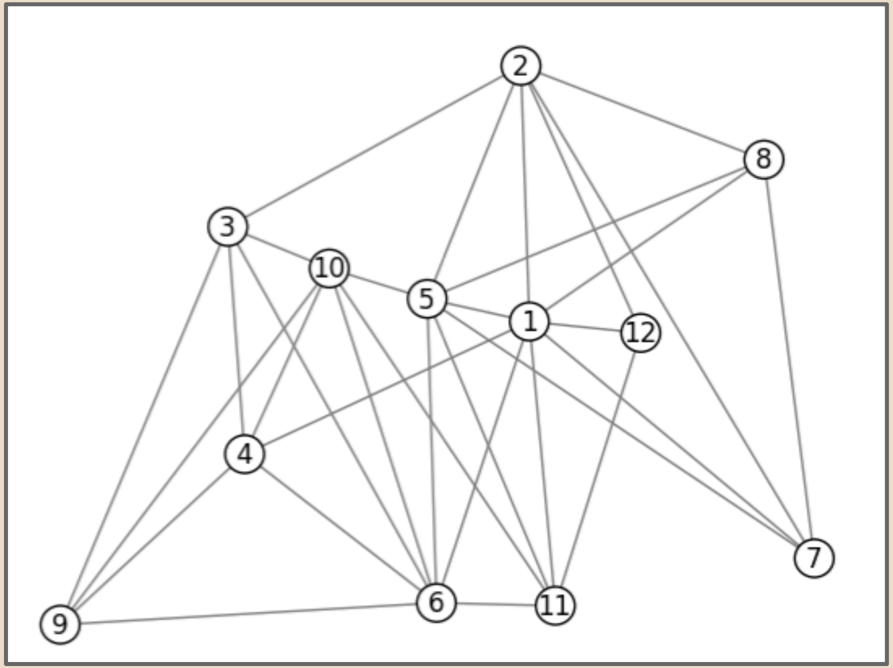
Time-Constrained Flooding

A. Mehta and E. Wagner

Time-Constrained Flooding: Problem Definition

- Devise an algorithm that provides a subgraph containing all possible paths from source to destination with **total edge latency** at most *L*, whose **nodes** are all on some simple path from source to destination.



An example input.

Two-Step Algorithm

- **Phase 1:** Eliminate all edges that are not on paths within maximum total latency *L*.
- **Phase 2:** Remove all nodes that are not on at least one simple path from source to destination within the time constraint.
- Why do we only care about nodes on simple paths?
    - A given node will only send packets once. If it receives the same packet twice, it will not resend it. Since non-simple paths reuse a node, they do not add reliability.

Algorithm - Phase 1

- **Step 1:** Run Dijkstra’s algorithm once from the source and once from the destination to get the shortest distance to the source (d_s) and to the destination (d_d) for each node in the graph.

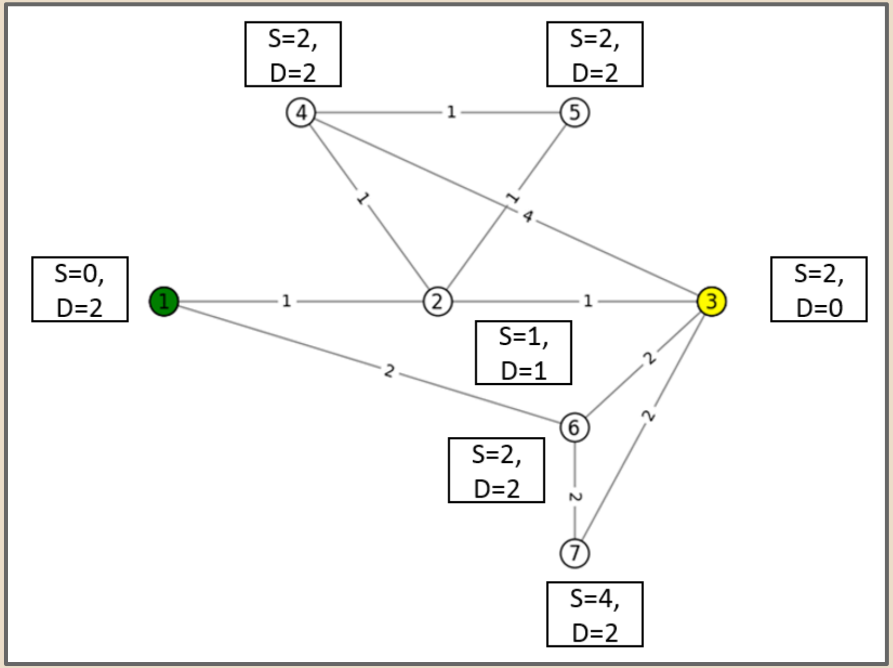


Fig : After Step 1.

Algorithm - Phase 1

- Step 2: For each edge *e* in the graph check if the following condition is true:

$$d_s(e.head) + latency(e) + d_d(e.tail) \leq L$$

- If so, the edge is included. Otherwise, it is not.

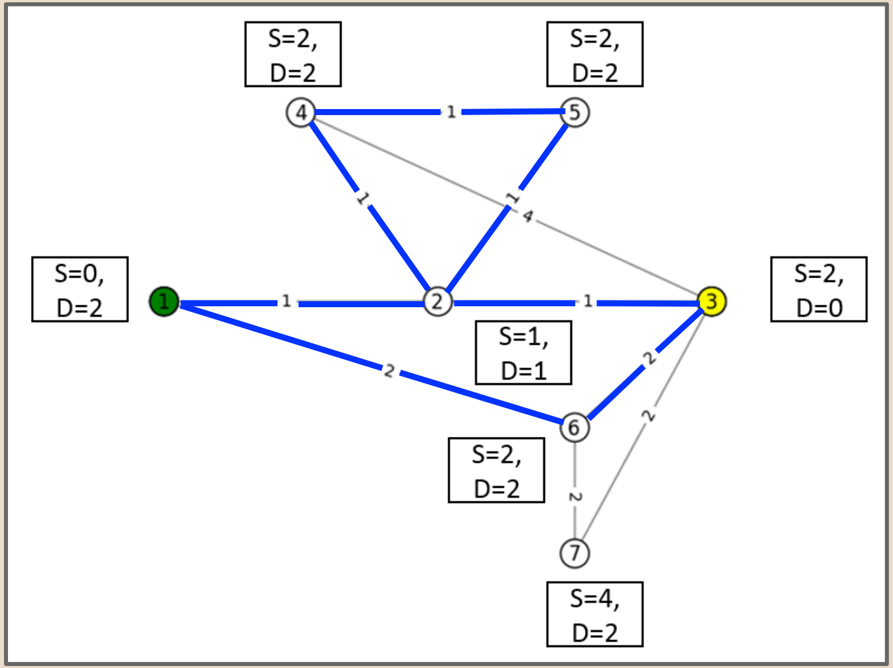


Fig : Output after Step 2 for Budget = 5.

Algorithm

- Now we have all edges that meet our time constraint.
- However, these may include cycles that do not increase reliability.

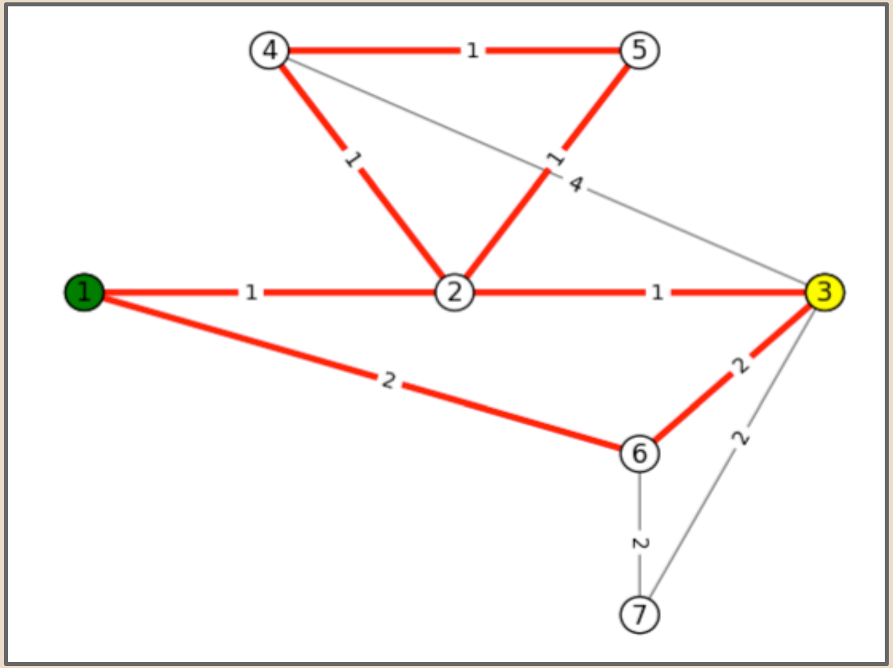


Fig : Output after Steps 1 and 2 includes a cycle.

Algorithm - Phase 2

- **Step 1:** Add a dummy node (*D*) to the graph with zero-latency edges to the source and destination. Add all the nodes in the graph except the source and the destination to a list of nodes called *list_eval*.

