Graph Definitions:

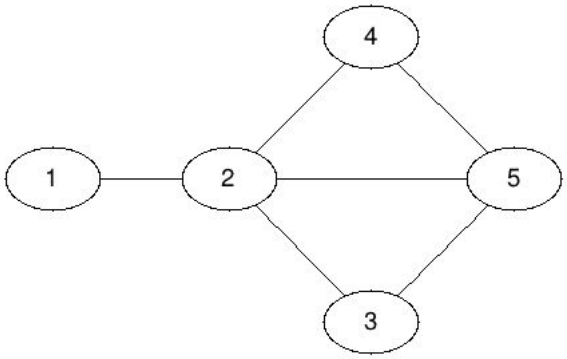
A ***graph*** `G = (V, E)`, where `V` is the set of vertices (or nodes) and `E` is a set of edges connecting vertices.

Note on sets: A ***set*** is an unordered collection of elements that uses braces to denote the elements within the set. For example, the set `H = {a, h, r, b }` is a set consisting of elements a, h, r, and b. The order of the elements in the set does not matter, so the set could also have been written as `H = { r, h, b, a }`.

A ***vertex*** is an element within the graph that represents some data or information. For example, a vertex could represent a location for an airport or a Boolean logic gate in a digital circuit.

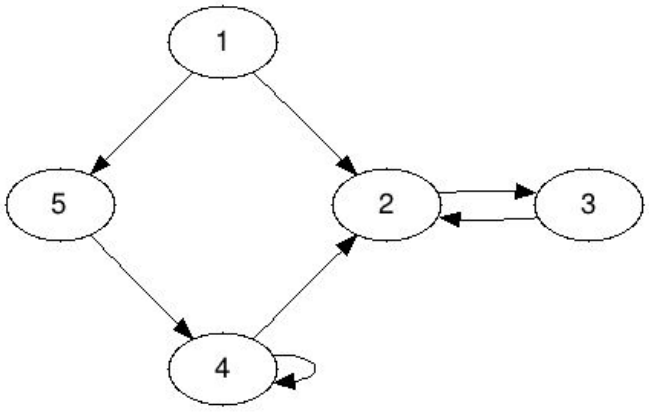
An ***edge*** `(u, v)` specifies a connection between vertex `u` and vertex `v`.

An ***undirected graph*** is a graph that contains edges that specify connection between vertices but do not imply a predecessor/successor relationship. Thus, in an undirected graph, and edge `(u, v)` is equivalent to an edge `(v, u)`.



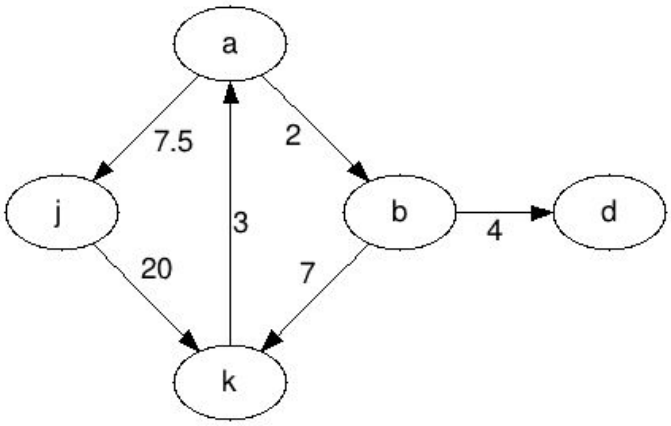

```
G = (V, E)
V = { 1, 2, 3, 4, 5 }
E = { (1, 2), (2, 4), (4, 5), (5, 2), (3, 2), (3, 5) }
```

A ***directed graph*** is a graph that contains edges that specify both a connection from a source vertex to a destination (or sink) vertex. A directed edge `(u, v)` is an edge from vertex `u` to vertex `v`.




```
G = (V, E)
V = { 1, 2, 3, 4, 5 }
E = { (1, 2), (1, 5), (5, 4), (2, 3), (3, 2), (4, 2), (4, 4) }
```

