Software Construction

RWTH AACHEN UNIVERSITY

The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Bachelor Thesis

A microservice framework for the generation and evaluation of Activity Network Diagrams

Ein Microservice Framework zur Generierung und Evaluierung von Aktivitätsnetzwerkdiagrammen

presented by

Dominik Lammers

Aachen, March 12, 2022

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Alex Sabau, M.Sc.

Acknowledgment

First of all, I would like to thank Prof. Dr. rer. nat. Horst Lichter for the opportunity to write my bachelor thesis at his chair. I also would like to thank him and Prof. Dr. rer. nat. Bernhard Rumpe for reviewing my thesis.

I would also like to thank my supervisor M.Sc. Alex Sabau for his guidance and constant support. I am grateful to him for his helpful suggestions and constructive criticism during the work.

Finally, I would like to thank my friends for their help and support with the last revision. Especially Marius Behrens, who was always there to discuss any questions or ideas that came up. Last but not least, I would like to thank my family for the tremendous support they have given me and the faith they have placed in me during my time of study and throughout this thesis.

Dominik Lammers

Abstract

Project management is indispensable in many fields, including in software development. Project networks are often used to visualize the various tasks and their relations within a project. A variant of project networks, the Activity-On-Node (AON) networks, can be used to schedule the tasks of a project and, thus, determine the total duration of the project as well as which tasks can not be postponed. As a central activity in project management, the understanding and application of AON networks is an essential part of the lecture "Software Project Management" of the Research Group Software Construction (SWC) at RWTH Aachen University. In this context, the SWC Research Group uses syntactic exercise and examination tasks (ex-tasks) on AON networks to deepen and evaluate the students' understanding. However, the creation and evaluation of such ex-tasks is time-consuming and error-prone, as there are no tools to support this process. This thesis focuses on the automatic generation of AON networks for ex-tasks. Moreover, due to the AON layout used in the lecture and ex-tasks, it is very challenging to generate such AON networks in this particular format, since known graph generation algorithms can not be used to generate graphs in the given format. We accomplish the generation and drawing of AON networks by leveraging existing approaches from graph theory and extending them for the domain of AON networks and our network structure. We develop a microservice framework to automatically generate AON networks and compute their properties. Using a microservice-driven approach, we are able to create an extensible prototype of this framework that can easily be extended in the future, to include additional features, such as automatic evaluation of ex-tasks.
Using this framework, we are able to create practical ex-tasks that only need minor manual adjustments to be usable. In addition, the framework allows us to import previously created networks, calculate their properties and potentially improve their structure. Although we focus on generating relatively small AON networks, our framework is generalized enough to also handle significantly larger networks.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

Planning and scheduling of tasks is part of our everyday life. Typically those tasks are not big enough to need proper planning, and it is sufficient to keep track of them using tools such as a calendar. However, this changes when it comes to more extensive projects, especially in the context of businesses. Such projects often consist of many different tasks involving several different people. Therefore, in order to first be able to plan all those tasks and their relationship between each other, and, second, not to lose track of them during the project duration, one needs to use proper tools. One of those tools is an *Activity Network Diagram (AND)*. An AND consists of two main parts, the activities and the dependencies between the activities. Activities are used to describe a task (i.e., description and estimated duration). The dependencies then describe the temporal relationship between the various activities. There are four different dependency types, each describing a slightly different relationship. For example, one dependency describes the case where one activity is only allowed to start after another is completed (e.g., one can only start building a house after finishing the foundation). The dependencies can then be used to calculate the earliest point in time where one can start with an activity and the latest point in time where it needs to be finished. In total, after successfully planning all the activities and their dependencies and calculating their additional properties, one knows when the project should be finished and which activities are critical (i.e., can not be delayed or extended without delaying the whole project).

