Sorbonne Université
Master 1 Mathématiques et applications

MU4MA016 Algorithms and Data Structures for Programming

Project

Task Scheduling

Name:	Thomas Gantz
Student Number:	21109004
Professor:	Didier Smets
Submission Date:	12.01.2025

Abstract

This project aims to develop and implement task scheduling algorithms for both sequential and parallel machines, considering cases with an unlimited or limited number of parallel processing units. While a topological sorting approach is suggested for the sequential case, all other scheduling methods, including the design of appropriate data structures, must be developed. The tasks and their dependencies form a finite directed acyclic graph (DAG), ensuring the existence of at least one valid schedule, though it may not be unique. The objective is then to minimize the total execution time.

Contents

1 Data Structures

This sections shows the different data structures constructed during the process, i.e. tasks and DAGs to encode the problem and a heap or queue used during the algorithms.

1.1 Task

A task is defined using a single struct:

```c
struct Task {
    double duration;    // Execution time of the task
    double end;         // Earliest completion time of the task
    int req;            // Number of prerequisite tasks
    int ns;             // Number of successor tasks
    int *successors;    // Array of indices of successor tasks
};
```

Listing 1.1: Definition of a task

The goal of the algorithms is to determine and set the correct value for `end`. Each task stores its successors, meaning $B \in \text{successors}(A) \iff A$ must be completed before B. Additionally, the number of prerequisite tasks (incoming edges) is stored. This value is updated dynamically during the execution of the algorithms.

1.2 DAG

All tasks together form a DAG. The DAG structure is implicitly defined and supported by helper functions.

```c
struct Task *dag;
struct Task *generate_random_DAG(int N);
void free_random_DAG(struct Task *dag, int N);
void set_req(struct Task *dag, int N);
```

Listing 1.2: Definition of a DAG

A crucial aspect of generating this DAG is ensuring that the adjacency matrix (if represented explicitly) is an upper triangular matrix, which guarantees that the graph remains acyclic. In code, this is implemented as follows:

```c
struct Task *generate_random_DAG(int N) {
    struct Task *dag = malloc(N * sizeof(struct Task));

    // Create N random tasks
    for (int i = 0; i < N; i++) {
        struct Task *task = malloc(sizeof(struct Task));
        int *neighbors = calloc(N, sizeof(int));

        // Fill adjacency array, only allowing successors from i
           +1 onward to maintain DAG property
        task->ns = fill_array(neighbors, N, i + 1);

        task->duration = random_double(1.0, 10.0);
        task->end = -1.0;
        task->req = 0;
        task->successors = malloc(task->ns * sizeof(int));

        // Copy successor indices
        int place = 0;
        for (int j = 0; j < N; j++) {
            if (neighbors[j] == 1) {
                task->successors[place++] = j;
            }
        }
        free(neighbors);
        dag[i] = *task;
        free(task);
    }
    return dag;
}
```

Listing 1.3: Random DAG generation

1.3 Queue

One topological sort algorithm (2.1.2) will use a queue to store tasks whose prerequisites have already been completed.

```c
typedef struct {
    int *data;
    int front;
    int rear;
    int capacity;
} CircularQueue;

void init_circular_queue(CircularQueue *q, int initial_capacity);
int is_empty(CircularQueue *q);
int is_full(CircularQueue *q);
void enqueue(CircularQueue *q, int item);
int dequeue(CircularQueue *q);
void free_circular_queue(CircularQueue *q);
```

Listing 1.4: Definition of a Queue

The implementation follows the structure of the circular buffer queue presented in TP, where the two indices, `front` and `rear`, move around an array in a circular manner. A key feature of this implementation is its dynamically resizable capacity, which grows only when necessary to optimize space complexity. The queue stores only integers, each corresponding to a `task_idx` in actual usage.

