Very Dynamic Steiner Trees:

Handling Graph Modifications in Dynamic Steiner Trees

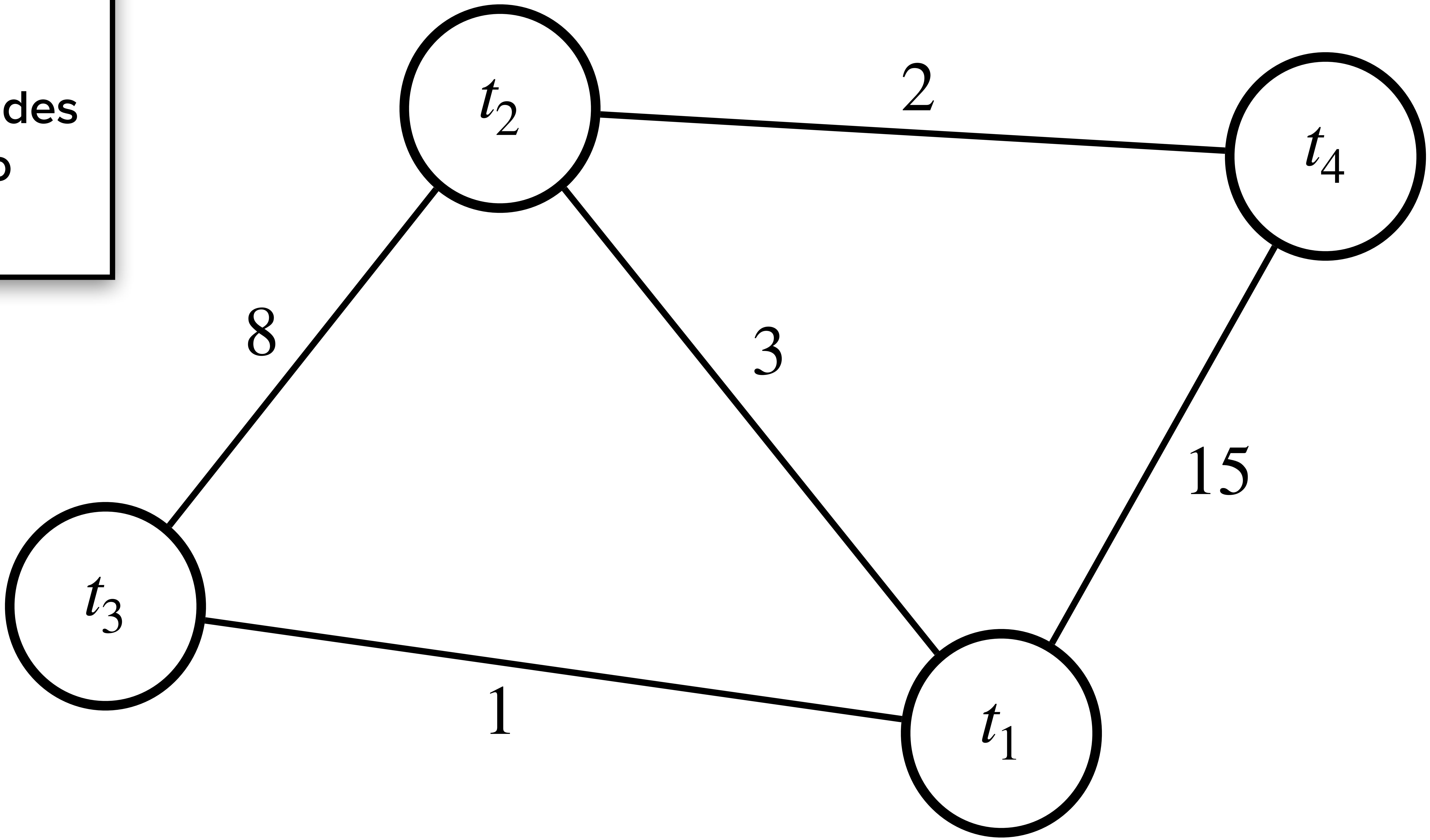
Mihir Garimella
Prathik Naidu
Pranav Sriram

Introduction

Q: What's the cheapest way to connect terminals?

$t_i \in T$

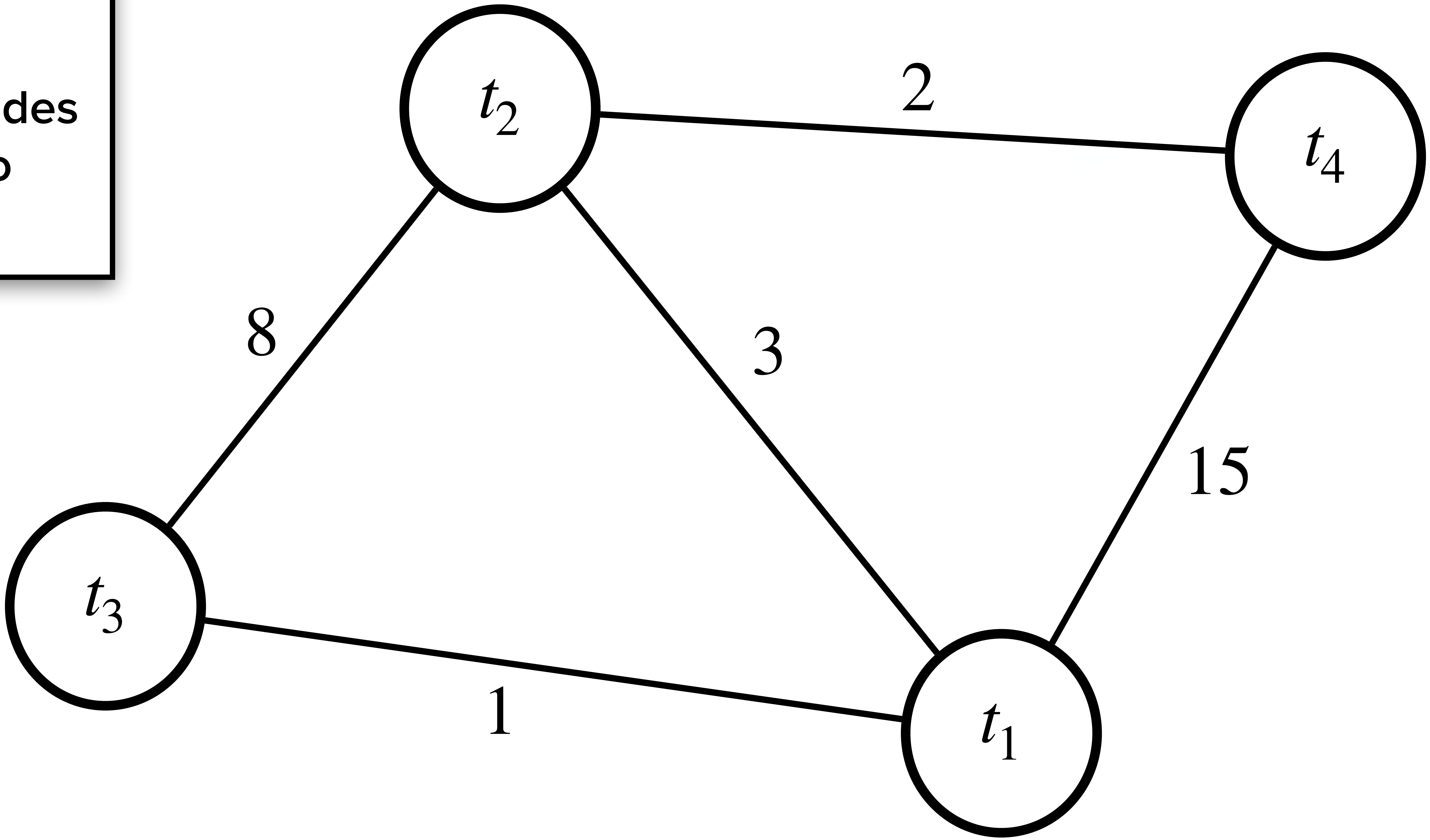
"Terminal" nodes we want to connect



A: Just find the MST!

$t_i \in T$

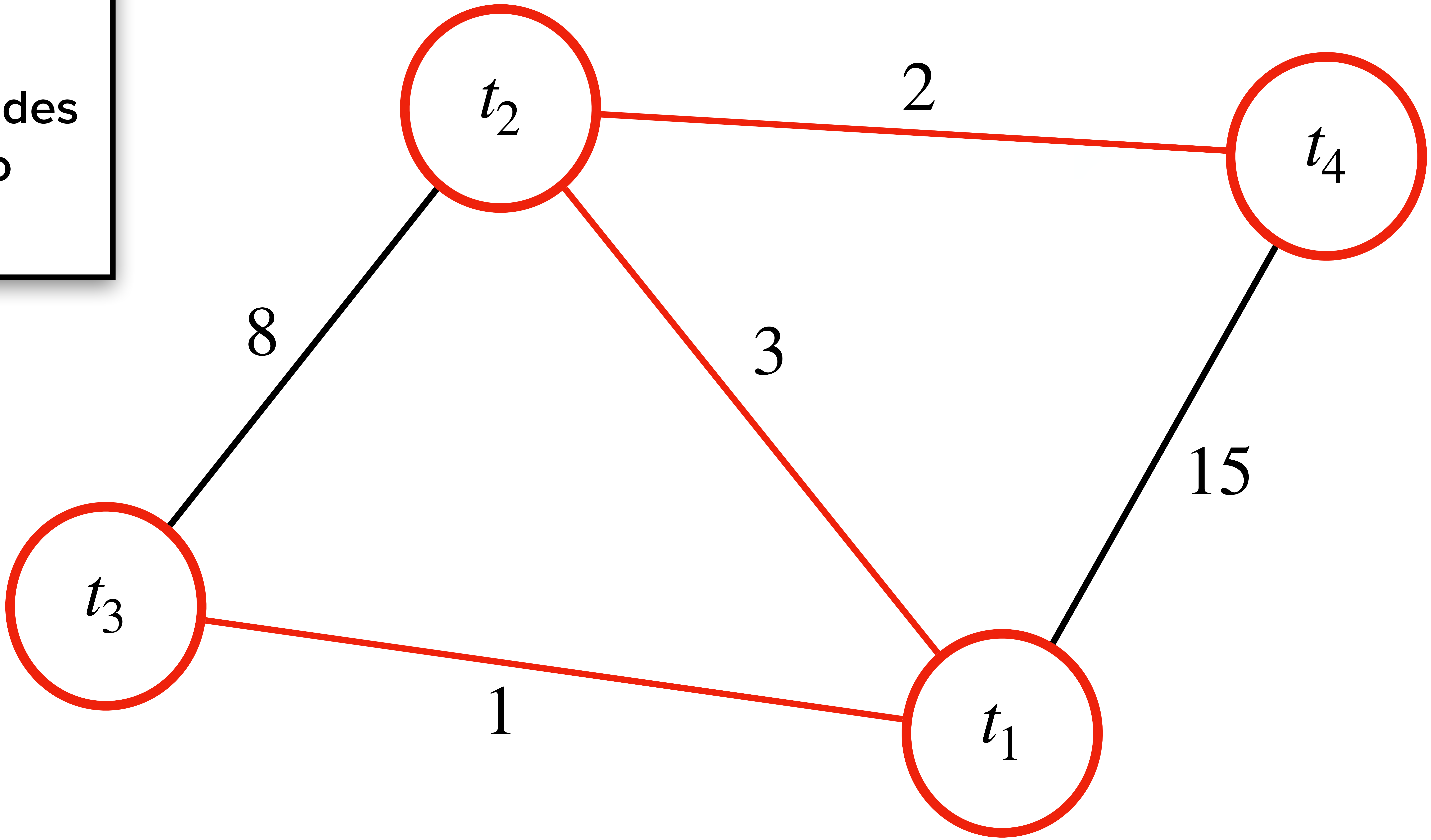
"Terminal" nodes we want to connect



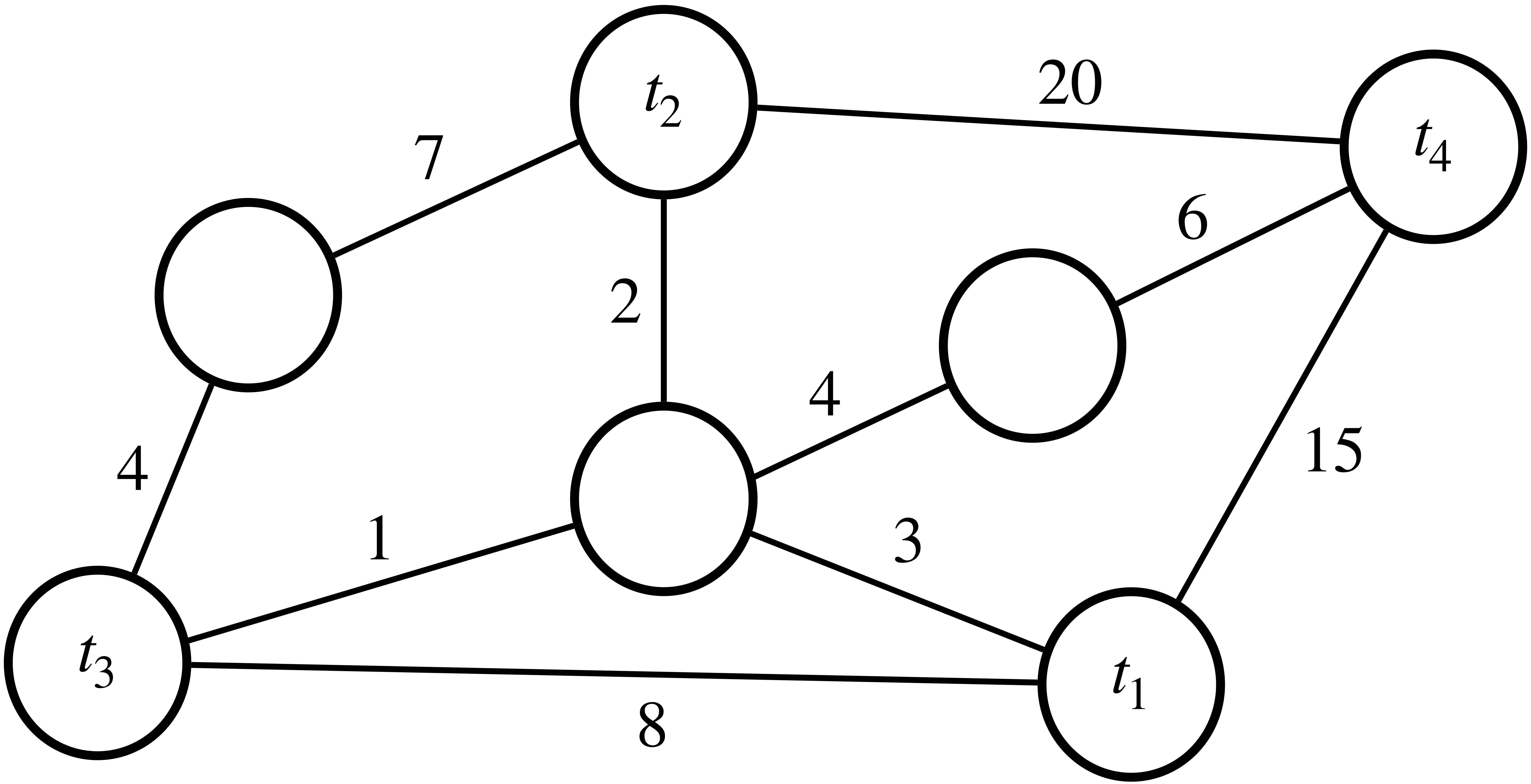
A: Just find the MST!

