ADC20
AUDIO
DEVELOPER
CONFERENCE

Introducing Tracktion Graph: A Topological Processing Library for Audio

Dave Rowland

1. Topological Graphs

Introduction

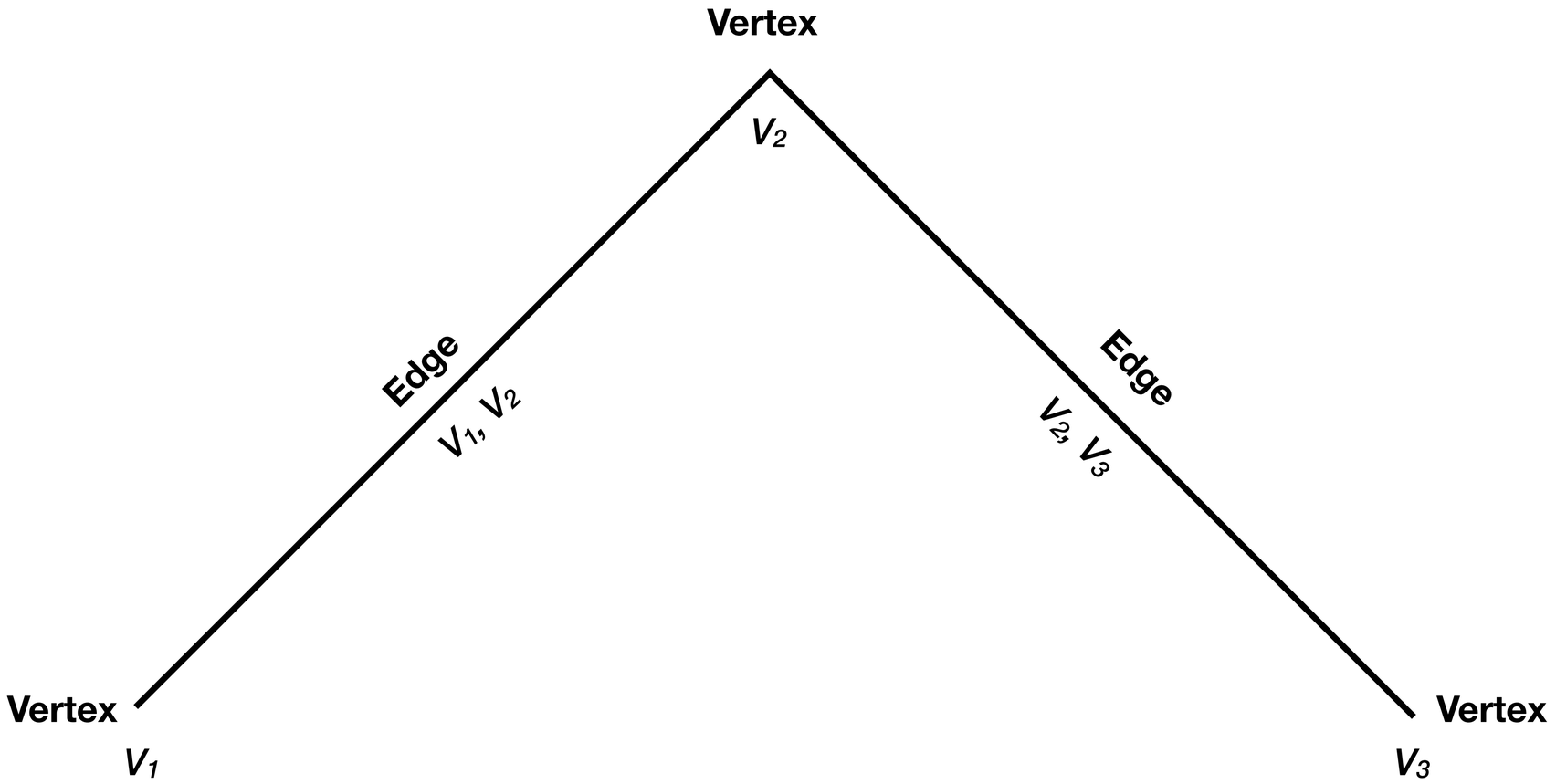
- What is a topological graph?

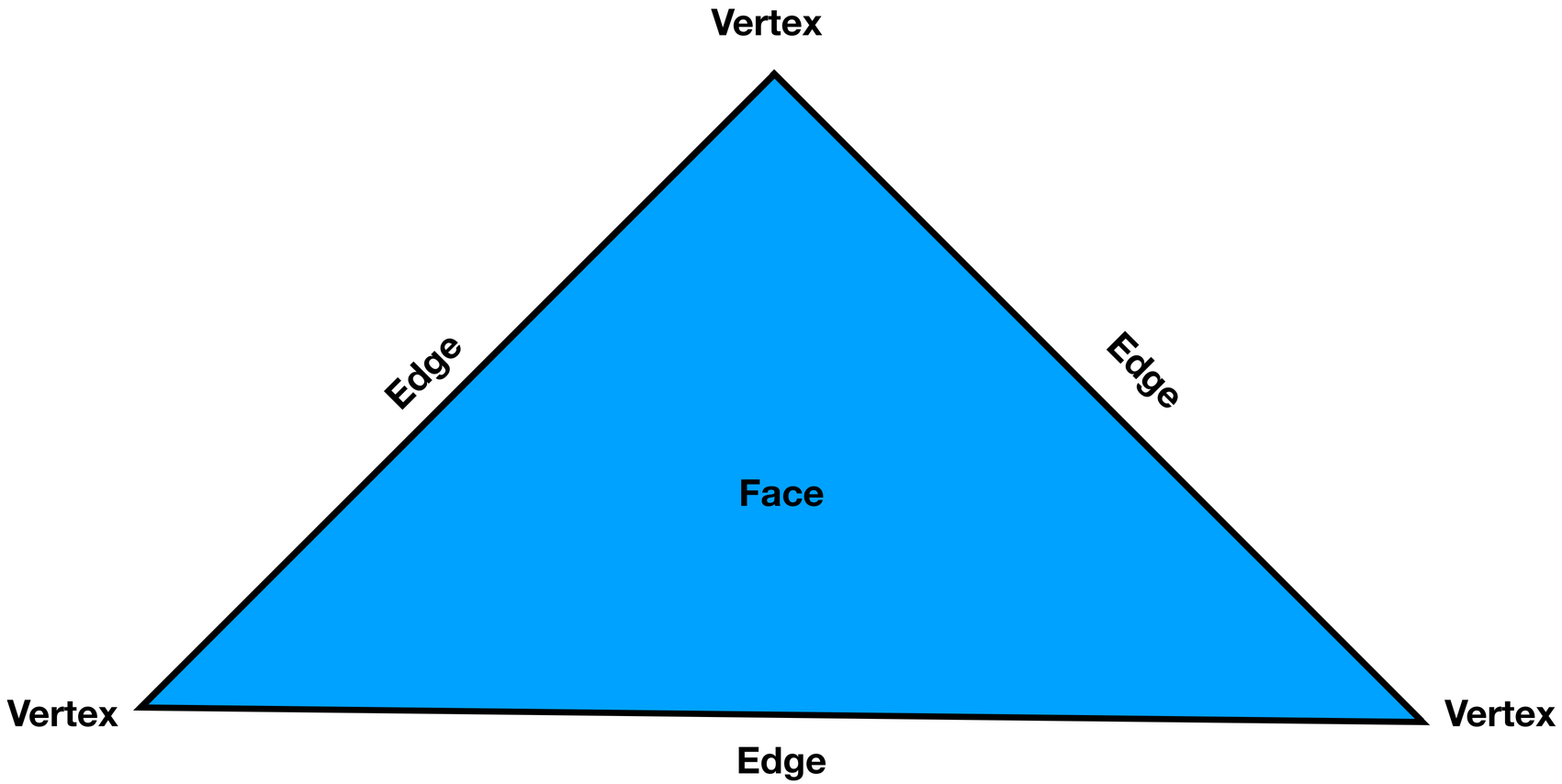
$$V = \{ V_1, V_2, V_3 \}$$

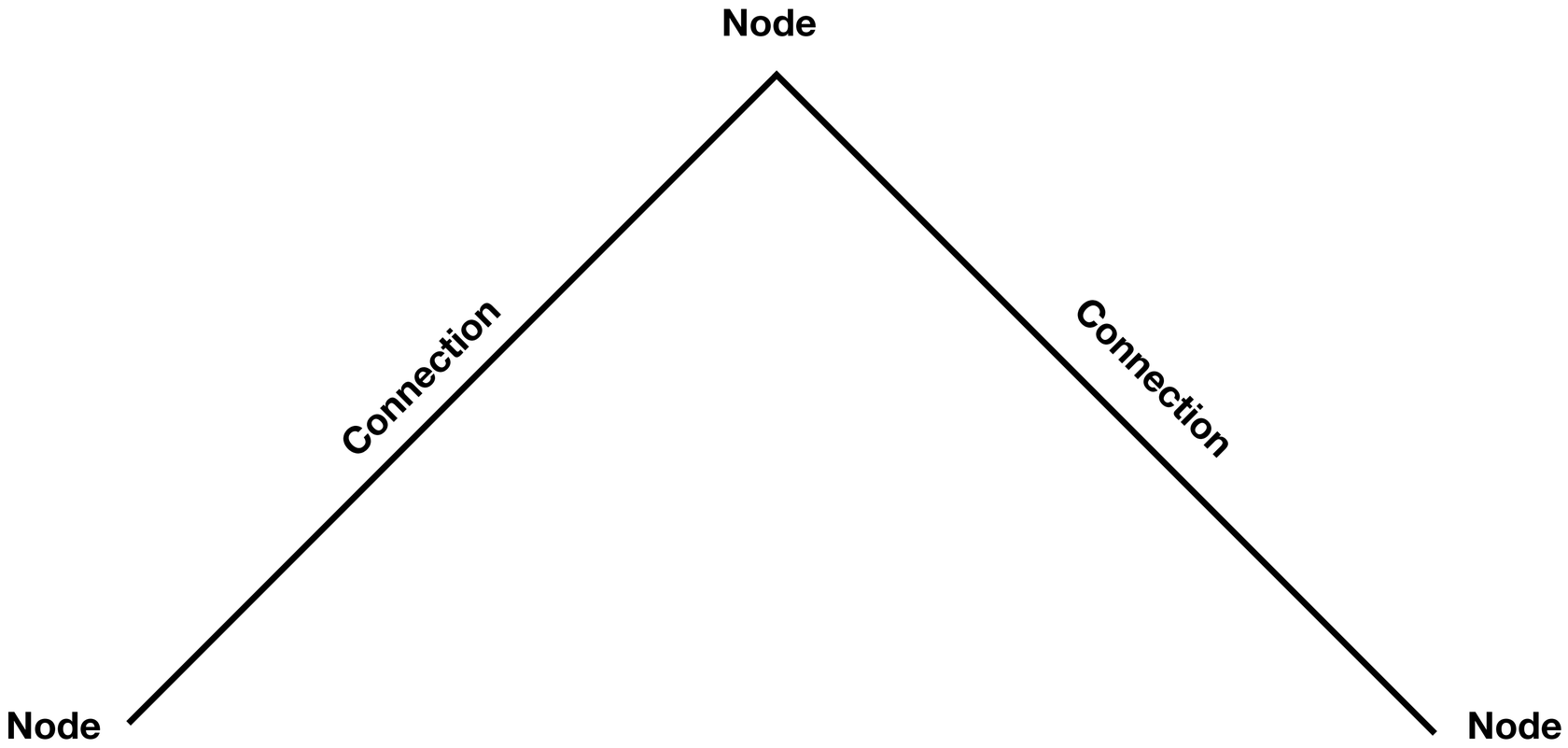
$$E = \{ \{ V_1, V_2 \}, \{ V_2 V_3 \} \}$$