1.4 Heap

In many of the the following algorithms, a heap will be used to order tasks by minimal duration or end time.

```c
// Definition of a heap node
typedef struct {
    int i;          // Integer identifier
    double val;     // Value used for heap ordering
} heap_node;

// Definition of the min-heap
typedef struct {
    heap_node* data; // Dynamic array storing heap nodes
    int size;        // Current number of elements
    int capacity;    // Maximum capacity of the heap
} min_heap;

min_heap* create_min_heap(int initial_capacity);
void free_min_heap(min_heap* heap);
void insert(min_heap* heap, int i, double val);
heap_node extract_min(min_heap* heap);
```

Listing 1.5: Definitition of a Heap

The heap follows the structure presented in class, where insertion and extraction operations maintain the heap property via sift-up and sift-down operations. The heap is stored as an array, with parent-child relationships defined by index formulas:

$$\text{parent}(k) = \left\lfloor \frac{k-1}{2} \right\rfloor, \quad \text{left\_child}(k) = 2k+1, \quad \text{right\_child}(k) = 2k+2.$$

A key feature of this implementation is its dynamically resizable capacity, which grows only when necessary to optimize space complexity. The heap stores only two elements per node: (`i`, `val`), corresponding to (`task_idx`, `duration`) or (`task_idx`, `end`) in actual usage.

2 Algorithms

2.1 Topological Sorting

Topological sorting organizes vertices in a linear order such that for every directed edge $u \to v$, vertex u comes before v in the ordering.

2.1.1 DFS

One common approach to topological sorting is an algorithm that leverages Depth-First Search (DFS).

Algorithm 1 Topological Sorting using DFS

```
1: Initialize an empty set visited
2: Initialize an empty stack order
3: function DFS(vertex)
4:     Mark vertex as visited
5:     for each neighbor in adjacency list of vertex do
6:         if neighbor is not in visited then
7:             DFS(neighbor)
8:         end if
9:     end for
10:    Push vertex onto order                    ▷ All dependencies processed
11: end function
12: for each vertex in $V$ do
13:     if vertex is not in visited then
14:         DFS(vertex)
15:     end if
16: end for
17: return order                   ▷ Stack represents reverse topological order
```

The time complexity of this algorithm is $\mathcal{O}(V + E)$, where V is the number of vertices and E is the number of edges in the graph, as it processes each vertex and edge exactly once.

2.1.2 BFS (Queue)

Another common method for topological sorting is based on Breadth-First Search (BFS). This algorithm uses a queue to process vertices with no incoming edges.

Algorithm 2 Topological Sorting using BFS

```
1: Initialize the in-degrees
2: Initialize a queue for vertices with no incoming edges
3: Initialize an empty list for the topological order
4: for each vertex with zero incoming edges do
5:     Add the vertex to the queue
6: end for
7: while queue is not empty do
8:     Dequeue a vertex from the queue
9:     Add the vertex to the topological order
10:    for each neighbor of the vertex do
11:        Decrease the in-degree of the neighbor by 1
12:        if in-degree of the neighbor is zero then
13:            Enqueue the neighbor
14:        end if
15:    end for
16: end while
17: return topological order
```

The time complexity is also $\mathcal{O}(V + E)$, as it processes each vertex and edge exactly once.

2.1.3 BFS (Heap)

An extension of the BFS approach involves replacing the queue with a heap data structure. This modification allows for more advanced scheduling strategies, such as prioritizing nodes based on specific criteria, like their duration, to implement Shortest Job First (SJF) scheduling.

The time complexity of this heap-based approach becomes $\mathcal{O}(V \log V + E)$, where the $\log V$ term arises from the insertion and extraction operations on the heap. These logarithmic operations increase the computational complexity when compared to the standard queue-based BFS. However, this trade-off allows for more refined control over the order in which nodes are processed.

2.2 Unlimited Workers

Now we consider the case of parallel execution of tasks with an unlimited number of workers. The goal is to minimize the total execution time, meaning that a task is executed as soon as all its prerequisites are completed.