$t_i \in T$

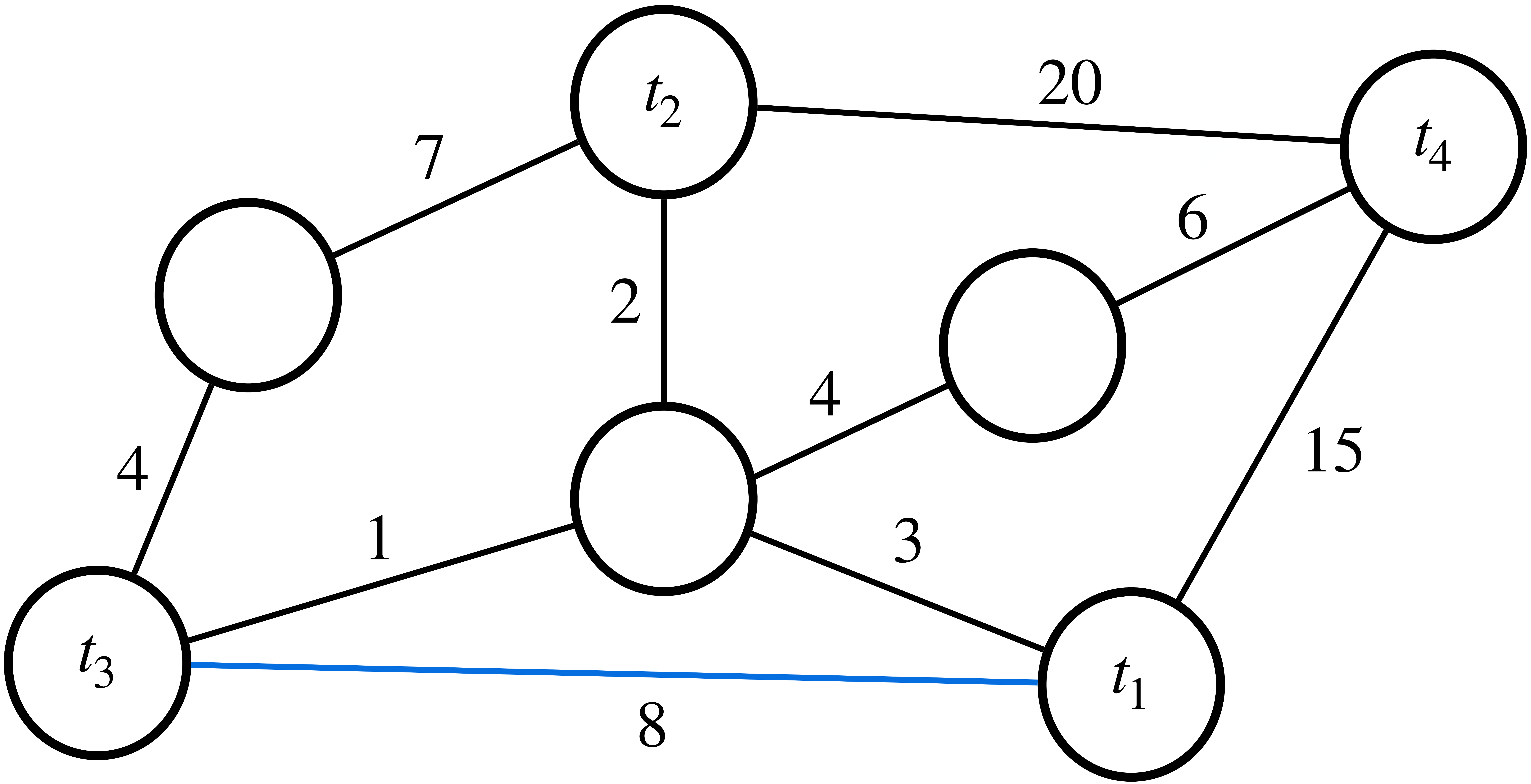
"Terminal" nodes we want to connect



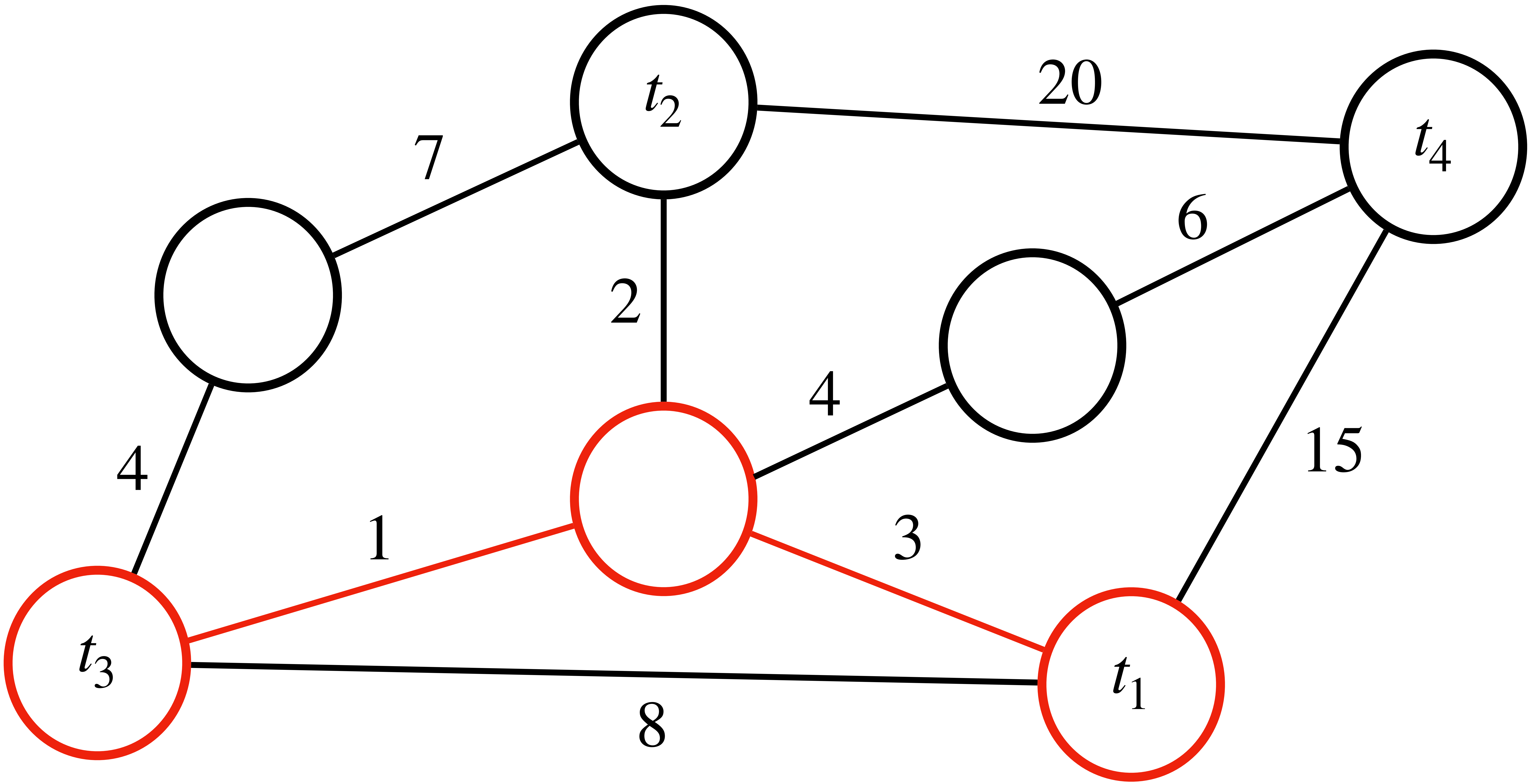
Q: What about now?



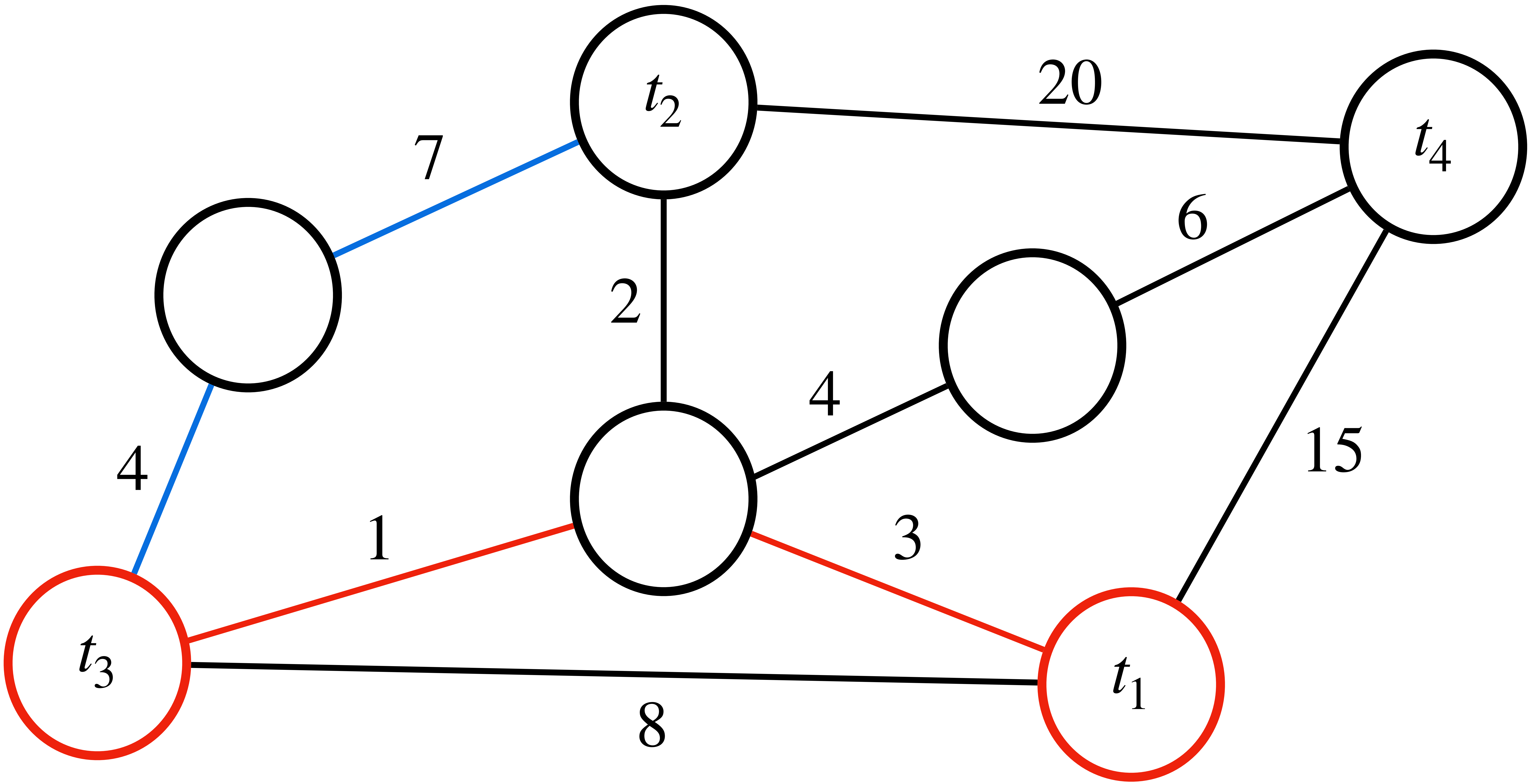
A: Well, we still want to connect all terminals...



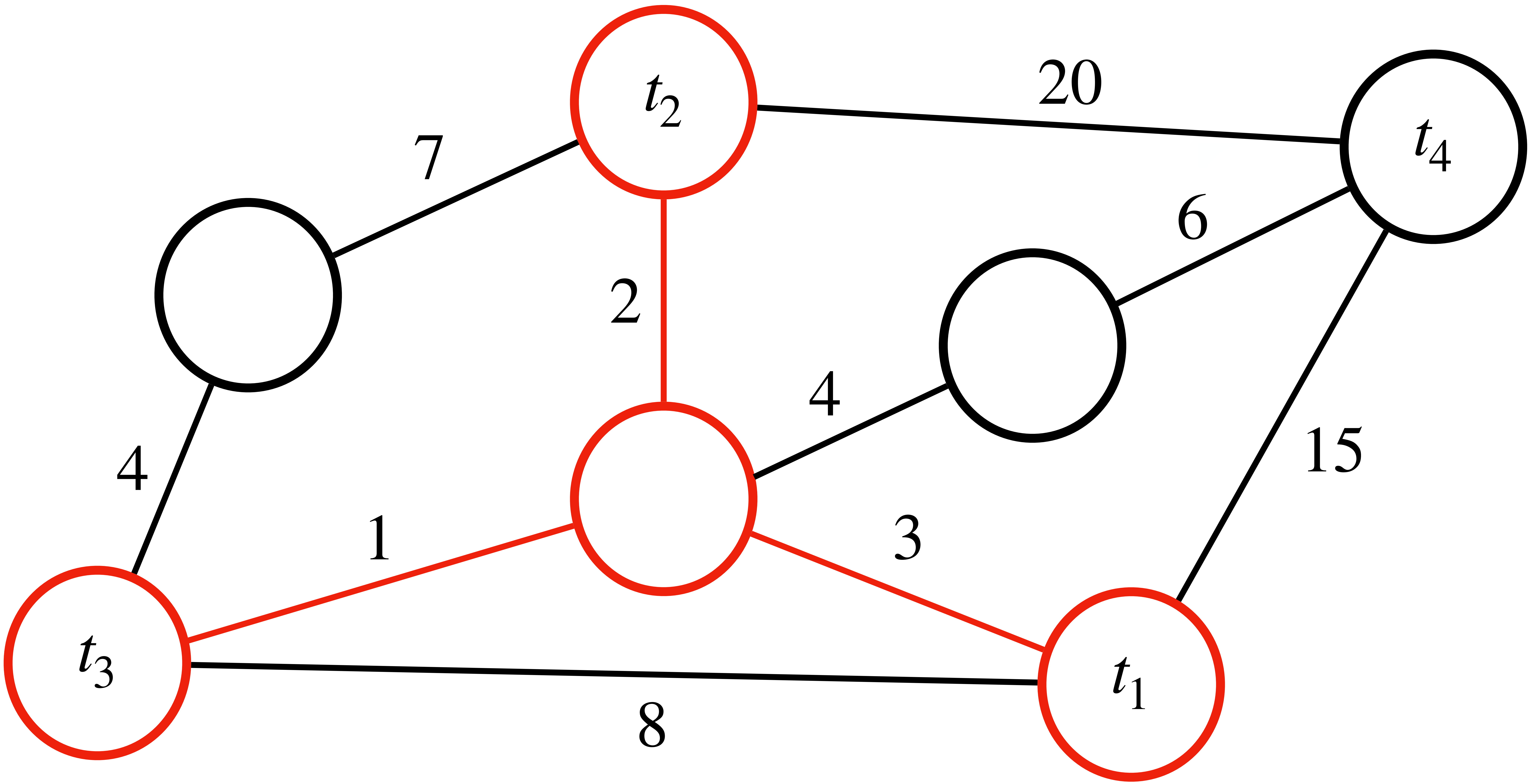
...but it might be cheaper to use other nodes as well!



