Task Scheduling

Algorithms and Data Structures for Programming 4M016

Nacef Ben Mansour, Ali Vincent Al-Azem
Sorbonne Université

January 2024

Contents

1 Graph Theory

1.1 General Introduction

In this project we are given a number of different tasks that have dependencies between themselves. Our first goal is to create a graph data structure to represent theses tasks and their dependencies. Let's begin by defining a graph.

Definition 1.1. *A graph, G, is formed by a pair G = (V, E), where V is a set of elements called vertices, and E are a set of paired vertices called edges.*

Definition 1.2. *Two vertices $x, y \in V$ are adjacent if they form an edge, i.e., $\{x, y\} \in E$.*

Notably, in a general graph there is no difference between $\{x, y\}$ and $\{y, x\}$, as x and y are adjacent. Because our tasks have dependencies on each other, we want a graph in which the edges have an orientation.

Definition 1.3. *A directed graph is a pair G = (V, E) in which V is a set of vertices and the edges are an ordered pair (i.e., have orientation).*

$$E \subseteq \{(x, y) \mid (x, y) \in V^2 \text{ and } x \neq y\}.$$

Importantly, in a directed graph (x, y) and (y, x) constitute two different (inverted) edges. We now wish to introduce our final constraint into our graph: no circular dependencies. First let's define a path.

Definition 1.4. *A path in a directed graph is a sequence of edges which connects a sequence of vertices in which the ending vertex of each edge is the same as the starting vertex of the next edge.*

Definition 1.5. *A path forms a cycle if the starting vertex of the first edge in the sequence is the same as the ending vertex of the final edge.*

We are now ready to introduce the graph data structure we will use throughout the rest of the project: a directed acyclic graph (DAG).

Definition 1.6. *A directed acyclic graph (DAG) is a directed graph that has no cycles.*

We finally have a graph which we can use to implement tasks. It is directed, so we can have dependency, and it is acyclic, so we can omit circular dependencies . In our structure tasks will be represented by vertices (or nodes), and the dependency of a task on another will be represented by an edge from the prerequisite task to the dependent task.

Below is the specific graph, $G = (11, 14)$, which we will be using throughout the rest of the report as an reference to apply and showcase our algorithms. In the actual code, the graph G is already implemented for each algorithm.

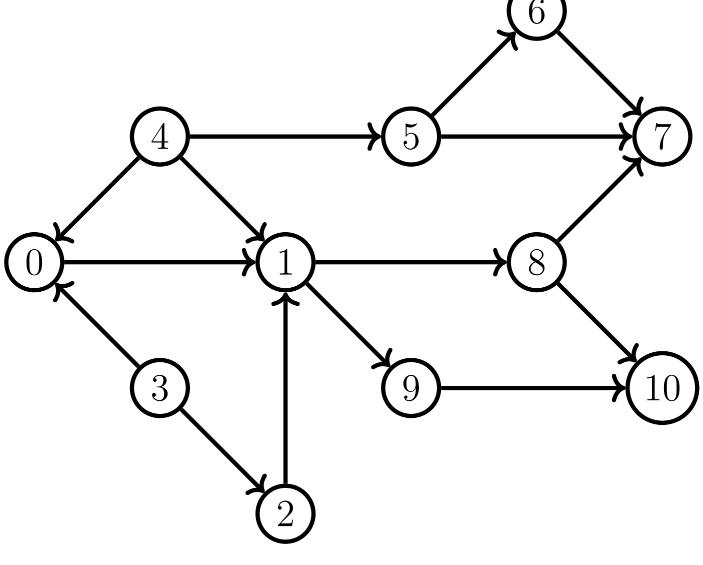


Figure 1: Selected Directed Acyclic Graph G

1.2 Implementation Details

To implement a DAG, we need a task and graph structure. First we introduce some data structures that will be critical to the rest of the project. In particular, in a graph context, an adjacency list uses both these concepts in conjunction.

Definition 1.7. *An array is a data structure consisting of a sequence of elements of the same data type stored at contiguous memory locations.*

Definition 1.8. *A linked list is a data structure consisting of a sequence of elements called nodes, which contain data and a reference to the address of the next node. We call the first node the head node and the last node the tail node.*

The main difference between a linked list and an array is that in an array each element must be stored contiguously, whereas in a linked list the order is provided by the links from one node[1] to the next. Linked lists have flexible memory storage, and in our purpose we will exploit this property to flexibly add or remove nodes from the linked list

Task Structure - Intuition and Implementation: In our project a task is a structure with an ID, an time and a pointer reference to another task.

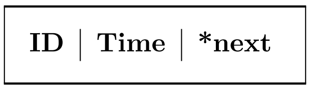


Figure 2: Anatomy of a Task

```
\\Struct of a Task
struct task{
\\Struct of a Task
int id;
int time;
struct task *next
};
```

A few key points :

- ID is represented by an integer. When we first started we tried to use a string so we could name the tasks (e.g., putting on socks). However, we immediately ran into complications with referencing and looking up the task in our code. An integer is much easier to look up, especially when it corresponds to an index in an array.
- Included is a time, represented by a integer. This is important for the second part of the project, in which we will need to consider the time each task takes.
- Within each task structure contains a pointer to another (next) task. This will be important later in the adjacency list structure to establish dependencies.

Graph Structure - Implementation: Now we implement the structure of a graph.

```
\\ Struct of a Graph
struct graph{
int number_vertices;
struct task **adjacency_list;
struct task *all_tasks;
};
```

[1]Throughout this report node, vertex and task are used more or less interchangeably. Generally, the intention is task is used when referring to algorithms, vertex when using the context of a graph, and node under the context of an adjacency list. This will not be followed strictly.

A few key points :

- The adjacency list is a double pointer. This is because the adjacency list is an array of pointers which point to a linked list of dependent tasks, so when we want to point to it we require a double pointer.
- We included an array *all_tasks* within the graph structure that includes all of the tasks themselves, rather than only having an adjacency list whose indices correspond to pointers which represent the tasks. This will be important for referencing the tasks themselves, and especially their respective components defined in the task structure.

More about the Adjacency List - Motivation and Representation The adjacency list is at the core of our graph implementation, serving as a fundamental data structure. This implementation is preferred over alternatives such as an adjacency matrix or an adjacency set due to its efficiency in representing sparse graphs. In the case of our task scheduling project, where tasks may have a limited number of dependencies, the adjacency list offers a more space-efficient representation. More formally here is a definition :

Definition 1.9. *An adjacency list of a graph is an array of linked lists, with each linked list representing a vertex in the graph, and each node in the linked list representing the other vertices of which the vertex shares an edge.*

We start with an array of pointers, whose index corresponds to a vertex in the graph. Each of these pointers are head nodes, which point to a linked list of its neighbors. This translates to the nodes of which the vertex shares an edge.
If the vertex in the array has no dependents/neighbors, its corresponding pointer points to **NULL**.

Below we include a figure which represents more intuitively the concept of an adjacency list.

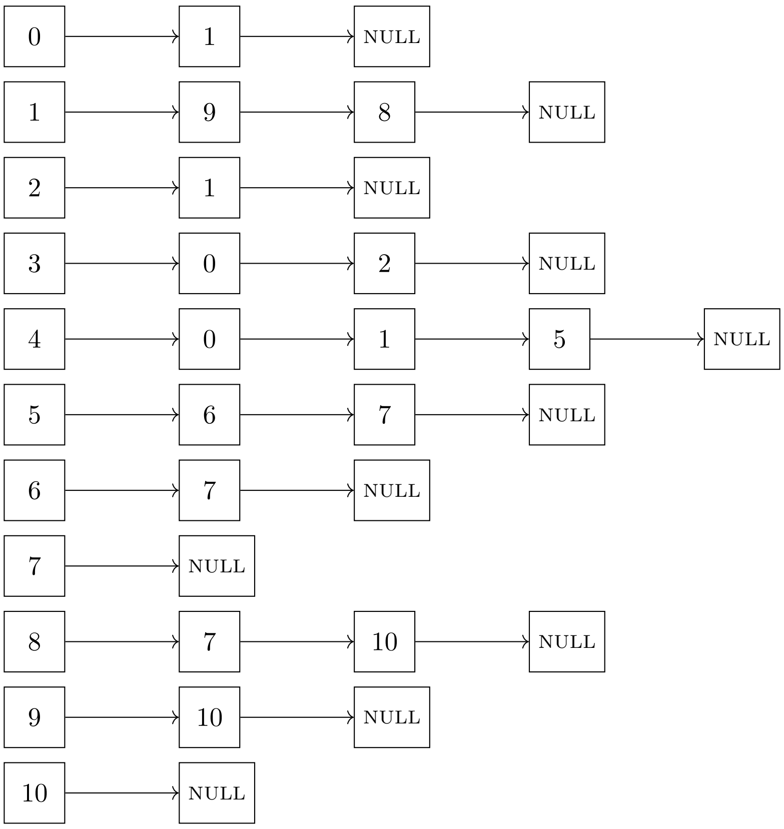


Figure 3: Adjacency List for Graph G

In the illustration above the column on the far left represents the array of pointers the adjacency list is made of. The rows represent the linked lists of tasks that the adjacency list elements point to.
To take an example, task 1 has 2 neighbors, task 9 and task 8, so they are therefore in the linked list of index 1 (in our case indices start at 0). In this way, the entire structure of graph G is now represented. Please note that for this figure, only an ID and a pointer is displayed (as marked by the arrow), however in our actual project the nodes resemble Figure 2.

