Graph Application in Multiprocessor Task Scheduling

Yusuf Rahmat Pratama, 13516062
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13516062@std.stei.itb.ac.id

***Abstract*—Task scheduling is commonplace for today's problems especially in the computer science field, including the optimization of computer's processing power to complete a certain task. In the search for optimal task scheduling, graph is often applied in many algorithms for such problem, and also to visualize the process of the task completion. This paper will explore the application of graph in task scheduling.**

***Keywords*—graph, task scheduling, application**

I. Introduction

Graph is a mathematical model which consists of vertices and edges that displays the connections between vertices by its edges. It is one of the most important and influential subject in Discrete Mathematics. As such, it has substantial usage and deviations in almost every field of study worldwide, especially in computer science and information technology.

One of the application of graph in computer science is in task scheduling. Task scheduling is the means to create a model that maps certain task that should be completed with minimal time, that is, with the smallest steps possible. The objective is done by optimizing the CPU to use its multiprocessing capability such that no task is conflicting with each other when discrete tasks are processed parallel to each other.

DAG (Directed Acyclic Graph) is a certain type of graph used for representing the task in task scheduling. The DAG is used to model the task needed to be completed in a certain order, with a possibility of dependencies among some tasks to another. Such dependencies require special care as a certain task needs to be completed before a dependent task could be processed. This is mainly the purpose of task scheduling.

II. Basic Theory

2.1. Graph

2.1.1. Graph Definiton

Acording to [1], a graph G = (V, E) consists of V, a nonempty set of vertices (or nodes) and E, a set of edges. Each edge has either one or two vertices associated with it, called its endpoints. An edge is said to connect its endpoints.

As defined, a graph couldn't contain a nonempty set of vertices, but could have an empty set of edges, which is called a *nulled graph*.

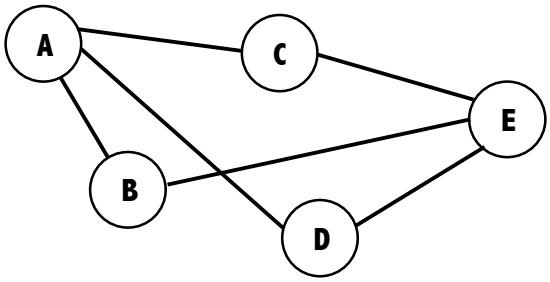

Figure 1. A graph of 5 vertices and 6 edges

2.1.2. Graph Terminology

Some terminologies regarding to graph are used to describe the properties, types, overall structure of a graph, as well as its characteristics.

1. Adjacent
   Two vertices of a graph are adjacent if and only if they are connected directly by an edge.
2. Incident
   An edge is incident to a vertex if the vertex is connected by the edge.
3. Isolated Vertex
   A vertex is isolated if and only if it doesn't have any incident edges.
4. Null graph
   A graph is a null graph if it contains an empty set of edges.
5. Degree
   Degree of a vertex is the quantity of edges incident to the particular vertex.

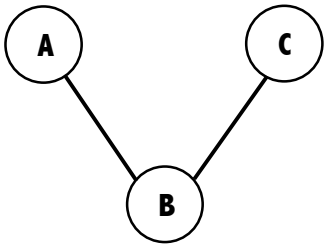

Figure 2. Vertex B has a degree of 2

6. Path
   A path with length n from a vertex v_0 to a vertex v_n in a graph G is a sequence $v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n$ such that $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, …, $e_n = (v_{n-1}, v_n)$.
7. Circuit/Cycle
   A circuit/cycle is a path in a graph G which starts and ends in the same vertex v_0.
8. Connected Graph
   A graph is connected if there is a path from every vertex to every vertex excluding itself.
9. Subgraph
   A graph G = (V, E) has a subgraph $G_1 = (V_1, E_1)$ if and only if $V_1 \subseteq V$ and $E_1 \subseteq E$. A subgraph G_1 is called spanning subgraph if $V = V_1$.