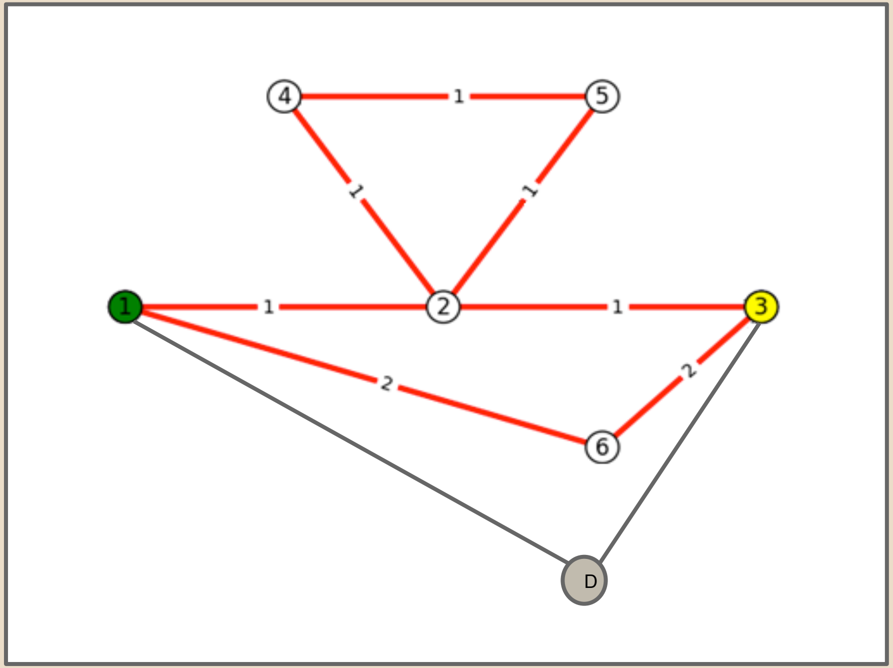


Fig : Output after Steps 1 and 2 includes a cycle.

Algorithm - Phase 2

- **Step 2:** For each node*(n)* in *list_eval*, use Suurballe's algorithm to find 2 node-disjoint paths whose total latency is minimized from *n* to *d*. 2 such paths may not exist.

- If 2 node disjoint paths do not exist, remove the node from the final graph and continue from **Step 2.** Otherwise go to **Step 3.**

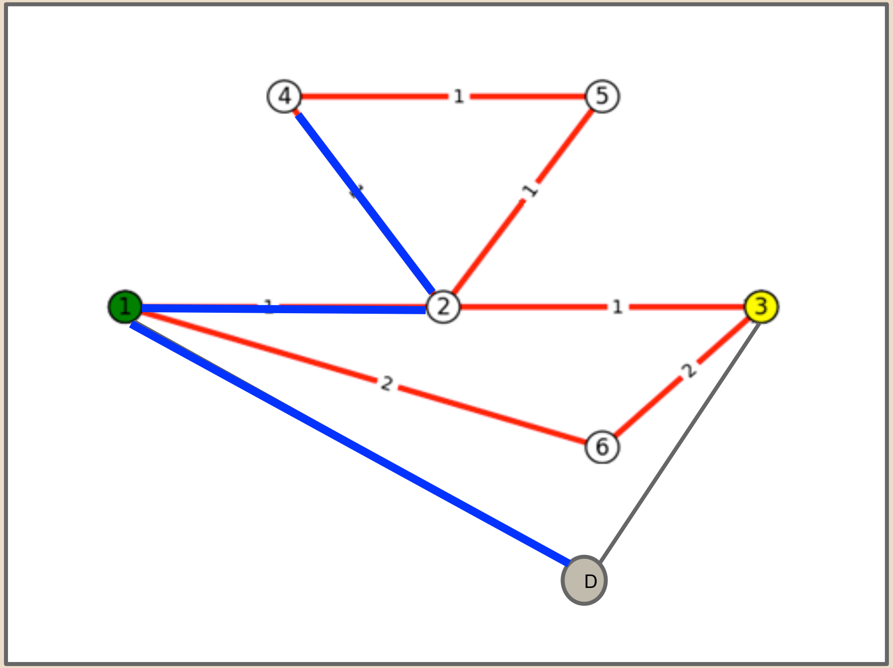


Fig : Step 2 for node 4.

Algorithm - Phase 2

- **Step 2:** For each node*(n)* in *list_eval*, use Suurballe's algorithm to find 2 node-disjoint paths whose total latency is minimized from *n* to *d*. 2 such paths may not exist.
- If 2 node disjoint paths do not exist, remove the node from the final graph and continue from **Step 2.** Otherwise go to **Step 3.**

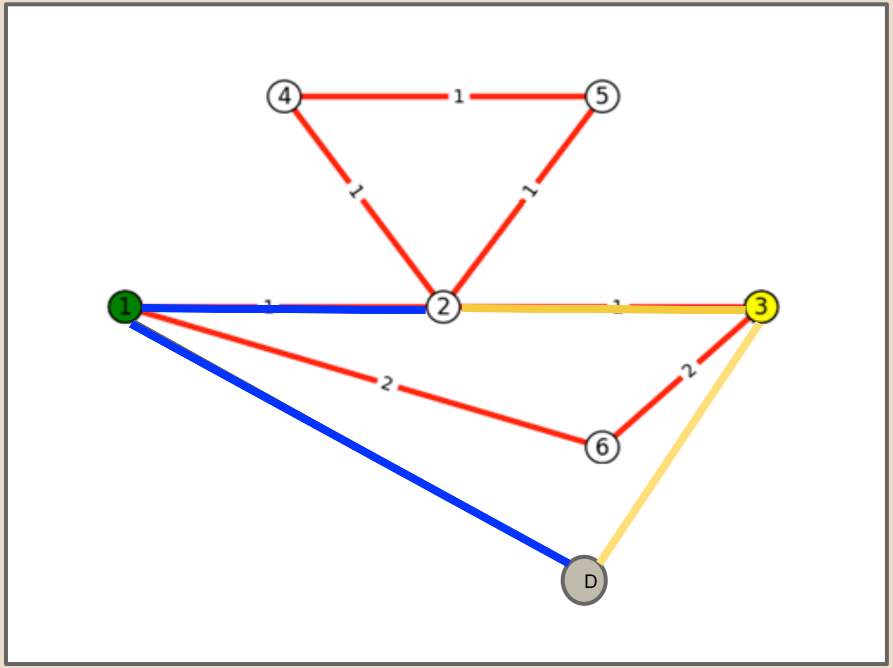


Fig : Step 2 for node 2.

Algorithm - Phase 2

- **Step 3:** Check if the total latency of the paths obtained in **Step 2**, which is necessarily minimal, is within the time constraint.
- If so, remove all the nodes on either path from *list_eval*, as these are all known to be on a valid simple path from source to destination. Otherwise, remove the node from the final graph. Repeat from **Step 2** until *list_eval* is empty.

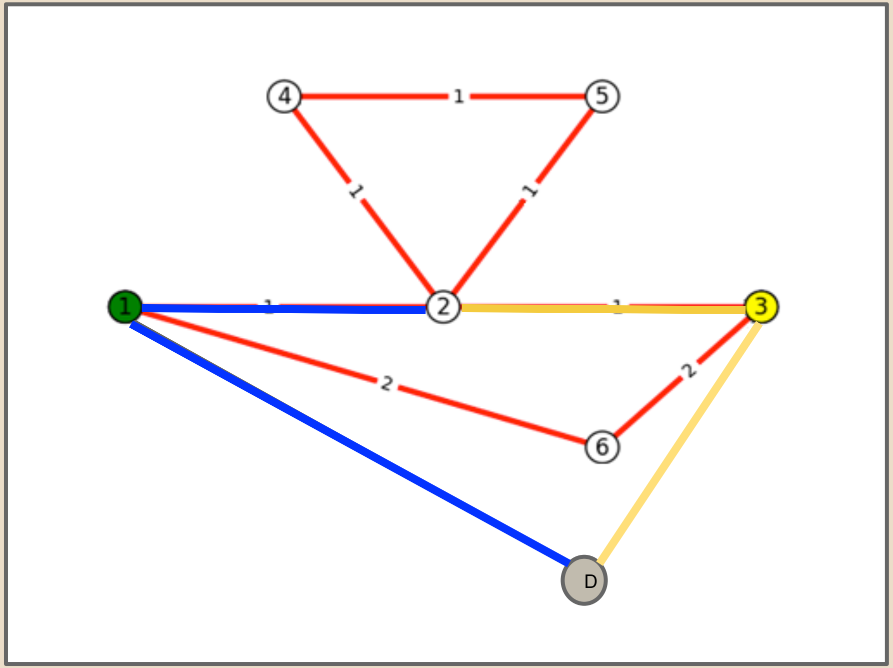


Fig : Step 2 for node 2.

Algorithm - Phase 2

- **Step 3:** Check if the total latency of the paths obtained in **Step 2**, which is necessarily minimal, is within the time constraint.
- If so, remove all the nodes on either path from *list_eval*, as these are all known to be on a valid simple path from source to destination. Otherwise, remove the node from the final graph. Repeat from **Step 2** until *list_eval* is empty.