Foundational Functions: This next part is dedicated to present functions that will be of constant use to create and populate a graph. This will be very helpful for the the next section : **Random Graph Creation**.
Next, using the task structure, we introduce a function to create a task. A subtle point: this function returns not an actual task, but a pointer to a task structure. This will be helpful when used to create our graph.

Listing 1: New Task Function

```c
struct task* new_task(int ID, int Time) {
  struct task* task_new = malloc(sizeof(struct task));
  task_new->id = ID;
  task_new->time = Time;
  task_new->next = NULL;
  return task_new;
}
```

A few key points :

- Since we already have the task structure this function is essentially just a matter of allocating memory for said structure, filled in with the provided data.
- *next* is initialized to **NULL**. This will be used when we add an edge using an adjacency list.

Now we implement a function to create a graph. Our goal is a function that will generate a graph automatically by specifying several parameters : number of vertices and a range between which completions time will be assigned randomly. We utilize the graph structure defined earlier to allocate the number of vertices and to create an adjacency list that will be used next when we want to add edges.

Listing 2: New Graph Function

```c
struct graph* new_Graph(int number_vertices, int minimum_time, int maximum_time) {
// Seed Random Time
  srand(time(NULL));
  // Allocate memory for graph
  struct graph* graph_new = malloc(sizeof(struct graph));
  // Assign graph specified number of vertices
  graph_new->number_vertices = number_vertices;
  // Allocate memory for the adjacency list
  graph_new->adjacency_list = malloc(number_vertices * sizeof(struct task*));

  for (int i = 0; i < number_vertices; i++) {
  // Set all adjacency list pointers to NULL
    graph_new->adjacency_list[i] = NULL;
  }
  // Allocate memory for all_tasks array
  graph_new->all_tasks = malloc(number_vertices * sizeof(struct task));

  for (int i = 0; i < number_vertices; i++) {
  // Create a random time between the values provided
    int random_time = rand() % (maximum_time - minimum_time + 1) + minimum_time;
    Create a new task for every vertex specified and place it in all\_tasks
    graph_new->all_tasks[i] = new_task(i, random_time);
  }

  return graph_new;
}
```

A few key points :

- The elements of the adjacency list are initialized to **NULL**. We do this because the initial graph has no edges, so the header nodes of the adjacency list have no other nodes to point to.

- For every vertex specified, new_task is called to create a task with a random time. Every time a task is created in the loop, it is further added to its respective index in *all_tasks*. In this way, we have an array that contains the tasks themselves.
  This will be crucial later, when we need information of a specific task, we can simply call *all_tasks*[ID] and access all of the information contained in its structure.

And finally we design a function to add edges.

Listing 3: Add Edge Function

```c
void addEdge(struct graph* graph, int src, int dest) {
// Get destination task time from all_tasks
  struct task* task = graph->all_tasks[dest];
  int time = task->time;
  // Create a new task to act as a representative node in the adjacency list
  struct task* new_task = new_task(dest, time);
  // Make the rep node point to what the source task is already pointing to
  new_task->next = graph->adjacency_list[src];
  // Now point the source task to the new task
  graph->adjacency_list[src] = new_task;
}
```

A few key points :

- We use new_task again because we need to create a new node that is a representative of the destination task. This is because we use adjacency list to keep track of edges, so we only need a *copy*[2] that has the same ID and time of the original.The *copy* is the exact same, the only difference being its address in memory.

- If we were to use the *original* task itself, we wouldn't be able to have more than one vertex point to any given task, meaning, only one edge for every task. For example, if we wanted to create the edges $u \rightarrow v$ and $u \rightarrow w$, this would be impossible as u could only point to either v or w in the adjacency list.

- Notice the importance of all_tasks here. We use it to copy the time into the destination node. This is the only part of the graph that contains the original tasks themselves.

- The algorithm to add an edge to the adjacency list is a bit tricky. Recall that a task structure contains a pointer **next* to another node. Here we point the destination node's **next* to the the source pointer in the adjacency list. If the source node has no neighbors, this would be **NULL**. Then we make pointer of the source task's adjacency list point to the copy, now that we've already used its address for the destination node to point to.

  In this way, we place the node in between the pointer and its reference, adding to the linked list. The result is the adjacency list element respective to the ID of the source task is now pointing to the destination task, while retaining its other neighbors further down the line in the linked list.

[2]Please note we mean copy in the sense of a new task that is the exact same as the original. This does not mean copy in the C language sense (e.g., memcpy).

1.3 Random Graph Creation

1.3.1 Motivation

We have now acquired functions to create a graph with specified edges that can be used to represent tasks and their dependencies. Before we start the primary purpose of this project which is to design algorithms to place these tasks in a topological order, we desire a function in which the user can choose their own graph.
While we successfully tested the algorithms on the fixed graph G[3], we are confident they will perform on any graph provided. Therefore, we want a way to automatically create a graph with a specified number of tasks and edges that allow for large scale testing but still preserve the DAG structure.

First Some Theory:

Theorem 1.1. *For any DAG with n vertices, the maximum number of edges is $n(n-1)/2$. We call a DAG with this number of edges complete.*

Proof. Take any vertex of G. In order for it to be acyclic, a vertex cannot have an edge to itself, so the vertex can point to at most $n-1$ other vertices. Taking another vertex, it can point to at most $n-2$ vertices to remain acyclic. Continuing in this way, we yield a maximum of $n(n-1)/2$ edges. □

Definition 1.10. *A sink in a graph is a vertex with an out degree of 0, i.e., it is not the starting vertex of any edge.*

Lemma 1. *If the maximum number of edges are added, a DAG must have only one unique sink.*

Proof. Take a complete DAG and assume towards a contradiction there exists multiple sinks. Since each vertex is connected to every other vertex, there must be a sink with at least one outgoing edge. But this contradicts the definition of a sink. Thus, there must only be one sink. □

1.3.2 Truly Random Implementation

Intuition The purpose of this part is to implement a function that designs a *truly* random graph. The idea is to assign edges randomly. The user specifies the number of vertices and edges, then the function creates a graph with the desired vertices and assigns random edges between them until all edges are in the graph.

Straightforward Steps

- 1. Create the structure of the graph with desired number of vertices.
- 2. Pick two random vertices u and v
- 3. Verify if the edge $u \to v$ already exist.
- 4. If not, add the edge.
- 5. Check if it forms a cycle. If it does, delete the edge.
- 6. Otherwise, add the edge to graph.

The following page shows a more detailed presentation of the `createTrueRandomGraph` pseudocode, following the steps iterated above.

[3]cf provided code

Algorithm 1: Create True Random Graph

```
Data: Number of vertices (number_vertices), Number of edges (number_edges), Minimum time
      (minimum_time), Maximum time (maximum_time)
Result: Randomly generated graph
Function createTrueRandomGraph (number_vertices, number_edges, minimum_time,
 maximum_time):
    // Create a new graph with specified vertices and random times
    graph *graph ← newGraph (number_vertices, minimum_time, maximum_time)
    // Generate random edges for the graph
    for edge_count ← 0 to number_edges do
        // Generate two random vertices
        random_vertices ← generateTwoRandomInt (number_vertices)
        src ← random_vertices[0]
        dest ← random_vertices[1]
        // Check if the vertices are already neighbors
        if !areTheyNeighbor (graph, src, dest) then
            // Add an edge between the vertices
            addEdge (graph, src, dest)
            // Check for cycles in the graph
            if hasCycle (graph) then
                // Remove the added edge to avoid cycles
                task *task_just_added ← graph→adjacency_list[src]
                graph→adjacency_list[src] ← task_just_added→next
                free(task_just_added)
                // Decrement the edge count to retry the same edge
                $edge\_count \leftarrow edge\_count - 1$
            end
        end
        else
            // Decrement the edge count to retry a new edge
            $edge\_count \leftarrow edge\_count - 1$
        end
        // Free the memory allocated for random vertices
        (random_vertices)
    end
    return graph
```