Due to the importance of ANDs in project scheduling and progress tracking, they are covered in many project management lectures. Therefore, *exercise and examination tasks (ex-tasks)* are needed for these lectures to deepen and evaluate the understanding of students. However, creating such tasks is, on the one hand, time-consuming and, on the other hand, error-prone. This is because it does not make sense to repeatedly use the same or very similar tasks, so one needs to create new tasks every semester. In addition, the creation of such tasks is challenging by the fact that a small change (e.g., changing a duration of an activity) leads to significant changes within the AND, meaning it is not feasible to try out many different variants of an AND.

This thesis will address this problem by automating the creation of such tasks and provide the possibility to modify and recalculate them by using a graph modeling tool. The generation is performed by first creating an initial AND, then performing several steps needed to fulfill aesthetic criteria, to then ultimately draw it. We develop this functionality using a microservice-driven approach, to allow easy extension of our prototype.

1.1. Structure of this Thesis

In chapter 2, we briefly introduce the necessary background, which later chapters build on. Then, in chapter 3, we provide the motivation and problem statement for this thesis and also introduce our research questions. In chapter 4, we introduce some related work. Before beginning with our work, we provide some conceptual foundations our work is based on in chapter 5. Afterward, in chapter 6, we introduce the various steps we performed to create an AND, and transform it into a proper output. In chapter 7, we look at the design and the most interesting aspects of our prototype. Then, in chapter 8, we introduce the technologies we used and describe the graph modeling tool we used to work with ANDs. We evaluate how well we achieved the goal of generating tasks in chapter 9. In the end, we summarize the results of this thesis and give some ideas that can be done in future work, in chapter 10.

2. Background

This chapter provides the background that is needed in later parts of this thesis.

2.1. Graphs

The following section first gives a general overview of graph theory to, then, further introduce one special graph type. This section is mostly based on [Deo17].

Directed Graphs

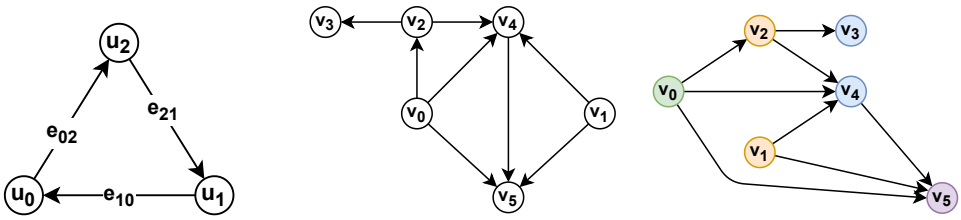
A *directed graph (digraph)* is usually denoted as $G = (V, E)$ containing the set of vertices V, and edges E. In this thesis, we can limit ourselves to non-empty finite vertex sets and finite edge sets. The vertices $V = \{v_1, ..., v_n\}$ can contain any arbitrary object, while the edges $E = \{e_1, ..., e_m\}$ are ordered pairs of vertices; thus, describing a connection only in one direction.

The degree of a vertex v is split into the *in-degree*, denoted as $d^-(v)$, and *out-degree*, denoted as $d^+(v)$. The in-degree is the number of in-going edges, while the out-degree is the number of out-going edges, of a vertex. For example, in Figure 2.1b the vertex v_2 has an in-degree of one and an out-degree of two.

Furthermore, there are different ways to traverse a digraph. First, a *path* defines a sequence of vertices and edges where each of them is only allowed to occur once (e.g., $(u_1, e_{10}, u_0, e_{02}, u_2$ in Figure 2.1a). A *circuit* is a special path where the sequence has to start and end in the same vertex (e.g., appending (e_{21}, u_1) in the previous example).

Directed Acyclic Graphs

We can, then, define a *directed acyclic graph (DAG)* to be a digraph that does not contain any circuit. A DAG has the special property that we can assign each vertex a layer so that all predecessors of a vertex are in a lower layer, and all successors are in a higher layer (e.g., we see in Figure 2.1c the layered representation of Figure 2.1b). Formally, we can define this as a partitioning into the sets $L_1, L_2, ..., L_h$ in such a way that if $(v, v') \in E$ and $v \in L_i$ as well as $v' \in L_j$, then $i < j$ must held. The number of layers (i.e., the number of partitions) is denoted as h, and is called the *height* of the DAG [HN02].