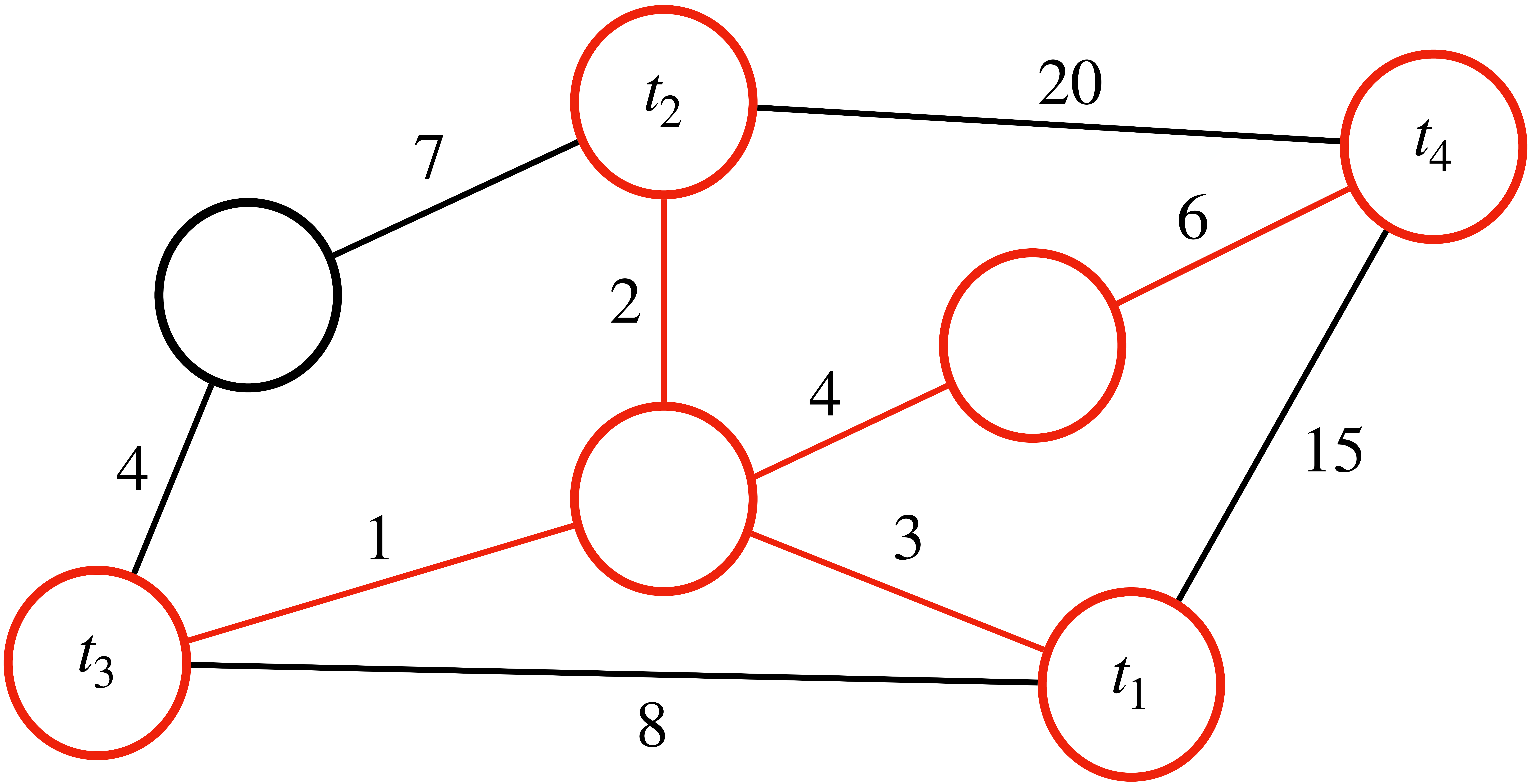
...but it might be cheaper to use other nodes as well!



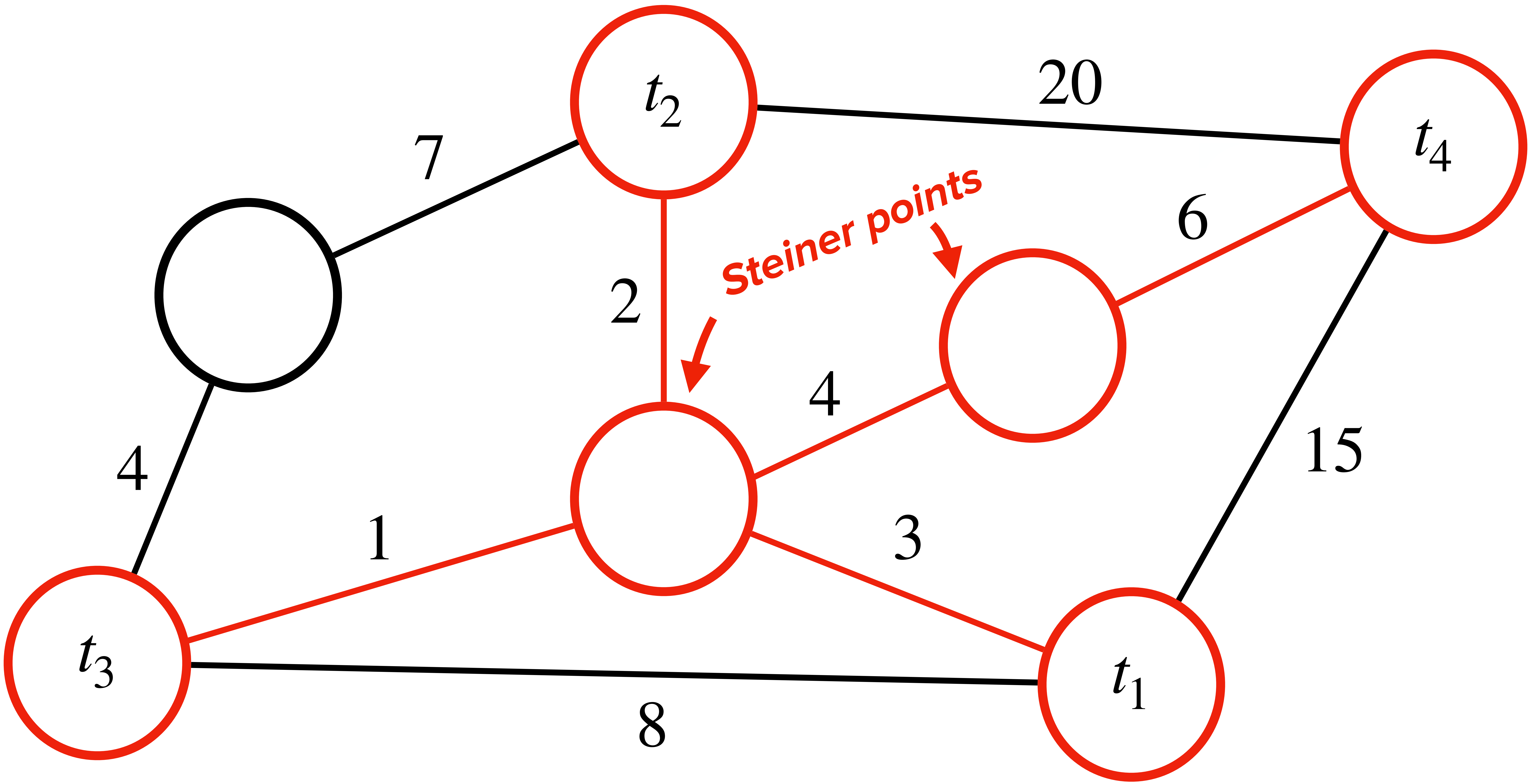
...but it might be cheaper to use other nodes as well!



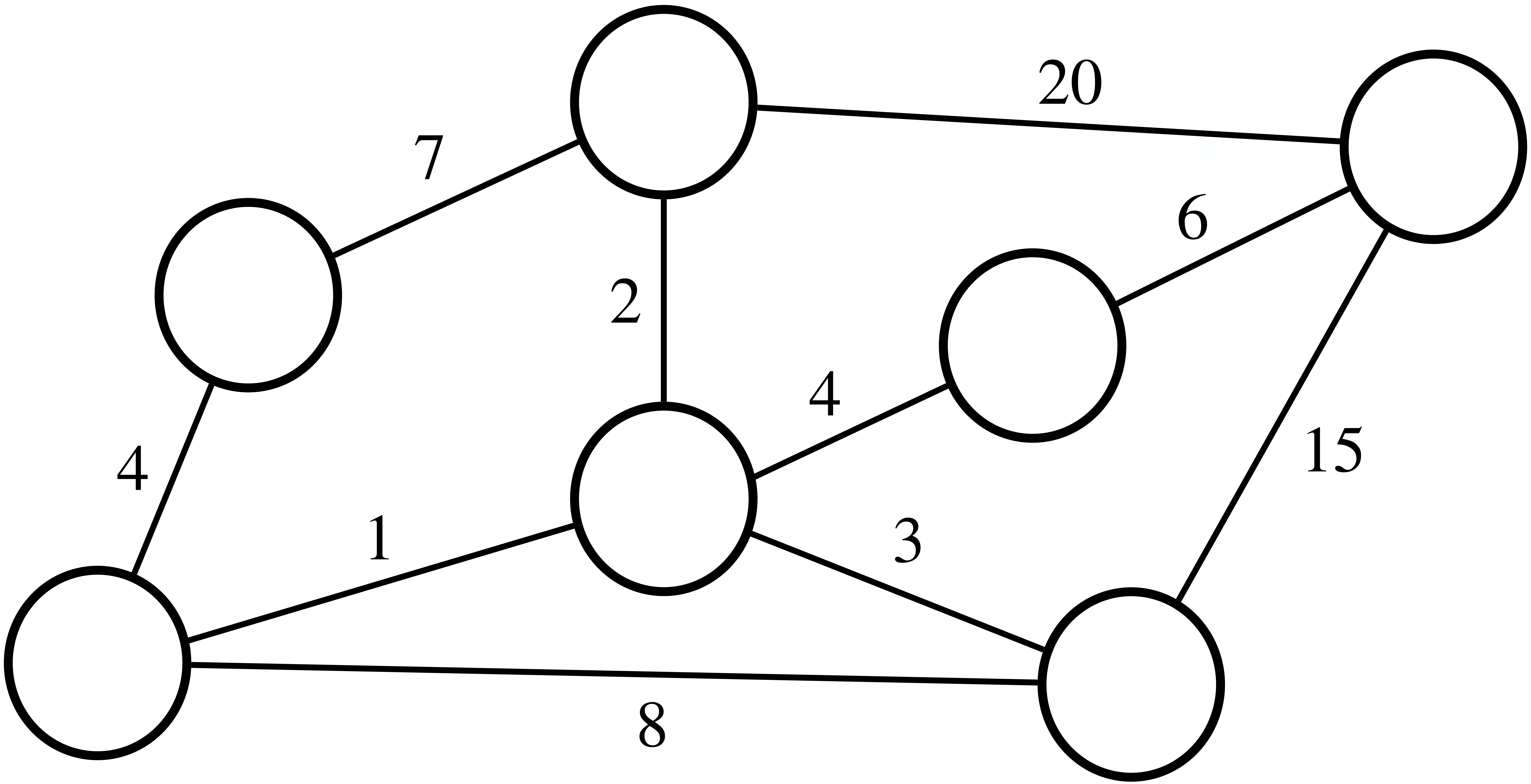
...but it might be cheaper to use other nodes as well!



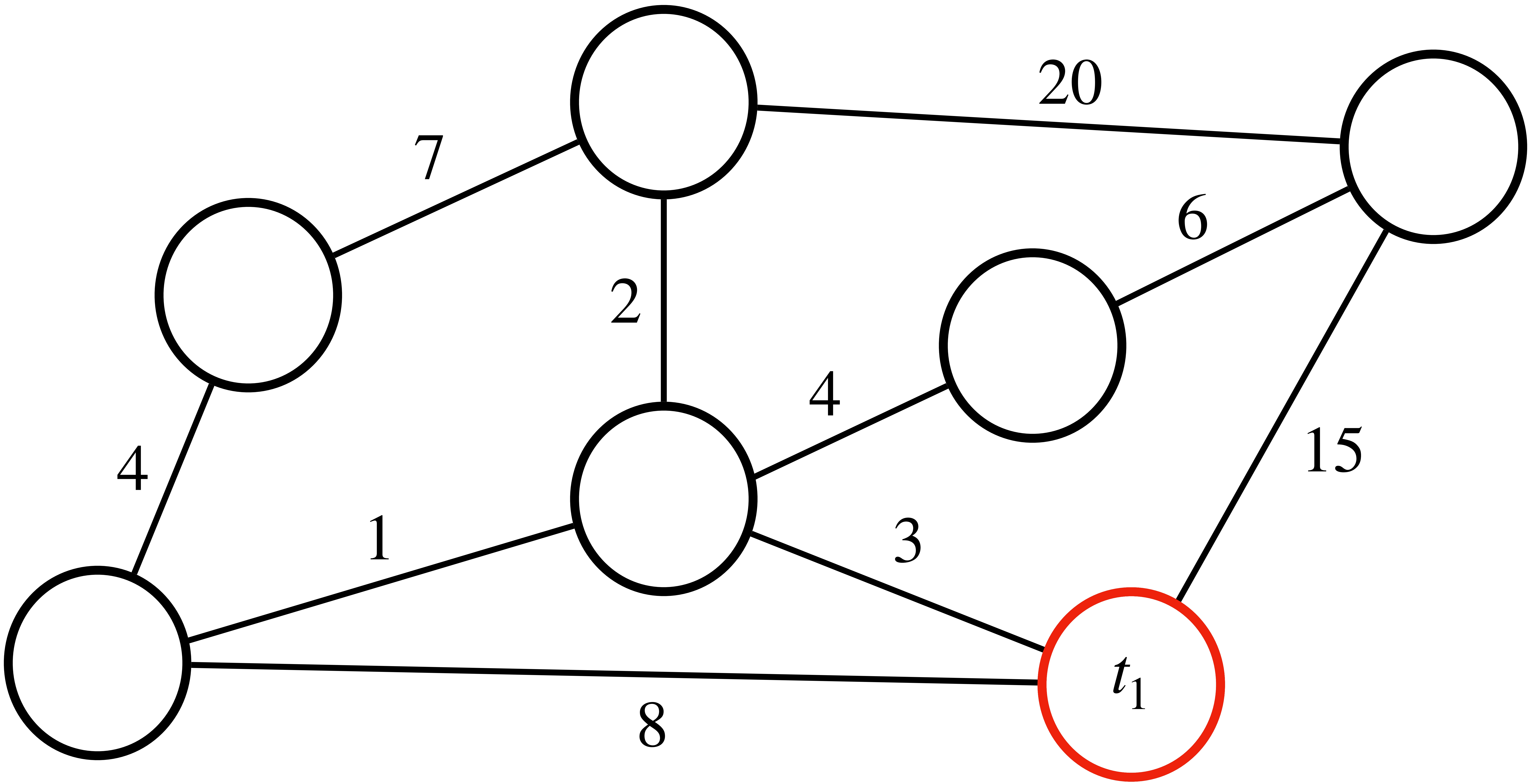
That's a *Steiner tree*!