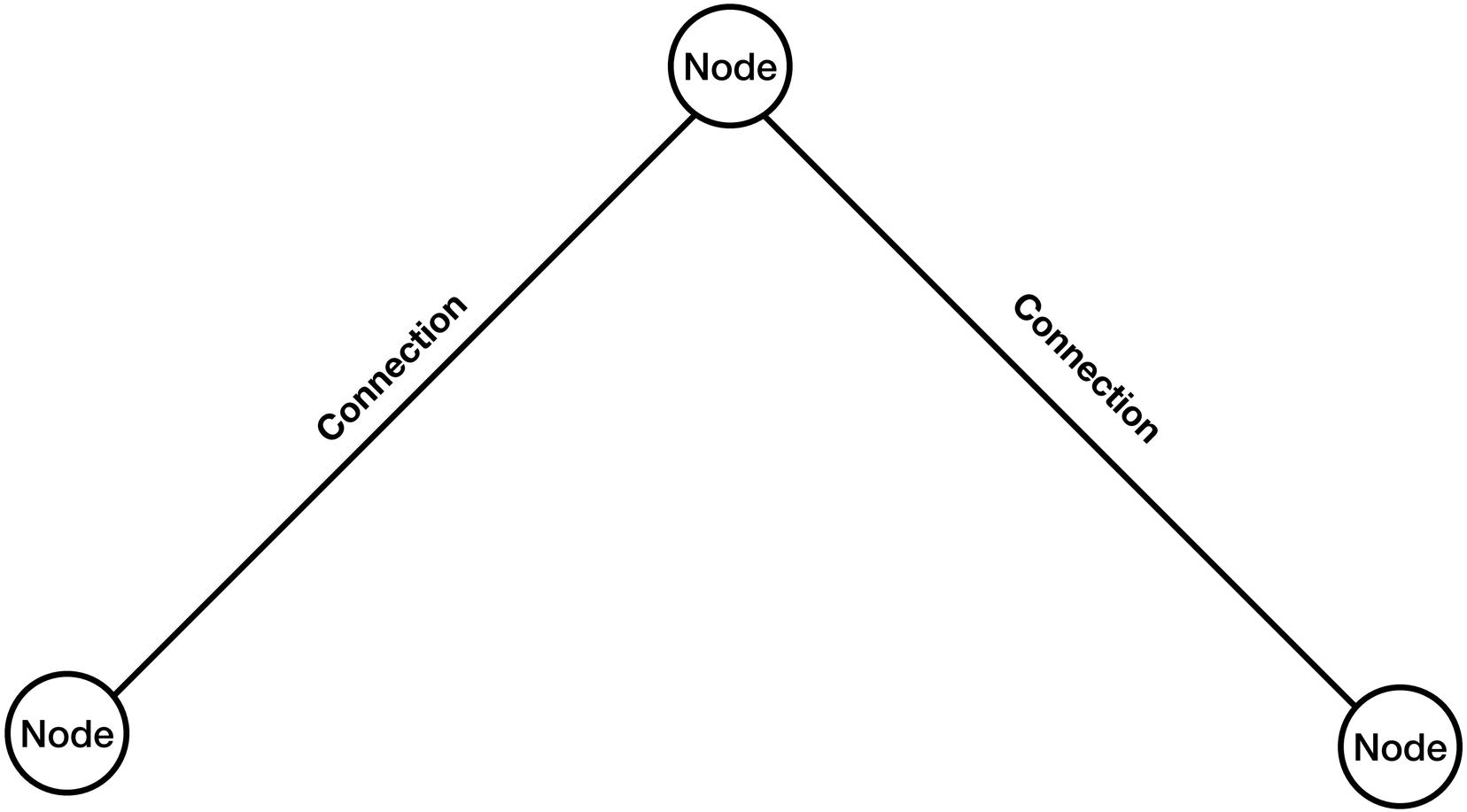
- *Formally:*
  "An ordered pair of sets (V, E), where the elements of V are called vertices or nodes and E is a set of pairs (called edges) of elements of V"

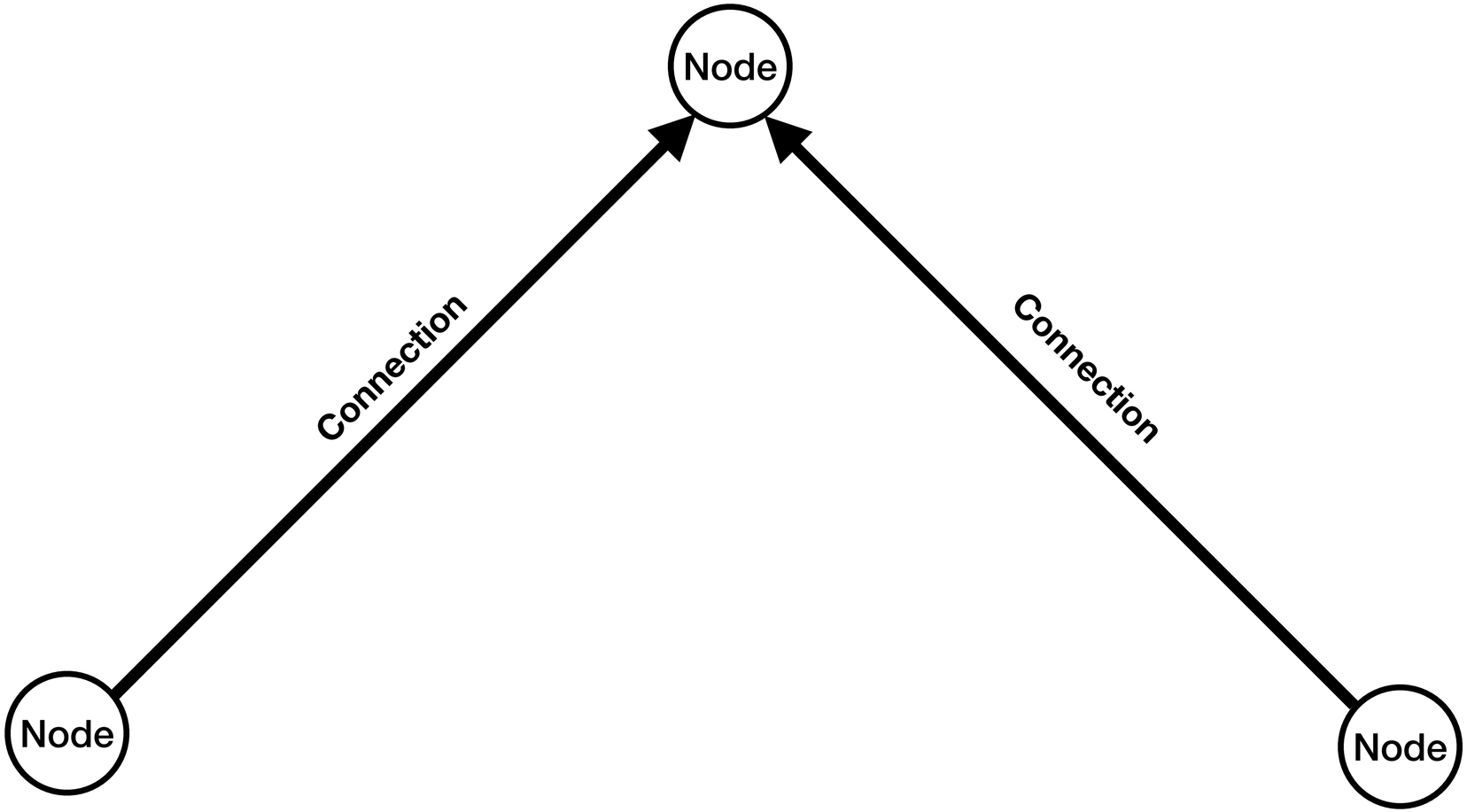
- *Informally:*
  "A set of vertices (or nodes) together with a set of edges that connect (some of) the vertices"