(a) Directed graph (b) DAG without layering (c) DAG with layering

Figure 2.1.: Example graphs

2.2. Project Management and Project Networks

Project management is the concept of planning, scheduling, and keeping track of the progress, of a project. The first, and most essential, step at the beginning of a project is to think about the different tasks the project consists of; thus, the planning [Hea16]. However, since our goal is to create synthetic ex-tasks where the structure and schedule are essential and not the actual meaning, we not detail the planning step.

The simplest and most general representation of a project is the *activity list*. In an activity list, all activities consist of four values: an identifier, a brief description, a duration, and a list of direct predecessor activities. The direct predecessor list of an activity contains all the activities that have to be completed before one can start that activity [HL05]. Table 2.1 is a general example of such an activity list.

The activity list is great for the planning step, but when it comes to scheduling and progress tracking, it is better to look at the project using a *project network*, which illustrates the flow of the project [HL05].

Identifier	Description	Direct Predecessor	Estimated Duration
A	Pour Foundation	-	4 weeks
B	Construct walls	A	2 weeks
C	Landscaping	A	3 weeks
D	Construct roof	B	1 week

Table 2.1.: Example activity list

2.2.1. Project Networks

A project network is generally used to visualize the *dependencies* between different activities. It can be seen as a DAG $G = (V, E)$ that has to visualize the following three properties: the activity identifier, the estimated duration, and the dependencies between activities. In general, two different formats of such a project network exist: the *Activity-On-Node (AON)* and the *Activity-On-Arrow (AOA)* network. In this thesis, we only use a variant of the AON networks, which we introduce in the following[1].

[1]For more information on AOA networks, refer to the following papers: [LG96; WG04]

In an AON network, the vertices describe the activities, and the edges describe the dependencies between activities. Therefore, each vertex consists of an activity identifier as well as a duration [Hea16]. It is essential to mention that an AON network usually contains exactly one start and one end activity. If one has more than one start activity, one must add a dummy start activity with dependencies to all initial start activities. Similarly, when more than one end activity is present, one must add a dummy end activity and create dependencies from the initial end activities to the dummy end activity. Both dummy activities have a duration of zero.

2.2.2. Dependency Variants for Activity-On-Node Networks

We already introduced a single *dependency type*, in Table 2.1, indicating that an activity is only allowed to start after all its predecessors are finished. Here, we introduce a total of four different dependency types used by AON networks [LG96].

1. *Finish-to-Start (FS)* - Figure 2.2. A FS dependency states that the target can only be started after the source activity has been finished for at least α time units.

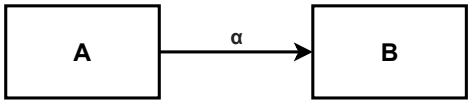


Figure 2.2.: Finish-to-Start dependency

2. *Start-to-Start (SS)* - Figure 2.3. The SS dependency states that the target is only allowed to start β time units after the source activity has started.

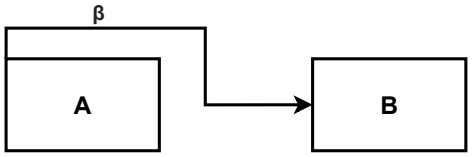


Figure 2.3.: Start-to-Start dependency

3. *Finish-to-Finish (FF)* - Figure 2.4. A FF dependency states that the target may only be completed if the source has already been finished for at least γ time units.

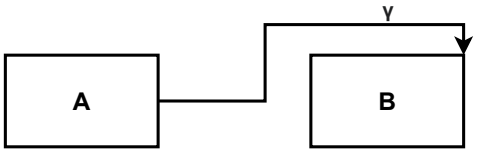


Figure 2.4.: Finish-to-Finish dependency

4. *Start-to-Finish (SF)* - Figure 2.5. A SF dependency states that the target is only allowed to finish after the source activity has been started for at least δ time units.