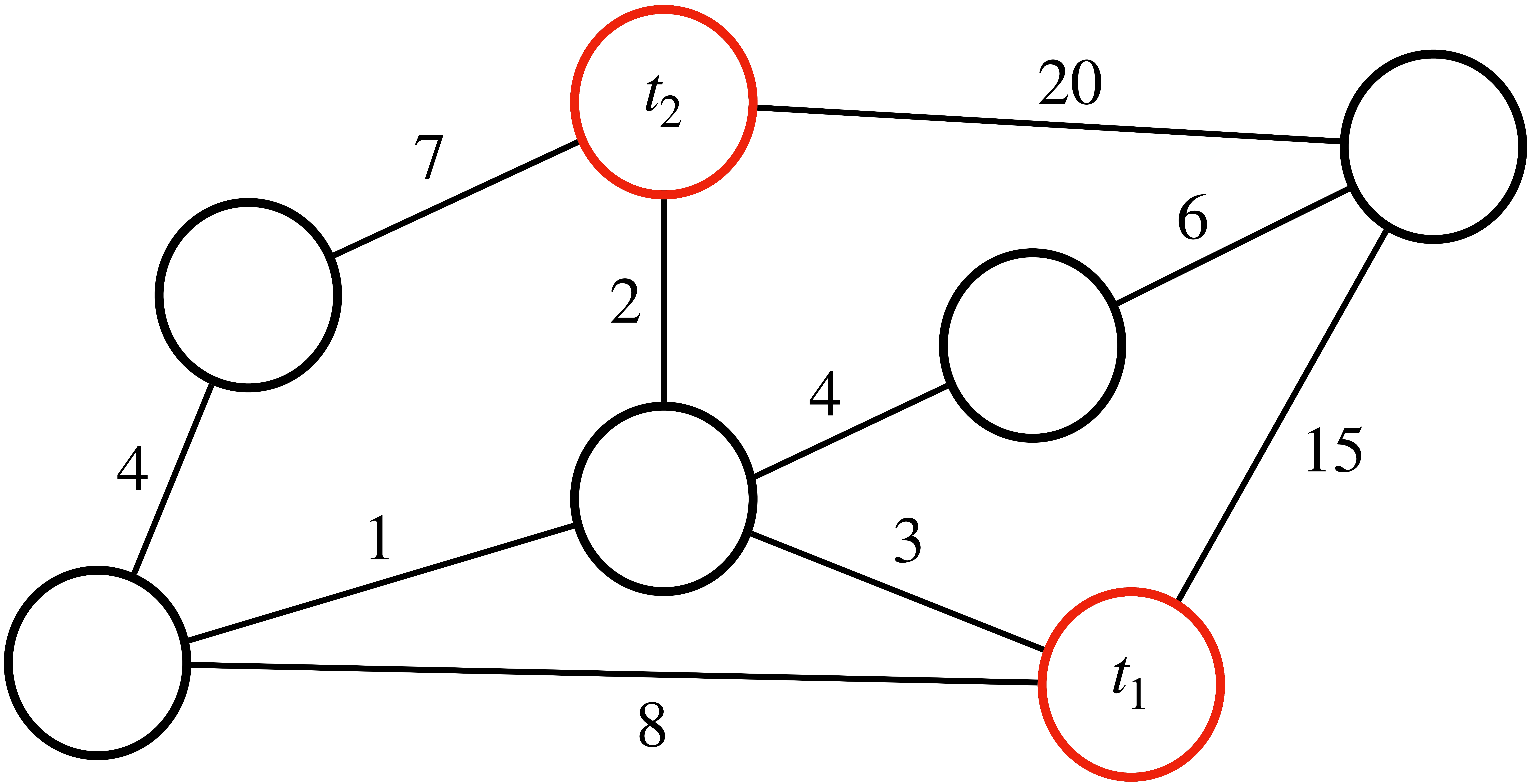
Q: Now, suppose we just start with a regular graph...



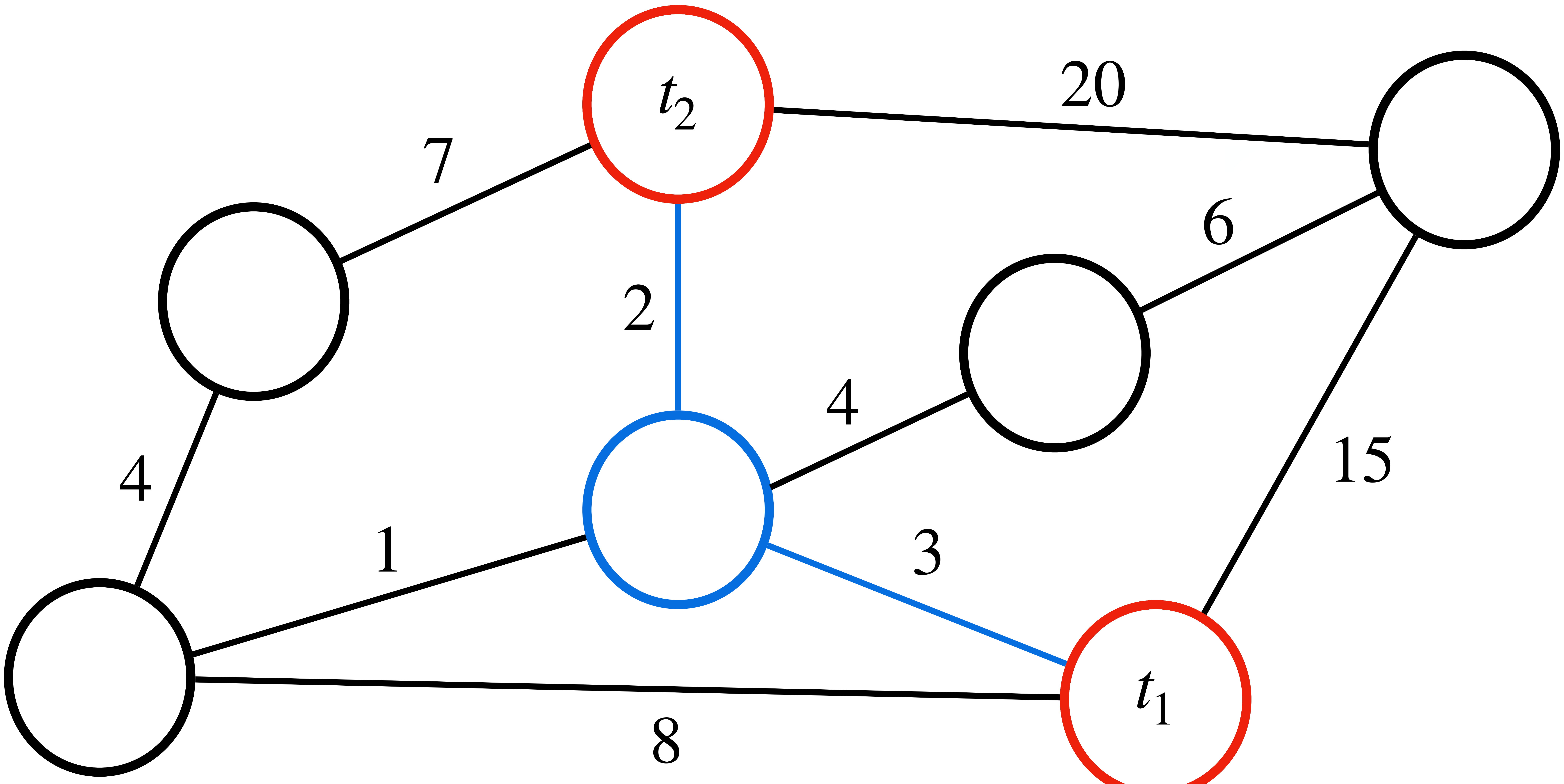
...and we're given terminals one-by-one?



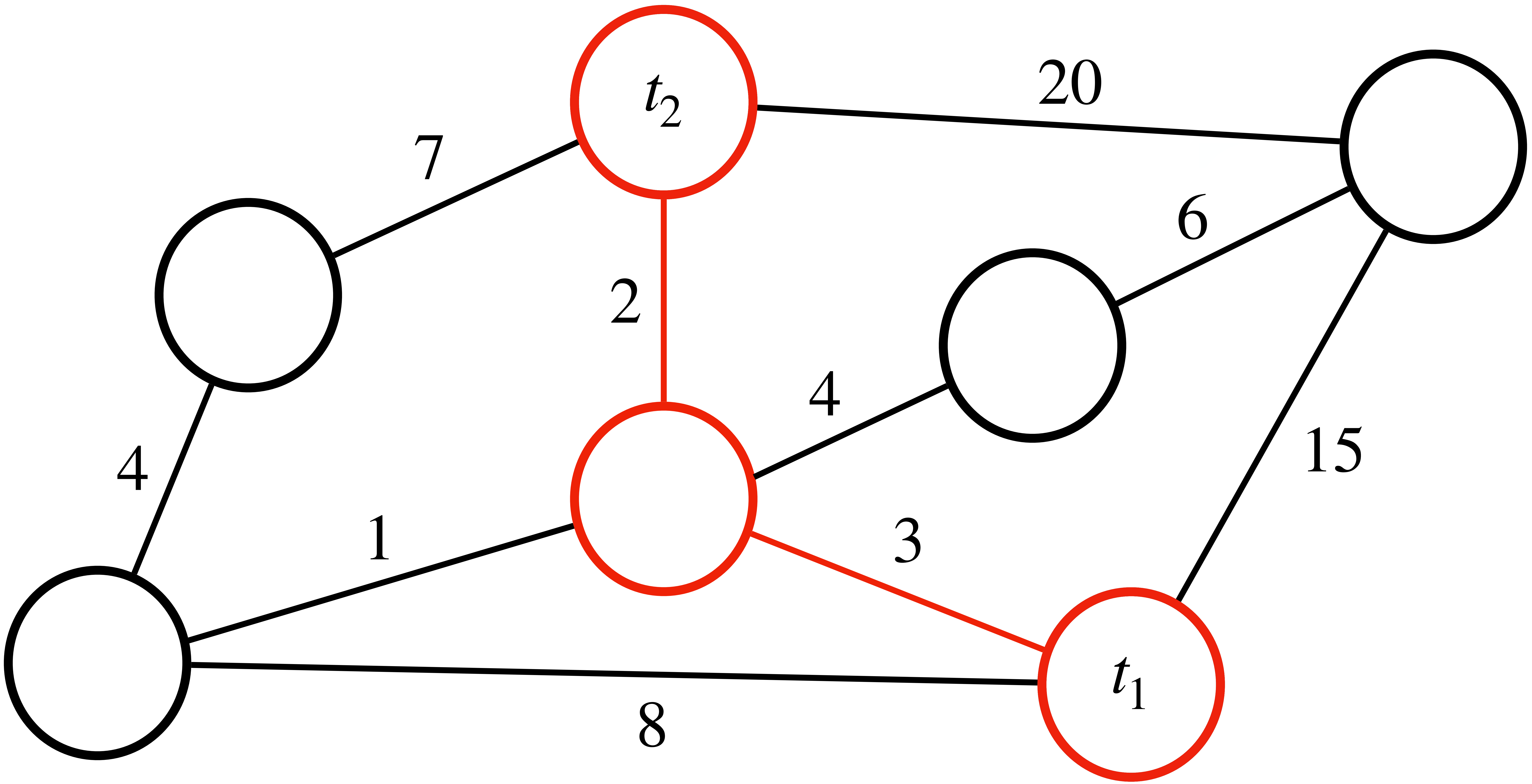
...and we're given terminals one-by-one?



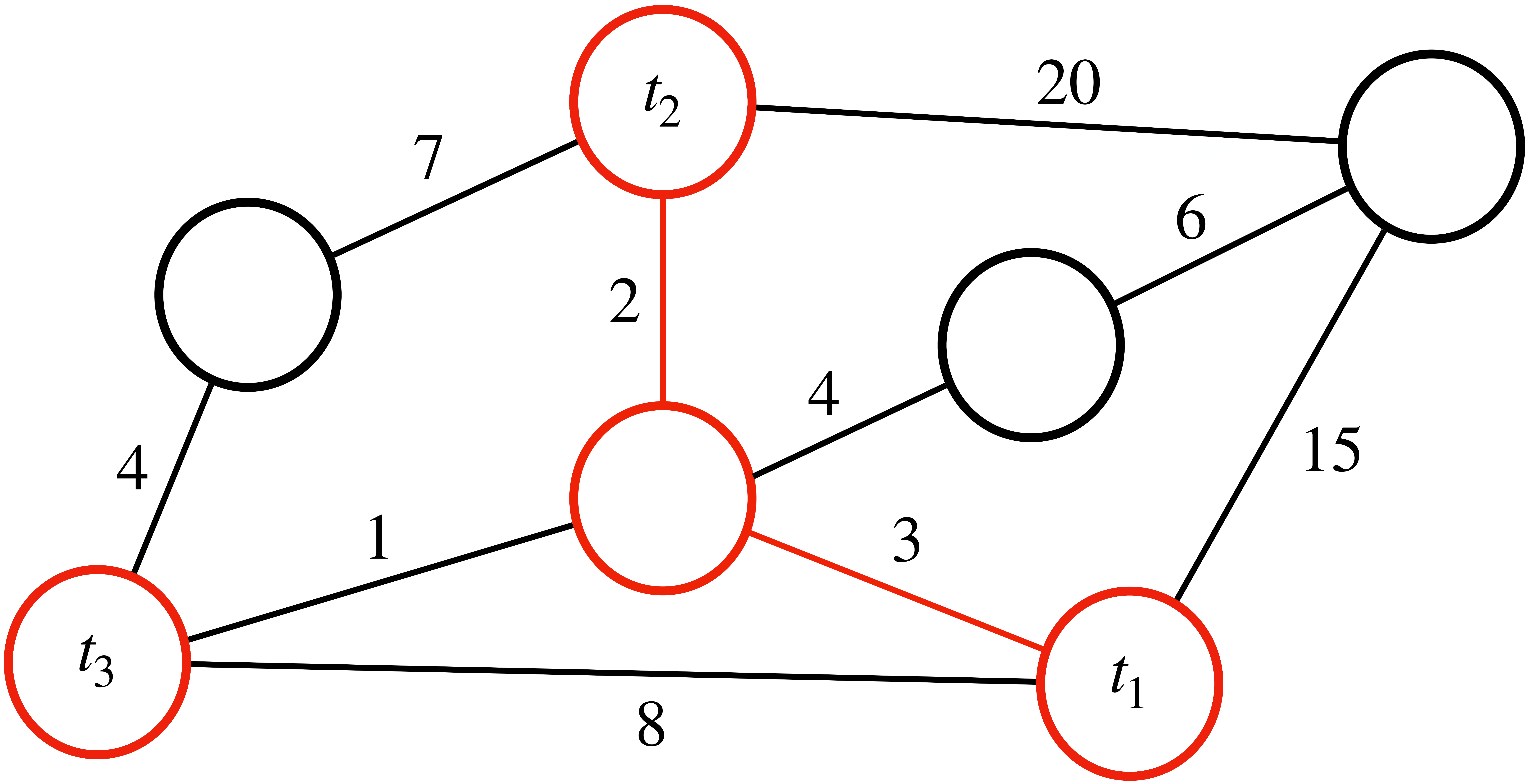
...and we're given terminals one-by-one?