Vertex
V_2
Edge
V_1, V_2
Edge
V_2, V_3
Vertex
V_1
Vertex
V_3

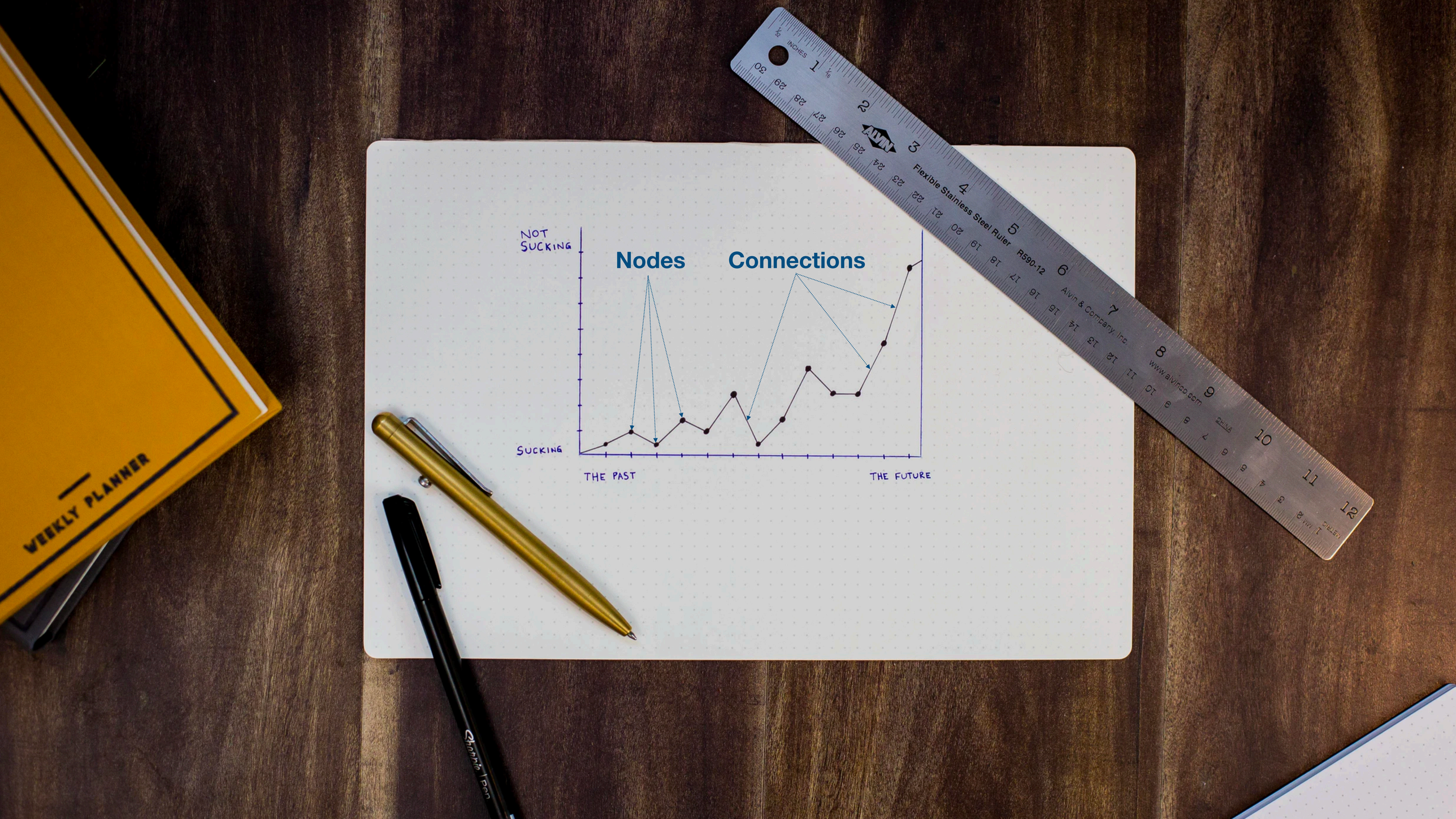

Vertex
Edge
Edge
Face
Vertex
Vertex
Edge

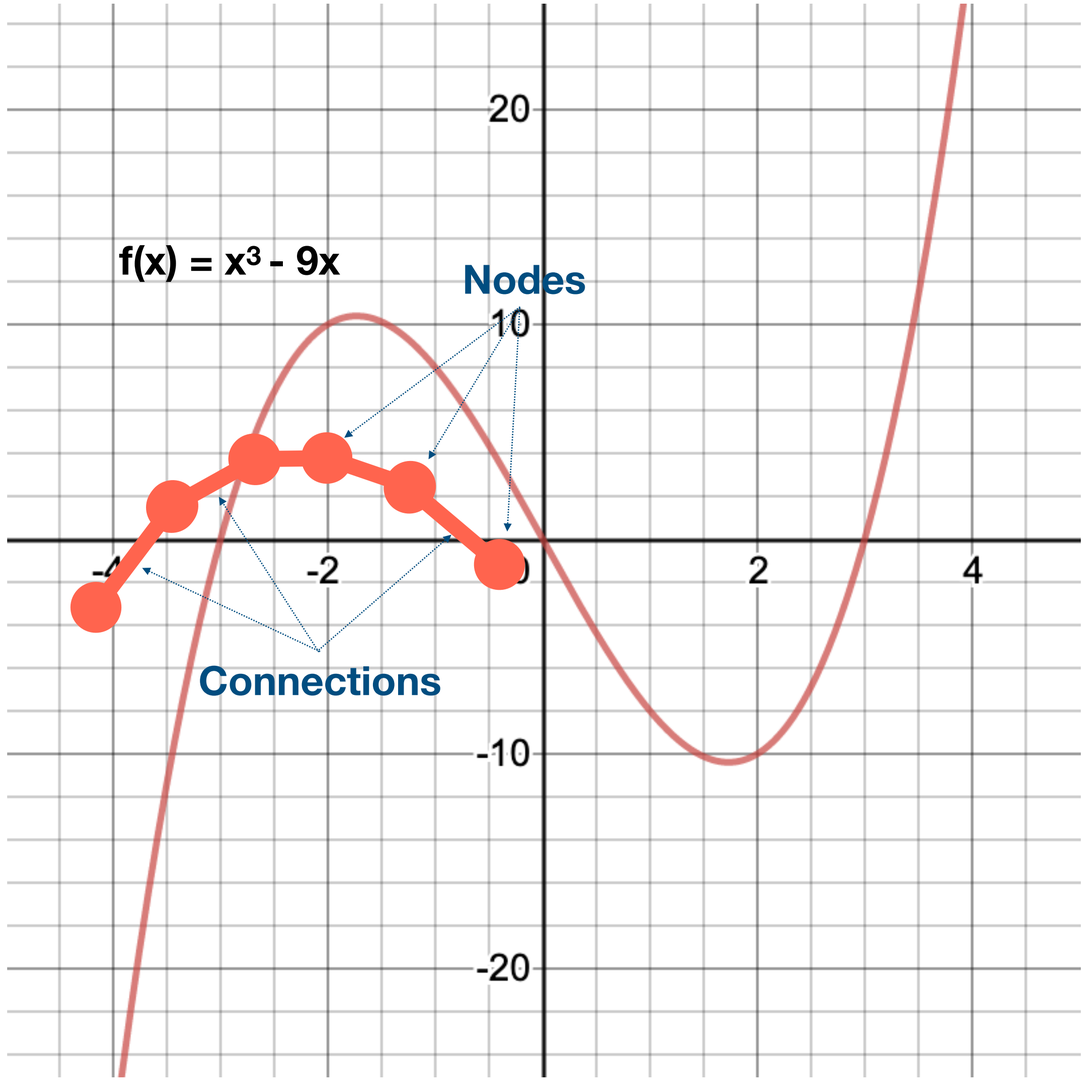

Node
Connection
Connection
Node
Node

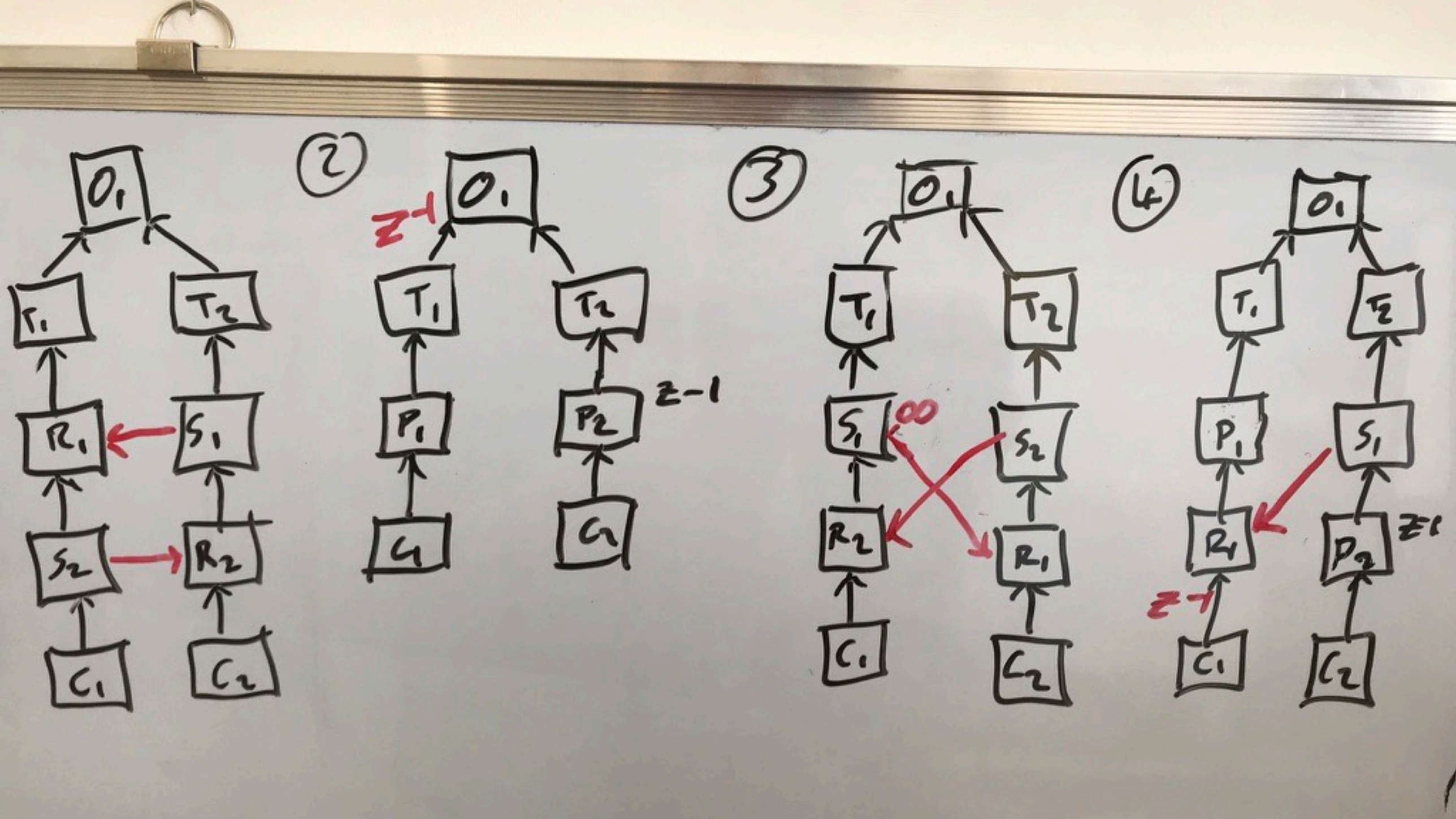

Node
Connection
Connection
Node
Node

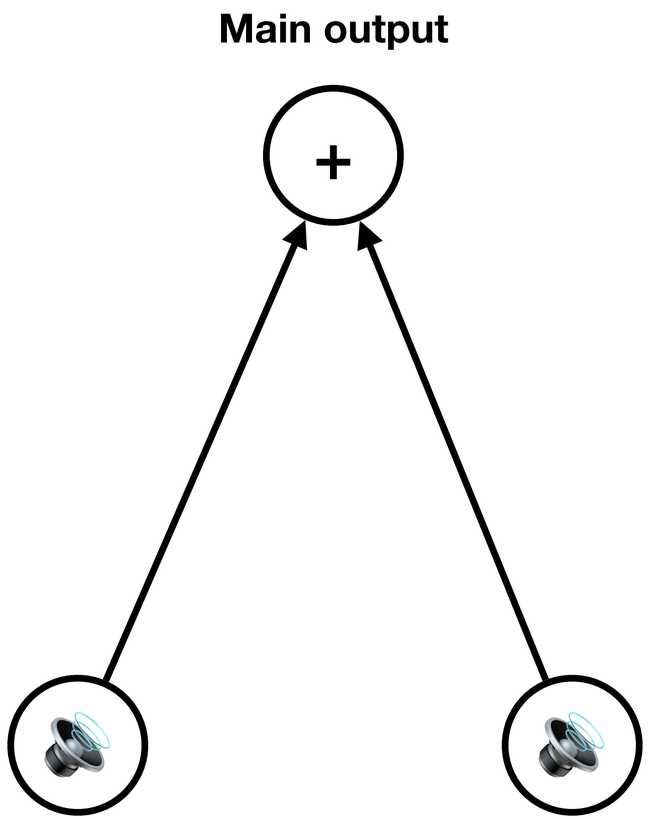


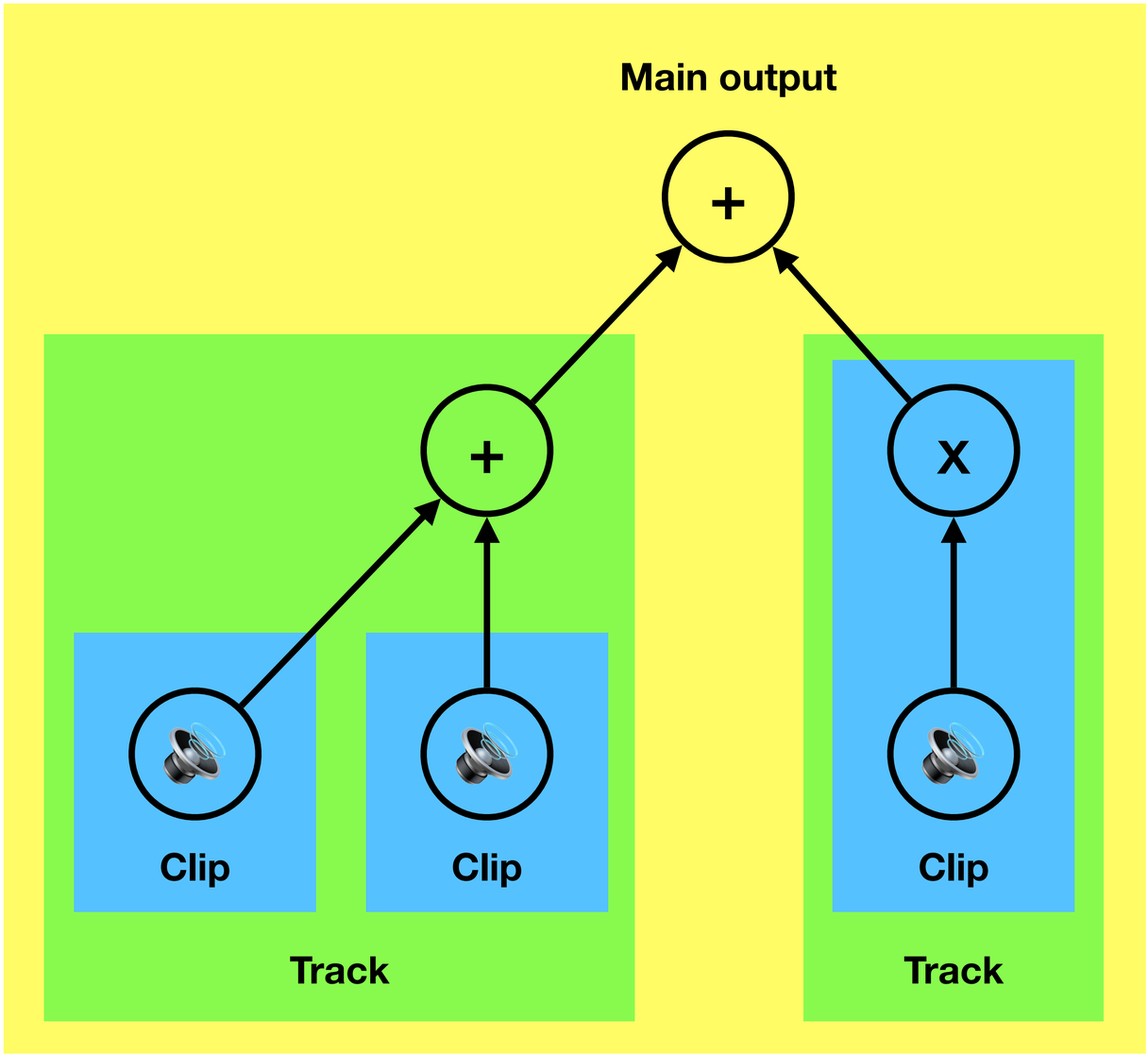
Directed Graph


NOT SUCKING
SUCKING
Nodes
Connections
THE PAST
THE FUTURE
WEEKLY PLANNER
Flexible Stainless Steel Ruler
R590-12
Alvin & Company, Inc.
www.alvinco.com