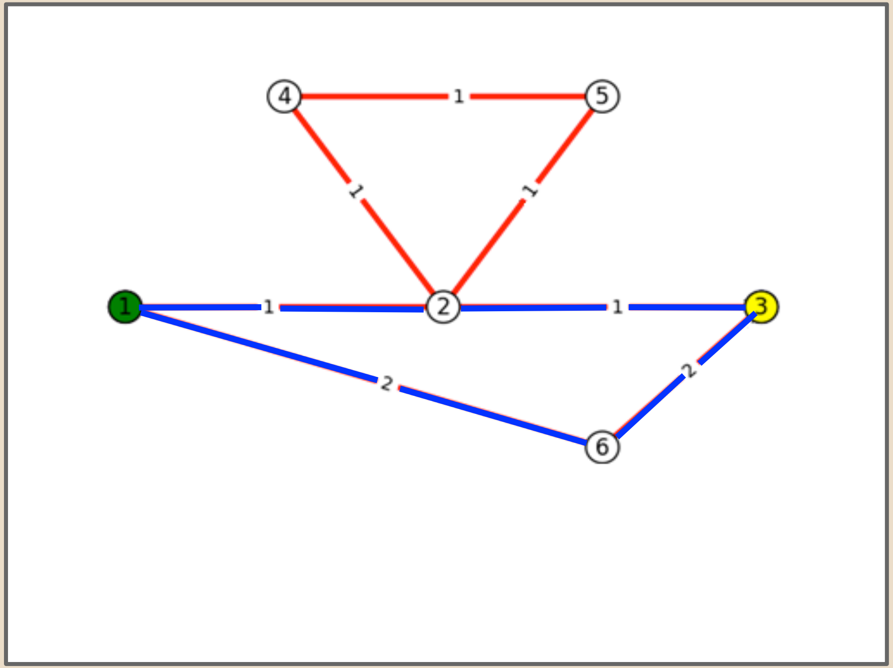
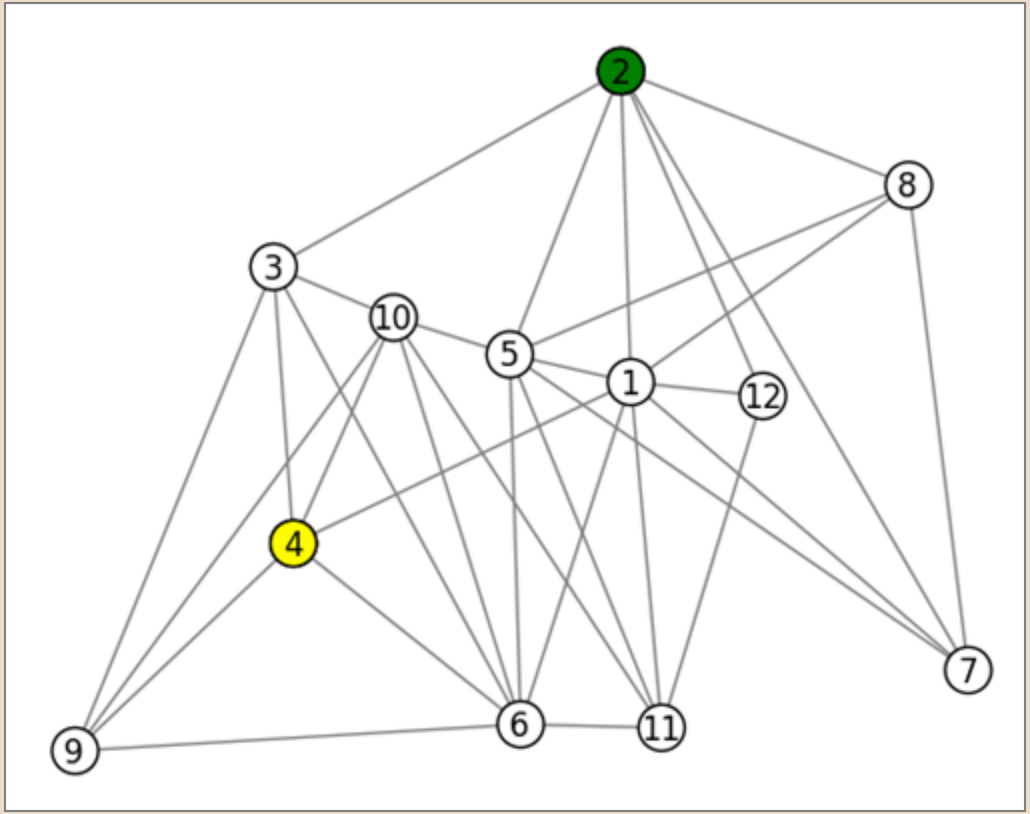


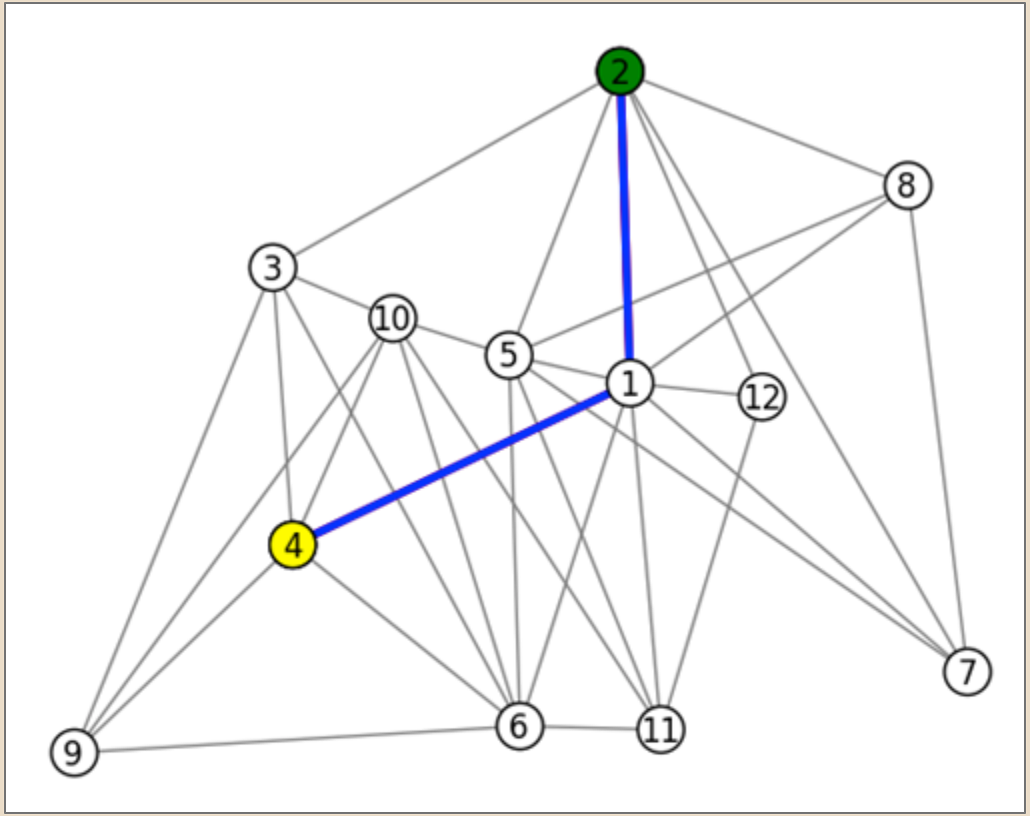
Fig : Final Output for Budget = 5.

Example Graph: Practical Topology



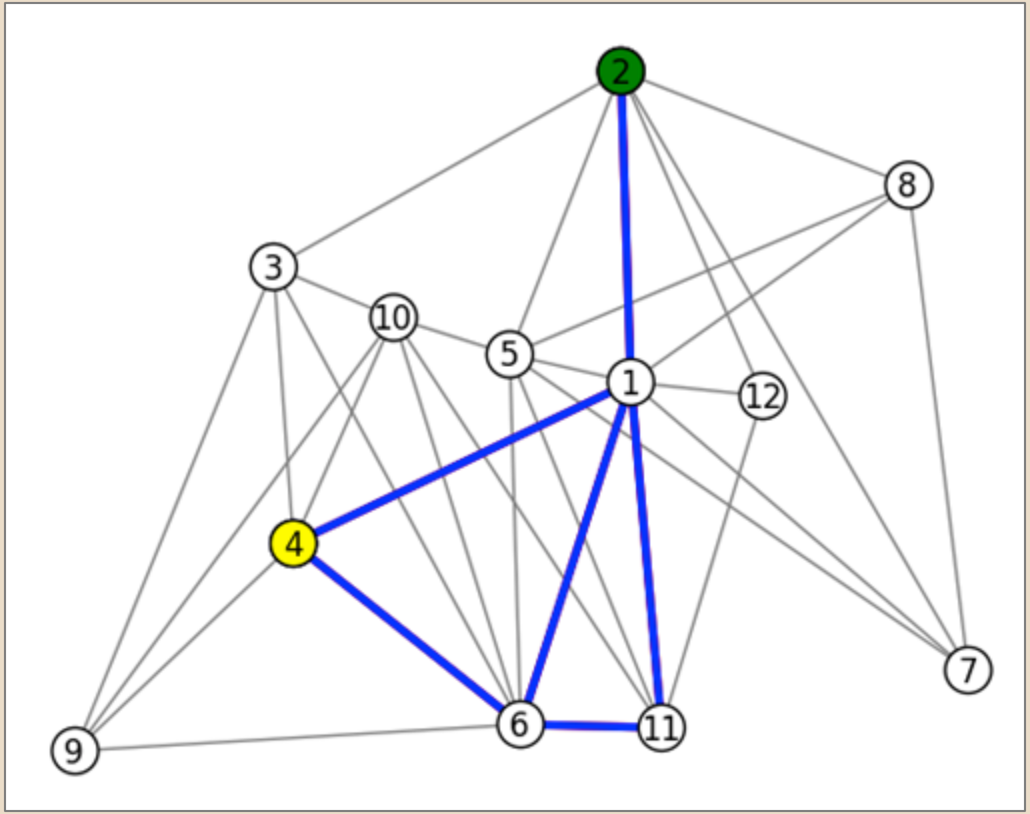
With source node 2, destination node 4, and a budget *b* = 34.75 ms, no paths are possible.