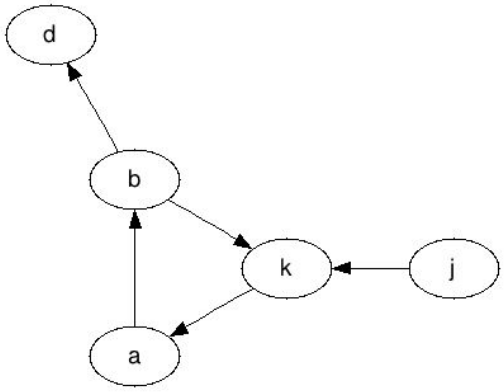
An ***weighted graph*** is a in which all edges are assigned an edge weight. The ***edge weight*** represents a numerical value specific to the problem being represented using the graph. An edge weight `w(u, v)` is the weight of an edge `(u, v)`.




```
G = (V, E)
V = { b, a, d, k, j }
E = { (a, j), (a, b), (b, k), (k, a), (j, k), (b, d) }
w(a, j)=7.5, w(a, b)=2, w(b, k)=7, w(k, a)=3, w(j, k)=20, w(b, d)=4
```

An ***unweighted graph*** is a graph in which all edges do not have an edge weighted. In other words, all edges has a uniform weight. An unweighted graph can be represented as a weighted graph in which all edges `(u, v)` have a weight `w(u, v) = 1`.

A ***path*** is sequence of vertices traversed by following the edges between vertices. A path from vertex `u` to vertex `v` is a sequence of vertices $\langle v_0, v_1, \ldots, v_{k-1}, v_k \rangle$, where $v_0 = u$, $v_k = v$ and for all $i = 1$ to k the edge $(v_{i-1}, v_i) \in E$. For example, a valid path in the directed graph below is $\langle k, a, b, d \rangle$.



A ***cycle*** is path in a graph in which the start vertex of the path and the end vertex of the path are the same. For example, the above directed graph contains a cycle $\langle b, k, a, b \rangle$.

A ***cyclic graph*** is a graph that contains at least one cycle. An ***acyclic graph*** is a graph that does not contain a cycle.

Graph Algorithms:

Breadth First Search (BFS)

A breadth first search of an unweighted graph will traverse the vertices of a graph starting from a start vertex to compute the distance from the start vertex to all vertex that are reachable from the start vertex. The computed distance is the *minimum* number of edges that must be followed to reach each vertex from the start vertex.

In other words, given a graph `G = (V, E)` and start vertex s, compute the distance from the vertex `s` to to all vertices `v ∈ V - {s}`.

The BFS algorithm will utilize a ***queue*** data structure for keeping track of the vertices that have been found so far in the graph. A queue is a data structure that only allows elements to the added (on ***enqueued***) to the back of a queue, and removed from the front of the queue (***dequeued***). Commonly, the underlying implementation for a queue is a linked lists. The enqueue operation can be implemented by inserting an element after the tail of the list, and the dequeue can be implemented by removing the element from the head of the list. The Standard Template Library (STL) in C++ has a `queue` class that implements the functionality of a queue. The common operations for the `queue` class are `push_back()`, `pop_front()`, `front()`, and `back()`.
To keep track of the status of a vertex, we will assign a color to the vertices, where *white* indicates a vertex has not been found or visited, *gray* indicates a vertex has been found but not visited, and *black* indicates a vertex has been visited.

BFS (Graph `G`, Vertex* `s`)

1. for each vertex `u` ∈ `G.vertices`
    a. `u->color` = white
    b. `u->distance` = ∞
2. `s->color` = gray
3. `s->distance` = 0
4. `Q`.enqueue(`s`)
5. while `Q` is not empty
    a. `u` = `Q`.dequeue()
    b. for each edge `e` ∈ `u.edges`
        i. `v` = `e->adjVertex`
        ii. if `v->color` == `white`
            1. `v->color` = `gray`
            2. `v->distance` = `u->distance` + 1
            3. `Q`.enqueue(`v`)
    c. `u->color` = black

Depth First Search (DFS)

A depth first search of a graph will traverse the vertices of a graph stby visiting each newly discovered vertex as soon as the vertex is discovered. The depth first search algorithm does not require a start vertex as the algorithm will ensure all vertices in the graph are visited once. However, the DFS algorithm can be easily adapted to perform a depth first search given a start vertex. Recursion is a natural and straightforward way to implement the DFS algorithm.