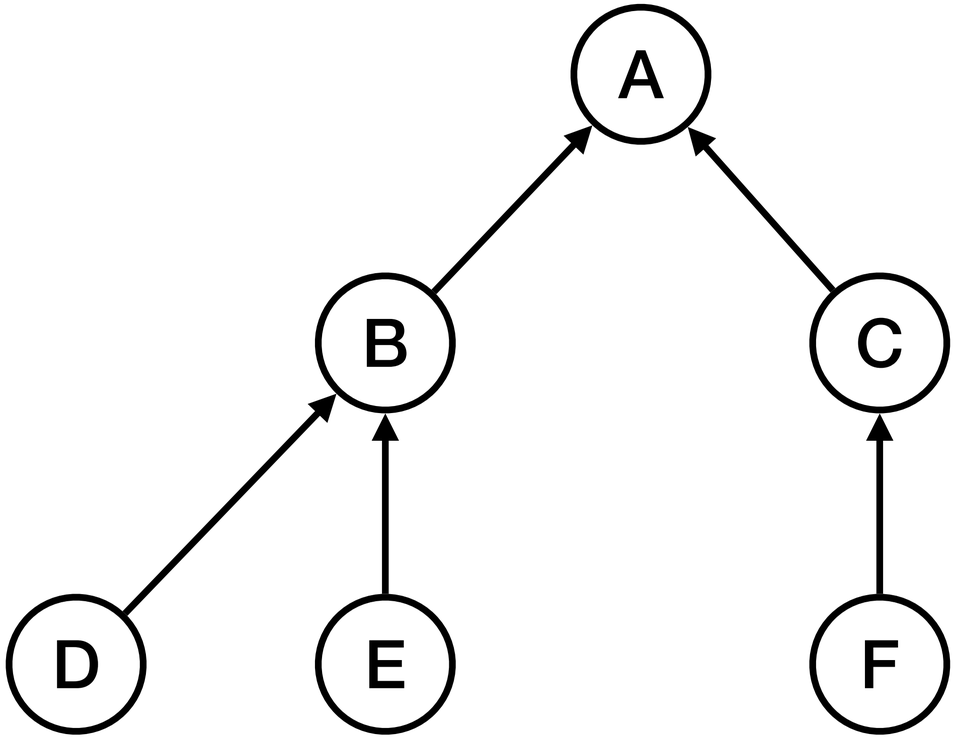

f(x) = x³ - 9x
Nodes
Connections
20
10
-10
-20
-2
2
4


O1
T1
T2
R1
S1
S2
R2
C1
C2
2
O1
z-1
T1
T2
P1
P2
C1
C2
z-1
3
O1
T1
T2
S1
∞
S2
R2
R1
C1
C2
4
O1
T1
T2
P1
S1
R1
P2
z-1
z-1
C1
C2

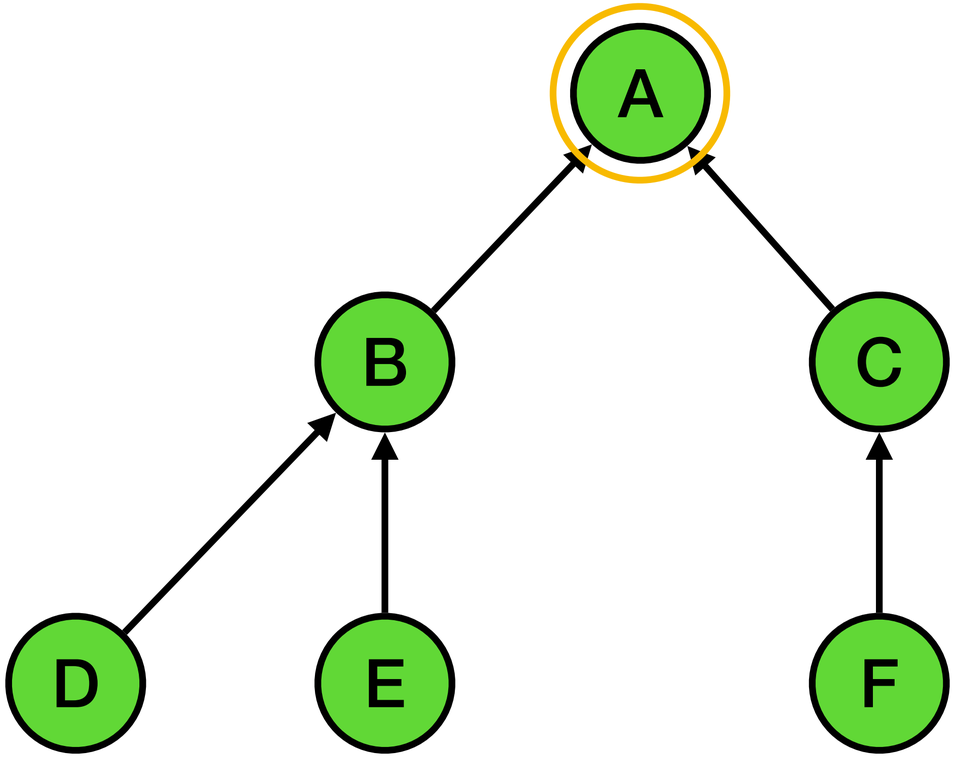

Main output
+


Main output
+
+
x
Clip
Clip
Clip
Track
Track

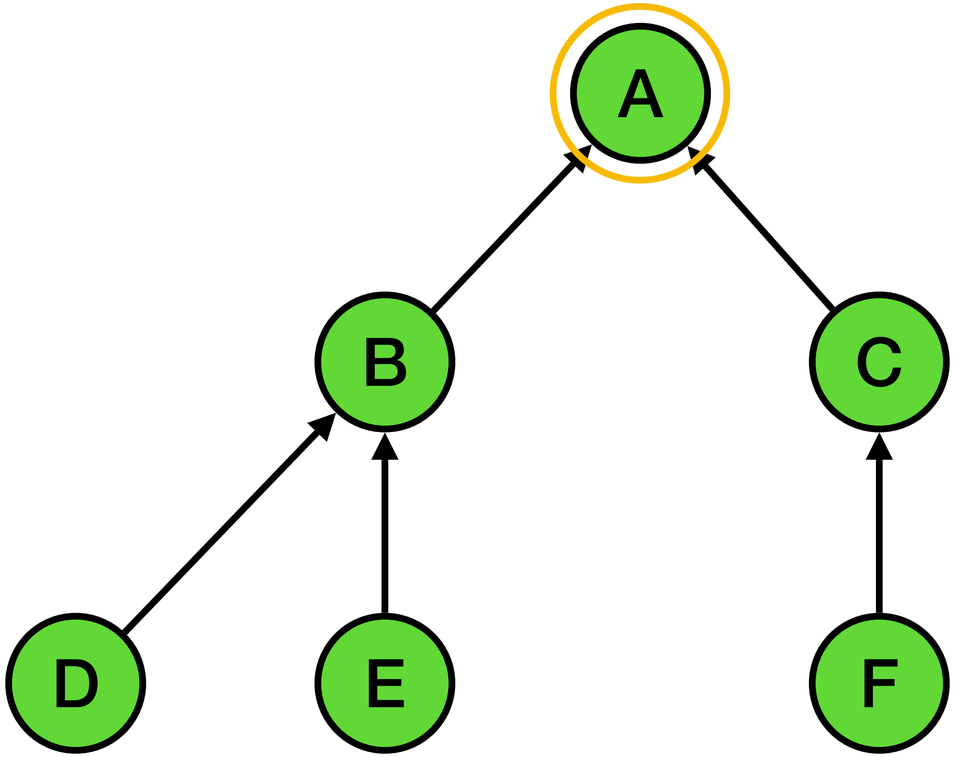
- Intuitively:
  D, E & F
  B & C
  A

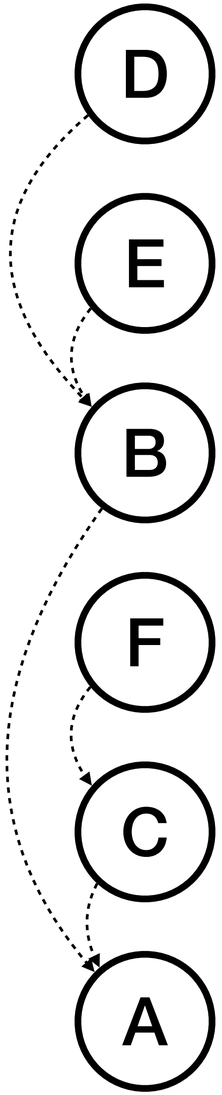


- **D**epth **F**irst **S**earch (**DFS**):

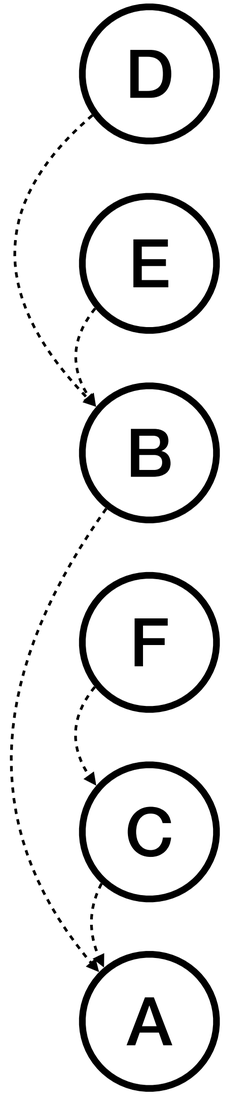


- **Post**-ordered DFS:

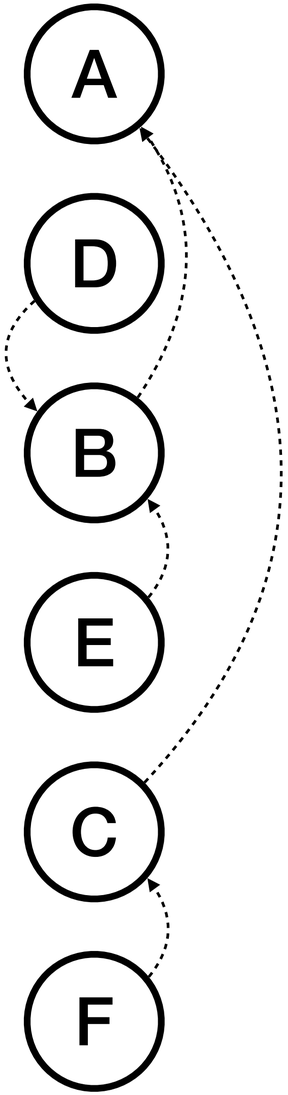


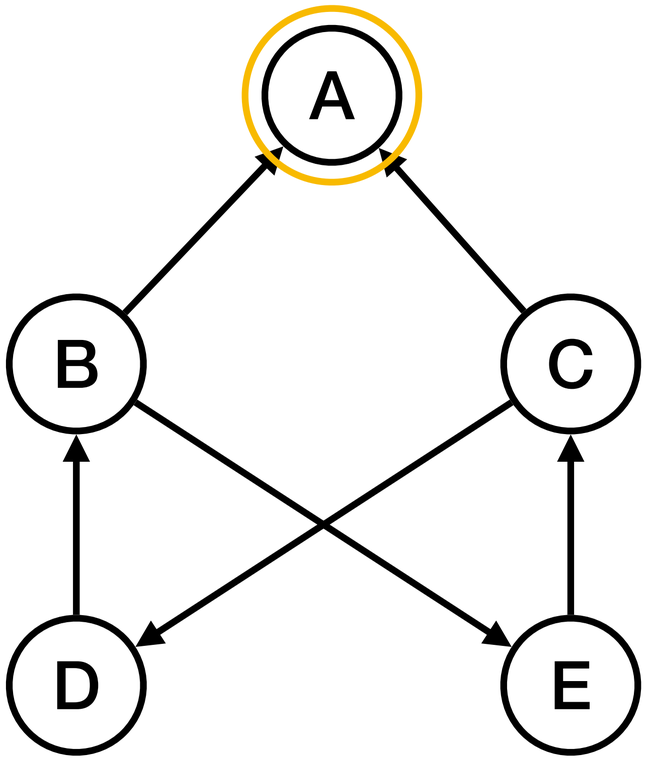

D
E
B
F
C
A

Post-ordered DFS: ✅

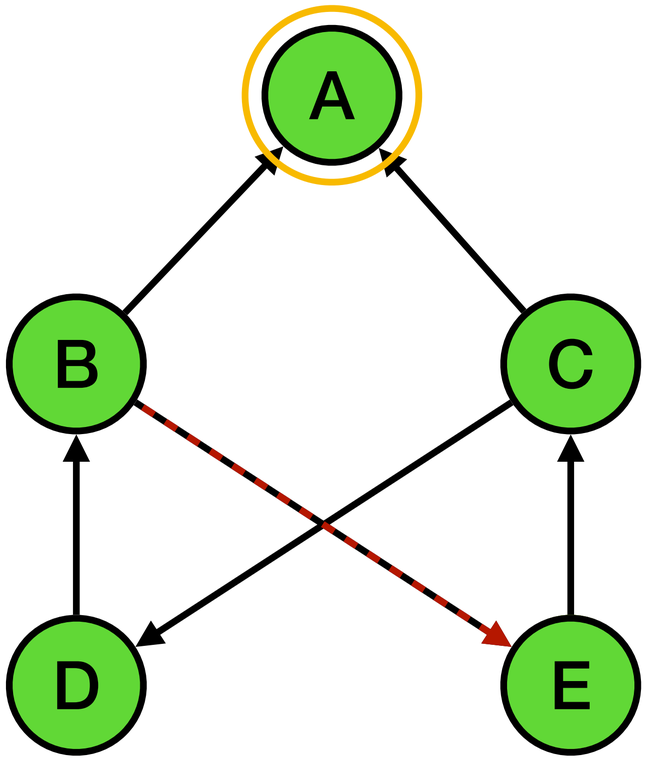


Pre-ordered DFS: ❌

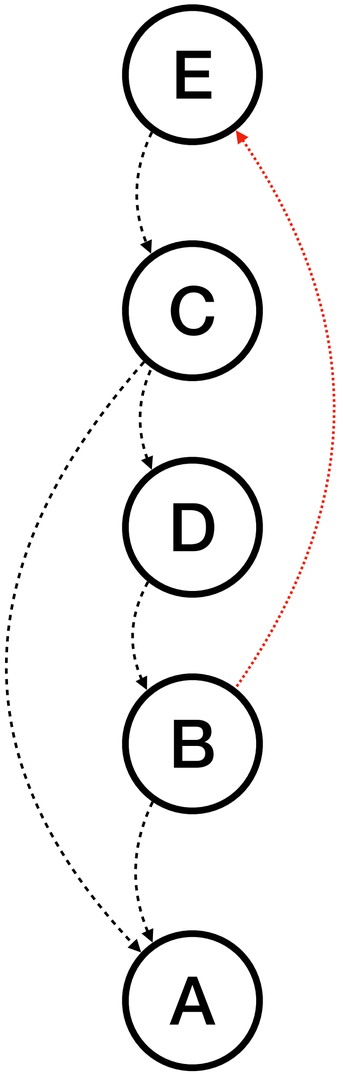



A
B
C
D
E

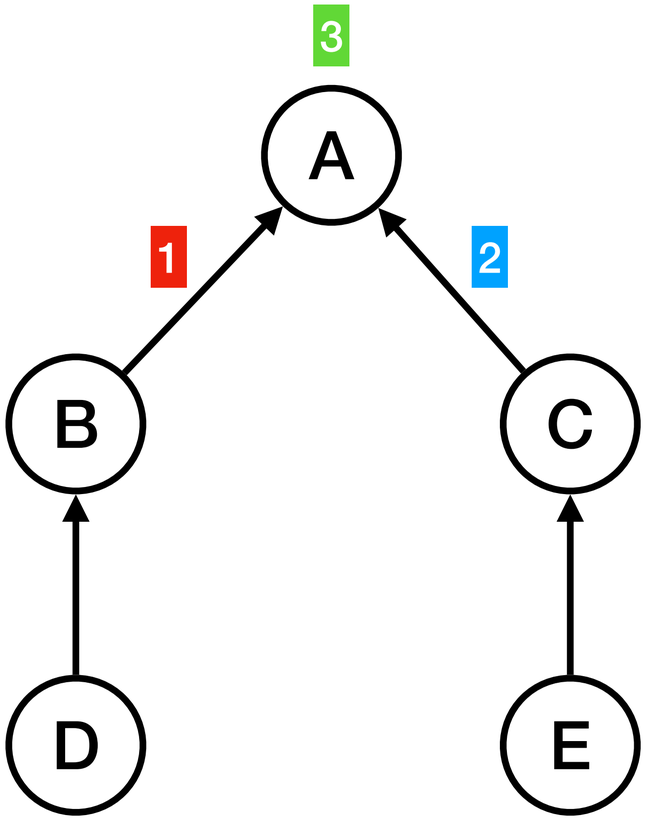
- Post-ordered DFS:

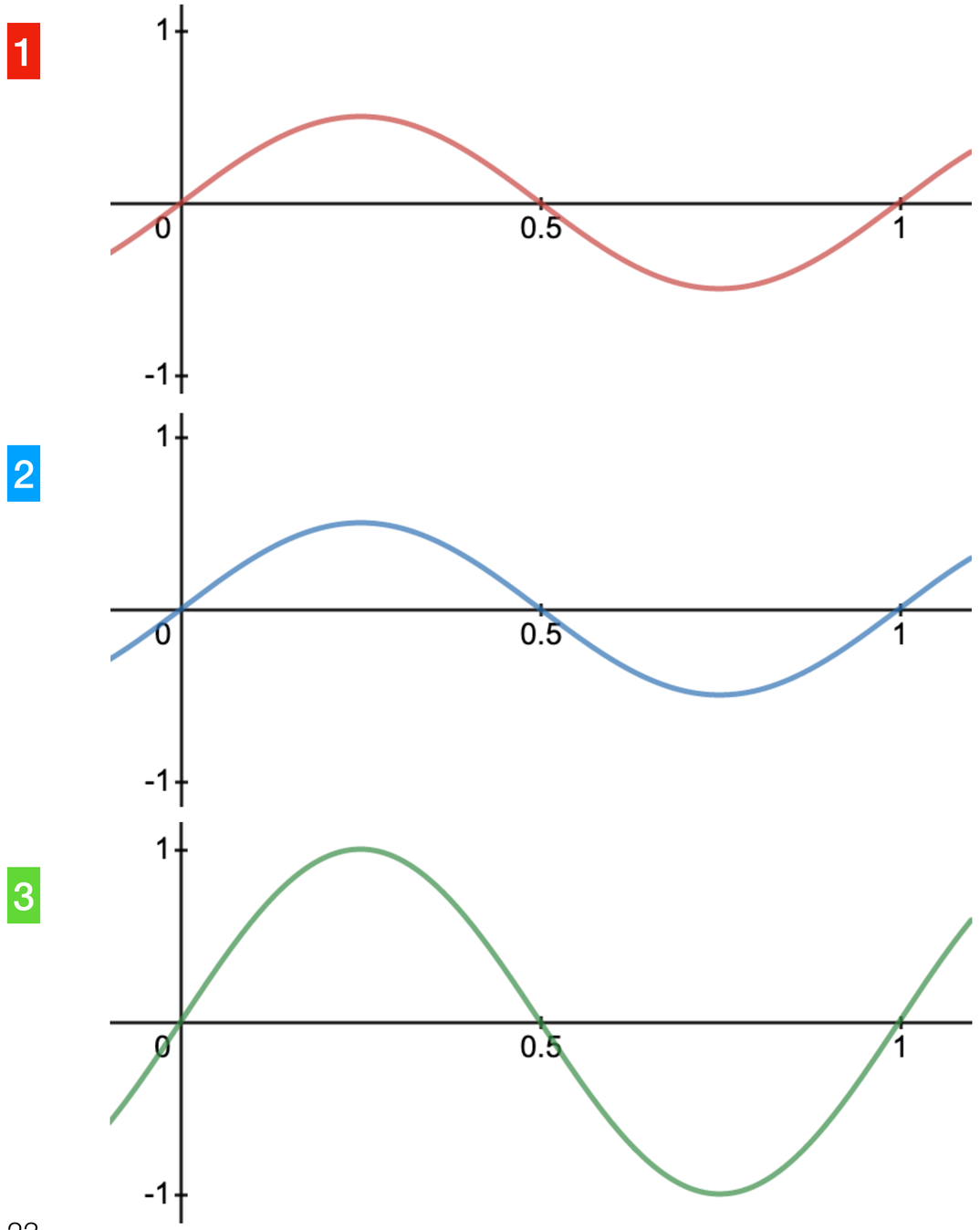


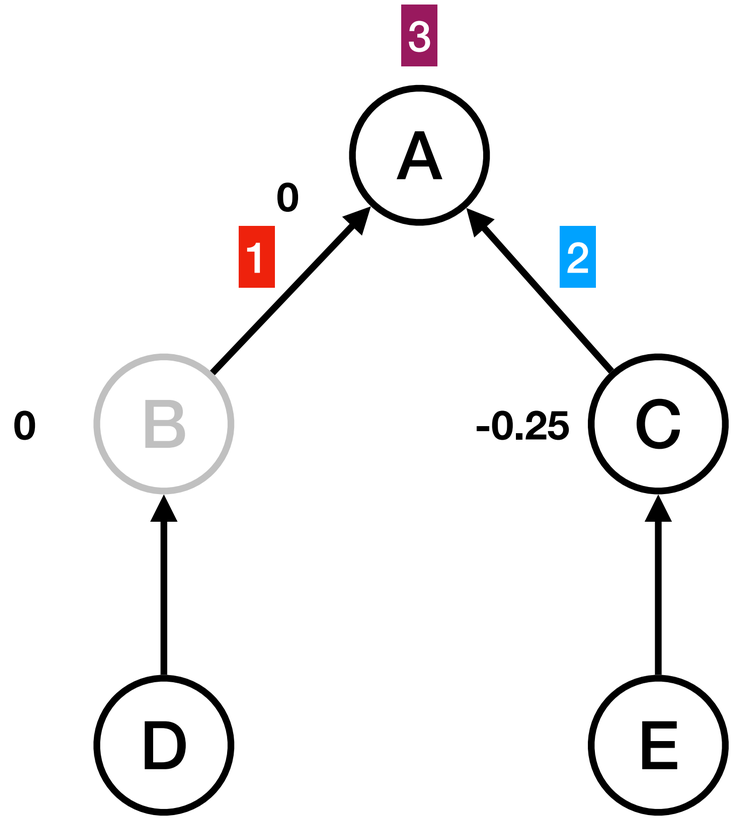
- Post-ordered DFS:
  E
  C
  D
  B
  A

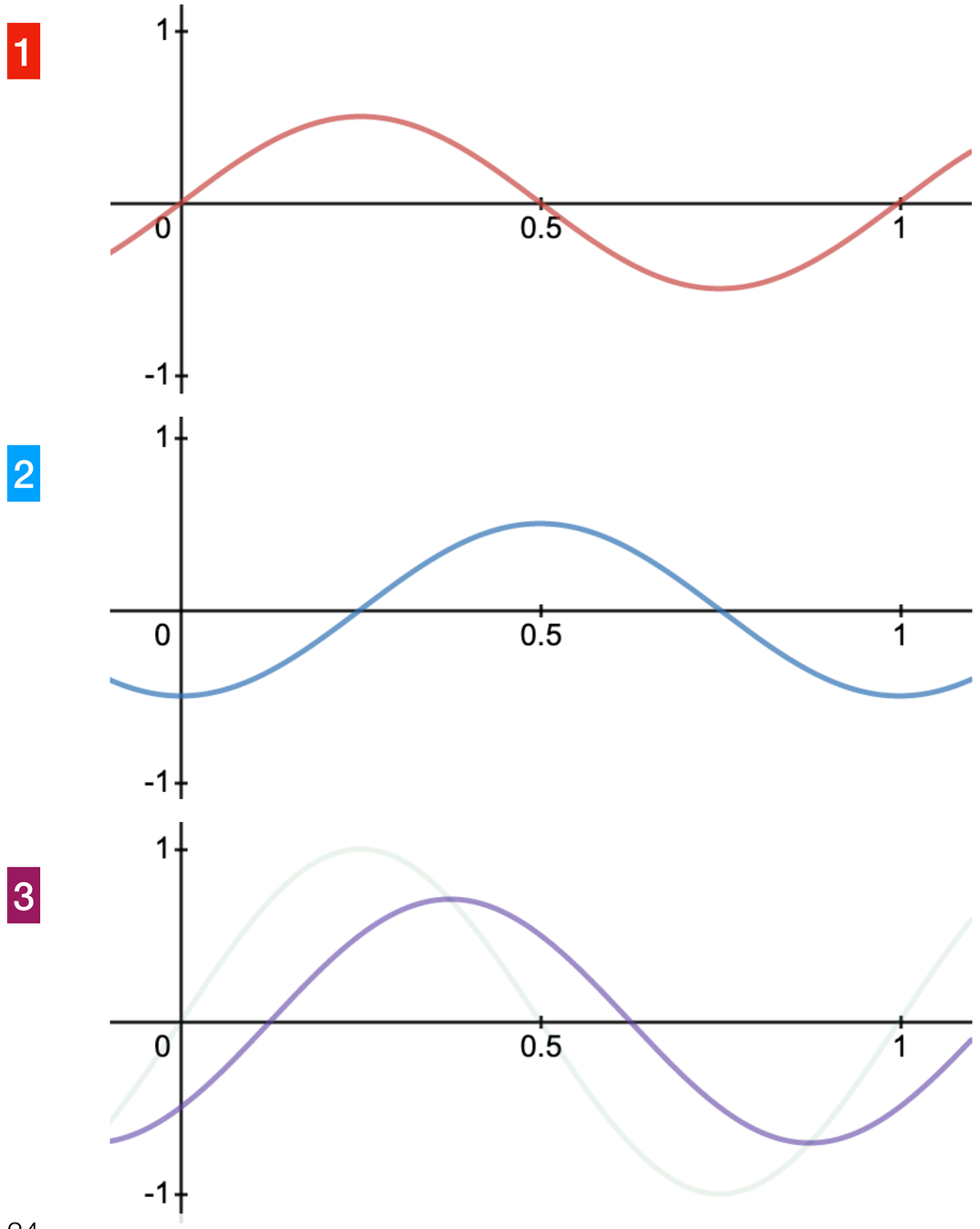


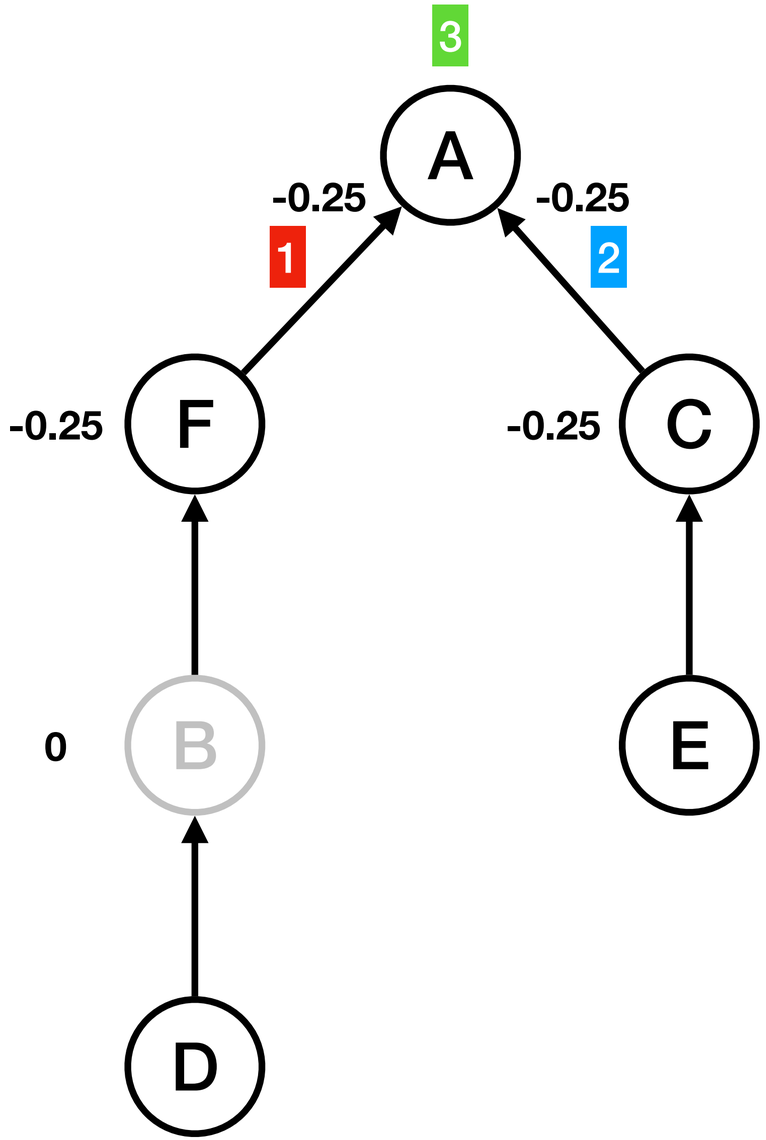
Latency

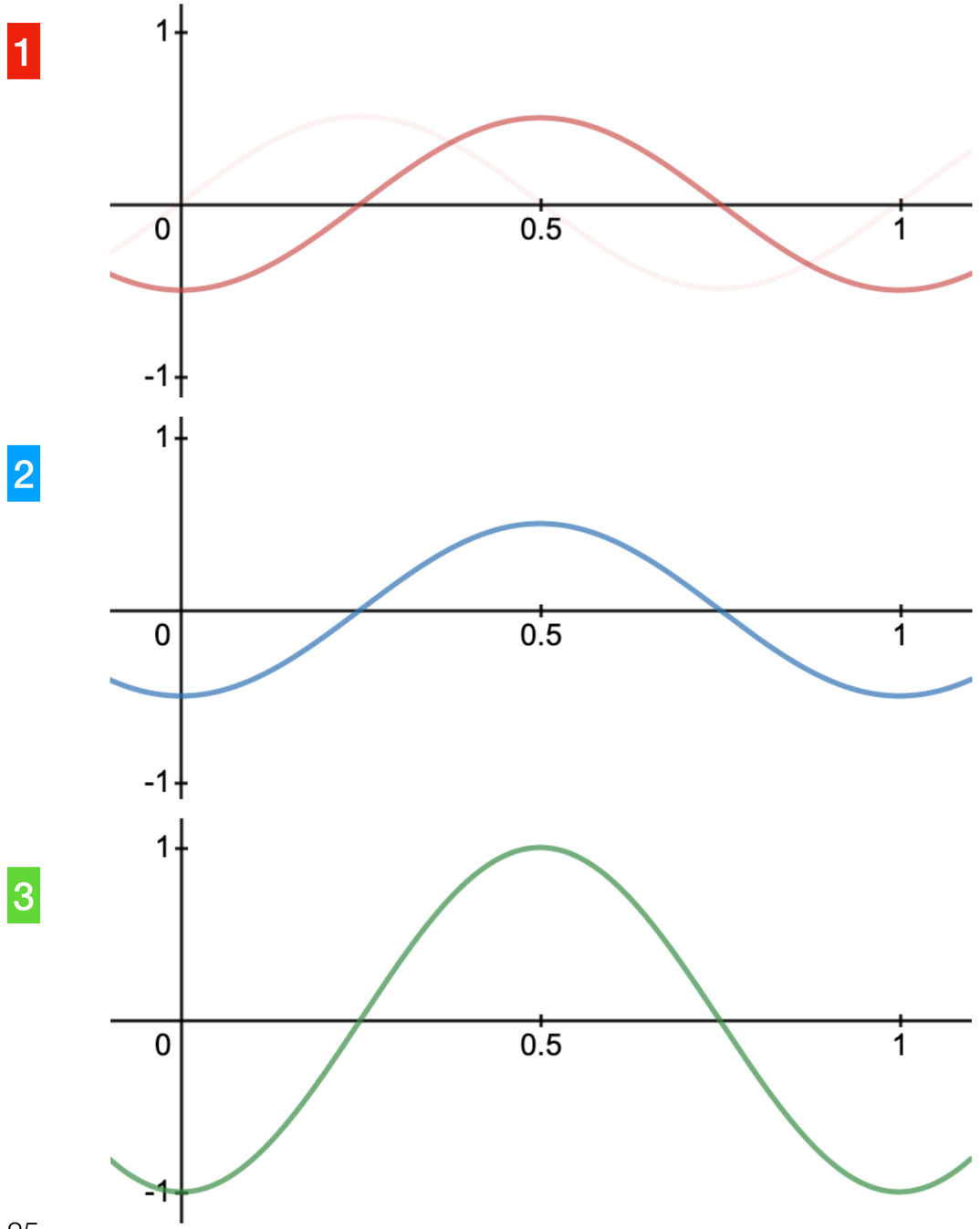

3
A
1
2
B
C
D
E

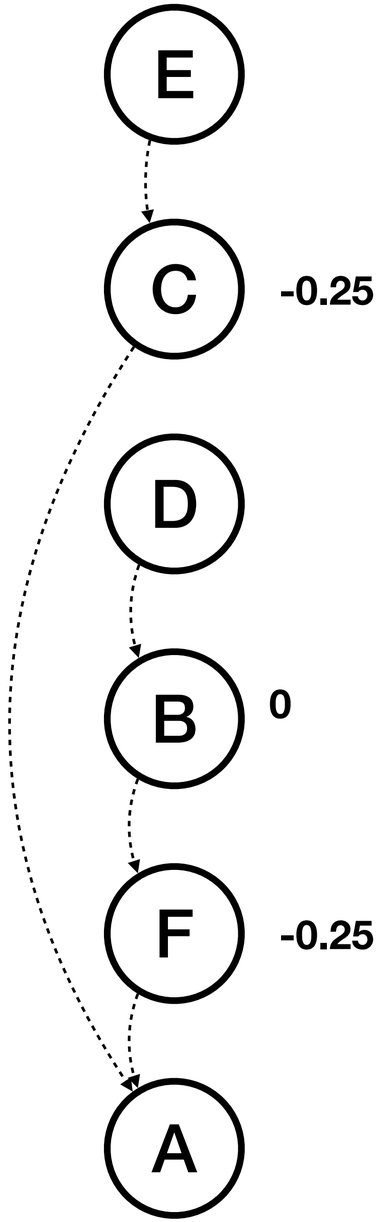

1
1
0
0.5
1
-1
2
1
0
0.5
1
-1
3
1
0
0.5
1
-1


3
A
0
1
2
0
B
-0.25
C
D
E


1
1
0
0.5
1
-1
2
1
0
0.5
1
-1
3
1
0
0.5
1
-1


3
A
-0.25
1
-0.25
2
-0.25
F
-0.25
C
0
B
E
D


1
1
0
0.5
1
-1
2
1
0
0.5
1
-1
3
1
0
0.5
1
-1


E
C
-0.25
D
B
0
F
-0.25
A


Problems…

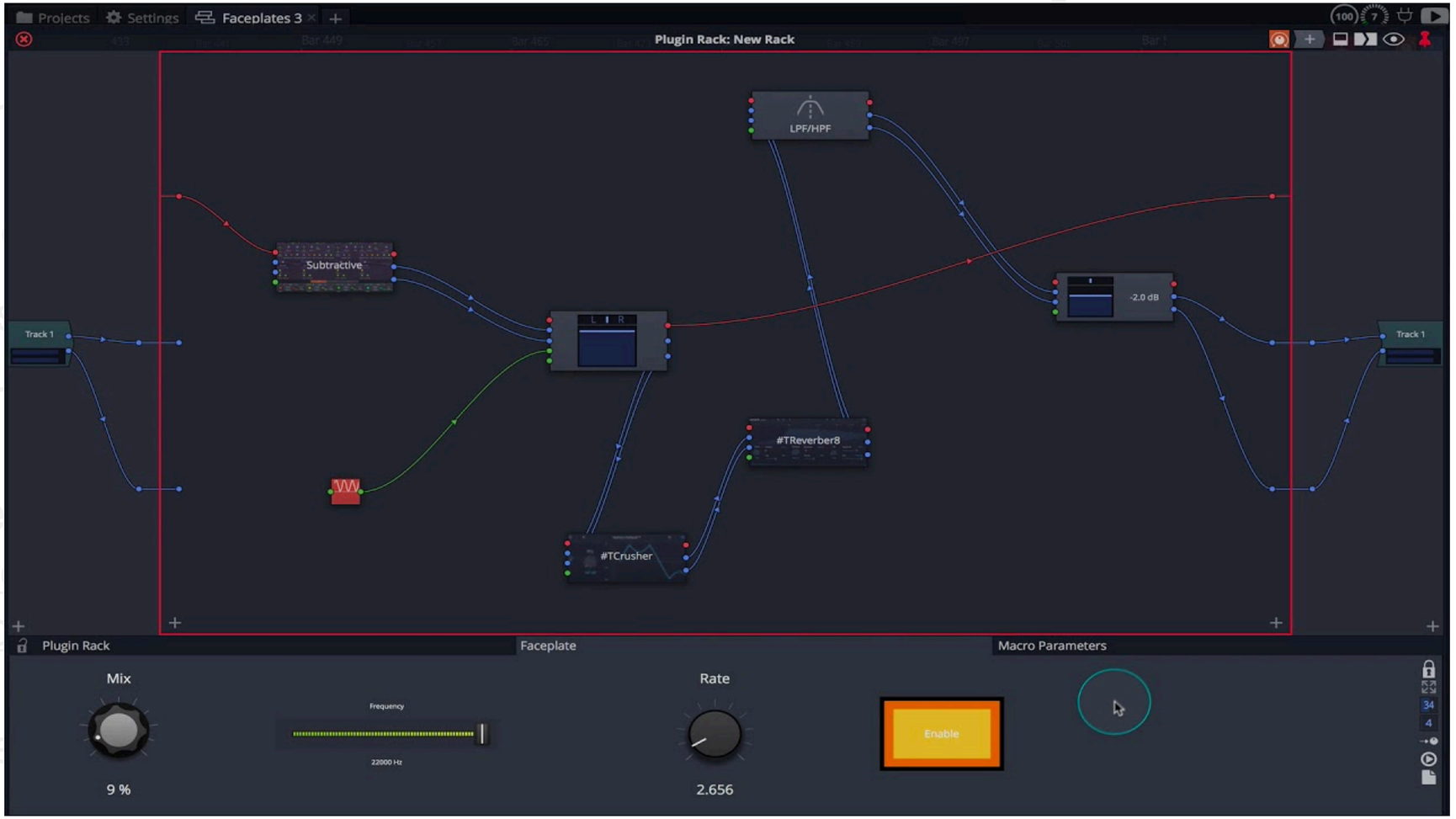
Continuity

- If the topology changes, the graph will need to be rebuilt
- If any nodes have latency, this means they will have a history of previous samples
- If this history is not persisted between graphs, there will be a gap/ inconsistency in playback and hence a glitch
- In order to avoid these discontinuities, any history buffers will need to be persisted between graphs
- This means each node must be uniquely identifiable and the same between graphs

2. Tracktion Graph Library

Tracktion Graph

- New pr
- Replac
  - Fixe
  - Impr
- Can be
  github.
  tracktio
- Will eve
- API is currently in flux



Aims

- Mid-level library with no ties to a specific model
  - *Easy to test*
- Ensure nodes can be processed multi-threaded which scales independently of graph complexity
  - *In theory this should give us the best CPU utilisation*
- Processing can happen in any sized block (up to the maximum prepared for)
  - *Enables reduced aliasing due to fast changing automation and handles looping more accurately*
- Processing in float or double
  - *Although the current engine can sum in doubles, having a complete 64-bit pipeline should provide the most headroom possible*

API: Easy to use, hard to abuse

- Slim API
  - *Avoids complexity and separates concerns*
- Process calls will only ever get the number of channels to fill that they report during initialisation
  - *This should avoid difficult decisions about what to do with extra channels*
- Process calls will always provide empty buffers so nodes can simply "add" in to them
  - *This makes processing simpler. Multiple sources can all just add in to the buffer without clearing it first and if a node doesn't need to process it can simply return. Clearing buffers is generally a quick CPU operation*
- Optimisations are opt-in
  - *Enables a easy to use and safe API whilst not sacrificing performance*

Node Overview

```
class Node
{
public:
    Node() = default;
    virtual ~Node() = default;

    //==============================================================================
    /** Call once after the graph has been constructed to initialise buffers etc. */
    void initialise (const PlaybackInitialisationInfo&);

    /** Call before processing the next block, used to reset the process status. */
    void prepareForNextBlock (juce::Range<int64_t> referenceSampleRange);

    /** Call to process the node, which will in turn call the process method with the
        buffers to fill.
        @param referenceSampleRange The monotonic stream time in samples.
                                    This will be passed to the ProcessContext during the
                                    process callback so nodes can use this to determine file
                                    reading positions etc.
                                    Some nodes may ignore this completely but it should at the
                                    least specify the number to samples to process in this block.
    */
    void process (juce::Range<int64_t> referenceSampleRange);

    /** Returns true if this node has processed and its outputs can be retrieved. */
    bool hasProcessed() const;

    /** Contains the buffers for a processing operation. */
