)$ space. Realize that for a node, computing the width of all its lineages takes $O(V)$ because we just need to do a single traversal up to the root and save the current maximum at every ancestral node. Thus preprocessing time is also $O(V^2)$.

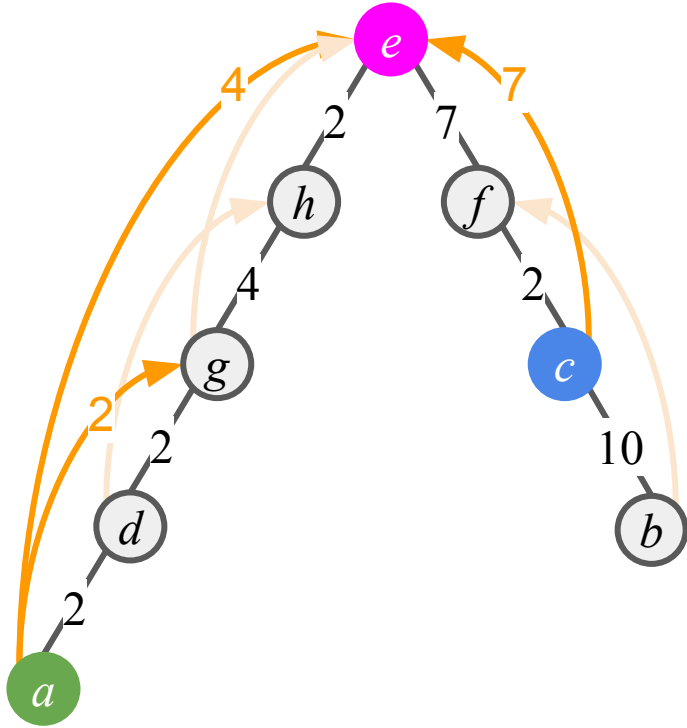
Precomputing widths—Skip pointers

Key idea

Realize we don't have to memoize all lineage widths. We can apply the idea of *skip pointers* here:

- Each node in the tree stores the pointers to ancestors $2=2^1$ hops up, $4=2^2$ hops up, $8=2^3$ hops up and so on
- Each node also stores the preprocessed widths along the lineages to these ancestors

Example: Skip pointers



Precomputing widths

For a node u , we define the following accessors:

- $u.\text{lineageWidth}(2^k)$: access the preprocessed width of the lineage to the ancestor 2^k hops up
- $u.\text{ancestor}(2^k)$: the ancestor 2^k hops up

Where k must be an integer greater or equal to 0. Note that when k is 0 then we are just looking at the parent of u and the `lineageWidth` is just the edge weight to the parent.

Test yourself!

How many lineage widths does each node have to store?

What is the total time needed to precompute them?

Test yourself!

How many lineage widths does each node have to store?

What is the total time needed to precompute them?

Answer: In the worst case, each node have to store $O(\log V)$ number of path widths and pointers thereby incurring a total of $O(V \log V)$ space. Since for a single node a $O(V)$ traversal up to the root can precompute all of the $O(\log V)$ widths along the way, total time needed for all nodes is $O(V^2)$.

Query—lineage width

Now we are ready to specify how to obtain the width of the lineage from a node S to the LCA at query time after the preprocessing scheme we outlined using skip pointers.