Example Graph: Practical Topology



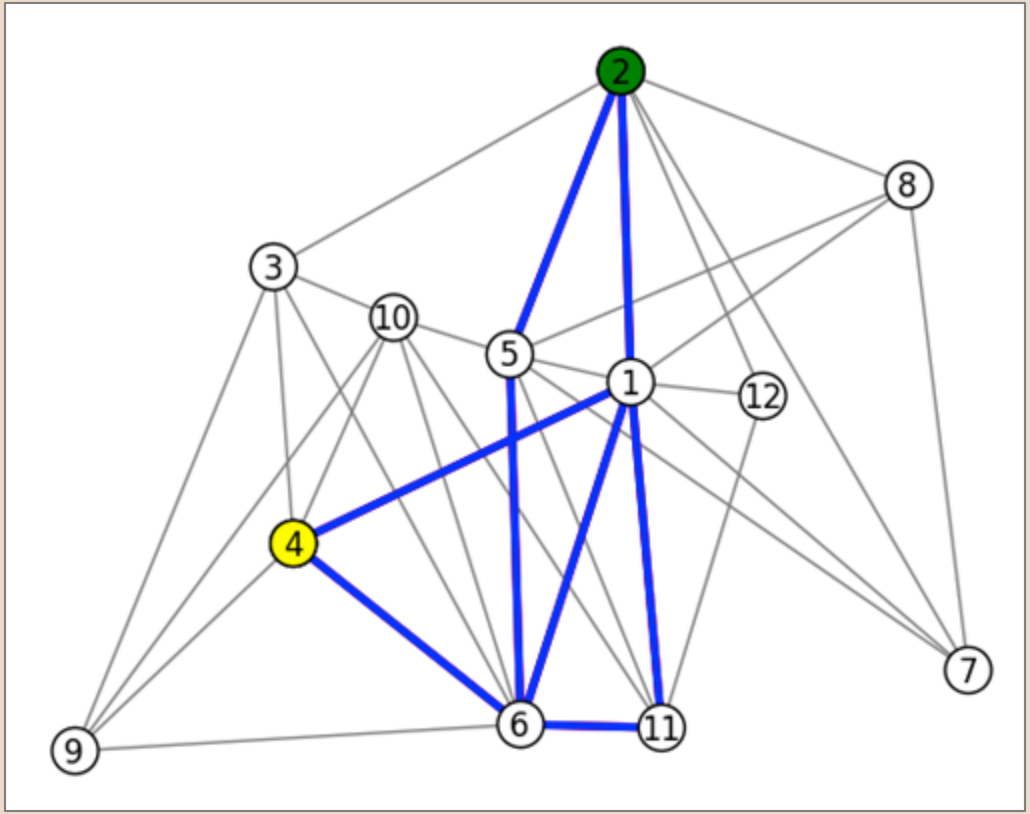
At *b* = 35 ms, a single path appears.

Example Graph: Practical Topology



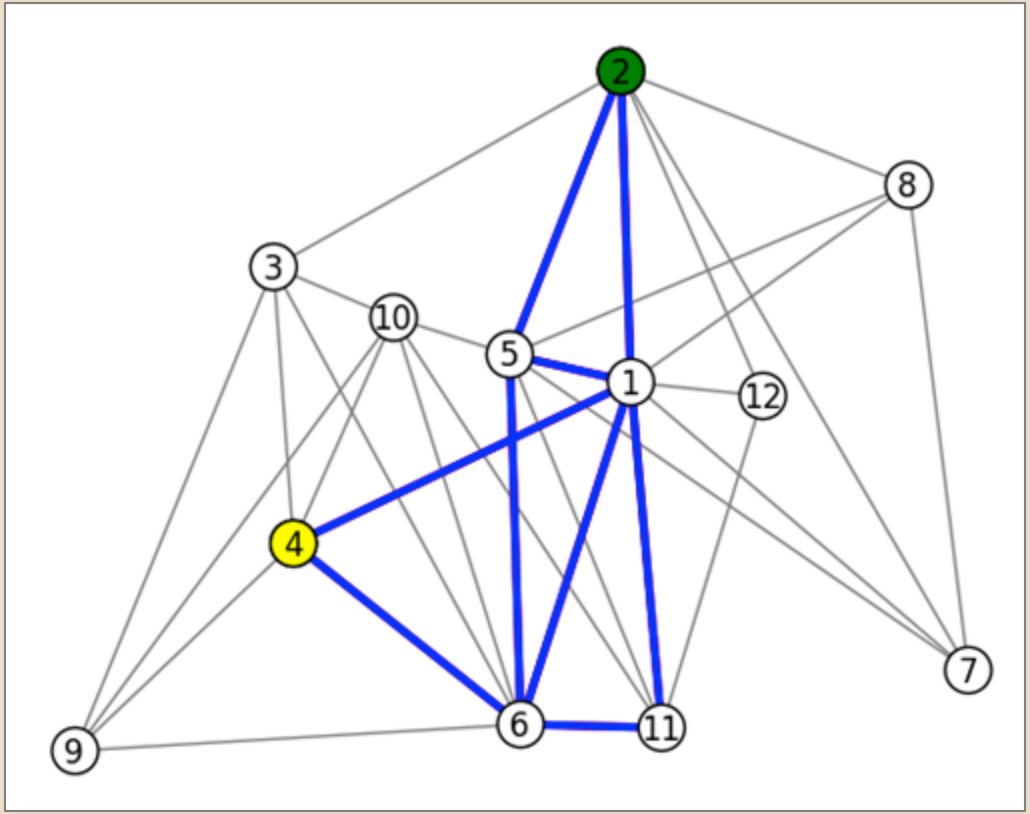
No further changes until *b* = 36 ms.

Example Graph: Practical Topology



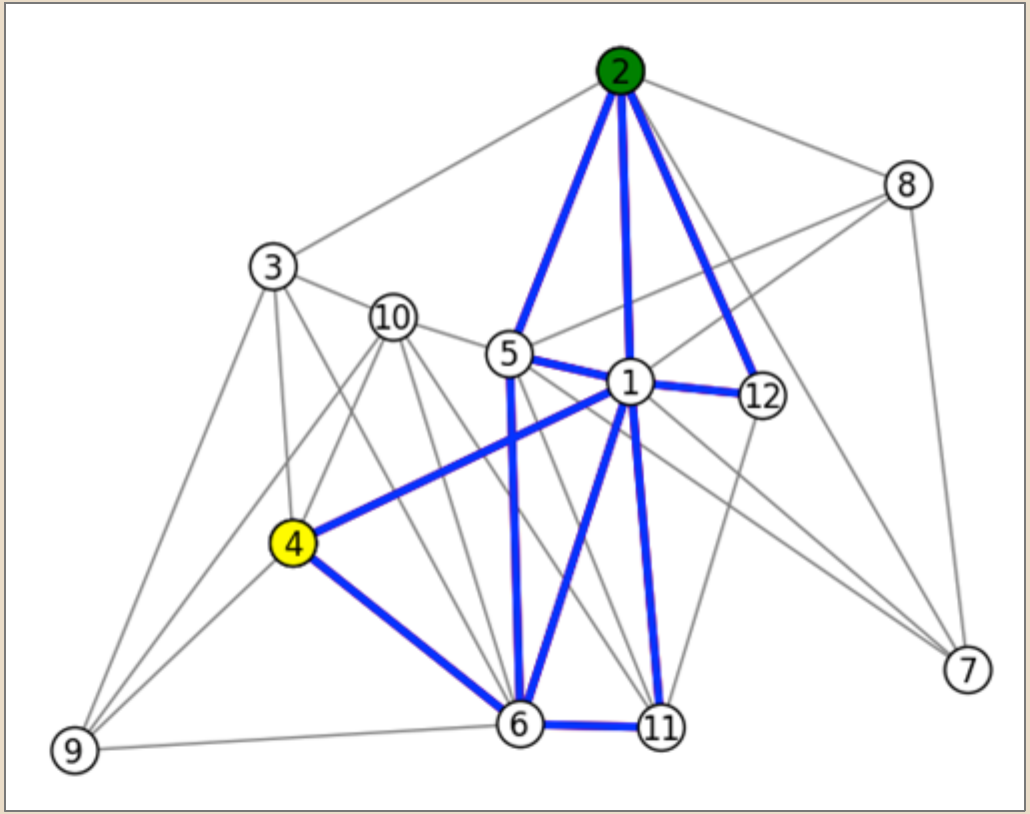
At b = 36.25 ms.