    struct AudioAndMidiBuffer
    {
        juce::dsp::AudioBlock<float> audio;
        tracktion_engine::MidiMessageArray& midi;
    };

    /** Returns the processed audio and MIDI output.
        Must only be called after hasProcessed returns true.
    */
    AudioAndMidiBuffer getProcessedOutput();
```

Implementing a Node Summary

- Declare dependencies (inputs)
- Declare properties
  (Has MIDI, num audio channels, latency, ID)
- Implement initialisation
- Implement pre-fetching
- Implement processing

Node: Public Virtual Methods

```
/** Called after construction to give the node a chance to modify its topology.
    This should return true if any changes were made to the topology as this
    indicates that the method may need to be called again after other nodes have
    had their topology changed.
*/
virtual bool transform (Node& /*rootNode*/) { return false; }

/** Should return all the inputs directly feeding in to this node. */
virtual std::vector<Node*> getDirectInputNodes() { return {}; }

/** Should return the properties of the node.
    This should not be called until after initialise.
*/
virtual NodeProperties getNodeProperties() = 0;

/** Should return true when this node is ready
    This is usually when its input's output buf
*/
virtual bool isReadyToProcess() = 0;
```

```
/** Holds some really basic properties of a node */
struct NodeProperties
{
    bool hasAudio = false;
    bool hasMidi = false;
    int numberOfChannels = 0;
    int latencyNumSamples = 0;
    size_t nodeID = 0;
};
```

Node: Protected Virtual Methods

```
/** Called once before playback begins for each node.
    Use this to allocate buffers etc.
    This step can be used to modify the topology of the graph (i.e. add/remove nodes).
    However, if you do this, you must make sure to call initialise on them so they are
    fully prepared for processing.
*/
virtual void prepareToPlay (const PlaybackInitialisationInfo&) {}

/** Called once on all Nodes before they are processed.
    This can be used to prefetch audio data or update mute statuses etc..
*/
virtual void prefetchBlock (juce::Range<int64_t> /*referenceSampleRange*/) {}

/** Called when the node is to be processed.
    This should add in to the buffers available making sure not to change their size at all.
*/
virtual void process (const ProcessContext&) = 0;
```

```
    /** Struct to de
    struct ProcessC
    {
        juce::Range<
        AudioAndMidi
    };
```

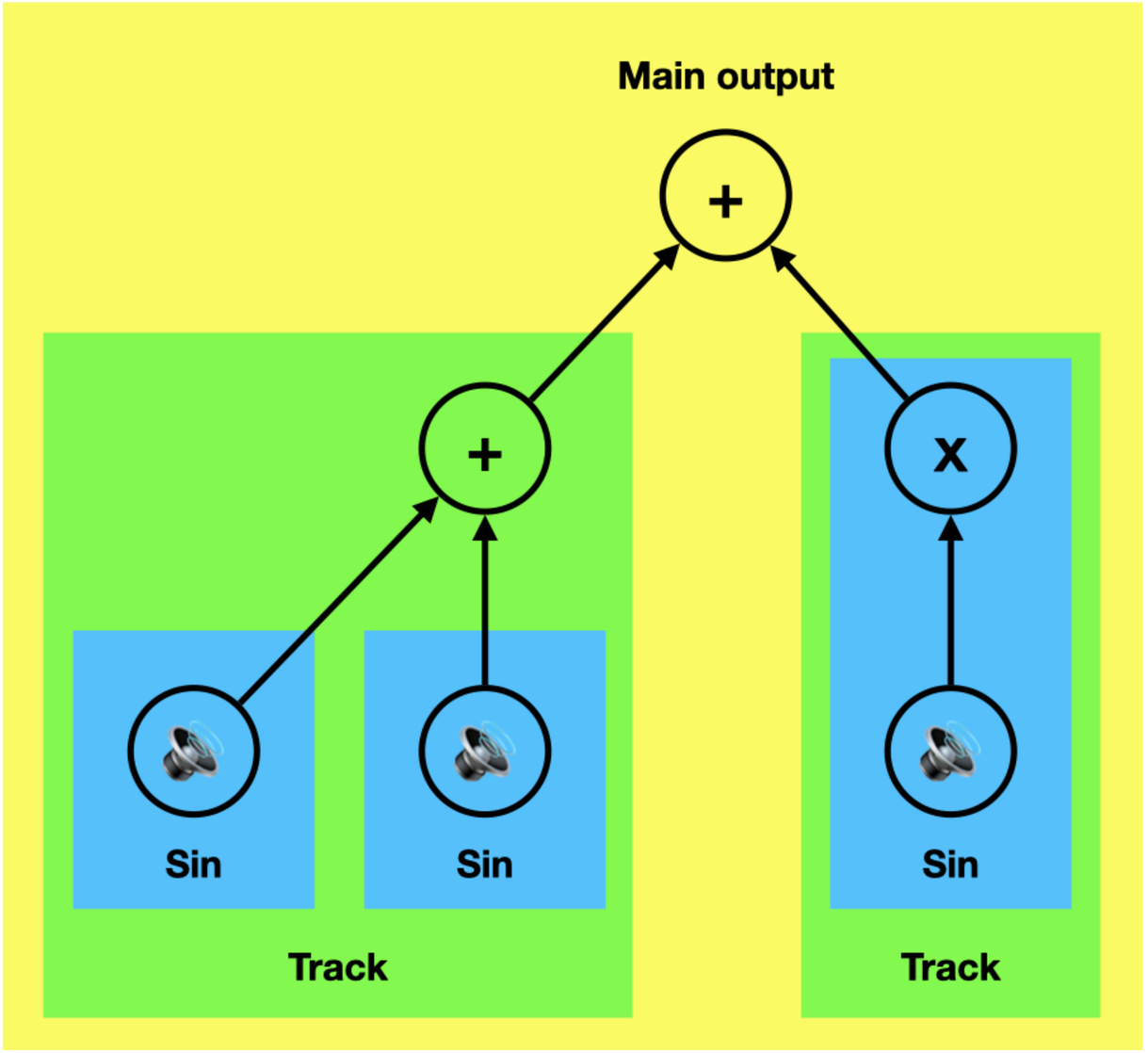
```
struct PlaybackInitialisationInfo
{
    double sampleRate;
    int blockSize;
    Node& rootNode;
    Node* rootNodeToReplace = nullptr;
};
```

Node Summary

- Declare dependencies (inputs)
- Declare properties
  (Has MIDI, num audio channels, latency, ID)
- Implement initialisation
- Implement pre-fetching
- Implement processing

`SinNode` Example

```
class SinNode final : public Node
{
public:
    SinNode (float frequency, int numChannelsToUse)
        : numChannels (numChannelsToUse)
    {
        osc.setFrequency (frequency, true);
    }

    NodeProperties getNodeProperties() override
    {
        NodeProperties props;
        props.hasAudio = true;
        props.hasMidi = false;
        props.numberOfChannels = numChannels;

        return props;
    }

    bool isReadyToProcess() override      { return true; }

    void prepareToPlay (const PlaybackInitialisationInfo& info) override
    {
        osc.prepare ({ double (info.sampleRate), uint32_t (info.blockSize), (uint32_t) numChannels });
    }

    void process (const ProcessContext& pc) override
    {
        auto block = pc.buffers.audio;
        osc.process (juce::dsp::ProcessContextReplacing<float> { block });
    }

private:
    juce::dsp::Oscillator<float> osc { [] (float in) { return std::sin (in); } };
    const int numChannels;
};
```