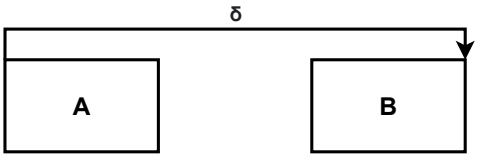


Figure 2.5.: Start-to-Finish dependency

2.2.3. Schedule Computation

Until now, we have only considered an activity's properties given prior to the scheduling (i.e., the activity identifier and the duration). We now extend the activity properties with information about the actual project schedule. For this, each activity gets the following *schedule properties*: *Early Start (ES)*, *Late Start (LS)*, *Early Finish (EF)*, and *Late Finish (LF)*. The ES and EF indicate the earliest point in time, and the LS and LF the latest point in time at which an activity can be started and completed [Hea16]. These properties are added to the visual representation of activities as shown in Figure 2.6.

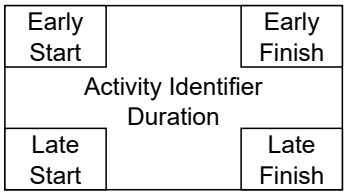


Figure 2.6.: Activity representation

Forward and Backward Pass

These four values are calculated in two phases, the forward pass and the backward pass. In the forward pass, the ES of an activity is calculated and used together with the duration to calculate the EF ($EF = ES + dur$). To calculate an activity's ES, one needs to consider all possible ES's resulting from in-going dependencies, and then set the activity's ES to the maximum of these values. This ensures that the activity can not start until all dependencies have been satisfied. Concretely, the forward pass starts by initializing the start activity with an ES and EF of zero and then calculates the ES and EF for each activity where all predecessors have already been considered [Hea16; LG96].

In the backward pass, the LF of an activity is calculated and used together with the duration to calculate the LS ($LS = LF - dur$). To calculate an activity's LF, one needs to consider all possible LF's resulting from out-going dependencies, and then set the activity's LF to the minimum of these values. Concretely, the backward pass starts by initializing the end activity's LS and LF with the ES and EF determined in the forward

pass, and then calculates the LF and LS for each activity where all successors have already been considered [Hea16; LG96].

Given the definitions for the different dependency types in section 2.2.2, we can define the following calculation rules that need to be used for the calculations in the forward and backward pass. As mentioned, we calculate the ES in the forward pass and the LF in the backward pass. We will use the notation X_A, with $X \in \{ES, EF, LS, LF\}$ to refer to the properties of an activity A, which has a duration dur_A.

1. FS dependency: $ES_B = EF_A + \alpha$ and $LF_A = LS_B - \alpha$.
2. SS dependency: $ES_B = ES_A + \beta$ and $LF_A = LS_B - \beta + dur_A$.
3. FF dependency: $ES_B = EF_A + \gamma - dur_B$, where $ES_B = 0$ if $EF_A + \gamma < dur_B$. In addition: $LF_A = LF_B - \gamma$.
4. SF dependency: $ES_B = ES_A + \delta - dur_B$, where $ES_B = 0$ if $ES_A + \delta < dur_B$. In addition: $LF_A = LF_B - \delta + dur_A$.

Float and Critical Path

The *float* of an activity is defined as the additional time that can be used for an activity. The float always considers starting at the ES of an activity. This means that if a previous activity has already used some of its float, one can no longer use the full float of the current activity. The float can be calculated as follows: $float_A = LF_A - ES_A - dur_A$ [LG96].

An activity is considered *critical* if it has a float of zero, indicating that this activity should not be postponed even for a single time unit. If this happens, it directly impacts the total project duration. The *critical path* is then defined as a path that only contains critical activities and starts at the start activity and ends at the end activity [HL05].