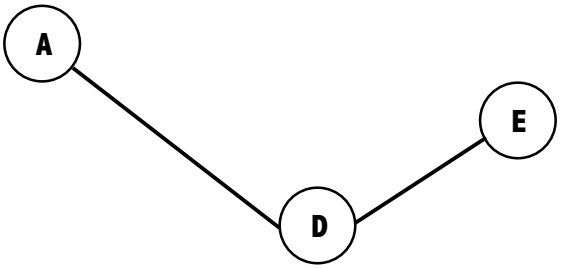

Figure 3. A subgraph of the graph in Figure 1

10. Weighted Graph
    A weighted graph is a graph in which every edge has a respective weight.

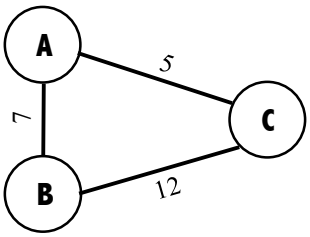

Figure 4. Weighted graph

2.1.3. Types of Graph

Graphs can be classified into many types based on their distinct properties, various numbers of its elements, or overall structures.

Based on the edge's characteristics, specifically the edge's direction, graphs can be distinguished into two types:

1. Non-Directed Graph
   A non-directed graph consists of edges that don't have specified direction. The graphs in Figure 1, Figure 2, and Figure 3 are all non-directed graph.
2. Directed Graph
   A directed graph has edges in which each edge has a specific direction.

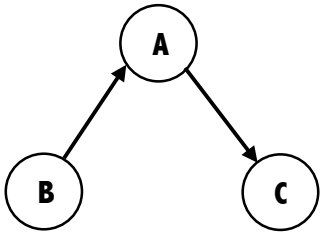

Figure 5. A directed graph

Based on the edge's types, graphs can also be grouped into three types:

1. Simple Graph
   A simple graph is a graph without any loops or parallel edges. Figure 1 is an example of a simple graph.
2. Multigraph
   A graph that has multiple edges connecting the same vertices is called a multigraph.
3. Pseudograph
   Pseudograph is a graph which has loops or multiple edges connecting the same vertices.

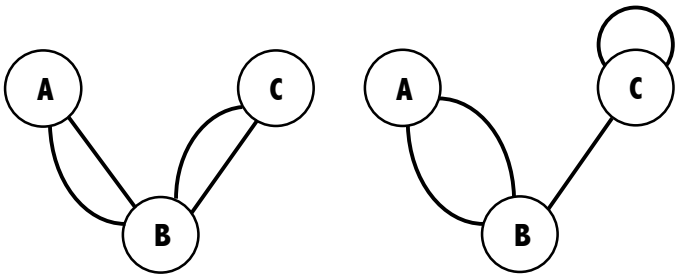

Figure 6. A multigraph (left) and a pseudograph with multiple edges in A-B, and a loop in vertex C (right)

Some simple graphs can also be classified based on its unique characteristics. Such graphs are called *specific graph*. Some of the specific graphs are:

1. Complete Graph
   Graphs in which each vertex is adjacent to every other vertex in the graph is called a complete graph. Such graphs are denoted by K_n, n representing the number of vertices.

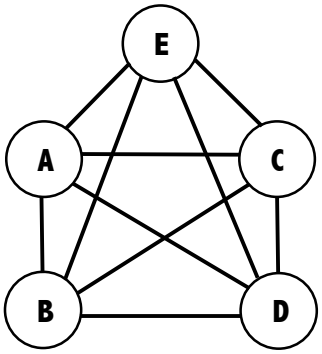

Figure 7. A complete graph K_5

2. Regular Graph
   A regular graph is a graph in which every vertex has the same degree. A regular graph with n-degree for each vertex is called an *n-regular graph*.
3. Cyclic Graph
   A cyclic graph is a graph that contains at least one cycle.

2.2. Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a special type of graph in which every edge in the graph has a specific direction, and no cycles exist in the graph, hence acyclic. Every DAG has a *topological sort* form, in which for all vertices, each vertex *v* always appears earlier before every other vertex reachable from *v*. Consequently, in a DAG there could be vertices that are *dependent* on other vertices. Such dependencies are important pieces of information especially for DAG's applications.

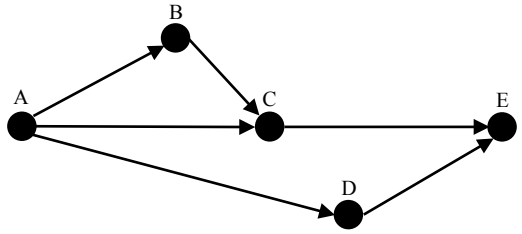

Figure 8. Direct Acyclic Graph

There are substantial applications for Directed Acyclic Graph, especially in the field of computer science. Some applications are in scheduling, data processing networks, structure modelling, genealogy and version history modelling, data compression, and more usage exist in other field of study in many variations of the DAG.