The DFS algorithm provides a generic framework for visiting vertices within a graph in a depth-first order. Depending of the specific problem you are trying to solve, the values that are computed for each vertex as part of the depth-first search can be modified. The structure and the recursive implementation of the DFS algorithm are the important points to understand about the DFS algorithm. To illustrate the behavior of the DFS algorithm, we will compute a ***discovery*** time and ***finishing*** time for each vertex. A global ***time*** value will be maintained that represents the steps of the algorithm. Every time we discover a new

vertex or finish processing a vertex, we will increment the time. The discovery time is the time step in which a node was first discovered, and the finishing time is the time the algorithm finishing visiting all adjacent vertices from the current vertex.

In other words, given a graph `G = (V, E)` compute the discovery time and finishing time for each vertex `v ∈ V`.

We will again use colors to keep track of the status of a vertex, where *white* indicates a vertex has not been found or visited, *gray* indicates a vertex has been found but not visited, and *black* indicates a vertex has been visited.

DFS (Graph `G`)

1. for each vertex `u ∈ G.vertices`
    a. `u->color` = white
2. `time` = 0
3. for each vertex `u ∈ G.vertices`
    a. if `u->color` == `white`
        i. DFSVisit(`G`, `u`)

DFSVisit (Graph `G`, Vertex* `u`)

1. `u->color` = gray
2. `u->discovery` = time
3. `time`++
4. for each edge `e ∈ u.edges`
    a. `v` = `e->adjVertex`
    b. if `v->color` == `white`
        i. DFSVisit(`G`, `v`)
5. `u->color` = black
6. `u->finishing` = time
7. `time`++

Shortest Path Algorithms

Shortest path algorithms are utilized to calculate the shortest path between vertices in a graph. Many variants of the shortest path problem exists (e.g., single destination, single source, all pairs). We exam two single-source shortest path algorithms.

Dijkstra's Single-Source Shortest Path Algorithm

Dijkstra's shortest path algorithm will efficiently compute the shortest path from a start vertex to all other vertices within the graph. Dijkstra's algorithm assumes the input graph is a weighted graph in which all edges has non-negative edge weights (i.e., edges must be greater than or equal to 0).

Dijkstra's algorithm will compute both a distance to each vertex and a predecessor for each vertex. The predecessor is a pointer to the previous vertex within the shortest path.

Given a graph `G = (V, E)` and a start vertex `s`, compute the shortest path from vertex `s` to each vertex `v ∈ V`.

Dijkstra's shortest path algorithm will utilize a ***priority queue*** data structure for keeping track of the vertices that have been found so far in the graph. A priority queue is a data structure that allows elements to the added (on ***enqueued***) to the queue, and removed from the queue based upon a numerical priority (***extracted***). In Dijkstra's shortest path algorithm, the priority used to extract a vertex from the priority queue is the current distance for the vertex. Specifically, for each iteration of the algorithm, the vertex with the smallest distance will be extracted from the priority queue.

DijkstraShortestPath (Graph `G`, Vertex* `s`)

1. for each vertex `u ∈ G.vertices`

    a. `u->distance = ∞`
    b. `u->pred = NULL`
2. `s->distance = 0`
3. for each vertex `u ∈ G.vertices`
    a. `PQ.enqueue(u)`
4. while `PQ` is not empty
    a. `u = Q.extract_min()`
    b. for each edge `e ∈ u.edges`
        i. Relax(`u`, `e`)

Relax (Vertex* u, Edge* e)

1. `v = e->adjVertex`
2. if `v->distance > (u->distance + e->weight)`
    a. `v->distance = u->distance + e->weight`
    b. `v->pred = u`

The runtime complexity of Dijkstra's shortest path algorithm depends on how the priority queue is implemented. Using a simple list implementation that requires the list to the searched each time to find the vertex with the smallest vertex (this has a complexity of O(V)), the runtime complexity for Dijkstra's shortest path algorithm is $O(V^2)$. Using other data structures to implement the priority queue (e.g. heaps), the runtime complexity can be improved to O((V+E)log V) and even O(VlgV + E).