Example Graph: Practical Topology



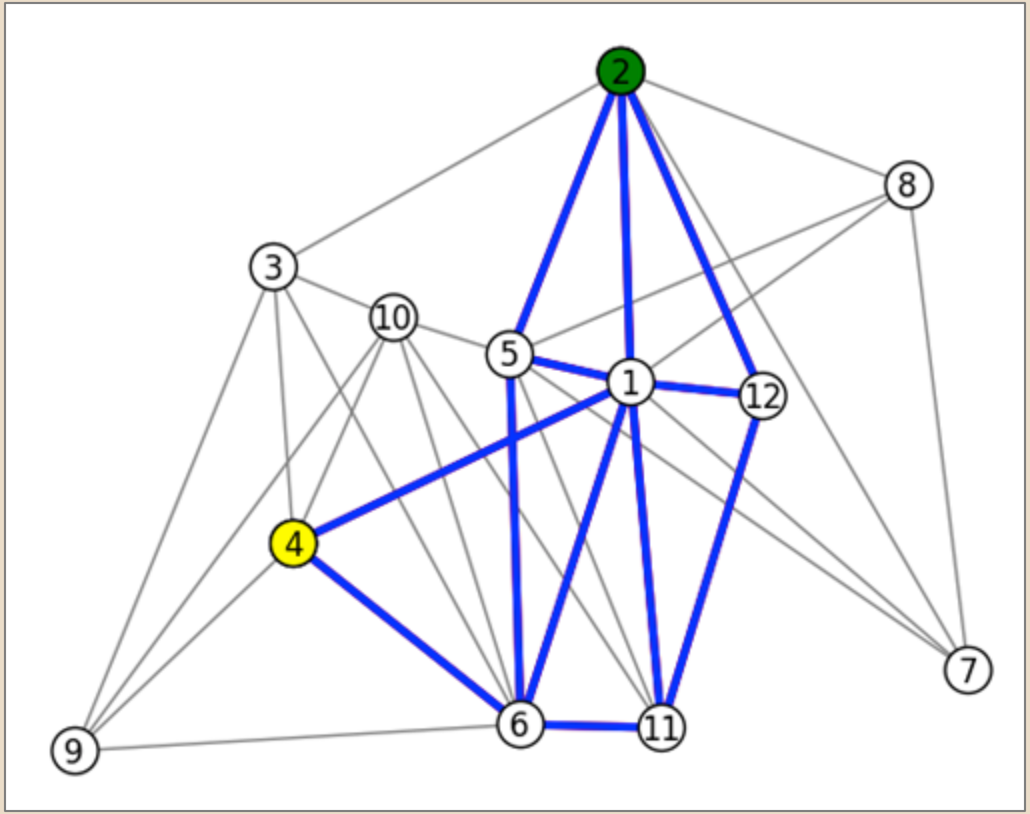
At *b* = 36.75 ms.

Example Graph: Practical Topology



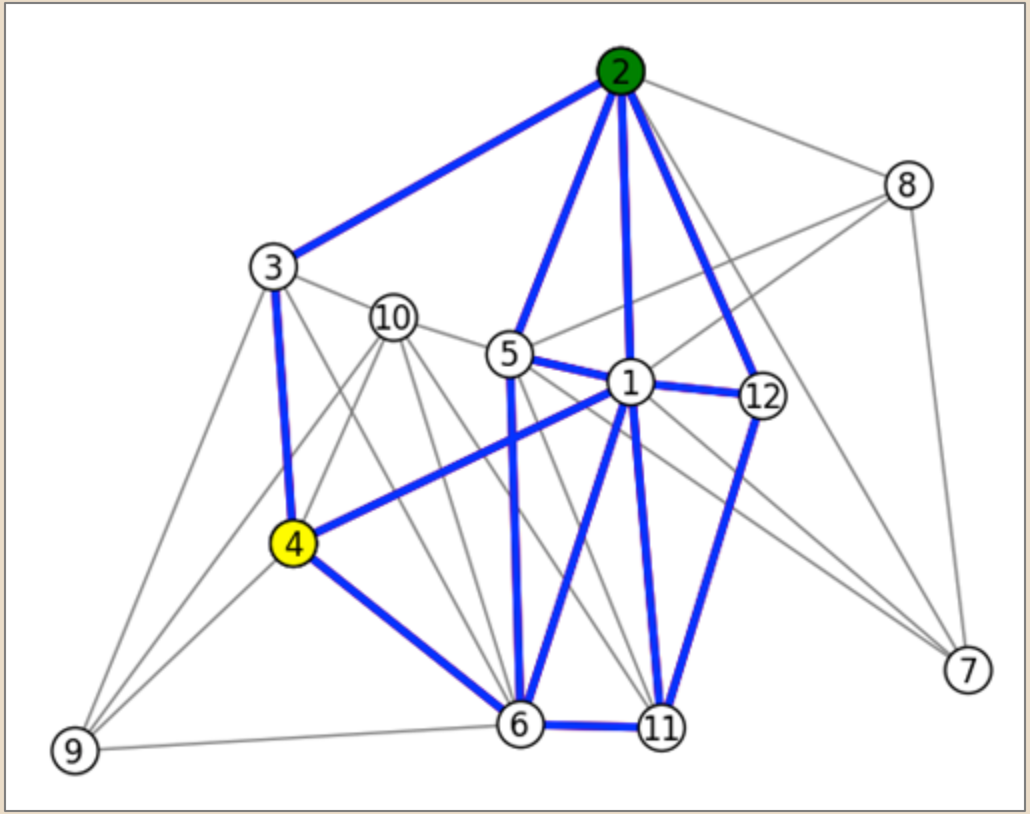
At b = 37.25 ms.

Example Graph: Practical Topology



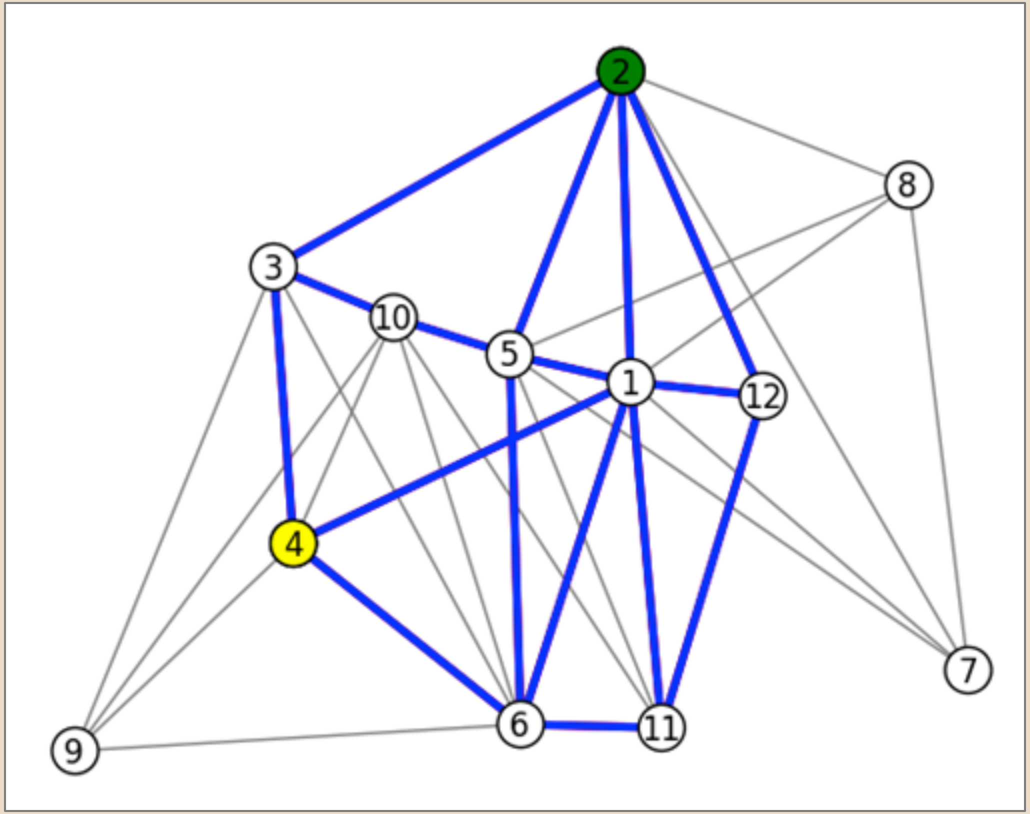
At *b* = 38 ms.

Example Graph: Practical Topology



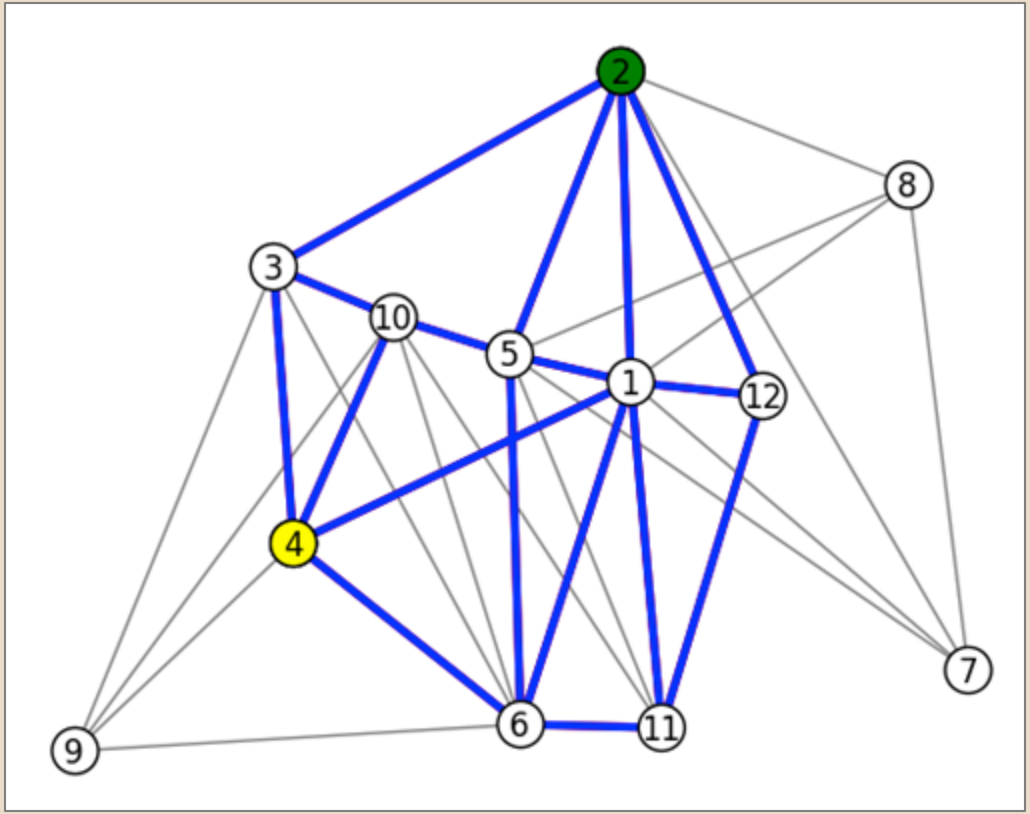
At *b* = 38.5 ms.