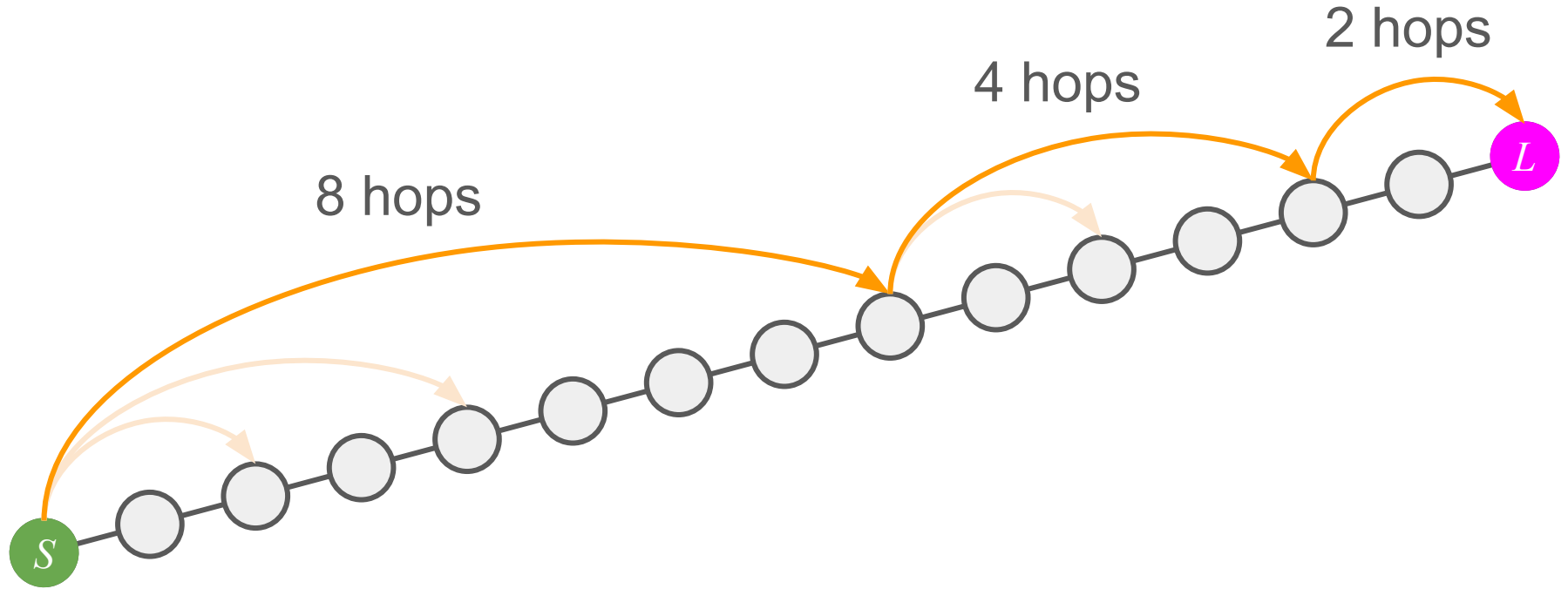
To do this, we shall define a $\text{getWidth}(S, \text{hops})$ function where *hops* refer to the number of edge-hops needed to go from S to the LCA.

Query—lineage width

$\text{getWidth}(S, \text{hops})$ is outlined as follows:

- Start at current node S with the current maximum width as zero
- Skip *as high up* to LCA as possible using the preprocessed exponentially-incremented pointers to ancestors (by updating current node to the ancestor node hopped to)
- For each hop, update the maximum width if the width due to the hop is greater than current maximum
- When we have exhausted all the hops (i.e. currently at LCA node), we can read off the maximum recorded width as the vertex-to-LCA width

Example: getWidth



Algorithm: getWidth

```
getWidth(u, hops)
```

```
return getWidthHelper(  
    u,  
    hops,  
    0  
)
```

```
getWidthHelper(currNode, hopsRemaining, currMax)
```

```
if hopsRemaining = 0 then  
    return currMax  
else  
     $kMost \leftarrow \lfloor \log_2 hopsRemaining \rfloor$   
    return getWidthHelper(  
        currNode.ancestor( $2^{kMost}$ ),  
         $hopsRemaining - 2^{kMost}$ ,  
        max(currNode.lineageWidth( $2^{kMost}$ ), currMax)  
    )  
end
```


Test yourself!

What is the time complexity of `getWidth`?

Test yourself!

What is the time complexity of `getWidth`?

Answer: $O(\log V)$. Similar analysis as exponentiation by squaring method (recall `MysteryFunction` in PS1).

Query—LCA

Now we are ready to look at how to find $L \leftarrow \text{LCA}(S, D)$.

Realize that `getWidth` expects the L to be provided in terms of the *number of hops* relative to each vertex.

Therefore in this problem, we need to define the function for LCA slightly differently to adapt it for use with `getWidth`. We shall thus define $\text{findLCA}(u, v)$ as the **number of hops up** from u to $\text{LCA}(u, v)$.