2.2.4. Activity Network Diagrams in the Lecture

In the following, we explain the particular features of the AON networks used in the *Software Project Management (SPM)* lecture at the *RWTH Aachen*. We refer to the SPM lecture's AON variant as *Activity Network Diagram (AND)*.

First, we call activities *work packages*. A work package consists of an identifier, a description, a duration, the four schedule properties, and predecessor relationships. Furthermore, we slightly change the layout of the dependencies we saw in Figure 2.2 to Figure 2.5. For us, a start-to-x dependency always leaves a work package on the left side, and an x-to-start dependency always enters a work package on the left side. The same goes for a finish-to-x dependency, which always leaves a work package on the right side and an x-to-finish dependency which always enters a work package on the right side.

In addition, we consider the FS dependency as the *standard dependency type* and the SS, SF, and FF dependencies as *special dependency types*; thus, most of the dependencies have the FS type. Additionally, we also omit α, β, γ, and δ for dependencies.

In Figure 2.7, we can see an AND that was used as an ex-task in a previous year, which was given as the *reference AND* for this thesis. We can see that we have exactly one start and one end work package. To better identify the critical work packages, they are highlighted in red. Furthermore, all special dependencies are labeled with their type.

Ex-Task Properties

The ex-task type for which we want to generate ANDs is solely used to check whether students can apply the calculations rules we introduced in section 2.2.3. Therefore, we omit the work package description since a unique identifier is sufficient in such ex-tasks. In the following, we describe the minimum requirements an ex-task should satisfy.

Since each dependency type has a different calculation rule, an ex-task needs to have at least one dependency of each type, to see if a student can apply the different rules. In addition, one must also include an activity with multiple in-going and an activity with multiple out-going dependencies to confirm if the student knows that one needs to take the maximum and minimum of the ES and LF, respectively.

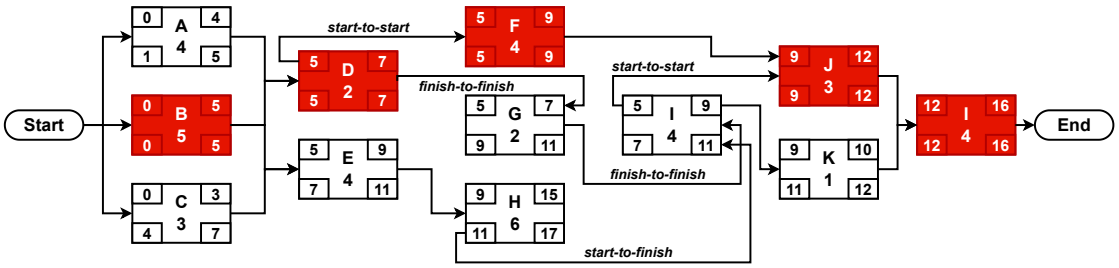


Figure 2.7.: Reference AND taken from a previous exam task

2.3. Microservices

One problem in large software projects is the increasing size of the project over time. Therefore, the codebase of such projects tends to get large. In a *monolithic system* this is mainly handled by ensuring that the code is as *cohesive* as possible; thus, code that is somehow related is grouped [New15]. This approach can also be called *Single Responsibility Principle (SRP)* which states that, on the one hand, everything that one needs to change due to a similar reason should be grouped, and, on the other hand, things that change for different reasons should be separated [New15]. Following this approach minimizes the effort of applying changes because one most likely only has to change the affected parts [New15; Lar+18].

The concept of *microservices* applies the SRP by dividing each functionality into an independent service. By doing so, one separates each functionality and focuses only on performing their designated task. This independence, and the small size of services, enables the use of different technologies within each service [Ric19], promotes an agile

development approach [SW21; Han10], and allows services to scale more dynamically [Ric19].

One challenge when working with microservices lies in the definition of their boundaries. In general, it is clear which functionalities should be divided into more minor services, however, how granular this division should be is usually not so simple. One rule that helps to check whether one should further split up a service is how many people are needed to manage that service. It is preferable to only have a small team be responsible for a service [Ric19].