Example Graph: Practical Topology



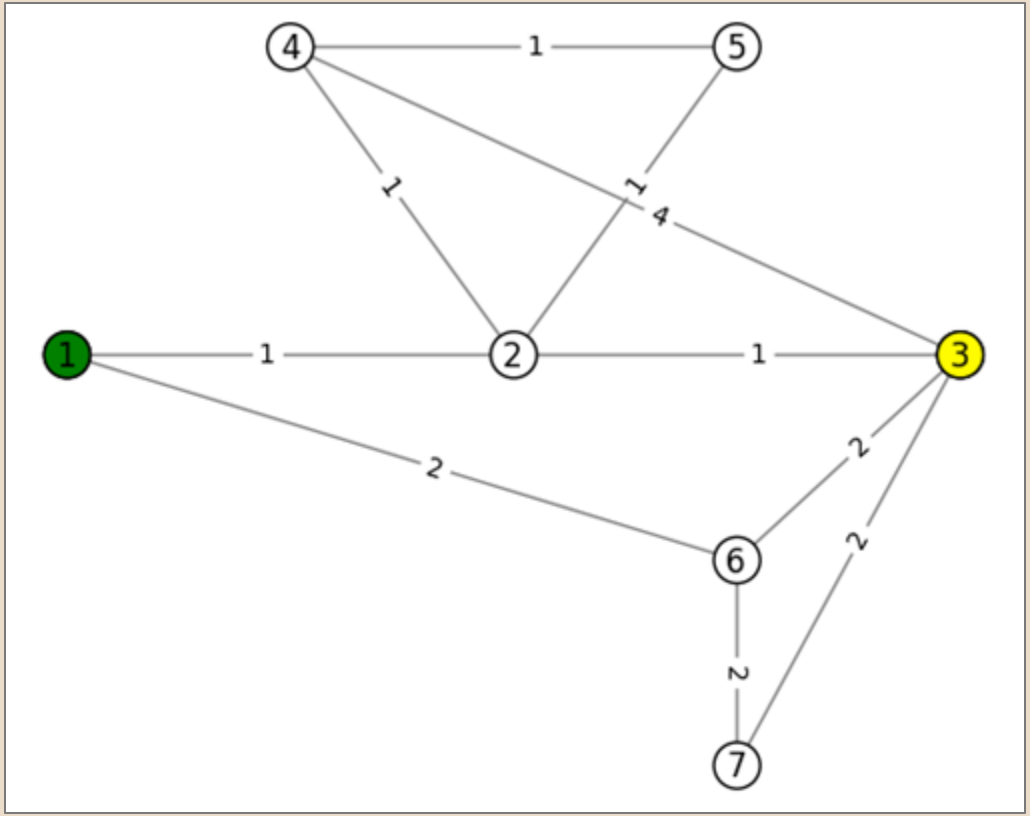
At b = 40.25 ms.

Example Graph: Practical Topology



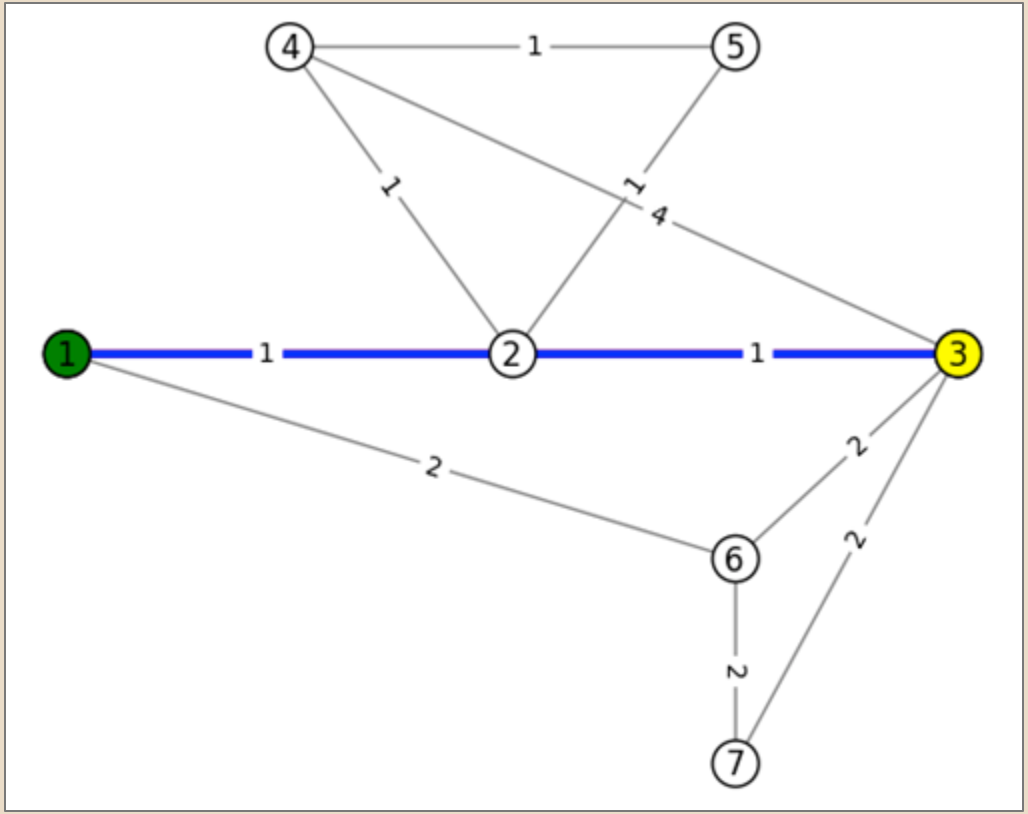
At *b* = 41.25 ms.

Example Graph: Loop Graph



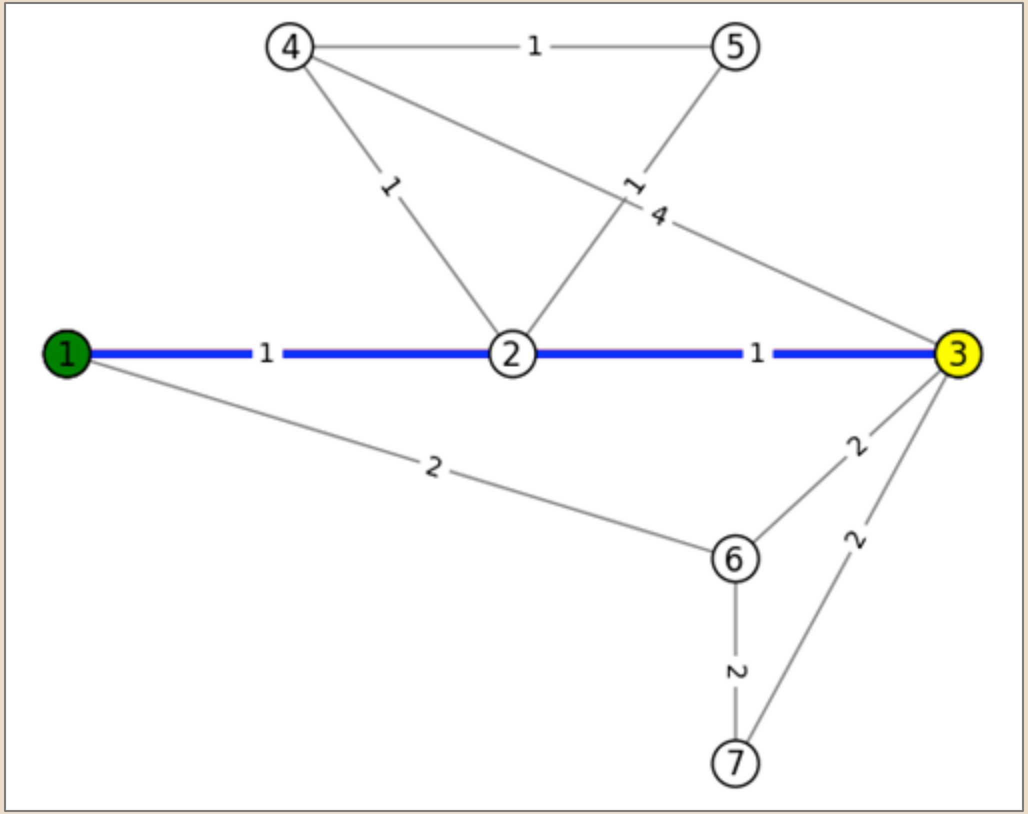
This graph will show the full power of algorithm that removes nodes only on non-simple paths from the source, node 1, to the destination, node 3.