Query—LCA

Let us first define a subroutine `getAncestor` which obtains in $O(\log V)$ time the ancestor that is *hops* hops away from given node *u*.

```
getAncestor(u, hops)  
currNode  $\leftarrow$  u  
hopsRemaining  $\leftarrow$  hops  
  
while hopsRemaining > 0 do  
    kMost  $\leftarrow$   $\lfloor \log_2 \text{hopsRemaining} \rfloor$   
    currNode  $\leftarrow$  currNode.ancestor( $2^{kMost}$ )  
    hopsRemaining  $\leftarrow$  hopsRemaining  $- 2^{kMost}$   
end  
  
return currNode
```

Query—LCA

Let us also define a subroutine `isAncestor` which checks in $O(\log V)$ if the given *currNode* is an ancestor of node *v*.

```
isAncestor(currNode, v)
```

```
checkHops  $\leftarrow$  currNode.depth  $-$  v.depth
```

```
if checkHops < 0 then
```

```
    return false
```

```
end
```

```
vAncestor  $\leftarrow$  getAncestor(v, checkHops)
```

```
return currNode = vAncestor
```

LCA—Naïve approach

findLCA(u, v)

$curr \leftarrow u$

$hops \leftarrow 0$

while not isAncestor($curr, v$) **do**

$curr \leftarrow curr.parent$

$hops \leftarrow hops + 1$

end

return $hops$

Realize the LCA is simply the node with the least number of hops up that is an ancestor to both u and v . We can therefore find this ancestor by incrementally hopping our way upwards to the root.

Test yourself!

What is the time complexity of this LCA algorithm?

Test yourself!

What is the time complexity of this LCA algorithm?

Answer: Each `isAncestor` call costs $O(\log V)$ time and it is incurred at every hop. In the worst case u will need $V-1$ hops and so this is a $O(V \log V)$ solution.

Test yourself!

How can we improve on this LCA algorithm?

Test yourself!

How can we improve on this LCA algorithm?

Answer: Realize this LCA algorithm is just doing a linear search from 1 hop away up to V hops away. We can thus optimize it via binary search!

LCA—Binary search approach

findLCA(u, v)

$lo \leftarrow 1$

$hi \leftarrow u.depth$

while $lo < hi$ **do**

$mid \leftarrow lo + \lfloor (lo + hi) / 2 \rfloor$

$ancestor \leftarrow \text{getAncestor}(u, mid)$

if $\text{isAncestor}(ancestor, v)$ **then**

$hi \leftarrow mid - 1$ // Search LHS

else

$lo \leftarrow mid + 1$ // Search RHS

end

end

return $hops$

Test yourself!

What is the time complexity of this binary search LCA algorithm?

Test yourself!

What is the time complexity of this binary search LCA algorithm?

Answer: Binary search takes $O(\log V)$ steps and each step takes $O(\log V)$ time due to `getAncestor` and `isAncestor` calls (each incurs $O(\log V)$). Hence total time is $O(\log^2 V)$.

Outline: Solution 8

Preprocessed MST + Binary search + LCA

Preprocessing stage

- Obtain MST T
- For each vertex in T , add $O(\log V)$ ancestral skip pointer to it and precompute lineage widths to them

Outline: Solution 8

Preprocessed MST+Binary search+LCA

Query stage (given S and D)

- Obtain $h_S = \text{findLCA}(S, D)$, the number of hops up from S to their LCA
- Obtain $h_D = \text{findLCA}(D, S)$, the number of hops up from D to their LCA
- Obtain $w_S = \text{getWidth}(S, h_S)$, the width of lineage from S to the LCA
- Obtain $w_D = \text{getWidth}(D, h_D)$, the width of lineage from D to the LCA
- The minimax distance is $\max(w_S, w_D)$

Analysis: Solution 8

Preprocessed MST+Binary search+LCA

| | | |
|-------------------------|------------|---------------------|
| Time complexity | Preprocess | $O(E \log V + V^2)$ |
| | Query | $O(\log^2 V)$ |
| Space complexity | Preprocess | $O(V \log V)$ |
| | Query | $O(1)$ |
| | DS | $O(V \log V)$ |

Thank you!

We would greatly appreciate if you could give us a review via Student Feedback!

Though there is a slight issue.. The system only permits you to review your assigned TA. So for those who have been attending another TA's sessions and intend to review your assigned TA with an average score, kindly refrain from doing so as it will unjustifiably harm their feedback score :(