Bellman-Ford Single-Source Shortest Path Algorithm

The Bellman-Ford shortest path algorithm is an extension to Dijkstra's shortest path algorithm to support graphs with negative edge weights. While the graph can have negative edge weights, if the graph contains a negative edge weight cycle, it is not possible to find the shortest path. A ***negative edge weight cycle*** is a cycle in which the sum of all edge weights is negative. In such a case, each time the cycle is traversed, the total distance will decrease. The Bellman-Ford shortest path algorithm will detect if a negative edge weight cycle exists.

Given a graph `G = (V, E)` and a start vertex `s`, compute the shortest path from vertex `s` to each vertex `v ∈ V`. Return `true` is no negative edge weight cycles exists, and return `false` otherwise.

bool BellmanFordShortestPath (Graph `G`, Vertex* `s`)

1. for each vertex `u ∈ G.vertices`
    a. `u->distance = ∞`
    b. `u->pred = NULL`
2. `s->distance = 0`
3. for `i = 0` to `G.vertices.size() - 1`
    a. for each vertex `u ∈ G.vertices`
        i. for each edge `e ∈ u.edges`
            1. Relax(`u`, `e`)
4. for each vertex `u ∈ G.vertices`
    a. for each edge `e ∈ u.edges`
        i. `v = e->adjVertex`
        ii. if `v->distance > (u->distance + e->weight)`
            1. return `false`
5. return `true`

The runtime complexity of the Bellman-Ford O(VE).

Topological Sort

A topological sort of a directed acyclic graph `G = (V, E)` is an ordering of vertices such that if G contains an edge `(u, v)` then u appears before v in the ordering. The topological sort algorithm is similar to the DFS algorithm. When a vertex is finished, the vertex is inserted at the front of the list.

Topological Sort (Graph `G`, List `L`)

1. for each vertex `u` ∈ `G.vertices`
    a. `u->color` = white
2. for each vertex `u` ∈ `G.vertices`
    a. if `u->color` == `white`
        i. TSVisit(`G`, `L`, `u`)

TSVisit (Graph `G`, List `L`, Vertex* `u`)

1. `u->color` = gray
2. for each edge `e` ∈ `u.edges`
    a. `v` = `e->adjVertex`
    b. if `v->color` == `white`
        i. TSVisit(`G`, `L`, `v`)
3. `u->color` = black
4. `L`.insert_front(`u`)