A few key points :

- The function generateTwoRandomInt takes in the number of vertices and returns an array with two values between 0 and the number of vertices.

```c
  int *generate_two_random_int(int number_vertices){
  srand(time(NULL));

  int *random_vertices = malloc(2*sizeof(int));
  random_vertices[0] = rand()%number_vertices;
  int second_int = rand()%number_vertices;

// To prevent having the same values in the array
  while(second_int == random_vertices[0]){
    second_int = rand()%number_vertices;
  }
  random_vertices[1] = second_int;
  return random_vertices;
}
```

- The boolean function areTheyNeighbor for a source vertex u and a destination v checks if the edge $u \to v$ exists. Simply, this works by accessing the adjacency list of the source, then traversing its linked list. If the destination node is found in the linked list, the function returns **TRUE**.

```
bool are_they_neighbor(struct graph *graph, int src, int dest){

struct task* source = graph->adjacency_list[src];
    while (source != NULL) {
    int neighbor = source->id;
    if (neighbor == dest) {
        return true;
    }
    // Go to the next neighbor in the list.
    source = source ->next;
}
return false;
}
```

- The way we implemented true randomness means the edge is added first with no conditions other than not adding the same edge multiple times. It is not guaranteed that adding a given edge doesn't form a cycle. Therefore, we check if there is a cycle using hasCycle, removing the edge if there is one.

- The hasCycle function aims to detect any cycle in the graph. It is used very extensively in this algorithm, at each iteration. The function is based on a modified recursive hasCycleDFS algorithm. A detailed explanation of the Depth-First Search algorithm is provided later in Section 2.1.
Essentially, it uses the *in_path* array to keep track of vertices in the *current* path and checks for back-edges during the traversal.
If it encounters a vertex that is already in the *current* path, it means there is a cycle.

- The closer the number of edges added is to the maximum, the more often hasCycle will return **TRUE**. This is due to the fact that the available edges to choose from without forming a cycle decreases, so the random edges added will have an increasing probability of forming a cycle.
For example, if the maximum number of edges are chosen, then for the last iteration in the loop, there is only one edge to choose from.

- We remove an edge by accessing the adjacency list of the source node using *task_just_added*. Because the destination node of a newly added edge is the first element in the linked list, we only need to now point the source node to the next node of *task_just_added*. Then the memory of *task_just_added* is freed to remove the node entirely. Lastly, we decrease *edge_count* by 1 so the loop tries adding an edge again.

Drawback While this does achieve a random DAG with the specified number of edges and vertices, there are huge complexity costs to this.
Intuitively, the drawback of the algorithm is inherent to the true randomness of the graph. We are choosing edges completely at random, and checking if they are valid for a DAG. The more and more edges are chosen, the less valid edges are available, so the loop must run until it *by chance* comes across a working edge.
In summary, throughout the process of adding edges, DFS is the computational bottleneck, but its nature changes as the number of edges added increases. At first, since nearly every randomly added edge is successful, it needs to search the entire graph, though the graph at this point is relatively small. But then, during the later stages, the randomness requires DFS to be increasingly called as the algorithm encounters challenges in finding suitable edges to avoid a cycle.

More theoretically, every time an edge is added, addEdge, areTheyNeighbor and hasCycle are called.

- addEdge has the least time complexity of $O(1)$, since it's only adding an edge.

- areTheyNeighbor has a complexity of $O(V)$, because at most it needs to traverse one linked list of vertices.

- hasCycle is the most costly, with $O(V + E)$, as it is recursive and needs to traverse, for every vertex, every sub graph spanned.

The overall complexity of the truly random graph is thus $O(E * (V + E))$, which is the complexity of `hasCycle` times how many times the loop calls it.
In practice, when dealing with large V and E, this graph is only usable if $number\_edges << V(V-1)/2$. As the user requests edges closer to the max possible, the degree of freedom for which edges are valid decreases. The amount of computations the algorithm must do to find one increases quadratically, as expressed in $(O(V * E + E^2)$ (the E^2 being the dominating term).

We now aim to simplify this function. Instead of using randomness to find a valid edge, we will introduce a constraint to guarantee one.

1.3.3 Less Random implementation

To simplify the complexity a bit, let's consider the following theorem.

Theorem 1.2. *In a directed graph G, let u and v be two vertices. If for every edge $u \to v$ we have $v > u$, then G is acyclic.*

Proof. We need to show that a cycle cannot exist in such a graph. A cycle from Definition 1.5 is a path in which the starting vertex of the first edge is the same as the ending vertex as the last edge. If all edges are directed in increasing value, then there is no way for this to happen as there would need to be an edge in which $v > u$. Therefore G must be acylic. □

So by forcing an edge to be added with increasing vertices, we eliminate the need for the functions areTheyNeighbor or hasCycle to be introduced in the truly random graph.

```
Algorithm 2: Create Less Random Graph
  Data: Number of vertices (number_vertices), Number of edges (number_edges), Minimum time
        (minimum_time), Maximum time (maximum_time)
  Result: Randomly generated graph
  Function createLessRandomGraph(number_vertices, number_edges, minimum_time,
   maximum_time):
      // Create a new graph with specified vertices and random times
      graph *graph ← newGraph (number_vertices, minimum_time, maximum_time)
      // Initialize Counter
      edges_added ← 0
      while edges_added is not equal to number_edges do
          // Generate two random vertices
          random_vertices ← generateTwoRandomInt (number_vertices)
          src ← random_vertices[0]
          dest ← random_vertices[1]
          if src > dest and not areTheyNeighbor (graph, src, dest) then
              // Add an edge between the vertices
              addEdge (graph, dest, src)
              edges_added ← edges_added + 1
          end
          else if dest > src and not areTheyNeighbor (graph, dest, src) then
              // Add an edge between the vertices
              addEdge (graph, src, dest)
              edges_added ← edges_added + 1
          end
          // Free the memory allocated for random vertices
          free(random_vertices)
      end
      return graph
```

A few key points :

- The main driver of this algorithm is now a while loop. It only needs to add an edge to the counter when an edge is added. The need to remove an edge after adding it is eliminated as it is ensured there will be no cycle before the edge is added.
- The placement of the source and destination tasks in areTheyNeighbor are placed according to which is greater.

Now we only need to call areTheyNeighbor, and it now enjoys a time complexity of $O(V)$, as in the worst case it only needs to check $V - 1$ vertices. Since this is in the worst case done for every edge, the overall time complexity is now $O(V * E)$, which is much better than $O(V * E + E^2)$.

We now have tasks with dependencies that are able to be customized depending on how many tasks or dependencies the user would like. The next step is where the real work begins, in designing algorithms that yield a meaningful sorting of these tasks. In the final section, our focus shifts to minimizing the overall completion time with the help of unlimited workers.

More about the completeness of a DAG : Recall in Lemma 1 that a complete DAG must have one unique sink. We wish to explore a bit how that unique sink and the general structure depends on the formation of the graph.

Note We advise the reader to imagine a graph here in the context of an adjacency list.

We claim that in a complete graph created by `createLessRandomGraph` the *unique* sink is always the last vertex. The proof of this is straight forward, as the last vertex cannot form an edge with another vertex since it is the highest one, and thus always a sink. In a complete graph, since the other vertices necessarily have edges, the last vertex must be the *unique* one.
The shape of a **complete** DAG in which for every edge $u \rightarrow v$, $v > u$, is also fixed. The first vertex only can have $V - 1$ edges, since it can only form an edge with higher vertices. The second $V - 2$, and so on until, when the last vertex is reached, no further edges can be added. The order in which the edges are added may be random in our function, but the structure will always remain the same.[4] Only when the edges added are less than $V(V - 1)/2$ are there any degrees of freedom to work with.

In a **complete** graph created by `createTrueRandomGraph`, the unique sink can be *any* vertex. As the graph adds more and more edges, the set of vertices that can possibly be the unique sink diminish in size. This can be thought of as the graph coalescing around one vertex.
Additionally, any vertex can be the one with $V-1$ edges, any vertex (besides the previous) can have $V-2$ edges, and so on. Such is the nature of a *true* random graph: if the shape was structured it wouldn't be random. As edges are added, however, the possibilities of which vertices can have how many edges decreases.

[4]If $u > v$ everything is reversed; the first vertex is the unique sink and the structure is flipped.

2 Topological Ordering of a Directed Acyclical Graph

Topological orderings find their primary application in scheduling tasks, where the completion of certain tasks is a prerequisite for the initiation of others. In this representation, vertices represent individual task, and the presence of a directed edge from vertex $u \to v$ signifies that task u must be finished before task v can commence.