Furthermore, due to their independence, services also need ways to communicate. Therefore, one must adequately plan what a service exposes, as to not reduce their independence (e.g., forcing services to use a specific technology). Usually, a service exposes its functionality with an *Application Programming Interface (API)* [Ric19]. However, often the user is only allowed to access a system through an *API-Gateway* which can be used to encapsulate the underlying services or to aggregate the results of some services [Ric19]. Thus, the gateway provides convenient methods to access the needed resources without exposing any information about the actual services behind the gateway to the user [Ric19].

3. Motivation and Problem Statement

Project networks are widely used to support project scheduling and progress tracking. As a result, they are usually covered in project management lectures, such as the SPM lecture this thesis focuses on; therefore, ex-tasks are required each semester to check students' understanding. However, there are no tools that support the creation of such tasks, apart from diagram drawing tools. This means one must manually create the work packages and dependencies and assign values to them. At first glance, this does not seem to be a big problem since one only needs to assign a duration to the work packages and create dependencies between them to then calculate the scheduling properties. However, in ANDs, the schedule properties are very sensitive to changes; thus, when a single duration is changed, or a dependency is created or modified, the schedule properties must usually be recalculated for most work packages. In addition, these calculations are very error-prone and time-consuming. This leads us to the first research question of this thesis.

> **RQ1:** *Can we automatically generate ANDs that are practical for ex-tasks?*

One problem when automatically generating graphs is the creation of an easy-to-read and understandable output. It is not helpful to generate ex-tasks for ANDs if it takes a long time to understand them or manually restructure them into an understandable structure. In addition, this would also reduce the benefit of being able to import an AND to calculate its properties since one would then have to restructure the output anew each time. This leads to the second research question:

> **RQ2:** *Can we achieve a similar output structure as that of manually created ex-tasks?*

3.1. Additional Requirements

Certain additional requirements for the generation and processing of ANDs were given. As mentioned above, along with being able to generate ANDs, we want to be able to import modified or self-created ANDs. Importing ANDs will allow us to adjust the values of generated or self-created ANDs without the need to recalculate everything manually. However, this means that we need a suitable graph modeling tool that allows us to work with ANDs manually and import and export them easily.

In addition, it is important that it should be possible to extend the functionalities we provide in this thesis easily. At the same time, creating such extensions should not necessitate a complete understanding of the system, to make the implementation

of extensions simpler, and thereby, faster. Therefore, we had to design the application using microservices that allow such easy extensibility as presented in section 2.3.

4. Related Work

In the previous chapter, we discussed the motivation of, and problem with, creating ex-tasks manually, which led to our research questions. In this chapter, we now look at related work concerning the generation of ANDs.

4.1. Task Graph Generation

We first discuss the generation of task graphs provided in [DRW98]. Dick et al. developed a generation approach for task graphs that can be used to evaluate and compare allocation or scheduling methods in embedded real-time systems and operation systems [DRW98]. Although this application area is in a different domain, the task graphs they generate have much in common with our ANDs. Task graphs are DAGs, the vertices describe a task, and the edges are the communication between tasks. Furthermore, the produced graphs also have exactly one start vertex (i.e., with no in-going edges) and do not contain duplicated edges. The generation process is also configurable so that one can define a boundary of how many vertices one wants to generate, and one can define their maximum in-degree, and out-degree [DRW98]. For example, in Figure 4.1 we can see a generated task graph with a maximum in-degree and out-degree of two, and the number of vertices was set between 8 and 12.