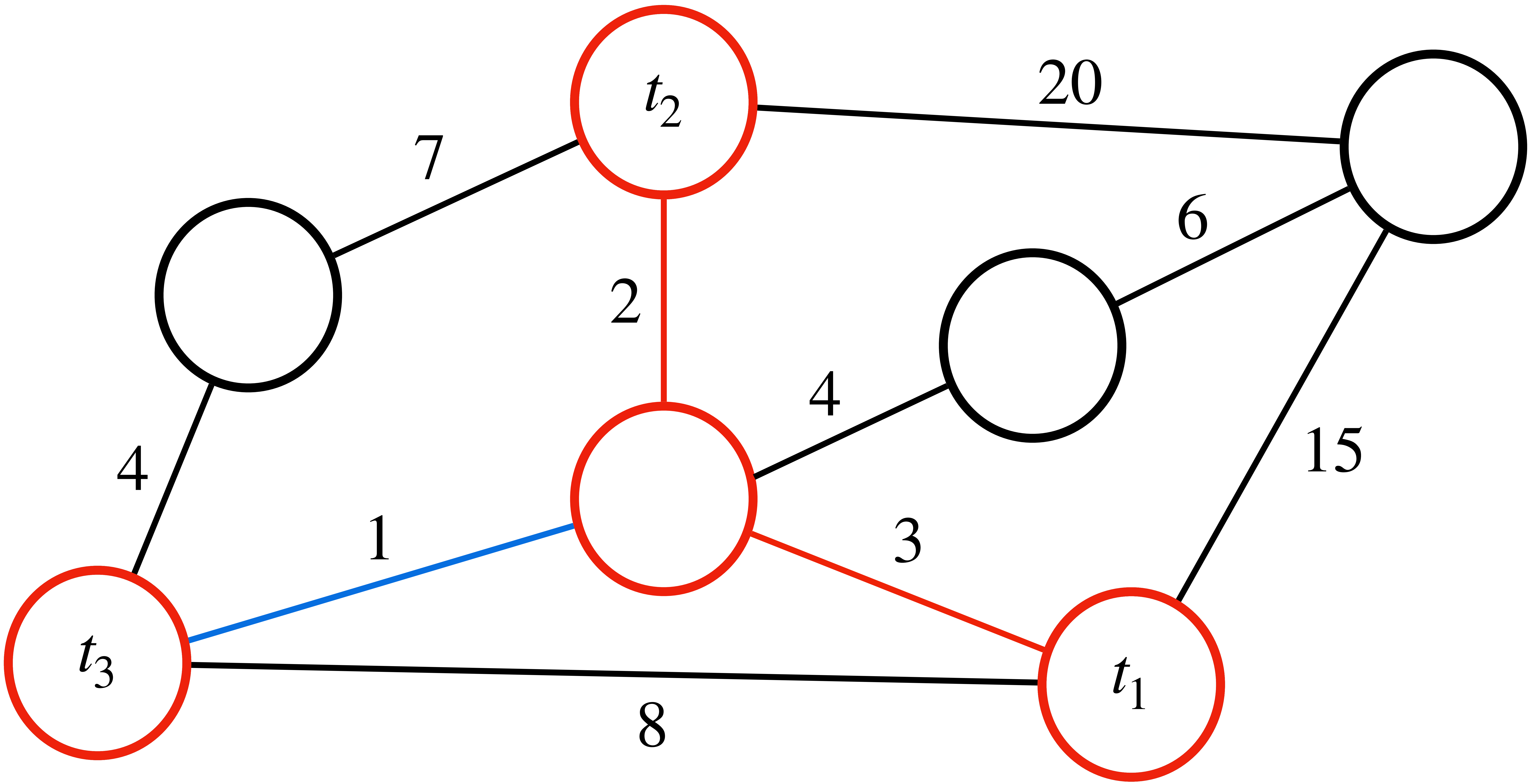
...and we're given terminals one-by-one?



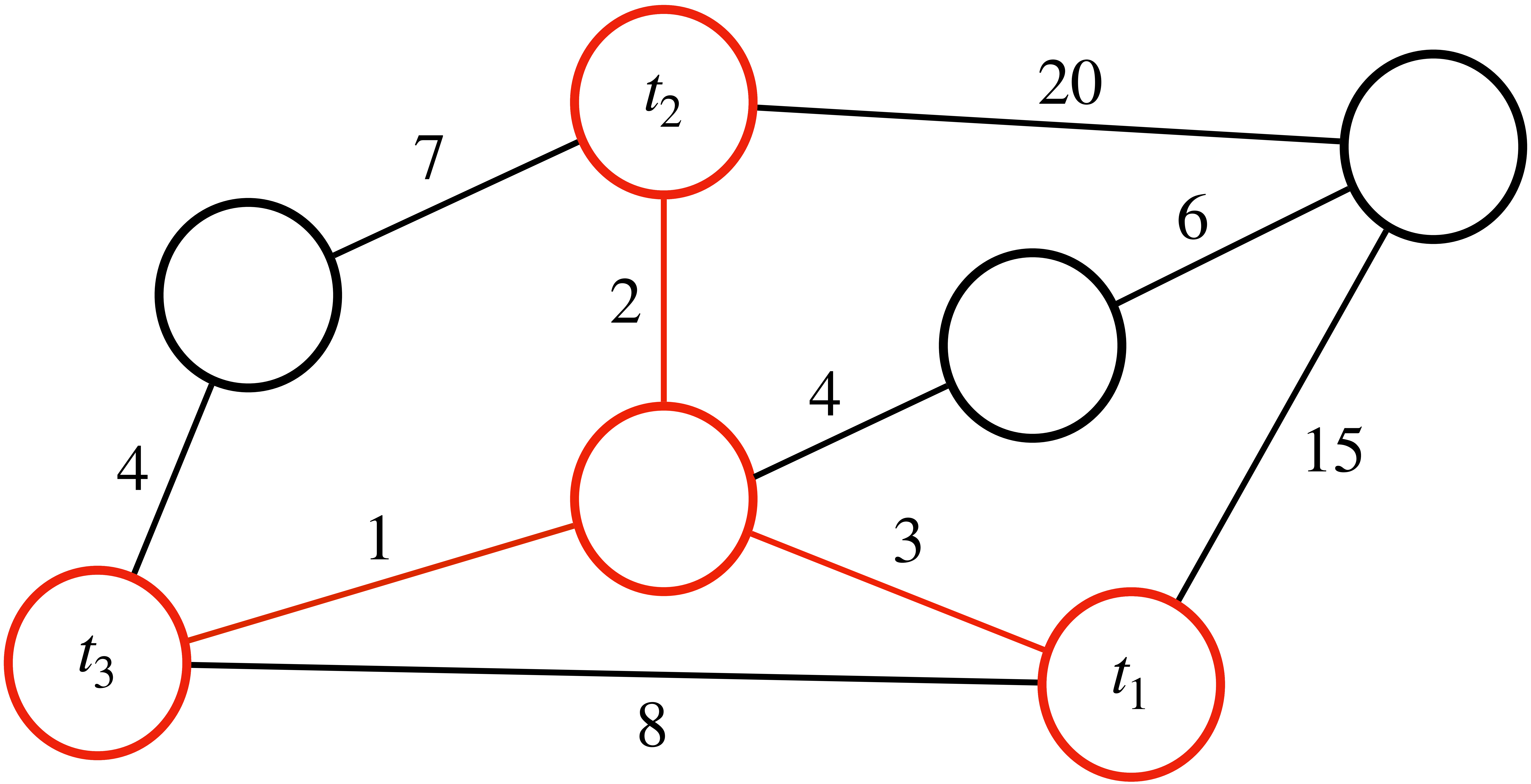
…and we're given terminals one-by-one?



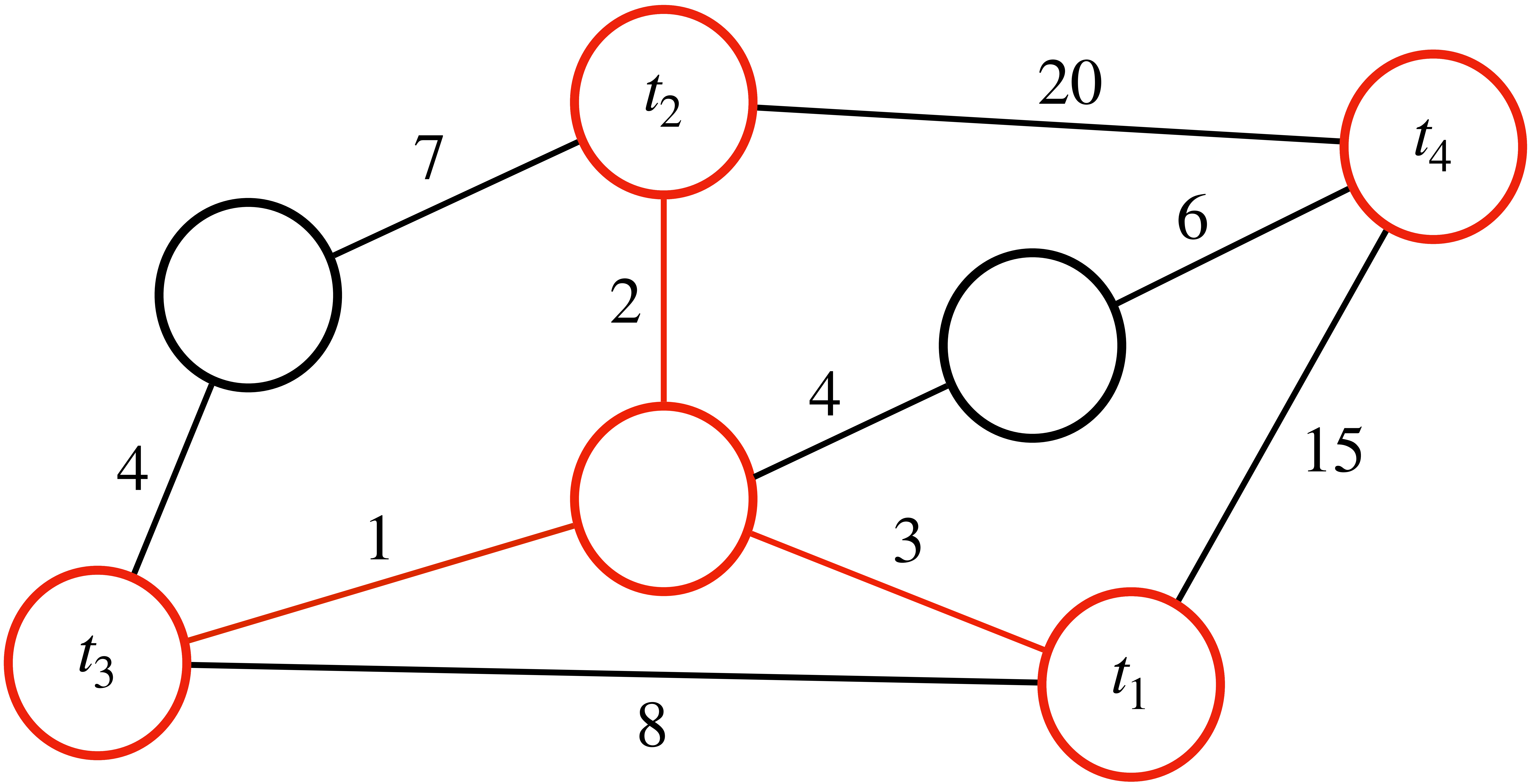
...and we're given terminals one-by-one?