For example, in the context of baking, a topological ordering establishes a sequential order for tasks, guaranteeing that prerequisites are satisfied, such as allowing the dough to rise before proceeding to the baking stage.

Definition 2.1. *A topological ordering of a directed acyclic graph (DAG) is a linear arrangement of the graph's vertices such that for each directed edge (u, v), vertex u precedes vertex v in the ordering.*

Using this definition, we want to be guaranteed a topological order for any DAG.

Theorem 2.1. *Let G be a directed graph. Then G is a DAG if and only if G admits a topological ordering.*

Proof. First the we show the reverse direction. Assume towards a contradiction that there exists a directed cyclic graph G that contains a cycle with a topological ordering. Let's choose the simplest cycle G can contain: a path of edges (u, v) and (v, u). Then any topological ordering of G would need to have u come before v and v come before u, which is impossible because an ordering must be a linear arrangement. Therefore if a directed graph has a valid topological ordering, it must be an acylic one (DAG).
Now that we showed the reverse direction, let's show the only if. We proceed by induction on the number of vertices, V.

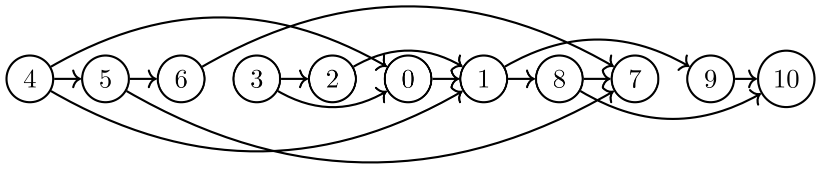
Base Case (V=1): If there is only one vertex then a topological ordering clearly exists because there are no edges to consider.
Inductive Step: Assume a DAG with V vertices has a valid topological ordering. We show a graph G with $V + 1$ vertices also has a valid ordering. Choose a vertex v with an indegree of 0[5] (i.e., a vertex with no incoming edges) from G and remove it and its outgoing edges from G. By the inductive assumption, the new graph G' now has a valid topological ordering $v_1, v_2, ..., v_n$. Now because v had an indegree of 0 we can add it back to G and thus obtain a valid ordering for G with $V + 1$ vertices by placing it in front: $v, v_1, v_2, ..., v_n$.

We conclude that G is a DAG implies it has a valid topological ordering. Thus the theorem is proved.

□

Lemma 2.1. *For a given DAG G, topological orderings are non-unique.*

For example, in our selected graph G, both of the following sequences constitute a valid topological ordering:



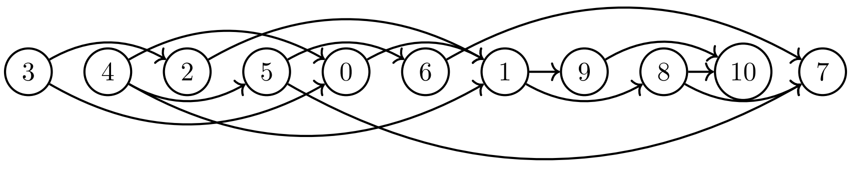


Figure 4: Two possible Topological Sort of the selected graph G

[5]See Theorem 2.2 for more on why we are guaranteed this vertex.

In each sequence, the arrow denotes the direction from one task to the next one in the topological ordering.

For a given DAG G with V vertices and E edges, two widely recognized methods for obtaining a topological ordering in $O(V+E)$ time involve either employing depth-first search method or iteratively removing vertices with no incoming edges (Kahn's Algorithm).

2.1 Depth-First Search Algorithm

2.1.1 Theory

Depth-First Search (DFS) is a methodical approach to traversing the nodes of both directed and undirected graphs. Similar to breadth-first search (BFS), DFS finds widespread applications in various Graph Theory problems (such as topological sorting or finding connected components) and stands as a foundational element in numerous graph algorithms.

DFS explores the vertices of a graph in the following manner: it commences by selecting an initial vertex, denoted as v. Then, DFS opts for an outgoing edge (v, w) towards a neighbor and recursively explores deeper into the graph until it hits a node without outgoing degrees. Then it backtracks returning to the most precedent vertex that hasn't been fully explored.
This process continues until there are no further unvisited vertices reachable from the current vertex. After exploring the entirety of the sub-graph spanned by the first vertex v (that we denote G_v), we chose another vertex. We check if it has been explored already, if not recursively apply DFS on it. We keep iterating this process until all node are visited.

DFS for Topological Ordering : In the context of topological ordering, Depth-First Search (DFS) operates as usual. As DFS backtracks, it appends vertices to the ordering, ensuring that each vertex is added to the list in the reverse order of its completion.

2.1.2 Implementation Details

While the recursive implementation appears to be the more intuitive approach, it's worth noting that iterative implementations of DFS also exist. However, for the purposes of this project, we have opted to utilize the recursive algorithm due to its clarity and conciseness.

Algorithm 3: Depth First Search (recursive version)

```
Data: Task (int), Visited (array), Order (int), Index (ptr to int), Graph $G$ with $n$ vertices
Result: Append task to Topological Order or recursively explore deeper
function DFS ($task, visited, order, *index, graph$)
// Mark the current task as visited
$visited[task] \leftarrow$ true
// Explore neighbors in the adjacency list
$current \leftarrow$ graph.adjacency_list[task]
while $current \neq NULL$ do
    $neighbor \leftarrow$ current.id
    // Recursively call DFS on unvisited neighbors
    if $not\ visited[neighbor]$ then
        function DFS ($neighbor, visited, order, index, graph$)
    end
    // Move to the next neighbor in the adjacency list
    $current \leftarrow$ current.next
end
// Add the current task to the topological order and increment the index
$order[index] \leftarrow task$
$index \leftarrow index + 1$
```

A few key points :

- The first thing we do in Depth-First Search Algorithm is to mark the current task as **visited** to ensure that during backtracking, we avoid revisiting tasks that have already been explored.
  This step also enables us to check, before delving deeper (i.e. recursively applying DFS), whether the neighboring child task has already been visited.

- Towards the conclusion of the algorithm, when a task is appended to the topological order array, it is crucial to determine the precise **position** for inserting the task in the array. Thus only using the **index** integer may seemed fine. But it doesn't work.
  If we use **index** just as regular integer (not a pointer), any modifications to it within the DFS function would not affect the original variable outside the function and would be lost during backtracking.

- Using a pointer in this function is necessary, as it enables dynamically changing the value of **index**. This ensure the persistence of modifications to the **index** value outside the function, ensuring precise tracking of the next available position in the order array.

  In C code, this translates to the following:

  ```c
  //Add the current node to the order array and increment the index
  order[(*index)++] = task;
  ```

Now, let's explore how this algorithm can be incorporated into the function topological sort.

Algorithm 4: Topological Sort using Depth First Search

```
Data: Graph $G$ with $n$ vertices
Result: Topological order of vertices
function TOPOLOGICAL-SORT ($graph$)
// Initialize an array to track visited vertices, setting all values to zero
$visited[number\_vertices] \leftarrow \{0\}$
// Initialize topological order array and index, setting all values to zero
$order[number\_vertices] \leftarrow \{0\}$
// Initialize index value to zero
$index \leftarrow 0$

// Iterate through all vertices
for $i \leftarrow 0$ to $number\_vertices$ do
    // If the current vertex is not visited, perform DFS starting from that vertex
    if not $visited[i]$ then
        function DFS ($i, visited, order, \&index, graph$)
    end
end
// Print the topological order
for $i \leftarrow 0$ to $number\_vertices$ do
    Print($order[i]$)
end
```

A few key points :

- The initial step involves creating and setting the visited array, the order array, and the index integer to zero.
- The next is to iterate through all vertices, check if the task has been visited if not apply DFS .
- Note that in this context, the DFS function not only explores the graph and marks the task as visited but also fills the order array while updating the index. Therefore, the topological sort function can be viewed as wrapper for the DFS function.
- Towards the end, after fully exploring the graph and completing the order array, we print the topological order by printing each element of the order array in reverse order.
- One might question why we used $\&index$ inside the for loop instead of just $index$. It's essential to note that in the C implementation of the DFS function, a pointer ***index** is required.
  Recall the function declaration :

  ```c
  //Notice the int *index
  void DFS(int task, bool visited[], int order[], int *index, struct graph* graph)
  ```

  Including & before $index$ provides us with the address of the integer $index$, which is, *in fine*, a pointer that can be seamlessly fed to the DFS function.

To summarize, we present a representation of the selected graph G depicting the algorithm's progression. We follow one of the possible paths that DFS may take, starting from task 0.