Algorithm 3 Unlimited Workers Scheduling Algorithm

```
Require: $G = (V, E)$ DAG
Ensure: A scheduled task's end time
 1: Initialize prerequisites count for each task
 2: Initialize a min-heap H (stores tasks sorted by end time)
 3: for each task $T \in V$ do
 4:     if $T$ has no prerequisites then
 5:         Insert $T$ into H with its duration
 6:     end if
 7: end for
 8: while H is not empty do
 9:     Extract task $T$ with the earliest end time and set it as task's end
10:     for each successor $S$ of $T$ do
11:         Decrease $S$'s prerequisite count by 1
12:         if $S$ has no remaining prerequisites then
13:             Insert $S$ into H with $T$'s end time + $S$'s duration
14:         end if
15:     end for
16: end while
```

Inserting a task into the heap can be seen as starting the task, and extracting from the heap represents the completion of a task. In this implementation, time progresses by extracting tasks from the heap, where each extraction corresponds to one time step. The complexity is $\mathcal{O}(V \log V + E)$ as in the heap version of topological sorting.

2.3 Limited Workers

In the case of limited workers, we encounter the problem that a task can only be started if a worker is available. To address this, we extend the previous approach by using a second heap to store tasks that could be started when a worker becomes available.

Algorithm 4 Limited Workers Scheduling Algorithm

```
Require: $G = (V, E)$ DAG, number of available workers $W$
Ensure: A scheduled task's end time
 1: Initialize prerequisites count for each task
 2: Initialize min-heap startable (tasks with no prerequisites, sorted by duration)
 3: Initialize min-heap started (tasks being executed, sorted by end time)
 4: Initialize active_workers $= 0$
 5: for each task in $V$ with no prerequisites do
 6:     Insert the task into startable with its duration
 7: end for
 8: while startable is not empty and active_workers $< W$ do
 9:     Extract the task with the shortest duration from startable
10:     Insert it into started with its duration
11:     Increment active_workers
12: end while
13: while started is not empty do
14:     Extract task $T_1$ with the shortest end time from started and set it as task's end
15:     Decrement active_workers
16:     for each successor $S$ of $T_1$ do
17:         Decrease $S$'s prerequisite count by 1
18:         if $S$ has no remaining prerequisites then
19:             Insert $S$ into startable with its duration
20:         end if
21:     end for
22:     while startable is not empty and active_workers $< W$ do
23:         Extract task $T_2$ with the shortest duration from startable
24:         Insert $T_2$ into started with $T_1$'s end time $+$ $T_2$'s duration
25:         Increment active_workers
26:     end while
27: end while
```

The complexity of this algorithm is $\mathcal{O}(V \log V + E)$, as each task is inserted and extracted twice from a heap, but constant factors disappear in the $\mathcal{O}$-notation. This approach minimizes the total execution time by keeping as many workers busy as possible. Short tasks are prioritized and executed first, leading to higher task availability in shorter periods. As a result, more workers are likely to be active at any given time.

3 Evaluation

We now evaluate the implementation, focusing on two main aspects: verifying the correctness of the algorithms and assessing whether their theoretical time complexity matches practical results.

To do so, we generate a random DAG using the function described in (1.3). The same DAG is used for all methods at a fixed input size. Specifically, for an input size N, we construct a graph with $V = 2^N$ vertices and $E \approx V^2/4$ edges. The adjacency matrix is upper triangular, and each potential edge in the upper triangle is included with a probability of $1/2$.