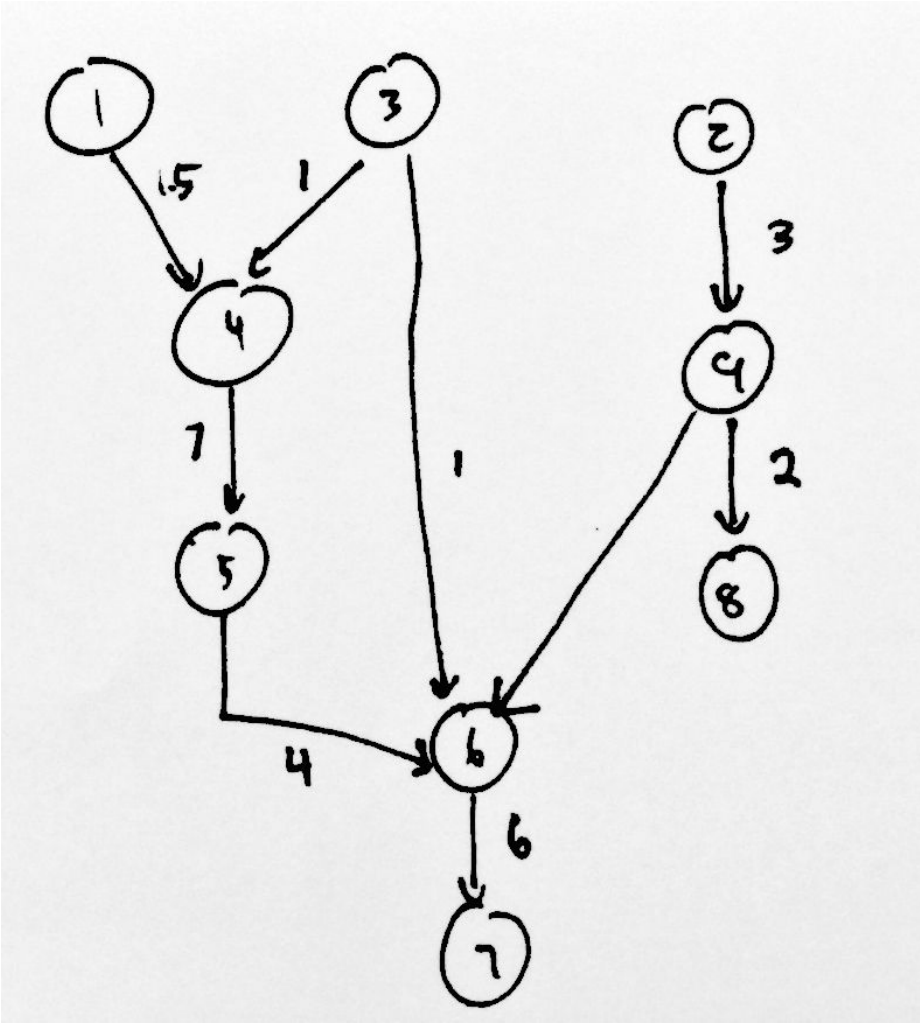
Longest Path in Directed Acyclic Graph

The longest path with a directed acyclic graph (DAG) can be computed by first determining a topological sort of the vertices and then searching for longer paths by examining the vertices in that order.

LongestPathDAG (Graph G)

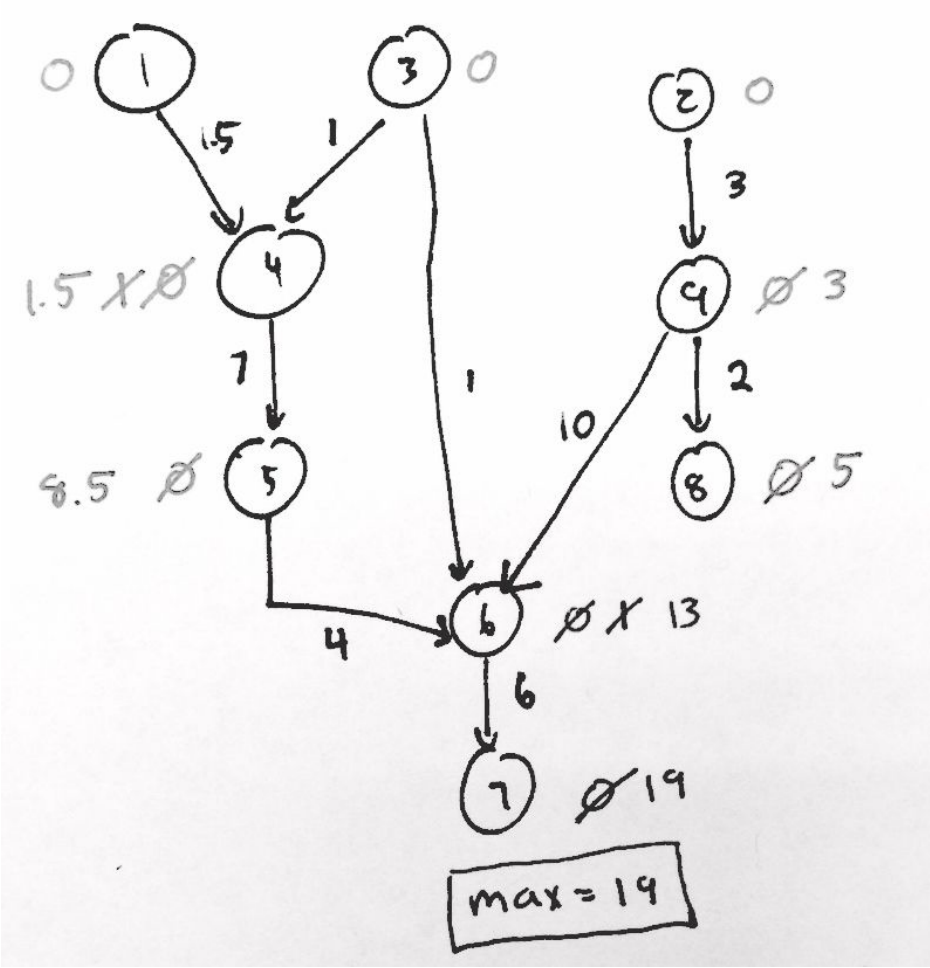
1. TopologicalSort(G, L)
2. for each vertex `u` ∈ `G.vertices`
    a. `u->dist` = 0
3. for each vertex `u` ∈ L
    a. for each edge `e` ∈ `u.edges`
        i. `v` = `e->adjVertex`
        ii. if (`u->dist` + `e->weight`) > `v->dist`
            1. `v->dist` = `u->dist` + `e->weight`
            2. `v->pred` = `u`
4. `max` = 0
5. for each vertex `u` ∈ `G.vertices`
    a. if `u->dist` > `max`
        i. `max` = `u->dist`