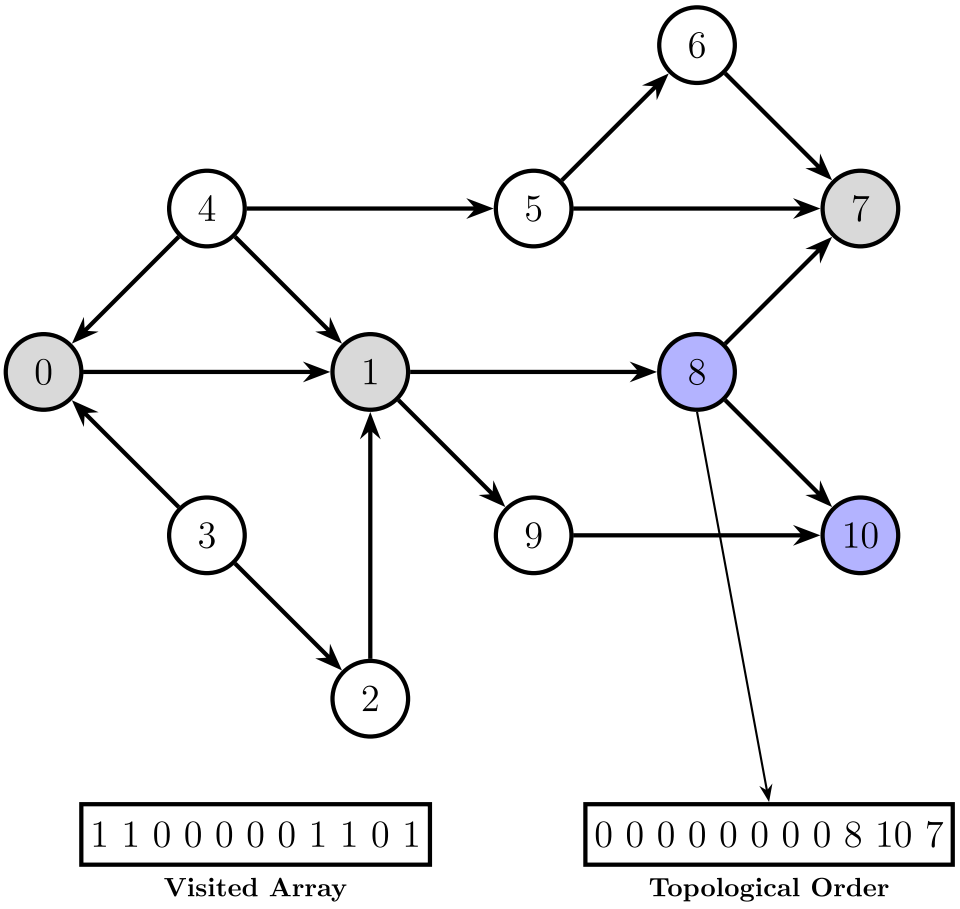


Figure 5: Topological Sort of graph G using DFS

In the illustration, the gray vertices represent the initial path followed by the DFS algorithm, starting from task 0 and ending up to task 7. Because 7 has no neighbors, 7 is placed in the order and DFS backs up to 8, this path now represented by blue. The only neighbor that is not visited for 8 is 10, and because 10 has no neighbors, it too is added to the order.
Now DFS backs up again to 8, but because it has no where else to go, it will be added into the order and the algorithm goes back to 1. From there the process continues until every vertex is added to the order.

Note It's important to note that this particular path is arbitrary and could have been different. It depends on the arrangement within the adjacency list, specifically on who is the first neighbor of task 1 and task 8 in the adjacency list.

2.2 Kahn's Algorithm

2.2.1 Theory

Introduced by Arthur Kahn in 1962, Kahn's algorithm is another method used for finding a topological ordering of a DAG. It works by finding vertices that can be added in the same order as the eventual topological ordering.
Unlike DFS, where you can start with any vertex and sort by adding the final vertex, in Kahn's algorithm, we make use of an "indegree" list. We first choose tasks with no prerequisites (i.e., have no incoming edge). After completing their execution, we update the outgoing edges, thereby freeing up additional tasks. The process then continues by identifying which next tasks can be started once their prerequisites are finished. In this way, we fill the topological sort in the forward direction as opposed to the reverse as in DFS.

Theorem 2.2. *Given a Directed Acyclic Graph (DAG), let the indegree of a vertex represent the number of incoming edges to that vertex. We are guaranteed to have at least one task with an indegree of 0.*

Proof. Start with a DAG G, and choose the longest path, P between vertices u and v. As G is acyclic, this path must be finite. As P is the longest such path between u and v, there must be no incoming edge to u. Therefore, G contains at least one vertex u with an indegree of 0. ☐

To start Kahn's algorithm, we first traverse the entire graph and find the indegree of every vertex. The vertices with an indegree of 0 are enqueued in a queue. While the queue is not empty, we dequeue a vertex from said queue and place it into the order array. Then using the adjacency list, subtract by 1 each of its neighbors' indegree in the indegree list. If the indegree is now 0, it is added to the queue. In this way, we go through all of the vertices of the graph one at a time until we have filled out the topological sort.

2.2.2 Implementation Details

To begin Kahn's algorithm, we first require an algorithm to calculate the indegree of the vertices of a given graph.

Algorithm 5: Indegree List

```
Data: Graph $graph$
Result: Array $indegree\_list$
function ComputeIndegreeList ($graph$) $number\_vertices \leftarrow graph.number\_vertices$
// Initialize the array, setting all values to zero
$indegree\_list[number\_vertices] \leftarrow \{0\}$
for $i \leftarrow 0$ to $number\_vertices$ do
    // Cycle through adjacency list
    $current \leftarrow graph.adjacency\_list[i]$
    while $current \neq NULL$ do
        for $j \leftarrow 0$ to $number\_vertices$ do
            // Add +1 for every node in adjacency list
            if $current.id == j$ then
                $indegree\_list[j] \leftarrow indegree\_list[j] + 1$
            end
        end
        $current \leftarrow current.next$
    end
end
return $indegree\_list$
```

Key point :

- Because the adjacency list contains pointers, the algorithm must loop through every vertex in the graph in order to find the index of the indegree list to increase. Updating the array then has a time complexity of $O(V * E)$, as for every edge (i.e., node in the adjacency list) the algorithm loops through every vertex until the condition $current.id == j$ is satisfied.
  Notably, $current.id == j$ is only satisfied exactly once, as each node in the adjacency list clearly corresponds to only one vertex in the graph.

Now we are ready to implement Kahn's Algorithm.

Algorithm 6: Topological Sort using Kahn's Algorithm

```
Data: Graph $graph$ with $n$ vertices
Result: Topological order of vertices
function KAHNTOPOSORT ($graph$)
// Create an Indegree List
$indegree\_list \leftarrow$ ComputeIndegreeList($graph$)
// Initialize a queue
$q \leftarrow$ queue_init($q, intsize, number\_vertices$)
// Enqueue Tasks with an indegree of 0
for $i \leftarrow 0$ to $number\_vertices$ do
    if $indegree\_list[i] == 0$ then
        queue_enqueue($q, \&i$)
    end
end
// Initialize an index to 0
$index \leftarrow 0$
// Initialize an order array with zeros
$order[number\_vertices] \leftarrow \{0\}$
while is_empty($q$) do
    queue_dequeue($q, \&dequeued\_task$)
    $order[index] \leftarrow dequeued\_task$
    $current\_task \leftarrow graph.adjacency\_list[dequeued\_task]$
    while $current\_task \neq NULL$ do
        $neighbor \leftarrow current\_task.id$
        $indegree\_list[neighbor] - -$
        if $indegree\_list[neighbor] == 0$ then
            queue_enqueue($q, \&neighbor$)
        end
        $current\_task \leftarrow current\_task.next$
    end
    $index + +$
end
printf("Topological order Using Kahn's Algorithm: ")
for $i \leftarrow 0$ to $number\_vertices$ do
    printf("%d ", $order[i]$)
end
printf("")
queue_dispose($q$)
free($indegree\_list$)
```

A few key points :

- A queue is introduced as a buffer between the indegree list and the topological sort. The important concept is a queue is a first-in-first-out (FIFO) data structure. That is, much like a real queue, an element "enqueued" (added) is only "dequeued" (removed) after all of the elements in the queue have been dequeued. See appendix for more.

  The upshot for our purposes is that, when tasks with an indegree of 0 are enqueued, they preserve the order in which they were added. When dequeued tasks are added into the order array they follow the desired order.