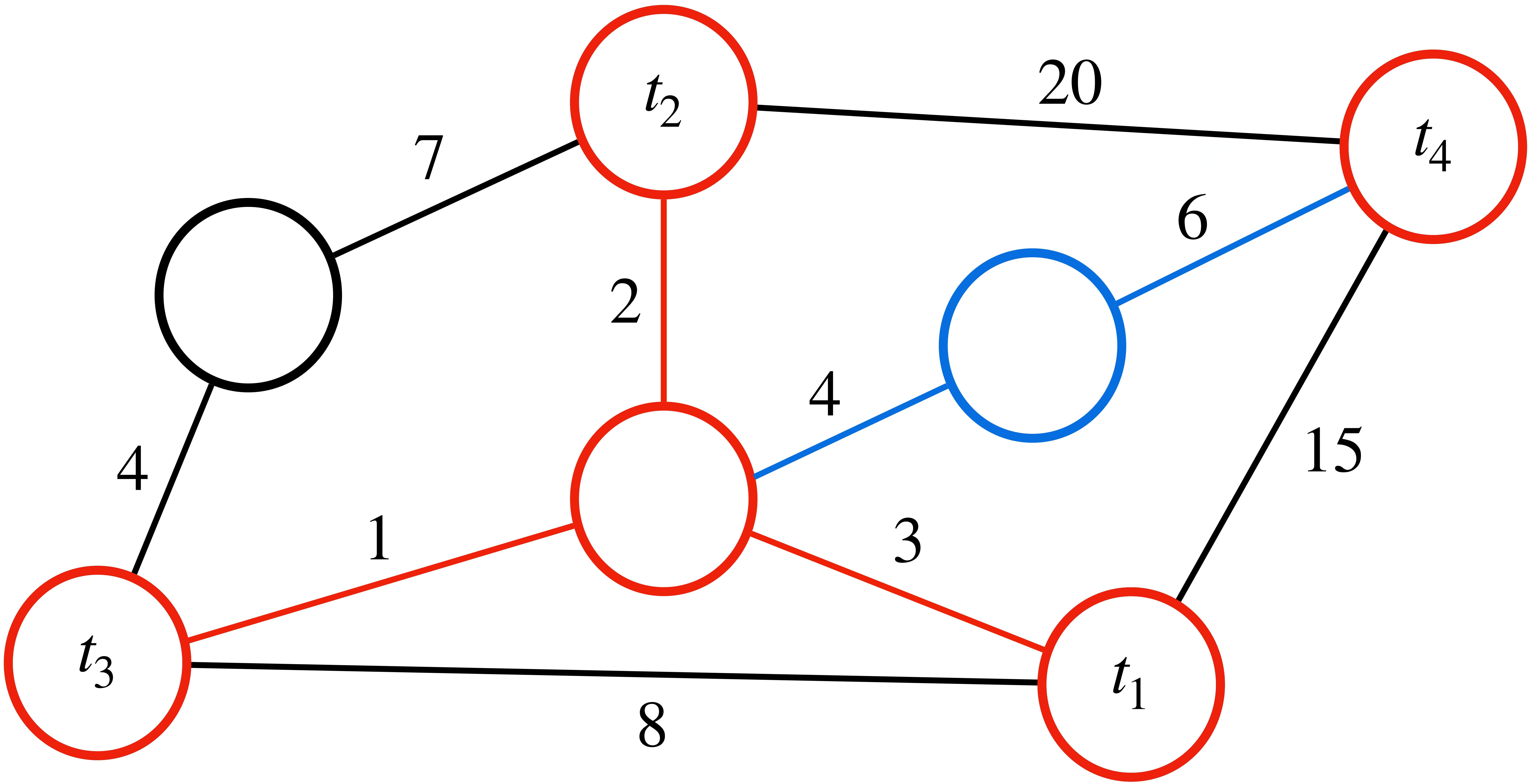
...and we're given terminals one-by-one?



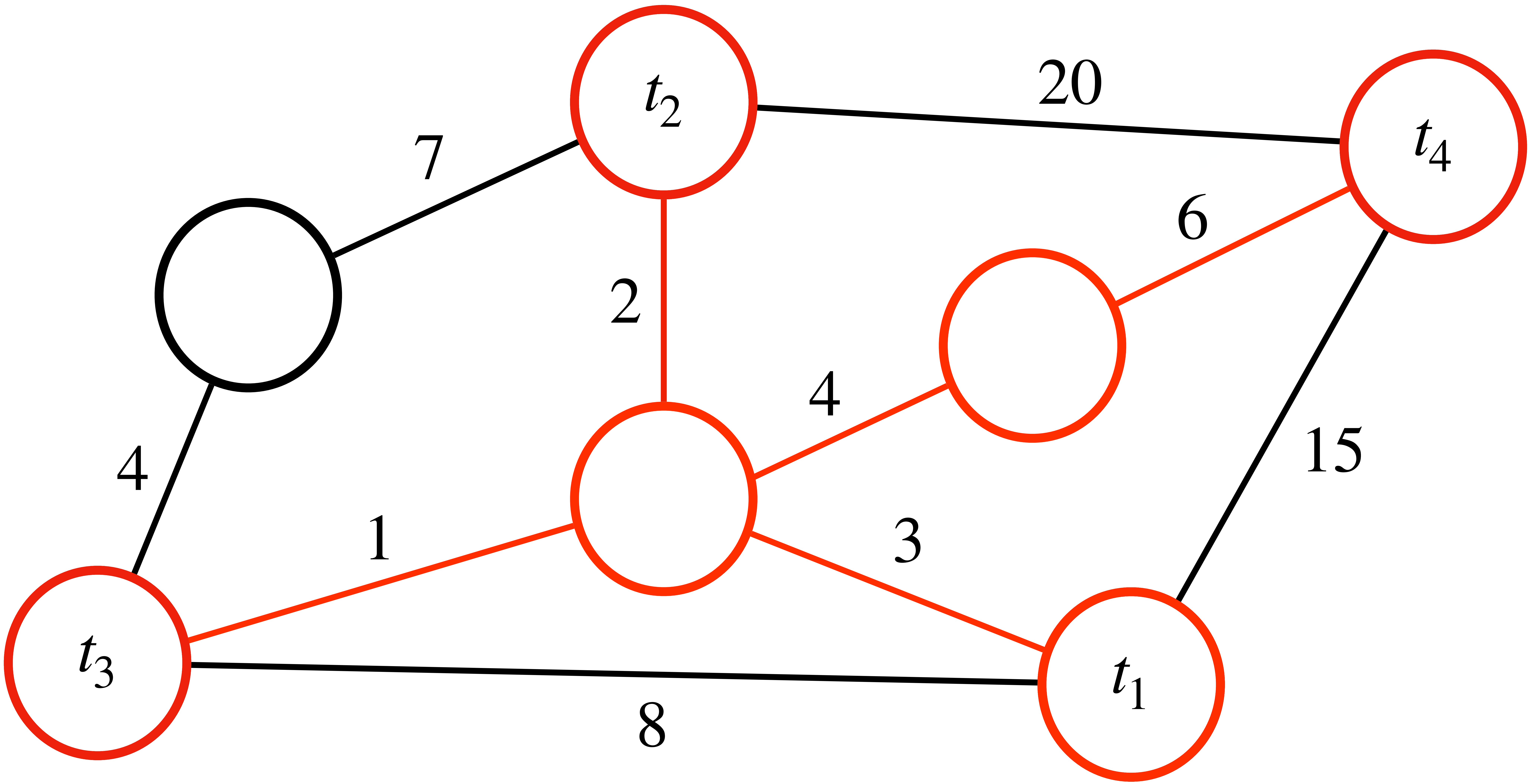
…and we're given terminals one-by-one?



…and we're given terminals one-by-one?

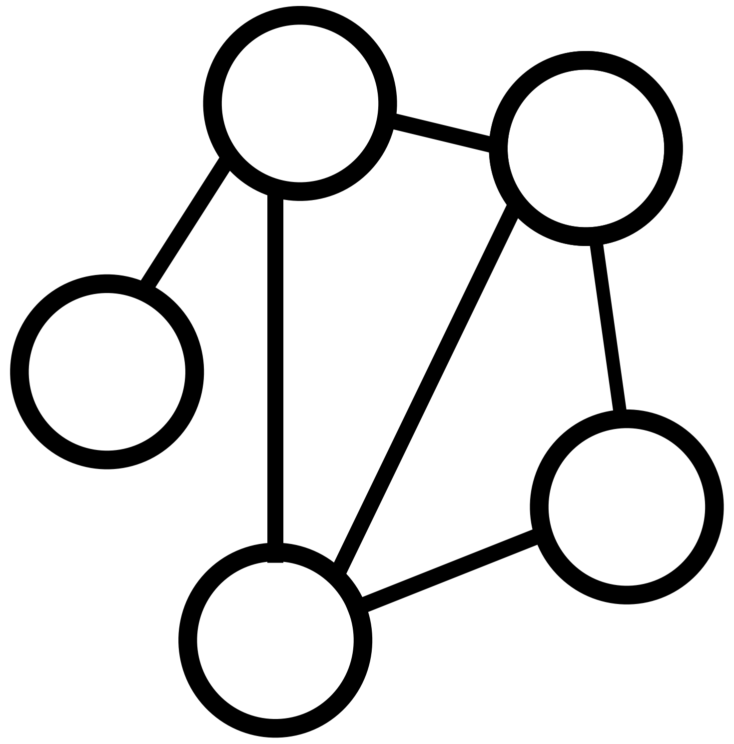


...and we're given terminals one-by-one?

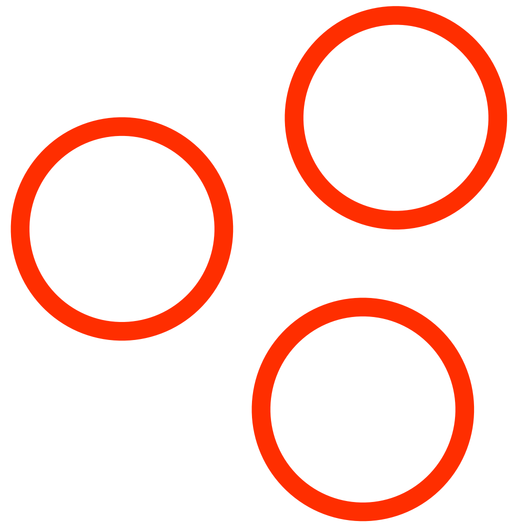


The Dynamic Steiner Tree (DST) Problem

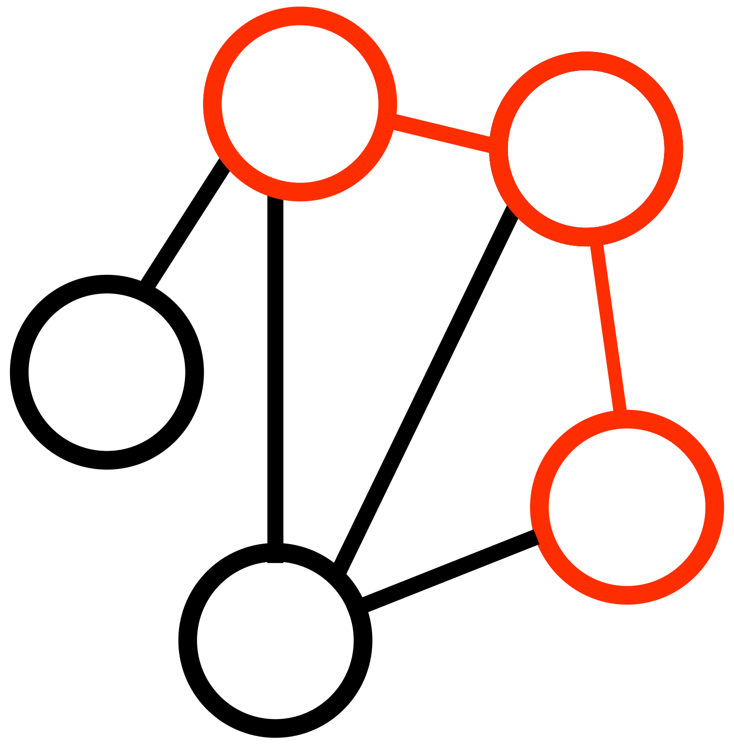
Given a graph $G = (V, E)$ with positive edge costs...



...and terminals $t_1, \ldots, t_j \in T$ arriving one-by-one,



maintain a subgraph T_i of G that contains all $t_i \in T$ and is close to the optimal Steiner tree after each new terminal.



Intuitively...

Let’s think about DSTs from the perspective of an adversary.

Intuitively...

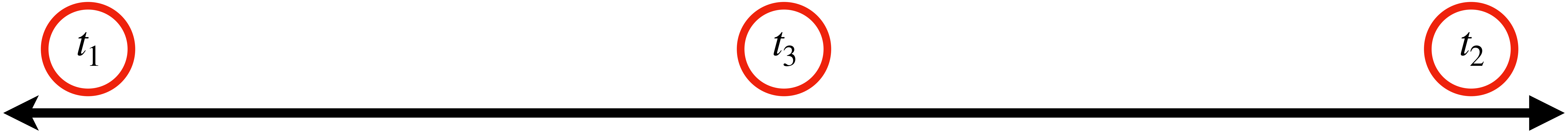
Let's think about DSTs from the perspective of an adversary.

Intuitively...

Let’s think about DSTs from the perspective of an adversary.

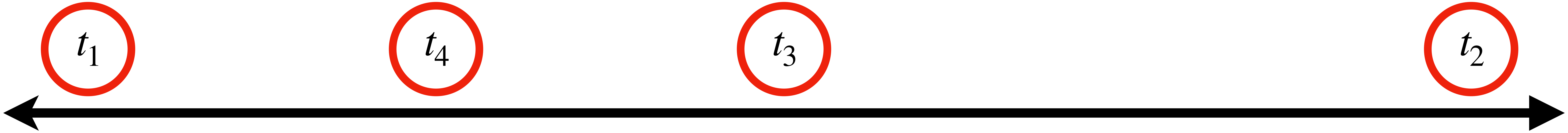
Intuitively…

Let's think about DSTs from the perspective of an adversary.



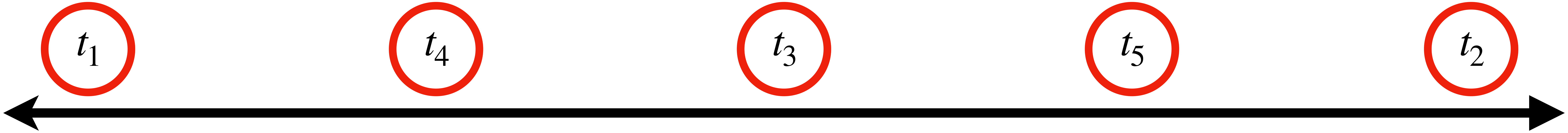
Intuitively…

Let's think about DSTs from the perspective of an adversary.



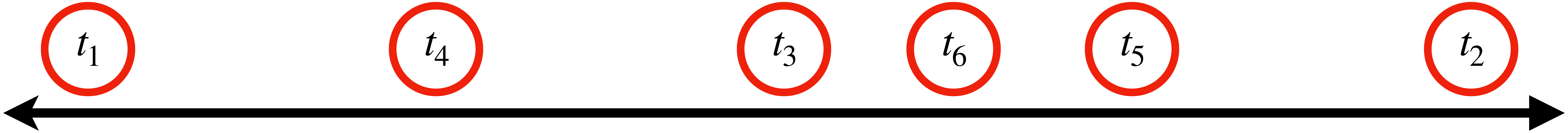
Intuitively...

Let’s think about DSTs from the perspective of an adversary.



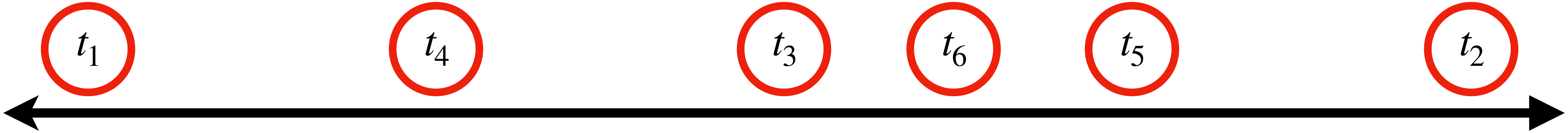
Intuitively...

Let’s think about DSTs from the perspective of an adversary.



Intuitively...

It's initially easy to place terminals far apart, but gets harder as more terminals arrive!



With many terminals, the cheapest path between a new terminal and any existing terminal probably won't be *that* expensive.

Imase & Waxman (1991)

Proposed a simple yet effective **greedy algorithm**:

> Add every new terminal by connecting it via the cheapest path to any of the existing terminals in the graph.