2.3 Task Scheduling

Task scheduling is the process of mapping the most efficient way of processing certain tasks until completion. The goal of task scheduling is to minimize the completion time for all the tasks, find an efficient execution of the tasks at hand, or maximize the throughput of the respective process.

Task scheduling has many implementations in a diverse set of fields of studies. Logistical problems, job scheduling problems, etc. can be based on task scheduling, with the same intention of finding the most efficient task processing. In computer science, task scheduling is crucial in compilers, mapping the most efficient way of disassembling the program and cross-linking registers to memory, to reduce compiling time and program execution time.

The task scheduling problem is modeled as a Direct Acyclic Graph (DAG). The vertices represent the tasks and their respective weight (the size of task computation), and the edges represent the dependency between two tasks.

Multiprocessing capabilities in most of today's computers make it essential in task scheduling. Thorough usage of parallelism could drastically increase time efficiency. However, the execution of some tasks may require the completion of other tasks (dependency). Therefore, the scheduling of dependent tasks needs to be considered such that the process will be optimized and the correctness of the output will be maintained.

III. Implementation of Direct Acyclic Graph in Multiprocessor Task Scheduling

In task scheduling, direct acyclic graph is almost always implemented as a model for the data of the tasks required to be processed, complete with their size of computation.

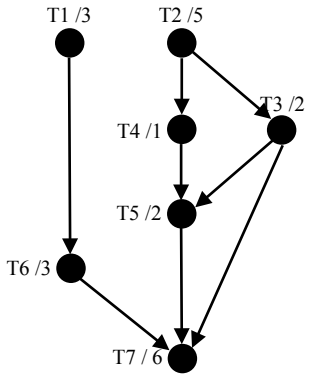

Figure 9. Implementation of DAG in Task Scheduling

Figure 9 shows an example of a task scheduling problem, presented in direct acyclic graph, with its vertices representing the tasks and their computational size, and its edges indicating the tasks' dependencies on another. In task scheduling, *indirect dependencies* may occur and can be viewed in the DAG. Such example in Figure 9 is the indirect dependency of T3 and T7. An indirect dependency of a task is when a task *v1* depends on *v2*, and the task *v2* depends on *v3*, then the task *v1* has indirect dependency on task *v3*.

Indirect dependencies are considered redundant in task scheduling, because an indirect dependency can only be completed when the tasks of direct dependencies representing the indirect dependency in the task are processed. Accordingly, the DAG of a task scheduling is usually *cleaned up* before proceeding into scheduling the tasks, so that it will not confuse the algorithm.

The output of a task scheduling is the model of assignments of the tasks in the processors, or the order of processing the tasks, so that the processors will have the instructions needed about which tasks should be done first, and whether if such tasks can be done in parallel. Note that the output can have many possibilities of combination with the same efficiency, not always a solution is absolute. Consequently, the goal is to acquire at least one possibility of solution to be implemented in processing the tasks.

In order to solve the task scheduling problem, including the multiprocessing nature of processing the tasks, some algorithms have emerged which can handle the problem

effectively and produce outputs that are efficient in task scheduling. Logically, direct acyclic graphs are always implemented in applying these algorithms to visualize, find patterns, and even to implement such algorithms visually by hand.

Below are some algorithms that are commonly used in solving the task scheduling problem with the implementation of the direct acyclic graphs.

3.1. The Coffman-Graham Algorithm

The Coffman-Graham algorithm is an algorithm used to solve multiprocessing task scheduling by arranging the tasks such that a task that is dependent on another task is assigned to later level, and each level doesn't exceed the number of processors available.

The Coffman-Graham algorithm assumes all tasks have the same amount of computational size.

The steps of the Coffman-Graham algorithm are [4]:

1. Represent the DAG by *transitive reduction*, that is, by cleaning up the graph of indirect dependencies so that only direct dependencies are represented. Transitive reduction is required to remove redundancies in the graph. The resulting direct acyclic graph may not be the same as the preceding graph.
2. Order the tasks so that a dependent task is assigned after the task it depends on. Furthermore, to maximize the parallel capabilities, assignments of each of the dependencies should be spaced apart as much as possible, as if a task *v1* is dependent on a task *v2*, they couldn't be processed in parallel, as processing of *v1* requires completion of *v2*. This leads to the idling of a processor, which reduces the efficiency of the scheduling. Altogether, *topological ordering* of the dependencies graph is required so that the starting tasks which doesn't have any dependencies are processed first, and then progressively process the tasks that are dependent only on them recursively, until all of the tasks are ordered efficiently.
3. Fill up available CPUs by assigning the tasks that have been ordered accordingly one by one until all available CPUs are used in the particular time. If in a particular time the task assigned has a dependency that is also assigned at the same round, the CPU has to be idled as the dependency has to be completed first before the dependent task could be processed.