- The main driver of the algorithm is the is_empty (q) while loop. In the first step, Theorem 2.2 guarantees at least one task has an indegree of 0 and will be enqueued. On every subsequent loop, at least one task will enqueued. This is because we subtract the indegree of the neighbors of the dequeued task on every loop. For the number of elements in the queue, there will be an indegree of at most that number, so that every indegree will be 0 before the queue is empty (proof ommited). Thus, every task will be added to the order.

To summarize, we present a representation of the selected graph G depicting the algorithm's progression. We here illustrate the first iteration of Kahn's algorithm.

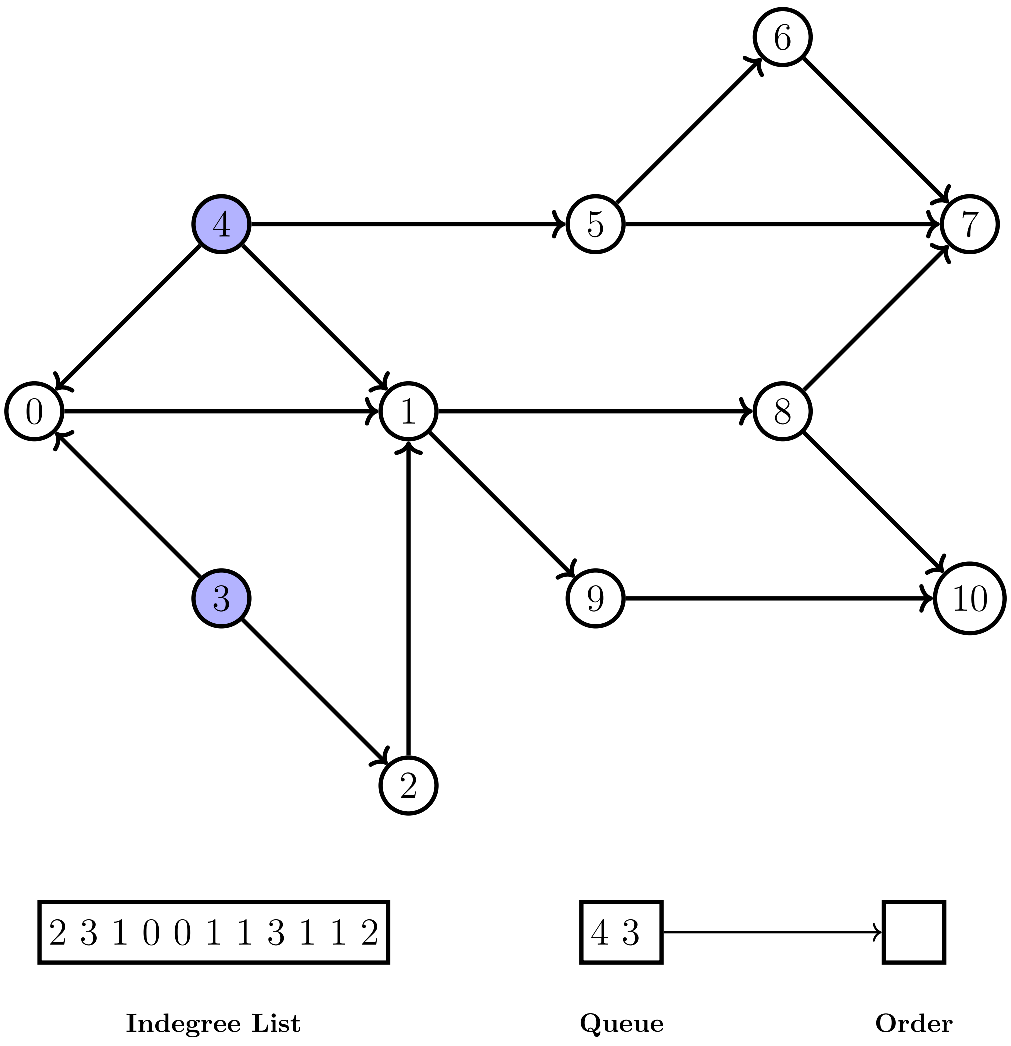


Figure 6: First Step of Kahn's Algorithm on graph G

In this representation, the blue vertices represent vertices with an indegree of 0. These are then enqueued in a queue, and when 3 in dequeued into the order array, 0 and 2 will both have their indegrees subtracted by 1. Since 2 has an indegree of 1, it will subtract to 0 and thus be enqueued. Then 4 will be dequeued and so on...

2.3 Comparaison

The first thing we'll discuss is what makes them similar: they both provide multiple valid topological orderings. This is not trivial, as we have seen topological orderings are not unique, so the algorithms need to be able provide multiple valid orderings. In Topological sort using DFS, the ordering depends on where you start DFS, as there are multiple sub trees that can be explored. For Kahn's, it is the order of which tasks of indegree 0 are enqueued.

Theoretically, for DFS you can start with any vertex for DFS, and any element of indegree 0 for Kahn's, each choice providing a different ordering. In implementation, both will always yield the same order as our for loop used in both algorithms iterates through the vertices from 0 to $number\_vertices$. So DFS always starts with the 0 vertex, and Kahn's always enqueues the vertex with smallest index first.

Now for the differences. In DFS there exists a unique final vertex to which a given path leads. All paths can very well lead to the same sink (in the case of a complete DAG), but one path cannot have multiple sinks. The degree of freedom for the algorithm comes from which neighbor of a vertex DFS chooses to search.

Because we use an adjacency list, this in practice is the node to which the vertex points, so the path DFS takes is determined by how the adjacency list is implemented. In our project, this means how we add an edge, which is random. In any case, when DFS finds a vertex to which it cannot go further, this is the natural candidate to be placed at in the topological sort (from decreasing order). Then when it starts to reverse out of the recursion, the rest of the order is similarly unique. This means that getting an order from DFS is quite simple.

In Kahn's, there is not a natural candidate to be placed in the order, which is why a queue must be used as a buffer between the indegree list and the order. Because indegree list has multiple candidates with no ordering, a queue gives the algorithm the order in which to proceed, enqueuing based on index then dequeuing with that same order so that the tasks with smallest index and indegree are placed in the order first. In the next section, time will be introduced so that the minimum time tells the algorithm what order to go, which will actually yield a unique topological ordering.

In terms of complexity, the algorithms both have a time complexity linear to the size of the graph, $O(V + E)$. This is because both visit/process each vertex and each edge once. The space complexity is also the same, $O(V)$. In DFS the visited array has at most V vertices, and in Kahn's the indegree list and queue both also have at most V vertices.

In practice, however, Kahn's has a smaller constant factor so is slightly faster as it doesn't require the recursion of DFS. In the next section, it is also easier to transition to ordering based on time. DFS, on the other hand, is more intuitive and easier to implement. (In any case as we have seen in our project it is creating a true random graph which really takes the most time.)

3 Parallel Task Scheduling

In the following phase, we explore a scenario where each task is assigned a known duration. Tasks can be executed concurrently, without restrictions on the number of workers, under the condition that a task initiates only when all its prerequisites are fulfilled.
One way to understand this scenario is to view each task as containing a dormant worker awaiting the signal to commence work on that specific task. This signal is triggered once all of the task's prerequisite tasks have been successfully completed.

Warning :
An unlimited workforce implies the ability to work on any number of tasks simultaneously, but the constraint remains that only tasks meeting their prerequisites can be initiated. Moreover, it doesn't permit the assignment of multiple workers to a single task to expedite its completion.

3.1 Optimal Solution Overview

In this section, we present an algorithmic approach for an efficient task scheduling strategy with the aim of minimizing the total completion time. We are confident that this approach holds the potential for achieving an optimal solution when having unlimited worker.
Additionally, we will delve into the implementation details of both a naive approach and a faster, more optimized implementation.

3.1.1 Theory

While the conventional Kahn's algorithm processes each task individually in a sequential manner, the modified parallelized version deviates from this paradigm. Instead, the parallelized Kahn's algorithm processes set of tasks with zero incoming degrees.
This method enables us to address the **maximum allowable** number of tasks at given time "simultaneously". By definition, other tasks not currently in progress cannot be addressed anyways, as they still have prerequisites to be fulfilled.

Understanding Time in this Context Understanding the concept of time as described here is essential. Although we write the term *simultaneously*, it does not imply genuine parallelized processing. Instead, it symbolizes a conceptual view of time where we freeze it and consider one unit of time as equivalent to the completion of one unit of work for each task concurrently in progress.
This justifies the loosely employed term *simultaneously.*

Illustration on the Selected Graph G For the sake of simplicity, let's assume that each task requires the same completion time: 1 unit. Additionally, we'll consider no travel time between tasks, implying that the signal instantly reaches the concerned child tasks.