Main output
+
+
x
Sin
Sin
Sin
Track
Track

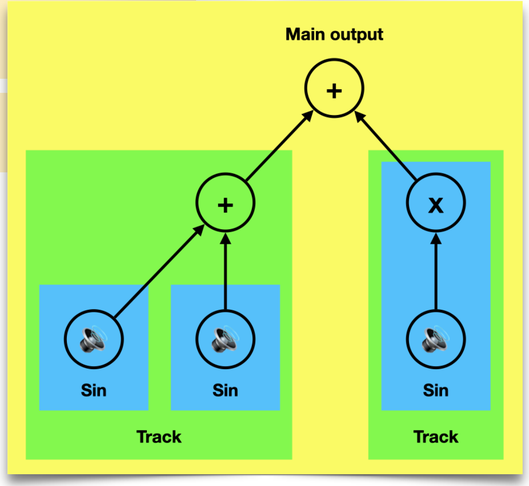
```cpp
// Make track one
std::vector<std::unique_ptr<Node>> trackOneClipNodes;
trackOneClipNodes.push_back (std::make_unique<SinNode> (220.0f, 1));
trackOneClipNodes.push_back (std::make_unique<SinNode> (220.0f, 1));
```

```cpp
auto trackOneNode = std::make_unique<SummingNode> (std::move (trackOneClipNodes));
```

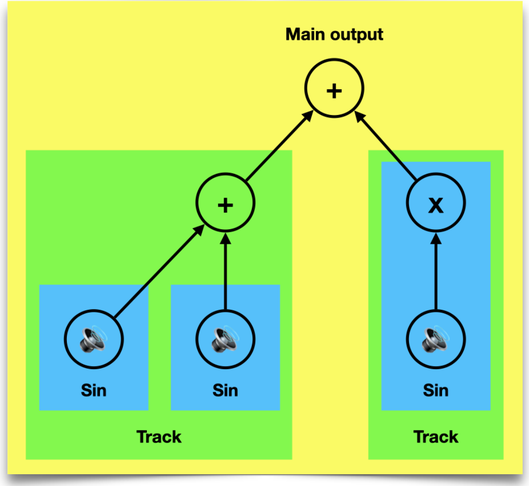
```cpp
// Make track two
auto trackTwoClipNode = std::make_unique<SinNode> (220.0f, 1);
float clipGain = 1.0f;
auto trackTwoNode = std::make_unique<GainNode> (std::move (trackTwoClipNode),
                                                [clipGain] { return clipGain; });
```

```cpp
// Make main output node
std::vector<std::unique_ptr<Node>> trackNodes;
trackNodes.push_back (std::move (trackOneNode));
trackNodes.push_back (std::move (trackTwoNode));
auto mainOutput = std::make_unique<SummingNode> (std::move (trackNodes));
```

```cpp
// Play mainOutput!
```



```
// Make track one
auto trackOneNode = makeSummingNode ({ makeNode<SinNode> (220.0f, 1),
                                       makeNode<SinNode> (220.0f, 1) });

// Make track two
float clipGain = 1.0f;
auto trackTwoNode = makeNode<GainNode> (makeNode<SinNode> (220.0f, 1),
                                        [clipGain] { return clipGain; })

// Make main output node
auto mainOutput = makeSummingNode ({ std::move (trackOneNode),
                                     std::move (trackTwoNode));

// Play mainOutput!
```



Summary of `NodePlayer` Class

- Prepare:
  - Transform
  - Initialise
  - Order Nodes
- Process:
  - Prepare for next block
  - Process

SimpleNodePlayer

```
/**
    Simple player for a Node.
    This iterates all the nodes attempting to process them in a single thread.
*/
class SimpleNodePlayer
{
public:
    /** Creates a player to play a Node. */
    SimpleNodePlayer (std::unique_ptr<Node> nodeToPlay)
        : rootNode (std::move (nodeToPlay))
    {
        assert (rootNode);
    }

    /** Prepares the Node to be played. */
    void prepareToPlay (double sampleRateToUse, int blockSizeToUse)
    {
        orderedNodes = node_player_utils::prepareToPlay (rootNode.get(), nullptr, sampleRateToUse, blockSizeToUse);
    }

    /** Processes a block of audio and MIDI data. */
    void process (const Node::ProcessContext&);

private:
    std::unique_ptr<Node> rootNode;
    std::vector<Node*> orderedNodes;
};
```

prepareToPlay

```
namespace node_player_utils
{
    /** Prepares a specific Node to be played and returns all the Nodes. */
    static std::vector<Node*> prepareToPlay (Node* node, Node* oldNode,
                                             double sampleRate, int blockSize)
    {
        if (node == nullptr)
            return {};

        // First give the Nodes a chance to transform
        transformNodes (*node);

        // Then find all the nodes as it might have changed after initialisation
        auto orderedNodes = tracktion_graph::getNodes (*node, tracktion_graph::VertexOrdering::postordering);

        // Next, initialise all the nodes, this will call prepareToPlay on
        const PlaybackInitialisationInfo info { sampleRate, blockSize, *node, oldNode };

        for (auto node : orderedNodes)
            node->initialise (info);

        // Finally return the Nodes in playback order
        return orderedNodes;
    }
}
```

SimpleNodePlayer::Process

```
void process (const Node::ProcessContext& pc)
{
    // Prepare all nodes for the next block
    for (auto node : orderedNodes)
        node->prepareForNextBlock (pc.referenceSampleRange);

    // Then process them all in sequence
    for (auto node : orderedNodes)
        node->process (pc.referenceSampleRange);

    // Finally copy the output from the root Node to our player buffers
    auto output = rootNode->getProcessedOutput();
    const size_t numAudioChannels = std::min (output.audio.getNumChannels(),
                                              pc.buffers.audio.getNumChannels());

    if (numAudioChannels > 0)
        pc.buffers.audio.getSubsetChannelBlock (0, numAudioChannels)
                        .add (output.audio.getSubsetChannelBlock (0, numAudioChannels));

    pc.buffers.midi.mergeFrom (output.midi);
}
```

Summary of `NodePlayer` Class

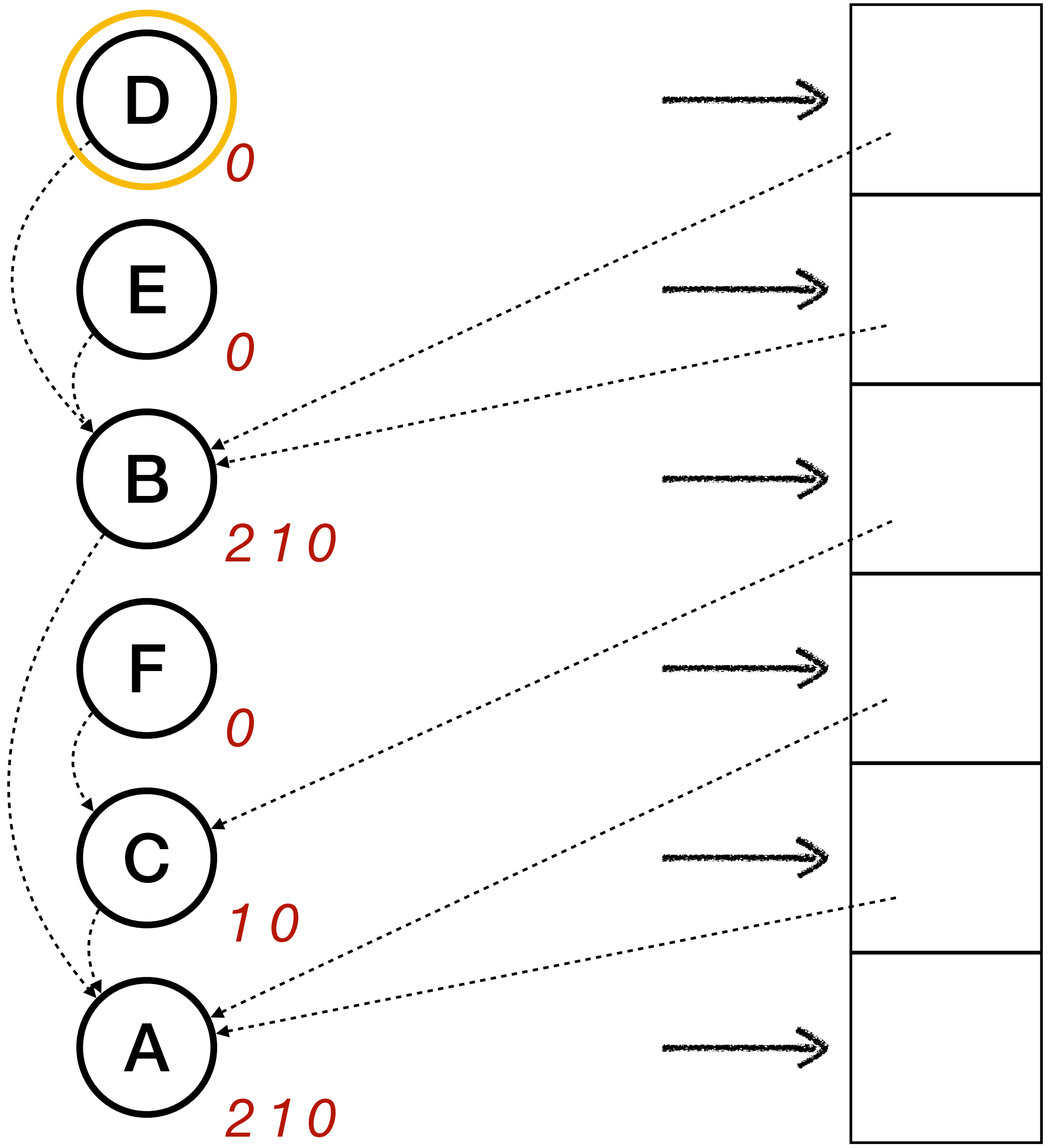
- Prepare:
  - Transform
  - Initialise
  - Order Nodes
- Process:
  - Prepare for next block
  - Process

Advanced `Node` Players

- `NodePlayer` class
  - Contains a `PlayHead`
  - Ability to set a new `Node` to play
  - Maintains continuity
  - Ability to change sample rate/block sizes
- Multi-threaded players
  - Uses multiple threads to process `Nodes` concurrently
  - Many possible algorithms…

Post-ordered DFS:

Ready to be processed:



Multi-threaded Strategies

- Multi-threading overview:
  - Single RT audio thread initialising and processing FIFO
  - Multiple “worker” threads processing the FIFO
- Questions:
  - How many threads do you start?
  - Trade-off between CPU use and throughput
- Fully real-time implementation means no system calls (locks, CVs, events etc.)
  - Requires worker threads spinning on the FIFO waiting for available `Nodes`
  - Worker threads can use CPU pause instructions
  - Or brief yields/sleeps - very difficult to get them to wake up at the right time to process
- Non-real-time solution can use condition variables to sleep/wake worker threads
  - NOT REAL-TIME SAFE!

Uses

- Tracktion Engine:
  - EditNodeBuilder.h/cpp files
  - Takes an `Edit` and builds a graph of `Nodes` to process it
  - Completely separates model from processing
- Tracktion Engine `Racks`:
  - `RackNodeBuilder`
  - Takes a `tracktion_engine::Rack` and returns a `Node` to process it
- Future:
  - Intermediate Rack-like format to generate a graph
  - `juce::AudioProcessGraph` -> intermediate format -> `Node`

Summary

- Graph representations of audio
- Ordering produces processable DAGs
- Common audio problems such as latency and continuity
- Tracktion Graph library implements these ideas
- Scalable way to process audio applications

Introducing Tracktion Graph

David Rowland

@drowaudio

Slides/video:

github.com/drowaudio/presentations

tracktion_graph code:

github.com/Tracktion/tracktion_engine/tree/tracktion_graph/modules/tracktion_graph