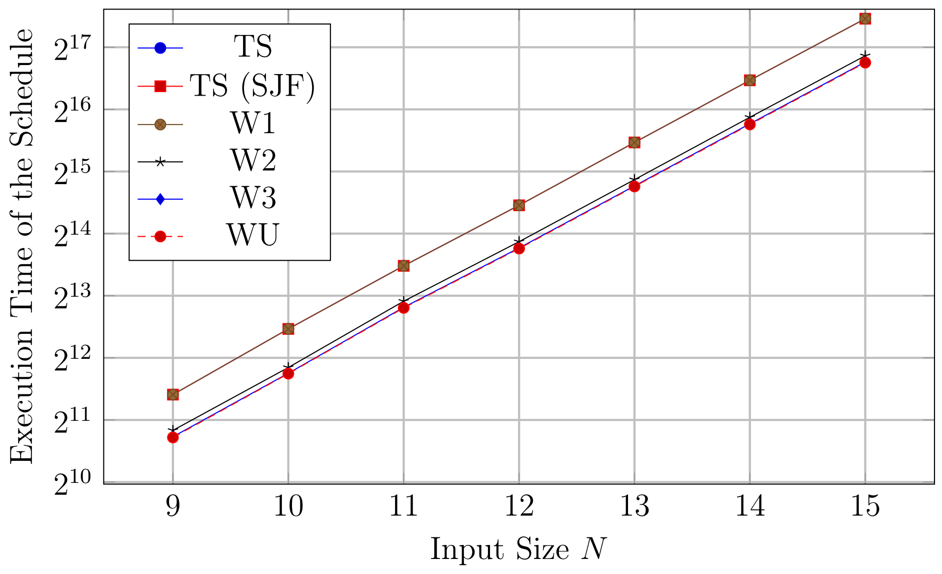
3.1 Correctness

To verify correctness, we first apply a test to ensure that each task's end time is scheduled correctly. The following function checks whether all successor tasks start only after their predecessors have finished:

```c
static bool test_correctness(struct Task *dag, int N){
    for(int task_idx = 0; task_idx < N; task_idx++){
        double predecessor_end = dag[task_idx].end;
        for(int k = 0; k < dag[task_idx].ns; k++){
            int successor_idx = dag[task_idx].successors[k];
            double successor_start = dag[successor_idx].end - dag
               [successor_idx].duration;
            if(successor_start < predecessor_end){
                return false;
            }
        }
    }
    return true;
}
```

Listing 3.1: Correctness test

We also measured the total execution times of the computed schedules:



N	TS	TS (SJF)	W1	W2	W3	WU
9	2715.10	2715.10	2715.10	1821.58	1696.28	1683.22
10	5649.64	5649.64	5649.64	3673.14	3444.66	3430.39
11	11419.31	11419.31	11419.31	7697.48	7187.82	7145.76
12	22436.58	22436.58	22436.58	14952.61	13944.14	13851.40
13	45238.14	45238.14	45238.14	29872.42	27846.39	27679.49
14	90498.26	90498.26	90498.26	59905.24	55715.69	55307.31
15	179944.13	179944.13	179944.13	119010.45	110935.94	110210.01