Theorem 3.1. *Let G be DAG, T be the total completion time of all tasks. Assume that we only have one worker, then*

$$T = \sum_{i=0}^{V} t_i \tag{1}$$

where V is the number of vertices in G and t_i the time of the task i.

Assuming that $\forall i < N, t_i = 1$ then in our selected graph G, with only one worker the total completion time T would be 11.

As previously discussed, in the context of the parallelized Kahn's algorithm, we now process tasks in batches. We begin with tasks $[3, 4]$, which have zero incoming degrees. Assuming all tasks require 1 unit of time, they are completed simultaneously, resulting in a total time of $T = 1$.
Subsequently, we advance to the subsequent batch of tasks $[0, 5, 2]$, now that their prerequisites have been satisfied, leading to a total time of $T = 2$. Similarly, for the batches $1, 6$, $[8, 9]$ and $[7, 10]$ the tasks within each batch are processed simultaneously, contributing to respective total times of $T = 3$, $T = 4$ and final time $T = 5$.
To summarize, we present a representation of the selected graph G depicting the algorithm's progression. Much like the (simple) topological ordering presented in Section 2, the following depicts the topological *batch* ordering.

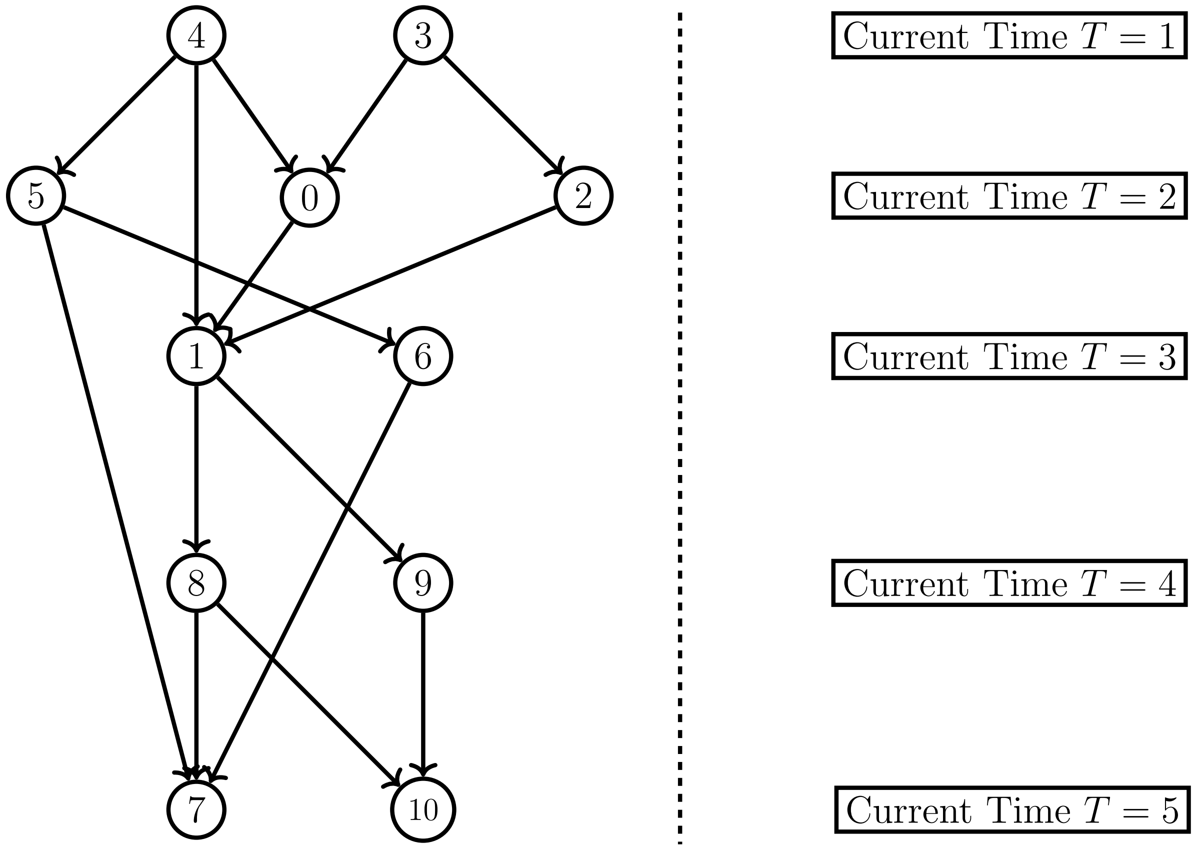


Figure 7: Topological Batch Ordering of G using parallelized Kahn's algorithm

This straightforward view of the graph G allows us to clearly see the distinct batches of task, their order and the time required for their completion.
In this version, the completion of all tasks requires only 5 time units, compared to the 11 time units in the basic single-worker version.

We have opted to only implement the efficient version of the parallelized Kahn's algorithm, since it is a clear improvement over the naive implementation.

Note In the description of the following implementations, we utilize random completion times for each task. Thus, in a "real life DAG", the scheduling process is not as straightforward as our toy example G. For instance, there may be significant overlaps between consecutive batches, as some tasks require more time for completion and consequently extending their presence in the *current batch.*

3.1.2 Naive Implementation

This implementation strategy revolves around incrementing time by one unit at the end of each iteration of the algorithm. Then one time unit is deducted from the completion time of each task in the current batch. We then check if there is any new task with zero incoming degree and update the batch accordingly. The algorithm continues this process until all tasks are completed.

Drawback Clearly the main flaw of the naive implementation lies in its slowness, particularly when confronted with tasks with high completion times. This slowness comes from the fact that the algorithm increments time by only one unit at each iteration. Thus, the processing of each batch of tasks only progress by small step of one unit of time.

Note The pseudocode for the naive implementation is not presented here as it does not contribute substantially to the project. It would be redundant, considering its high similarity to the efficient implementation, with the primary distinction being the handling of *current-time.* Please refer to the next section for further details on implementation.

3.1.3 Efficient Implementation

In this version of the implementation, a more efficient strategy is employed. Instead of incrementing time by one unit, the algorithm dynamically adjusts the time. During each iteration, the algorithm identifies the minimum completion time among the tasks in the current batch.
This minimum time *min-time* of the batch is then used to advance the time *current-time*, and the completion time of each task in the current batch is reduced by *min-time*. Following this, we update the current batch namely *selected-task* based on their incoming degree.
This approach ensures that, at each iteration, the algorithm processes at least one task (i.e. the ones that have completion time *min-time*).

Algorithm 7: Parallel Task Scheduling

```
Data: Graph $G$ with $n$ vertices
Result: Total completion time of the tasks
function PARALLEL-TASK-SCHEDULING($graph$):
// Calculate indegree for each vertex
$indegree\_array[number\_vertices] \leftarrow$ indegreeList($graph$)
// Initialize time array with task times
$time\_array[number\_vertices]$
// Initialize completed task array to track processed tasks
$completed\_task[number\_vertices] \leftarrow \{0\}$
// Initialize integer representing the sum of all the tasks completion time in G
$sum\_of\_all\_tasks\_time \leftarrow$ time_sum($graph, time\_array$)
// Initialize integer representing the current time
$current\_time \leftarrow 0$
while $sum\_of\_all\_tasks\_time \neq 0$ do
    // Select only the unprocessed tasks with zero indegree
    $selected\_task \leftarrow$ zero_indegree_task_id($graph, indegree\_array, completed\_task$)
    // Find the minimum completion among the current batch
    $minimum\_time \leftarrow$ min_time($selected\_task, time\_array$)
    // Update the completion time of the selected task
    updated_time(time_array, selected_task, minimum_time)
    // Update the in-degrees of the processed tasks's neighbors
    update_indegree_of_completed_task(graph, time_array, selected_task, completed_task,
     indegree_array)
    // Update the current time and the sum of all tasks completion time
    $current\_time \leftarrow current\_time + minimum\_time$
    $sum\_of\_all\_tasks\_time \leftarrow$ time_sum($graph, time\_array$)
end
return $current\_time$
```

A few key points:

- There are many ways to ensure and check the completion of all the tasks in a given graph G. The method we choose involves summing all of the integers in the *time-array* by using the time-sum function. Its purpose is to store the completion time of all tasks. We chose the array data structure with a pointer because it allows easy retrieval and modification of values.

```c
int *time_array = malloc(number_vertices*sizeof(int));

// We retrieve the time information by using the all_tasks array.
for (int i = 0; i < number_vertices ; i++){
 struct task task = graph->all_tasks[i];
 int task_time = task.time;
 time_array[i] = task_time;
 }
```

The *time-array* is the **cornerstone** in this implementation, it is used in the main loop and by several functions (time-sum , minimum-time and update-time for exemple) and gets updated at each iteration.