For example, consider the following DAG.



A topological sort for the above graph resulting the ordering 3, 2, 9, 8, 1, 4, 5, 6, 7.

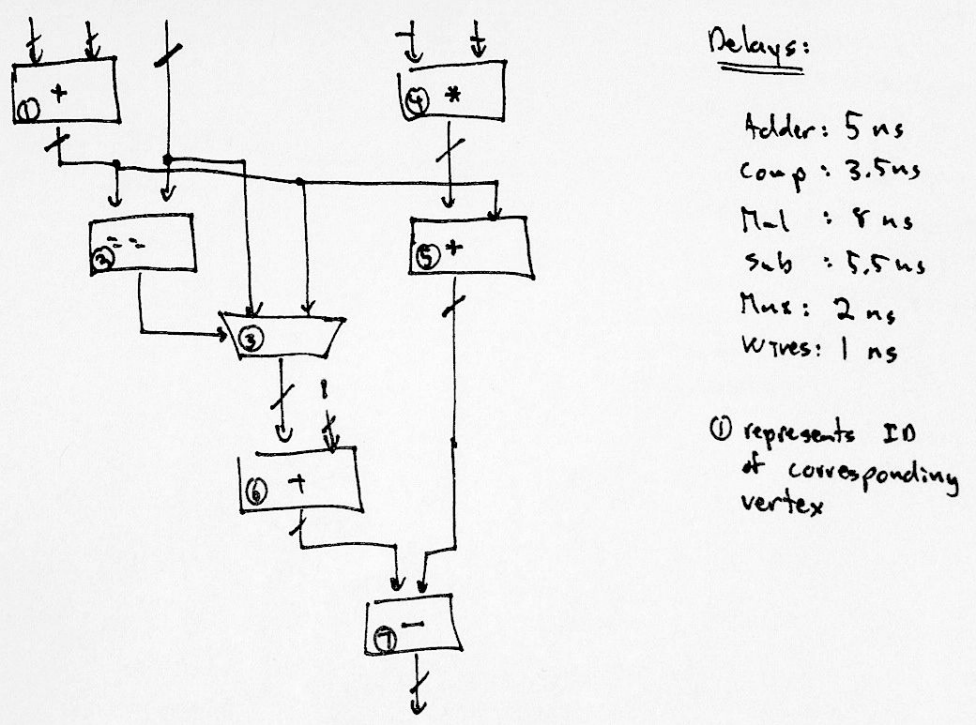
The following shows the result of the LongestPathDAG algorithm for that graph.