Example Graph: Loop Graph



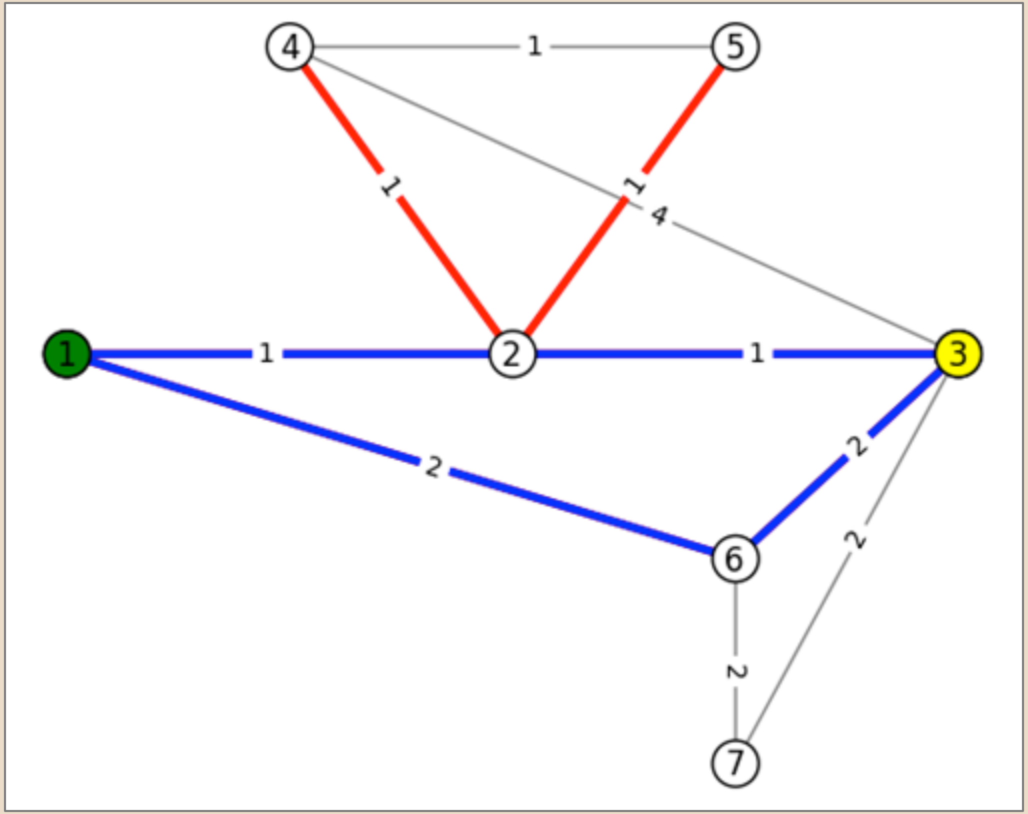
The first path appears at *b* = 2.

Example Graph: Loop Graph



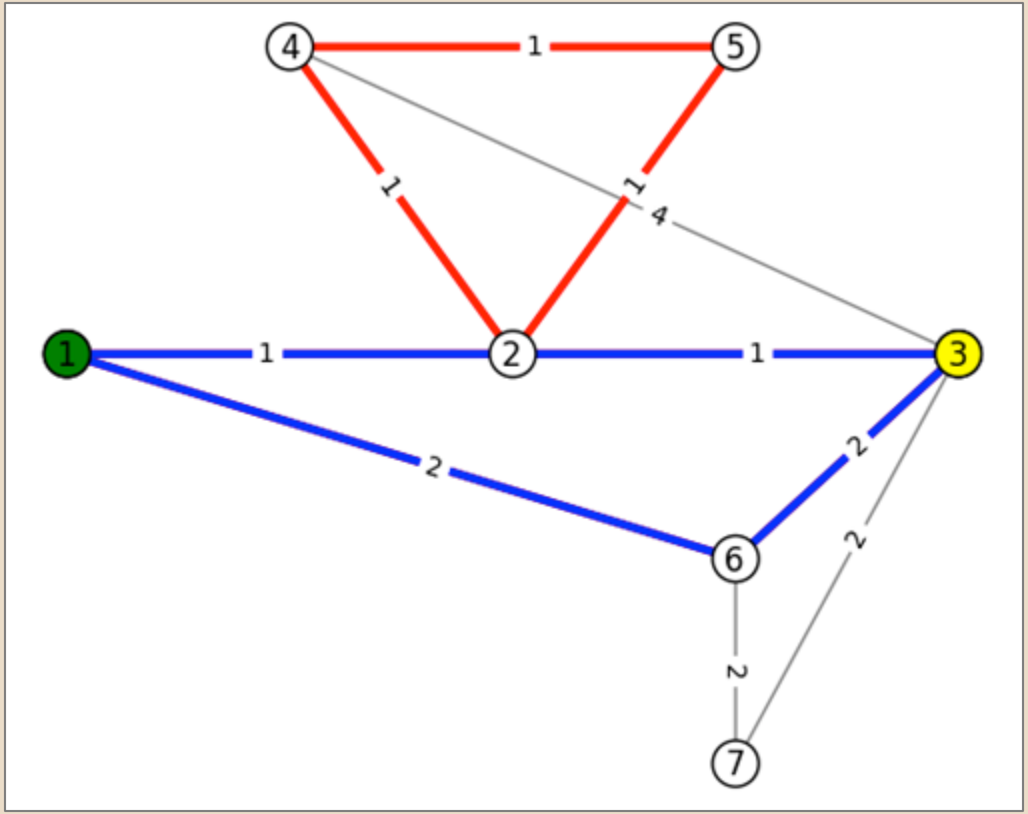
At *b* = 3.

Example Graph: Loop Graph



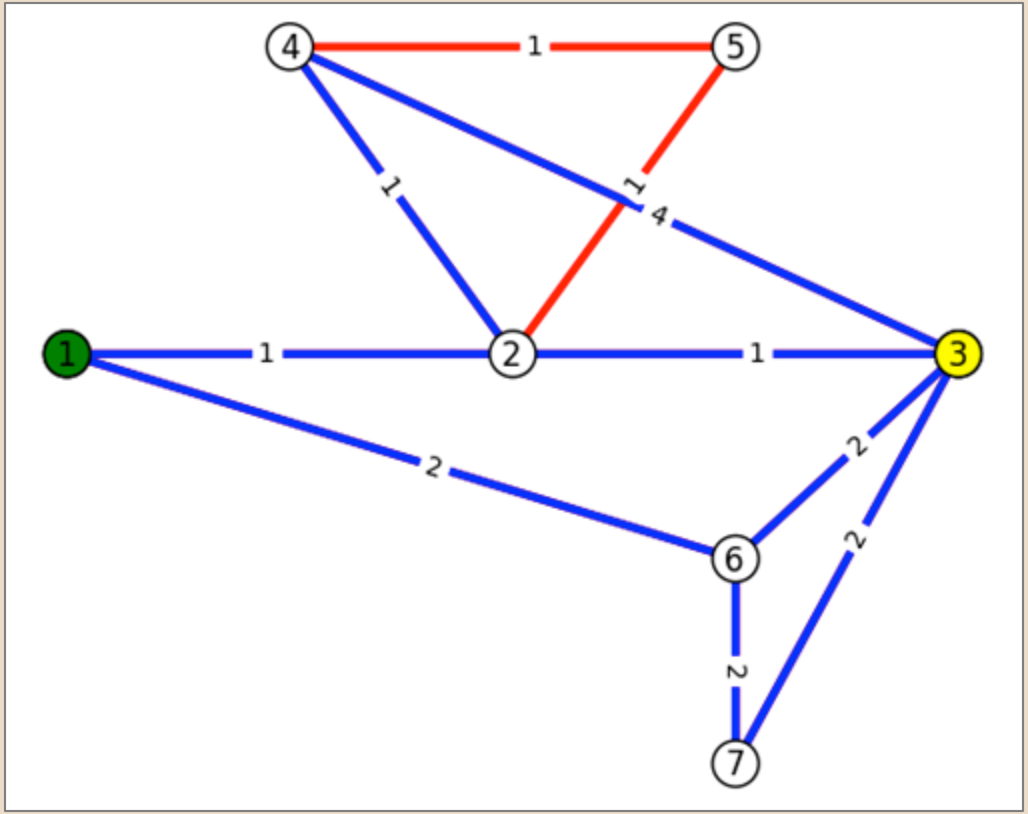
The first node removals happen at *b* = 4. They are shown in red. The bottleneck at node 2 precludes the existence of 2 node-disjoint paths.

Example Graph: Loop Graph



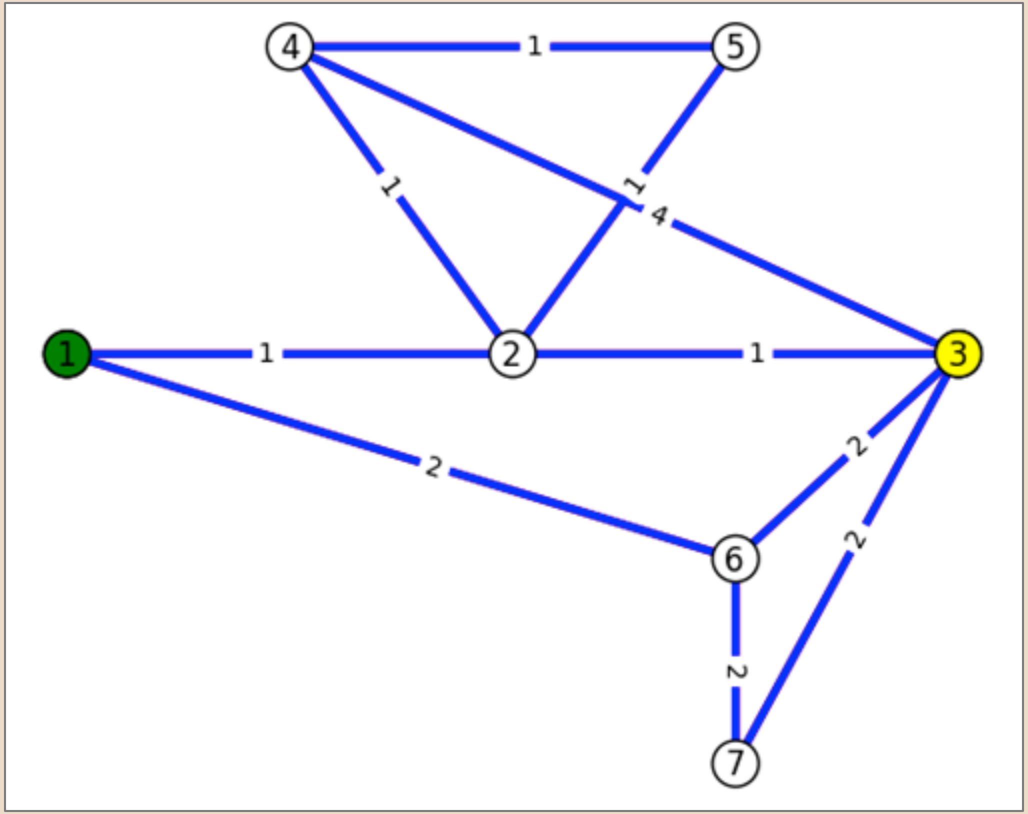
An additional edge is within the time constraint at *b* = *5*, but is removed due to the same bottleneck.