$2 \log k$ competitive, where k is the number of terminals

Key limitation: only really handles terminal addition

Our Work

VDSTGraph

Graph data structure that supports **five operations**:

Key idea: do some extra work here to make the fifth operation fast

1. Add node
2. Add edge (or decrease weight)
3. Add terminal (The only one of these that Imase & Waxman addressed!)
4. Remove terminal
5. Query Steiner tree

The *Metric* Steiner Tree Problem

"Metric" =

1. Graph is complete
2. Edge weights satisfy Triangle Inequality

Theorem: Any α-competitive algorithm for the **metric Steiner tree problem** is also α-competitive for the general Steiner tree problem.

How? Find the Steiner tree on the **metric closure**, then expand edges into full paths and break cycles.

How `VDSTGraph` Works

Intuition: Easy to handle graph updates in metric problem, but not so much in the general case.

So, we efficiently maintain the **metric closure** and **its Steiner tree** and lazily generalize the results to our original graph at query time.

To keep our data structure $2 \log k$ competitive, we maintain the **"favorite parent invariant"** during graph updates.

Convention: n = # of nodes, m = # of edges, k = # of terminals

`VDSTGraph.`**`add_node`**

Easy — node is initially disconnected from others and not a terminal.

1. Add node to graph and metric closure
2. Create direct path to all other nodes in metric closure with cost ∞

Time complexity: $O(n)$

`VDSTGraph.`**`decrease_edge_weight`**

Harder — might introduce many shortest paths around the graph.

1. Add this edge to metric closure
2. Using DP, solve *All Pairs Shortest Path* problem to “propagate” effect in metric closure
3. Update metric Steiner tree each time whenever we find a shorter path between two terminals

Time complexity: $O(n^2)$

`VDSTGraph.`**`add_terminal`**

Easy — just do what Imase & Waxman did!

1. Add node to metric Steiner tree
2. Using metric closure, find and connect to closest terminal

Time complexity: $O(k)$

VDSTGraph.**remove_terminal**

Harder — might cause other terminals to be orphaned.

1. Remove node from metric Steiner tree
2. For each child, reattach with `add_terminal` (respect timestamps!)

Time complexity: $O(k^2)$

VDSTGraph.**get_steiner_tree**

Easy — just rebuild general Steiner tree from metric one.

1. Expand paths in metric Steiner tree, adding edges to priority queue.
2. Run Kruskal's algorithm to find lowest-cost way to break cycles.

Time complexity: $O(m \log n)$

Cache queries to avoid recomputation if nothing changes.

Results

Evaluation Pipeline

Baseline: `NetworkX` Python package

`steiner_tree`(*G, terminal_nodes, weight='weight'*) [source]

Return an approximation to the minimum Steiner tree of a graph.

($2 - (2/k)$ competitive)

1. Build initial graph with 300 initial nodes and edge density ρ
2. Test both methods on randomized workflow of 500 operations

Add terminal

(first 40 operations)

then...

Query Steiner tree

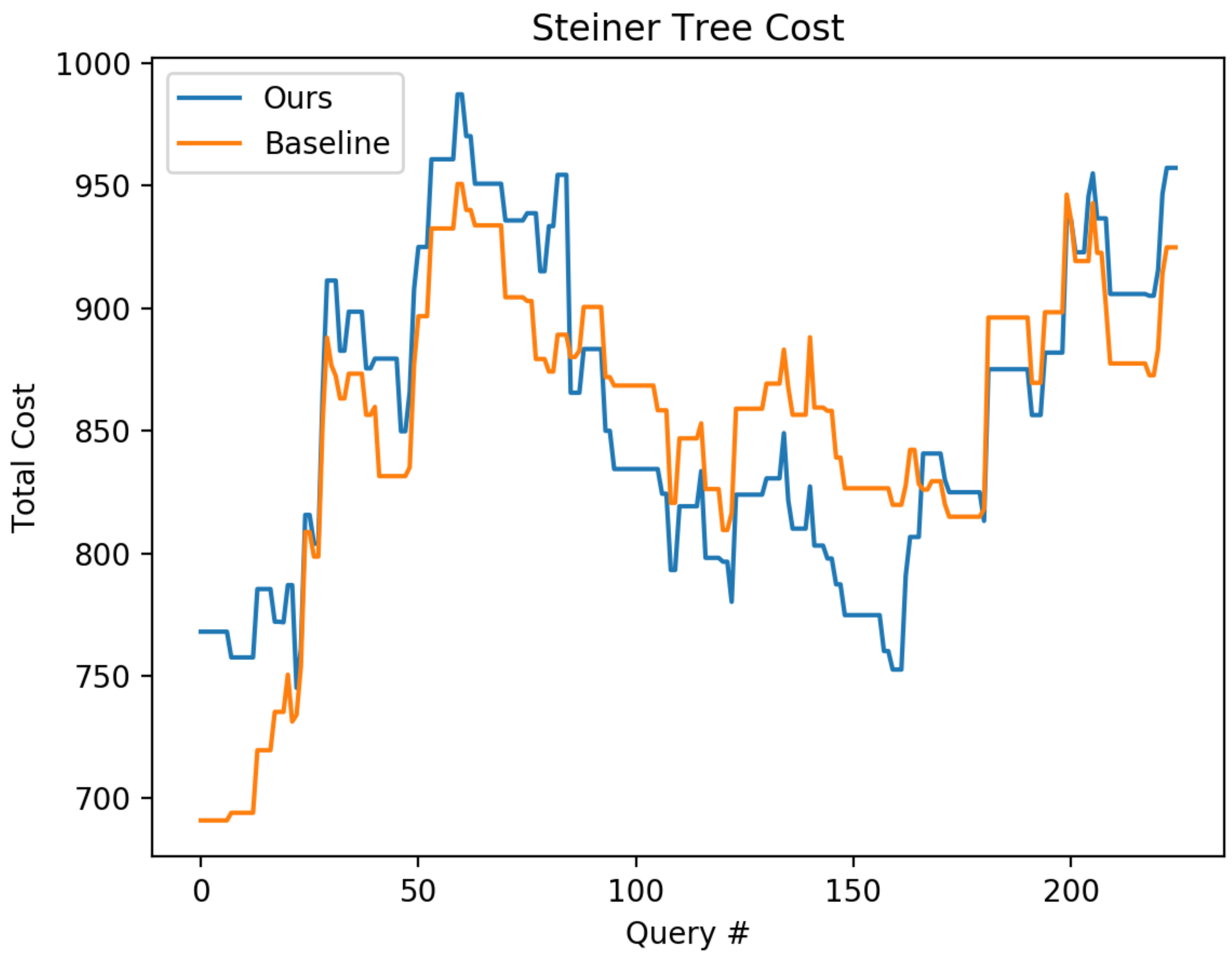
(with probability p)

Add terminal | **Add node + edges**

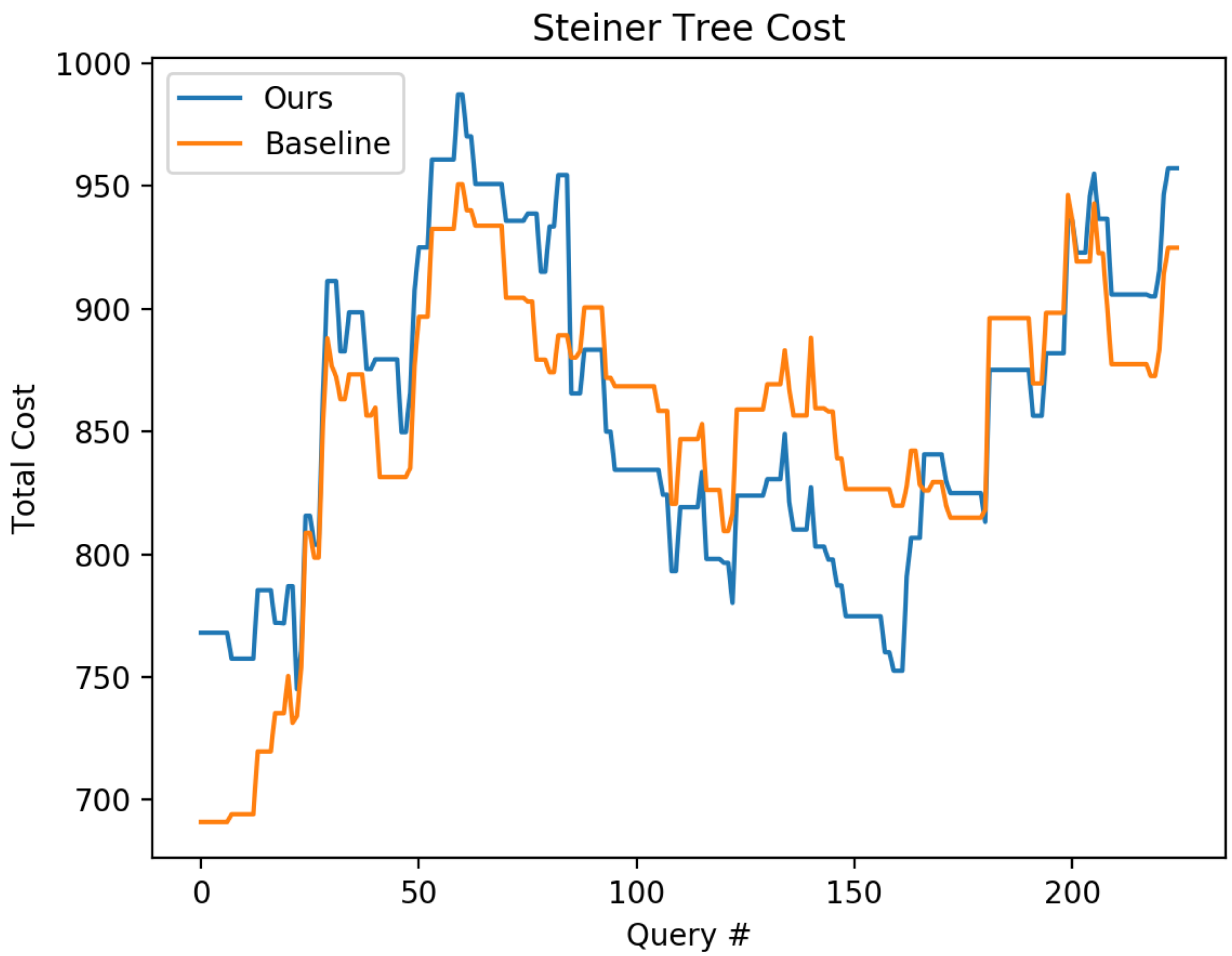
Remove terminal | **Decrease edge weight**

(with equal probability if we don't query)

Q: Does it work?



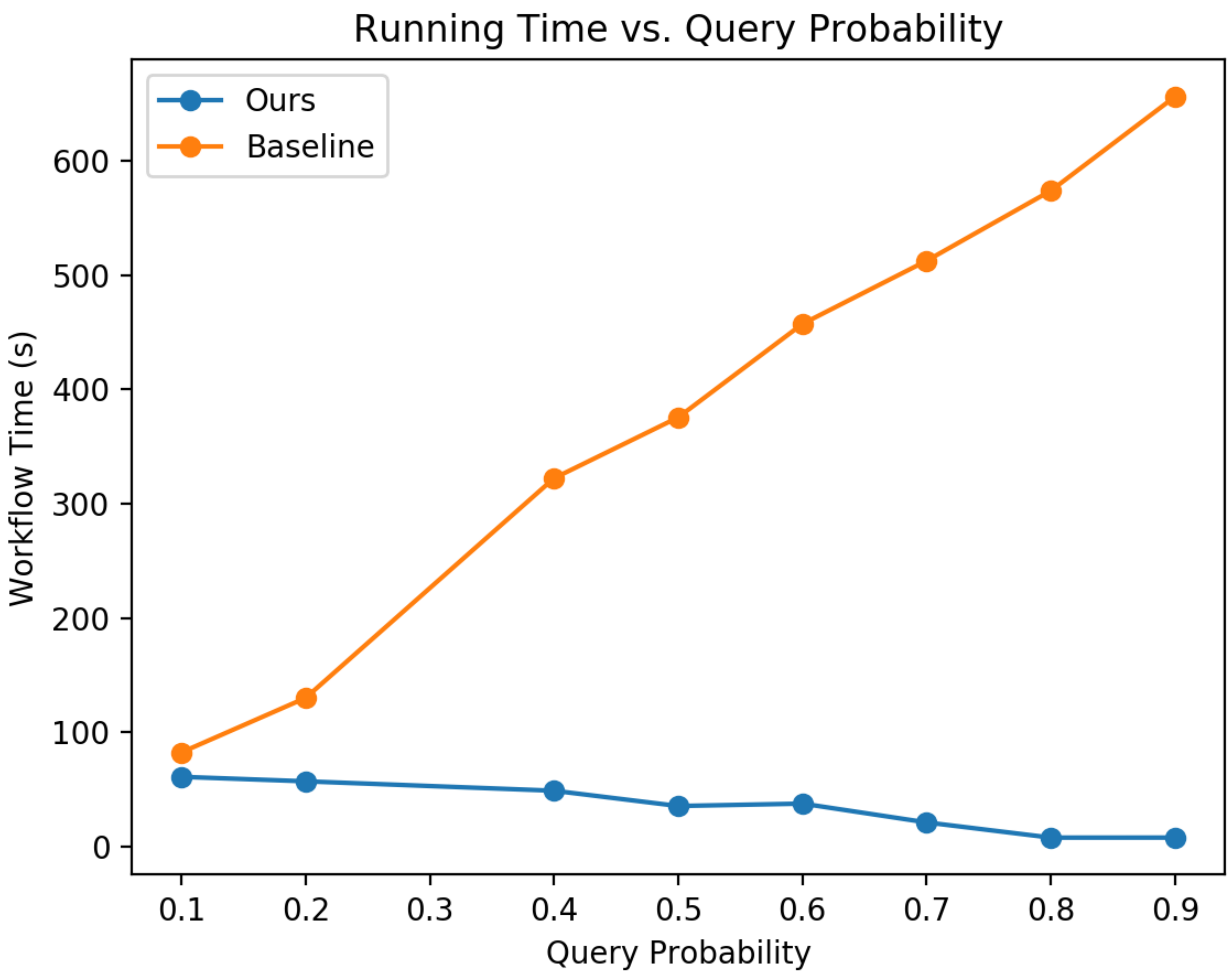
A: Yes!