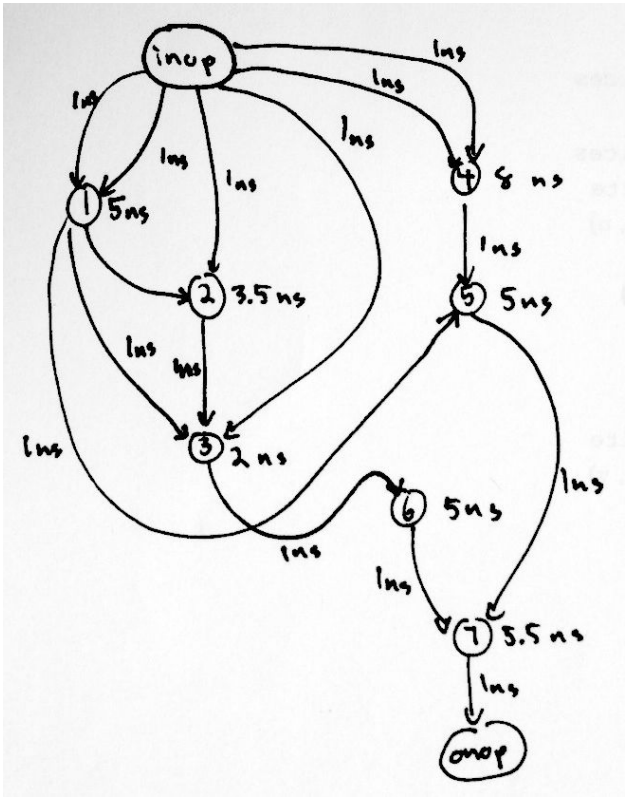
Computing Critical Path for Combinational Circuits

A combinational circuit can be represented using a graph, in which vertices represent logic elements (or datapath components) and edges represent wires (or nets) connecting the components together. Because delays in circuits come from both the combinational elements and wires connecting those elements, a weight can be associated with both the vertices (i.e., combinational component delay) and the edges (i.e., wire delay).

For example, consider the following circuit with the specified delays.



The circuit can be represented as:



The `inop` (input no operation) and `onop` (output no operation) vertices are additional vertices added to the graph to allow the delay of the wires for the inputs to and outputs from the circuit to be included.

Critical Path in Datapaths

To compute the critical path for a datapath, we need to find the longest delay between any two register. This can be accomplished by modifying the TopologicalSort and LongestPathDAG algorithms to calculate the longest path from a starting register to all register that can be reached through combinational components.

The following algorithms assume that each vertex maintains a type that define the ***type*** of component, with register marked as `reg` type. Additionally, each register has a ***weight*** (i.e., delay) that represents either the setup time for registers or the combinational delay of non-sequential components.

RegToRegTopologicalSort (Graph `G`, List `L`, Vertex* `u`)

1. for each vertex `u` ∈ `G.vertices`
    a. `u->color` = white
2. RegToRegTSVisit(`G`, `L`, `u`)

RegToRegTSVisit (Graph `G`, List `L`, Vertex* `u`)

5. `u->color` = gray
6. for each edge `e` ∈ `u.edges`
    a. `v` = `e->adjVertex`
    b. if `v->color` == `white` and `v->type` != `reg`
        i. RegToRegTSVisit(`G`, `L`, `v`)
    c. else if `v->color` == `white` and `v->type` == `reg`
        i. `v->color` = black
        ii. `L`.insert_front(`v`)
7. `u->color` = black
8. `L`.insert_front(`u`)

RegToRegLongestPath (Graph `G`)

1. `max` = 0
2. for each vertex `u` ∈ `G.vertices`
    a. if `u->type` == `reg`
        i. RegToRegTopologicalSort(G, L, u)
        ii. for each vertex `v` ∈ L
            1. `v->dist` = 0
        iii. for each vertex `v` ∈ L
            1. if (`v->type` == `reg` and `v` == `u`) or `v->type` != `reg`
                a. for each edge `e` ∈ `v.edges`
                    i. `w` = `e->adjVertex`
                    ii. if (`v->dist` + `e->weight` + `w->weight`) > `w->dist`
                        1. `w->dist` = `v->dist` + `e->weight` + `w->weight`
        iv. for each vertex `v` ∈ L
            1. if `v->type` == `reg` and `v->dist` > `max`
                a. `max` = `v->dist`