Example Graph: Loop Graph



At *b* = 6, the edge between 4 and 3 is now within the budget on the path 1-2-4-3, so there is no longer a bottleneck at node 2 for node 4.

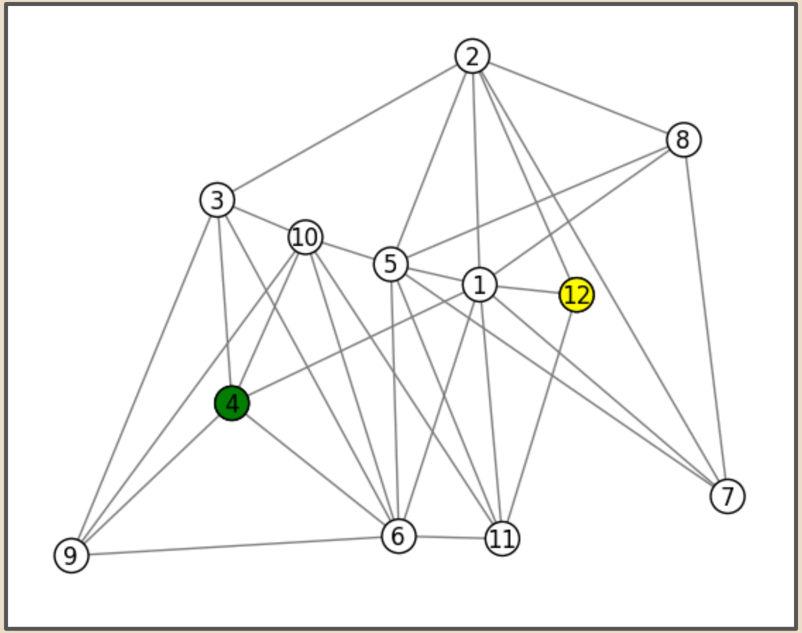
Example Graph: Loop Graph



At *b* = 7, the path 1-2-5-4-3 is now in the budget, so all nodes are now on some simple path from source to destination within the budget.

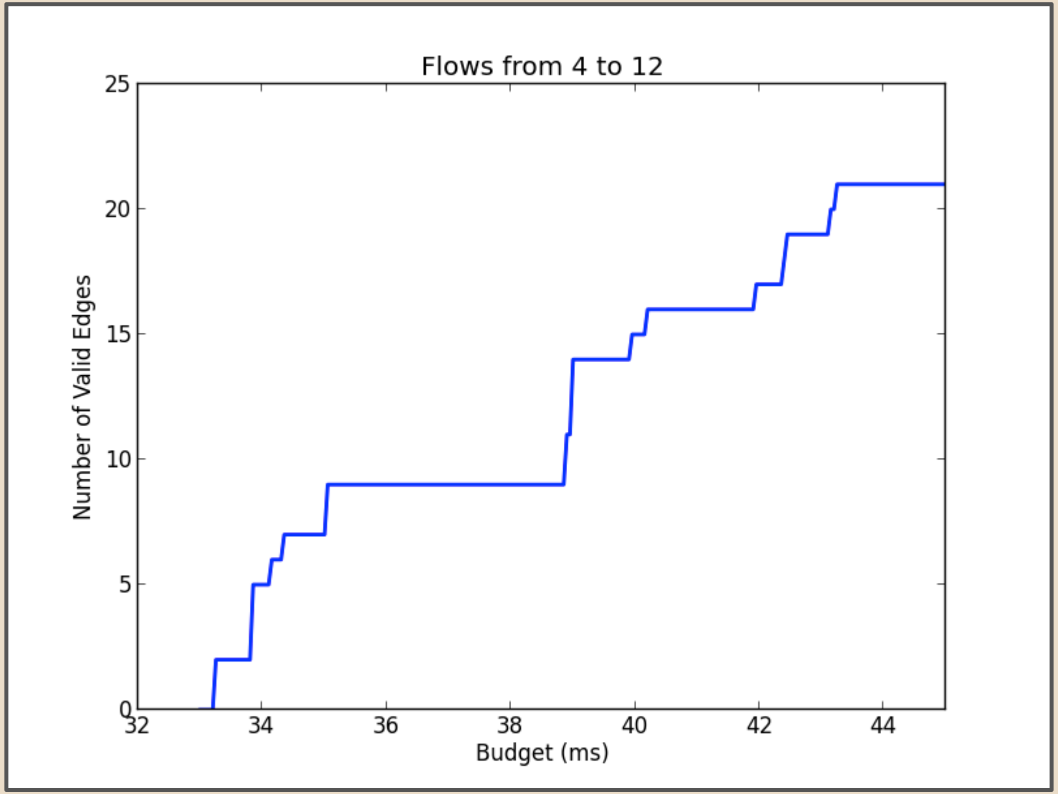
Demonstration: Subgraph Reachability

- When increased latency budget results in more edges in the subgraph, more nodes can fail without breaking *s-t* reachability.
- Small latency budget increases can result in large increases in the number of edges within the time constraint.



Live demo on practical network topology from west coast to JHU.

Demonstration: Subgraph Reachability



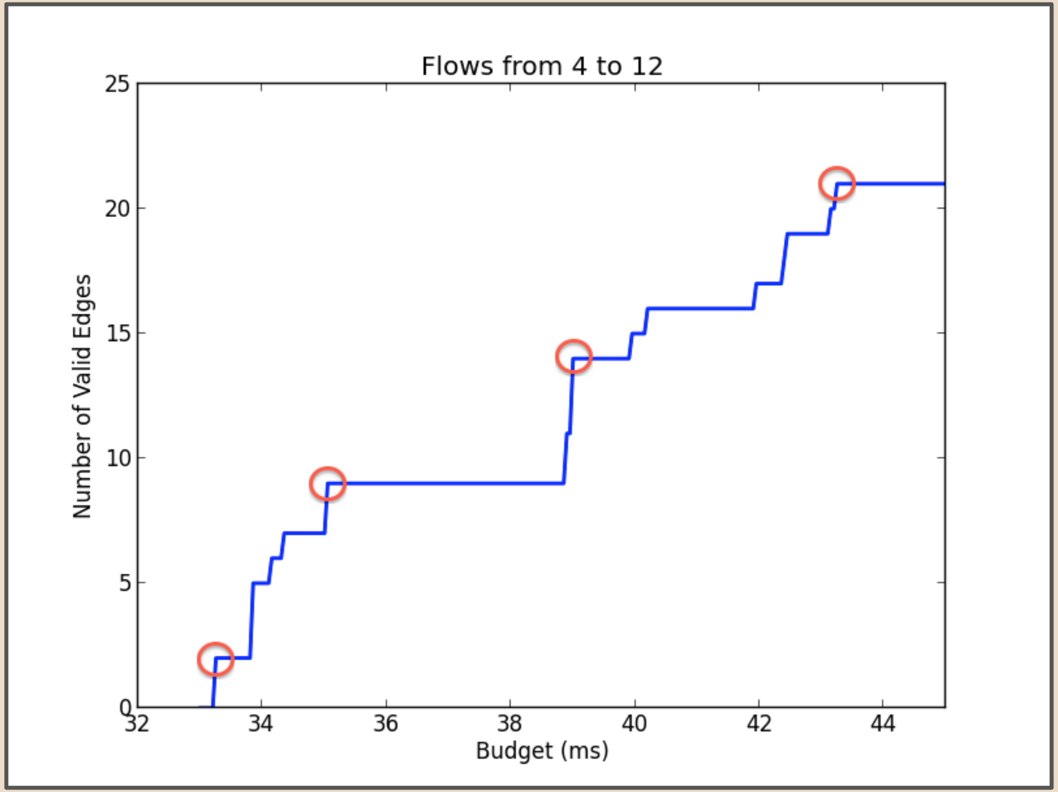
Demonstration: Subgraph Reachability

$b_1 = 33.75$ ms

$b_2 = 35.05$ ms

$b_3 = 39.0$ ms

$b_4 = 43.25$ ms



Applications of Time-Constrained Flooding

- Using packets sent over time-constrained flooding graphs, we can get an upper bound on reliability for a given source, destination, and time budget.
- The smallest latency for which a path is found with this algorithm can be used to determine the minimum bandwidth cost and the resulting reliability for a given source and destination.
- Thus for a given *s*-*t* pair, we can get a **lower bound** on cost and latency and find the associated reliability, and an **upper bound** on reliability at a given budget.