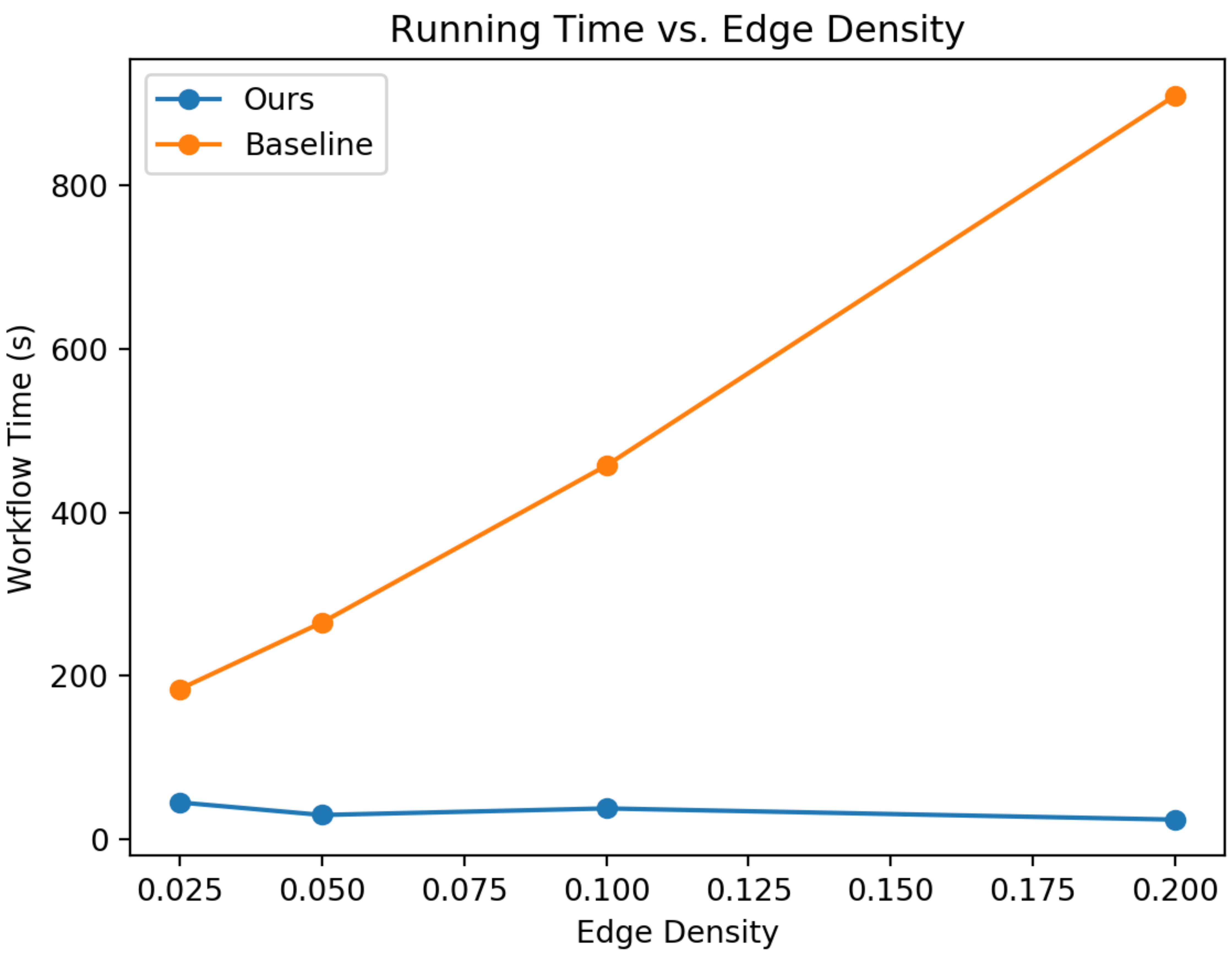


No noticeable reduction in quality compared to baseline (a more accurate approximation)!

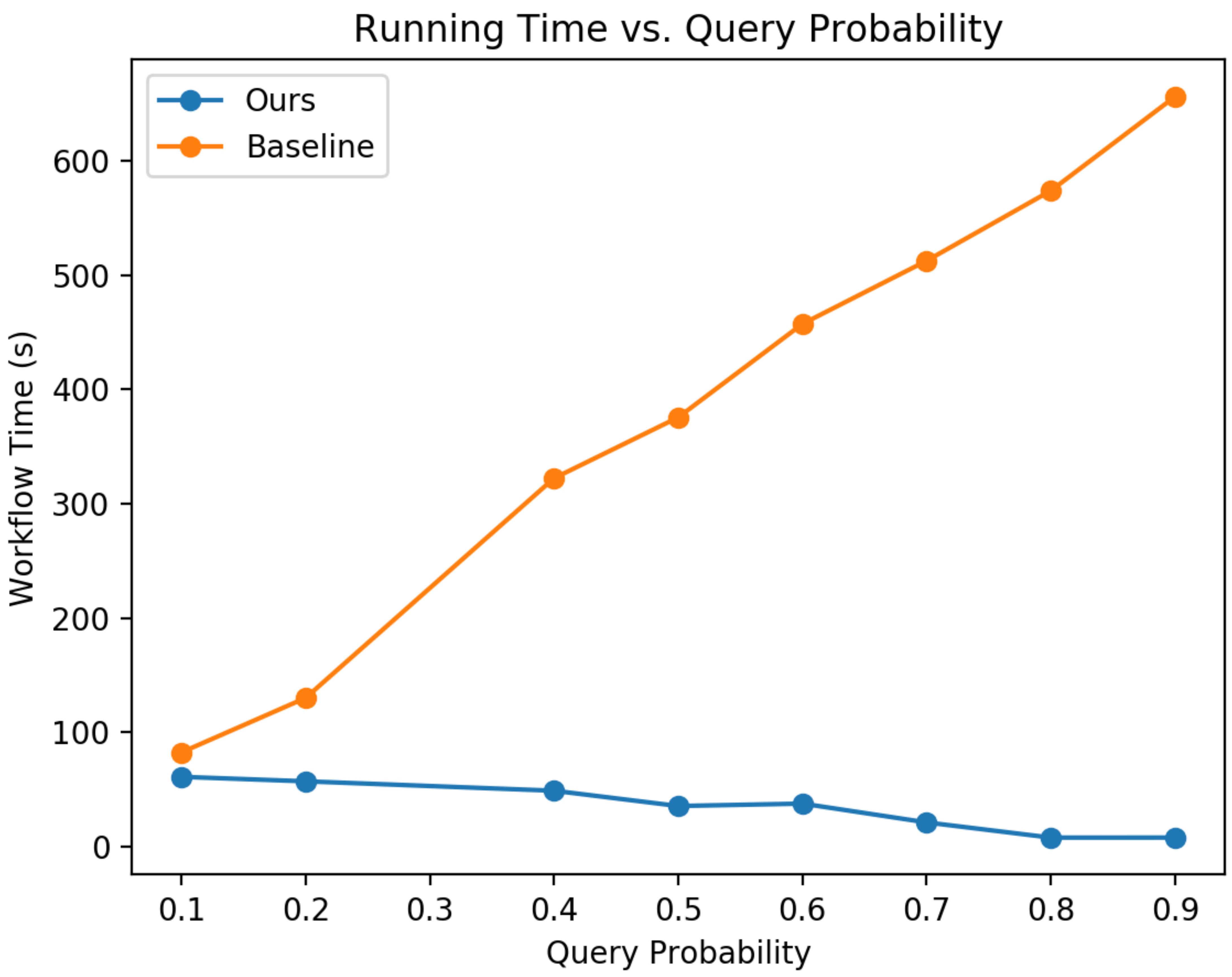
On average, returned Steiner trees with **1.049** times cost of those returned by baseline.

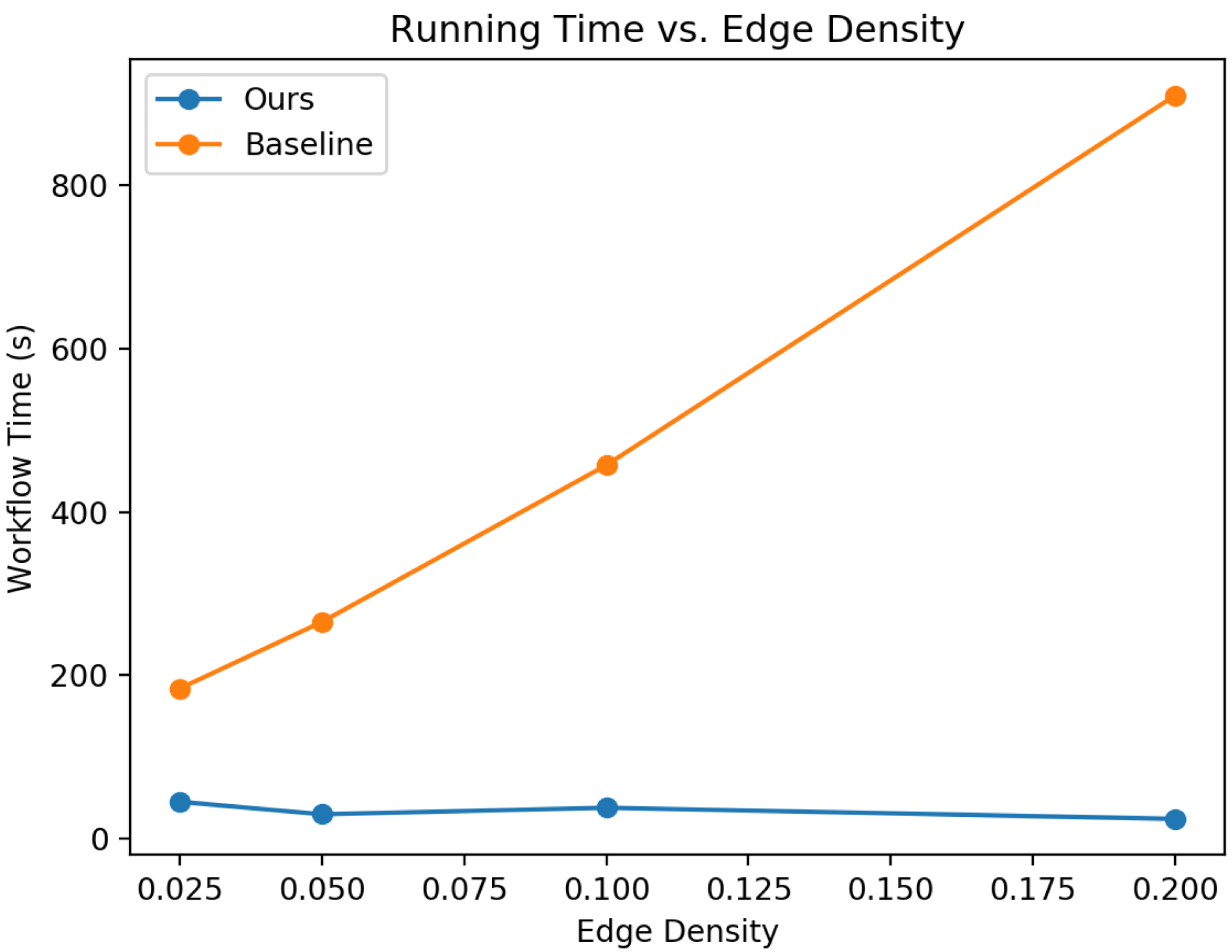
Q: Is it fast?


Running Time vs. Query Probability
Ours
Baseline
Workflow Time (s)
0
100
200
300
400
500
600
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Query Probability


Running Time vs. Edge Density
Ours
Baseline
Workflow Time (s)
0
200
400
600
800
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
Edge Density

A: Extremely!



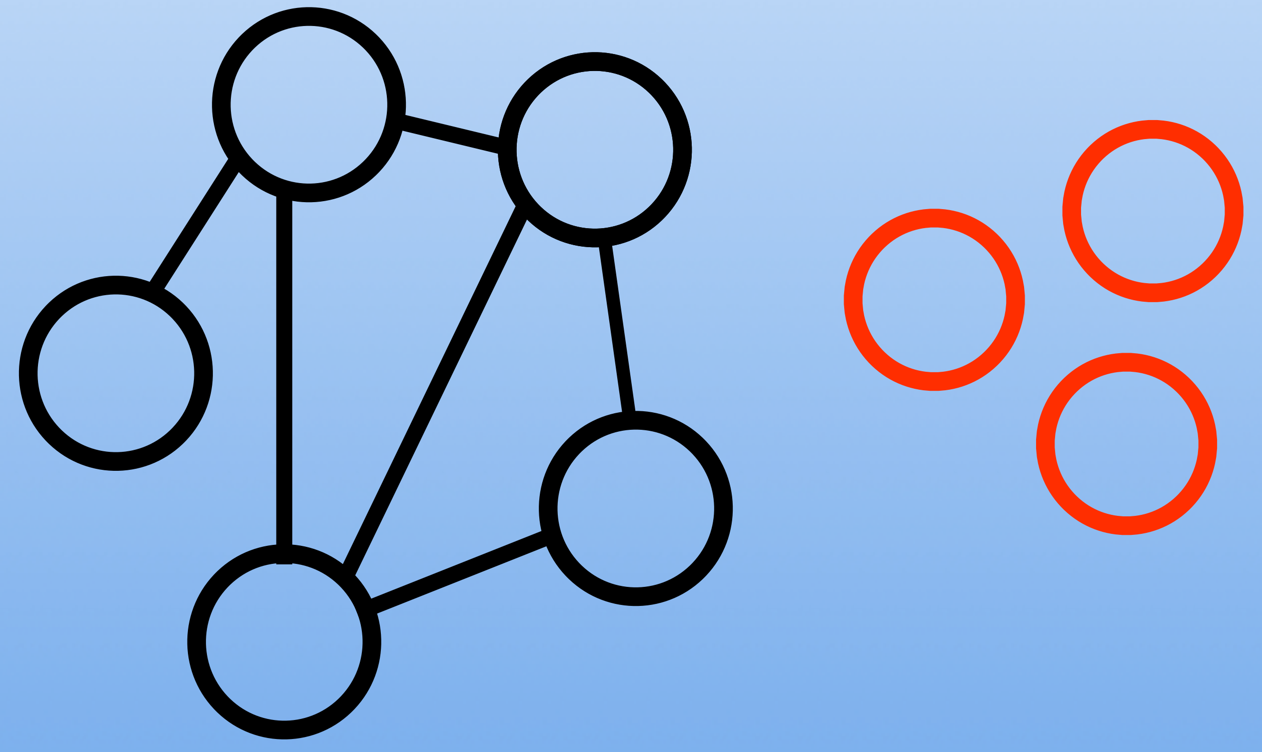


Faster as workflow includes more queries!

Scales extremely well to dense graphs.

Who cares?

Generalizing graph algorithms to the dynamic case is non-trivial



Theory

DSTs used in dynamic routing (Uber), large-scale distribution (Amazon), etc.

Practice

Future Work

1. Generalize metric Steiner tree without *de novo* recomptuation at query time (path expansion, Kruskal's)
2. Handle node and edge deletion
3. Handle batch insertions/deletions
4. Evaluate on real-world datasets (Oregon Networking or U.S. Road Networks)

Conclusion

To our knowledge, first to address graph modifications in the DST problem.

Algorithm handles dynamic modifications to terminal nodes, graph nodes, and edges.

No loss in accuracy, but significantly faster than existing algorithms.

Scales extremely well to dense graphs/lots of queries by **reusing more and more work**.