Note that the Coffman-Graham algorithm is optimal for W=2 number of processors. According to [5], for 2 processors, the relative performance of Coffman-Graham schedules (the product from the Coffman-Graham algorithm) is bound to 4/3. However, for W larger or equal to 3 processors, a lower bound of the worst-case relative performance is 3 - [6/(W + l)], in which the smaller number, the more efficient the performance is. Consequently, the more the processors, the less efficient the Coffman-Graham schedules will be.

Consider the DAG in Figure 9 to be processed with 2 processors. Assume the tasks have the same amount of computational size. To achieve the optimal result in processing the tasks, the Coffman-Graham algorithm is used.

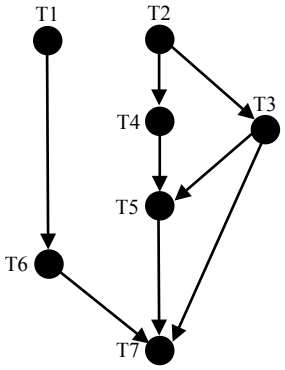

Figure 10. Modified DAG with no computational size

In the first step, transitive reduction is applied, removing all indirect dependencies of the tasks. Applying it to the DAG in Figure 10 results in a subset of the DAG.

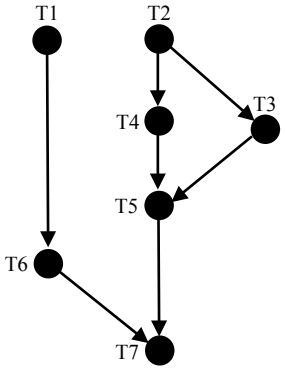

Figure 11. Step 1 of the Coffman-Graham algorithm

Secondly, the tasks are ordered with topological ordering, and other requisites stated on the second step. The output of the second step is a list of ordered tasks to be processed. Note that a handful of possible combinations may appear.

Table 1. One possible output of step 2 of the Coffman-Graham algorithm

T1	T2	T4	T6	T3	T5	T7

The instruction from the second step is then filled to the 2 CPUs to be completed. This will be done in step 3 with representation known as the *C-G schedules*.

Table 2. C-G schedules of the tasks. Note that 2 processes are tasked simultaneously in multiprocessors

P_1	T1	T4	T3	T7
	T2	T6	T5	*idle*
	t_1	t_2	t_3	t_4

From the C-G schedules above, it can be concluded that the tasks above require 4 unit-time for 2 processors. In all time, both CPUs are used except at the last processes where all the other tasks have been completed. This means that the algorithm was efficient for the example above in 2 processors.

3.2. Wave Front Method (WFM)

Wave front method are determined according to the level of vertices in the DAG, in which the vertices in each wave front is independent from each other, and are all assigned to different processors.

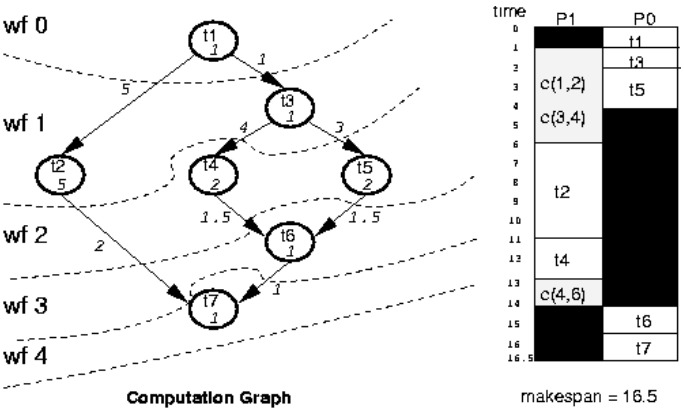

Figure 12. Example of WFM implementation in task scheduling. Notice that the tasks are processed in parallel for each level

Source: https://parasol.tamu.edu/

3.3 Critical Path Merge (CPM)

In critical path merge, a critical path in a DAG, which is the path from root to leaf with the most weight, is processed in the same processor, then the particular path is removed and the process is iterated for the rest of the DAG with different processors until all tasks are scheduled.