- The function min-time aims to determine the minimum completion time among the tasks in the current batch. It takes two arrays as input: *selected-task* which contains the IDs of the tasks in the current batch, and *time-array* holds the completion times of all tasks.

- The function update-time updates the completion times in *time-array* of the *selected-task* by substracting *min-time*.

- The function update-indegree-of-completed-task is complex function that performs multiples tasks. Its overall purpose is to maintain the accuracy of the incoming degree information for each task within the graph.
  First it iterates through the array of *selected-tasks* to identify those with a completion time of zero, marking them as completed in the *completed-task* array. Then, for each completed task, it updates the in-degree of its neighboring tasks by decrementing their respective in-degree values in the *indegree-list*. This is achieved by traversing the adjacency list of the completed task.

  This is how it translates to code :

```
void update_indegree_of_completed_task(struct graph *graph, int *time_array, int
    *selected_task, int *completed_task, int *indegreelist){
        int i = 0;
        while(selected_task[i]!= -1){
            int task_id = selected_task[i];
            if(time_array[task_id]==0){
             completed_task[task_id] = 1;
             //update in degree of neighbor now
             struct task *current_task = graph->adjacency_list[task_id];
             while(current_task != NULL){
                int neighbor = current_task ->id;
                indegreelist[neighbor]--;
                current_task = current_task->next; }}
  i = i+1; }}
```

- The function zero-indegree-task-ID initialize and returns an array containing the IDs of tasks with zero indegree that have not been completed. It use a graph, an indegree list, and an array indicating completed tasks as input. The resulting array represents the current batch of tasks ready for processing in the parallelized Kahn's algorithm.

  This is how it translates to code :

```
int *zero_indegree_task_id(struct graph *graph, int *indegree, int *completed_task){
    int number_vertices = graph->number_vertices;
    int* selected = malloc((number_vertices +1) * sizeof(int));
    int counter = 0;

    for (int i = 0; i < number_vertices; i++) {
        if (indegree[i] == 0 && !completed_task[i]) { //&& !completed[i]
            selected[counter] = i;
            counter++;
        }
    }
    selected[counter] = -1; // zone tampon.
    return selected;}
```

- Notice the buffer zone in the *selected-tasks* array. This extra element, marked as -1, serves as a buffer value to indicate the end of the list of selected tasks.
  It eliminates the need to dynamically determine the length of the *selected-tasks* array during iteration, as the algorithm can simply halt when encountering the negative value -1.

4 Future Improvements

4.1 Bug Fixes

4.1.1 Time Waster

In our project the main time user is creating the graph. However, when we called the functions creating the graphs in the main on their own (both the less and truly random), they take much less time than in our current version.

For some reason our code is slow - *really* slow. Even creating a truly random graph with a relatively large number of edges and doing DFS sort, Kahn's and parallel task scheduling, the program only takes some seconds. But in the version presented where we prompt the user to input their own variables, it takes an order of minutes in time. This is unfortunate as it is not due to fundamental problems in our algorithms or functions, but some bug that's holding everything up.

4.2 Schematics

We did not include as many schematics as would have been ideal to more clearly illustrate our project and how it works.

4.2.1 Adjacency Amelioration

The adjacency list doesn't convey that the array is just pointers continuously stored pointing to nodes that have an ID, a time and a reference.

4.2.2 Not Very Edgy

In the add edge function as well, an illustration of what's going on is really essential to properly understanding it. The same holds true for removing an edge.

4.2.3 Go with the Flow

Flow charts, too, would help better explain the recursive nature of algorithms such as DFS.

4.3 Top Notch User Experience

We would have liked to include a section to explain what the environment is like for the user, complete with how to read the results given and perhaps showing how we coded that in the main. A timer should also have been included to display the time it takes for our code to run.

4.4 A Tough Economy

What if we didn't have unlimited workers? That is, the assumption that every task is immediately available to be worked on after its prerequisites are complete. This would be a fun new constraint to add.

4.5 Le Temps

In the code presented in this report and the code itself, time is an integer. The algorithms, most importantly parallel task scheduling, will work the same with a double or a float as the data type (for example, in parallel task scheduling, the particular time data type is largely irrelevant since, because minimum time is subtracted, what matters is relative value more than the actual values). It is unfortunate this wasn't shown in practice, but everything works the same, even if time is irrational (see paragraph "Understanding Time in this Context" under section 3.1.1).

A Queues

A.1 General Introduction

Definition A.1. *A queue is a first-in-first-out data structure (FIFO) that can be modified by addition of elements to the front and the removal of elements from the back.*

Think of a queue in real life. Someone enters the queue, then someone behind them, and so on. The first person in the queue is processed, then the one behind them, etc. A queue in computer science works the same way. As opposed to a stack, the queue preserves the order in which elements were added.

A.2 Implementation details

Because C has no built in queue structure in its standard library, and this is a data structures course, we implement our own queue.

First we implement the structure of a queue. It has a front, where elements will be dequeued, a length corresponding to how many elements in contains, and a capacity corresponding to how many elements that can be added. We use malloc to allocate memory the size of element size times the element capacity.

Listing 4: Queue Structure

```c
struct Queue {
  size_t front;
  size_t length;
  size_t capacity;
  size_t elem_size;
  void *data;
};
```

Next we implement a function to initialize a queue. The front of length have value 0 because there are no elements yet enqueued, and the user is able to assign the capacity they desire.

Listing 5: Queue Initialization

```c
// Function to initialize the queue
int queue_init(struct Queue *q, size_t elem_size, size_t capacity) {
  q->front = 0;
  q->length = 0;
  q->capacity = capacity;
  q->elem_size = elem_size;
  q->data = malloc(elem_size * capacity);
  return 0;
}
```

To dispose a queue we simply free the data allocated in the initilization function.

Listing 6: Queue Disposal

```c
// Function to dispose of the queue
void queue_dispose(struct Queue *q) {
  free(q->data);
  free(q);
}
```

Here is an important function, enqueuing or adding an element to the queue. We want the element to be added in such a way that it is placed ”behind” the others. So the function calculates an index by modding the front + length with the capacity where the element can be added. This is because the queue is being treated as a circular buffer (an array of fixed length stored in a continuous loop). It finds the memory location destination where the element will be store by multiplying the index by element size and adding the data the queue takes up in memory. Finally memcpy copies the src element (in this case a pointer to an integer due to our project) into the destination dst, and increases length by 1.

Listing 7: Enqueue Operation

```c
// Function to enqueue an element into the queue
void queue_enqueue(struct Queue *q, int *src) {
  int idx = (q->front + q->length) % q->capacity;
  void *dst = q->data + (idx * q->elem_size);
  memcpy(dst, src, q->elem_size);
  q->length++;
}
```

Dequeue works is the reverse enqueue but is implemented differently. It calculates the memory address of the front of the queue. Then it uses memcpy to copy the content of this element to the destination *dest. This is important to retrieve the front element. Then the length is decreased and the front pointer moves to the next element in the queue.

Listing 8: Dequeue Operation

```c
// Function to dequeue an element from the queue
void queue_dequeue(struct Queue *q, int *dest) {
  void *src = q->data + q->front * q->elem_size;
  memcpy(dest, src, q->elem_size);
  q->length--;
  q->front = (q->front + 1) % q->capacity;
}
```

Our last function simply checks and returns true if the length is 0.

Listing 9: Check if Queue is Empty

```c
// Function to check if the queue is empty
int is_empty(const struct Queue *q) {
  return (q->length == 0);
}
```

References

[1] Directed acyclic graphs scheduling. URL: https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-spring-2015/mit6_042js15_session17.pdf.

[2] Graph (discrete mathematics), 05 2020. URL: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics).

[3] Topological sorting, 05 2021. URL: https://en.wikipedia.org/wiki/Topological_sorting.

[4] Burmeste. Topological sort. URL: https://www.cs.fsu.edu/~burmeste/slideshow/chapter23/23-4.html.

[5] Clrs. Topologically sorting a directed acyclic graph. URL: https://tildesites.bowdoin.edu/~ltoma/teaching/cs231/spring16/Lectures/11-topsort/topsort.pdf.

[6] Thomas Cormen. Topological sort. URL: https://ocw.snu.ac.kr/sites/default/files/NOTE/Week%2010_5.pdf.

[7] Shayan Gharan. Cse 421 topological ordering / greedy. URL: https://courses.cs.washington.edu/courses/cse421/18wi/lecture-9-slides.pdf.

[8] Jeffe. Depth-first search. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/book/06-dfs.pdf.

[9] Gang Wu. Topological sort of directed acyclic graph — baeldung on computer science, 07 2020. URL: https://www.baeldung.com/cs/dag-topological-sort.