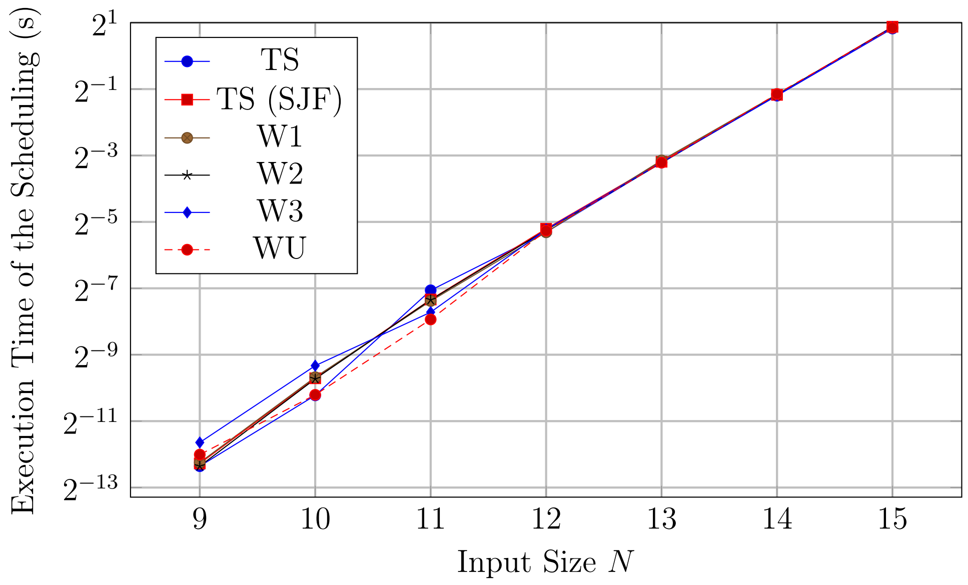
Table 3.1: Scheduled Times

One can observe that topological sorting with a queue (TS) and the SJF variant using a heap (TS (SJF)) result in the same total execution time. This is expected, as these methods only determine the execution order of tasks but do not affect the overall completion time. Similarly, the case with a single worker (W1) yields the same execution time. Increasing the number of workers reduces the total execution time, although this effect is bounded by the unlimited-workers case (WU). Notably, this bound is nearly reached with just three workers (W3). This occurs because, even with many workers, they may not be able to execute tasks simultaneously due to dependencies. In our graph, the dependency density is relatively high, leading to an early saturation.

It is also interesting to note that the execution time for all scheduling strategies scales in $\mathcal{O}(V)$. This is reasonable, given that the dependency ratio remains constant at $1/4$.

3.2 Complexity

Secondly, we examined the execution time of the scheduling algorithms themselves.



N	TS	TS (SJF)	W1	W2	W3	WU
9	0.000191	0.000202	0.000206	0.000191	0.000313	0.000243
10	0.000837	0.001196	0.001223	0.001181	0.001555	0.000846
11	0.007499	0.006189	0.006000	0.006149	0.004750	0.004082
12	0.025214	0.027080	0.025189	0.026390	0.026709	0.026021
13	0.108057	0.109822	0.112818	0.107184	0.107894	0.107733
14	0.435993	0.443397	0.447705	0.440896	0.438333	0.451488
15	1.771253	1.836509	1.796056	1.826926	1.812415	1.800244

Table 3.2: Scheduling Times

One can observe a complexity of $\mathcal{O}(V^2)$ for all algorithms, which aligns with the theoretical expectations. This is because the $\mathcal{O}(V^2)$ term, corresponding to the number of edges $E = \mathcal{O}(V^2)$, dominates the $\mathcal{O}(V)$ or $\mathcal{O}(V \log V)$ term present in the scheduling strategies for this specific DAG. Nevertheless, one can also see that using a heap instead of a queue increases the execution time, as expected. At the same time, the other cases remain at a similar level. This might be due to the fact that the unlimited-workers case employs a second heap, but the size of the `started` heap is effectively bounded by the number of workers (which, in this case, is minimal) while the `startable` heap in the unlimited-workers scenario evolves similarly to the heap used in the other cases.

4 Conclusion

The goal of this project to develop and implement task scheduling algorithms for three different cases, along with the necessary data structures was successfully achieved. Below is a summary of the key implementation steps:

Implementation Overview	
Data Structures	
Task	Stores successors and the number of incoming edges
DAG	Implicit representation, DAG via upper triangular adjacency matrix
Queue	Circular buffer queue (as in TP), but dynamically resizeable
Heap	As in TP, but dynamically resizeable
Scheduling Algorithms	
Topological Sort	Implements BFS using a queue
Topological Sort (SJF)	Replaces the queue with a heap
Unlimited Workers	Uses a heap to track task execution over time
Limited Workers	Two heaps: `started` and `startable`
Evaluation	
Correctness	Verified as expected (TS and WU as edge cases)
Complexity	Matches theoretical expectations ($\mathcal{O}(V[\log V] + E)$)

Table 4.1: Project Summary

Future work could explore whether the proposed limited worker scheduling strategy yields optimal schedules, i.e., minimizing total execution time. Currently, shorter tasks are scheduled first under the assumption that this releases dependent tasks more quickly. However, prioritizing tasks with more successors could be another approach.

In the specific DAG structure used here, edges in the upper triangular adjacency matrix appear with uniform probability 1/2. Due to this structure, tasks with smaller indices tend to have more successors and may enable more tasks to start sooner. On the other hand, task durations are also randomly (uniformly) chosen, meaning that shorter tasks are not necessarily those with the most successors.