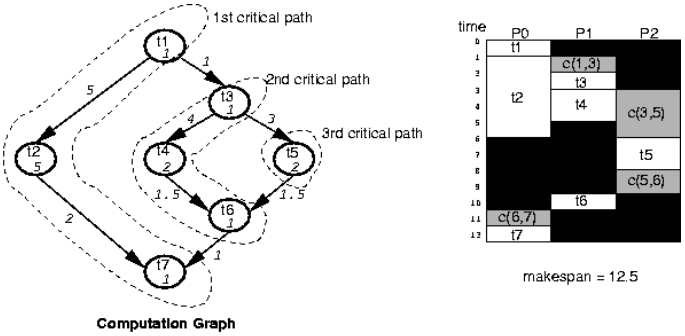

Figure 13. Example of CPM implementation in task scheduling with 3 processors.

Source: https://parasol.tamu.edu/

3.4. Heavy Edge Merge (HEM)

Some DAG used for task scheduling may have weight for representing the communication cost for the dependency of the task. HEM can be used to schedule task based on these edges' weights. HEM works iteratively by sorting the edges in non-increasing order. Every sorted group are assessed by one processor. After all are sorted, the makespan is calculated by merging the path that includes the endpoints of the graph.

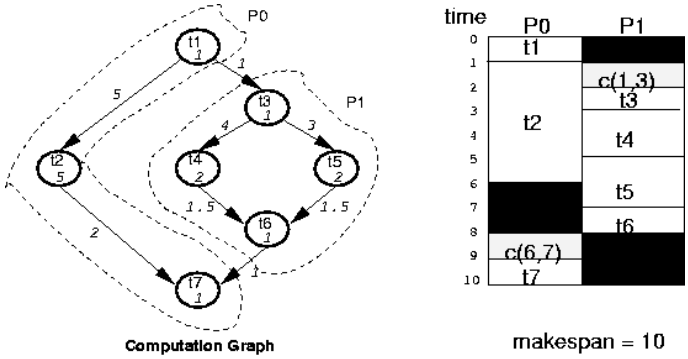

Figure 14. Example of HEM with 2 processors.

Source: https://parasol.tamu.edu/

IV. Conclusion

Graphs have many applications and implementations across all fields of study, including in solving the task scheduling problem. For task scheduling, a special type of graph is used, which is the directed acyclic graphs (DAG). These graphs are useful for task scheduling as it represents the tasks required to be completed and the dependencies between each task. From the DAG, several types of algorithms can be used to find the most efficient procedure of scheduling the task, so that multiprocessor capabilities can be used in its full extent to minimize the total time consumed in finishing the task. All in all, DAG plays an important role in modelling the task scheduling problem, and in solving the problem with various algorithms available.

V. Acknowledgment

The writer would like to thank God for without Him nothing could ever be achieved. The writer would also like to thank Mr. Rinaldi Munir as the coordinator of the Discrete Math lectures and initiator of the task, as well as Mrs. Harlili as Discrete Math's lecturer. The writer thanks every author in which his or her works are referenced in the writer's paper. Last but not least, the writer thanks his parents for continually supporting the writer's journey in life.

REFERENCES

[1] Rosen, Kenneth H. *Discrete Mathematics and Application to Computer Science 7th edition*. New York: McGraw-Hill, 2012.

[2] Bang-Jensen, Jørgen. "Acyclic Digraphs," *Digraphs: Theory, Algorithms and Applications, Springer Monographs in Mathematics*. Berlin: Springer-Verlag, 2008.

[3] Munir, Rinaldi. *Diktat Kuliah IF2120 Matematika Diskrit*. Bandung: ITB, 2006.

[4] Coffman, E. G., Jr; Graham R. L. "Optimal scheduling for two-processor systems," *Acta Informatica 1*. Berlin: Springer-Verlag, 1972.

[5] Hanen, C.; Munier, A. "Performance of Coffman-Graham schedules in the presence of unit communication delays," *Discrete Applied Mathematics*. Paris: Laboratoire LIAFA: 1998.

[6] Lehman, E.; Leighton, F. Thomson; Meyer, Albert R. *Mathematics for Computer Science*. Massachusetts: MIT OpenCourseWare, 2015.

[7] Fidel, A.; Amato, N.; Rauchwerger L.; Adams, N. "Task scheduling." Texas A&M University. https://parasol.tamu.edu/groups/amatogroup/research/scheduling/ accessed on 3 December 2017.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2017

Yusuf Rahmat Pratama - 13516062