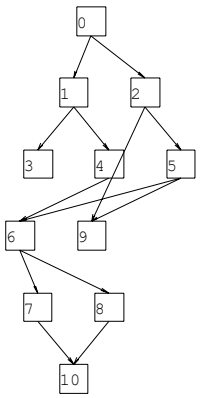


Figure 4.1.: Example graph generated by Task Graphs For Free (TGFF) [DRW98]

As the authors themselves state, the generation relates to the domain of embedded real-time and operating systems but can be applied more generally to many scheduling domains [DRW98]. An extension of their initial method, proposed by Vallerio et al. in [Val08], provides the possibility of having exactly one vertex with no out-going edges,

which is then even more similar to our AND structure. Thus, when we consider ANDs, we would only need to convert tasks into activities having our properties and assign a dependency type to the edges to obtain a valid AND.

However, these generation approaches are not suitable for generating ex-tasks. The reason being, that their main application is to provide a standardized generation tool for evaluating and comparing scheduling methods. In doing so, the main goal of the authors was to provide a tool capable of generating different types of graphs and configuring many properties of the generated graphs [DRW98]. However, the output as seen in Figure 4.1 is far away from the layout we saw in the reference AND in Figure 2.7; thus, the graph drawing is not usable for us. In addition, we later introduce certain restrictions we want to have during the creation, which can not be assured when using this generation tool (e.g., restricting the edge length or the number of layers).

4.2. Summary

We have presented a generation tool suitable for generating syntactically valid ANDs, but since it was developed for a different domain, one would need to convert vertices and edges into activities and dependencies with their respective properties. Furthermore, although it is highly configurable, it lacks configuration options needed for ex-tasks.

There is also a generator for AOA networks proposed in [AEH96], and one for AON networks in [DVH03]. However, both approaches focus on creating hard project management problems (e.g., considering limited resource and their allocation) to evaluate the performance of planning methods, and since the lecture ANDs do not consider resource allocation, they are of no use for us.

Furthermore, due to the lecture's particular AND structure (e.g., dependencies go out on the left and right depending on their type), there exist no tools or methods for automatically creating a drawing of such an AND. Hence, we need to introduce a proper method to do this.

5. Conceptual Foundations

This chapter introduces concepts that we will use or extend in the following chapters. We first introduce a method for generating graphs. Then we introduce a general set of rules that define when a graph is considered readable and explain typical steps that a graph goes through to satisfy these rules.

5.1. Generation Method

This section introduces the graph generation method that we base our AND generation on, and why we can not directly use it.

The *layer-by-layer* method is used to create DAGs. To do this, one first defines the number of vertices n and the number of layers k that one wants to generate. The method, then, considers every possible forward edge (i.e., an edge from a layer x to a layer $x + i$ with $i > 0$) and creates them with a predefined probability p. For example, a vertex in the first layer considers all vertices from layer 2 to layer k and creates an edge to each with a probability of p [Cor+10; CSH19].

The problem with this method is that it produces DAGs with properties that are unfavorable to us. For example, it always considers all possible forward dependencies without any constraints. Therefore, the vertices in the first layer will most likely have the most edges, and the further back a vertex is, the fewer edges it will have. For this reason, we will use the basic idea of the layer-by-layer method to only generate forward edges and extend it to match the constraints we later define.

5.2. Draw a Directed Graph

This section describes the general approach for drawing digraphs. In doing so, we introduce a general set of rules that define the properties that a well-readable and understandable graph drawing should satisfy. These rules are necessary because *well-readable* and *understandable* are subjective notions. We then present the typical steps taken to satisfy them, which we later adapt for our needs.

5.2.1. General Digraph Drawing Rules

There are several rules used in literature that defines a well-readable and understandable graph drawing. We will now use the rules succinctly defined by Bastert et al. in [BM01]:

- *Edges should primarily point in a single direction*
- *Vertices should be evenly distributed*
- *Long edges should be avoided*
- *Crossings between edges should be avoided as much as possible*
- *Edges should contain as few bends as possible*

Various related procedures are used to create a graph drawing that complies with the given rules, most of which are based on the *Sugiyama Method* [STT81]. The Sugiyama method consists of four steps: transforming the graph into a proper hierarchy, reducing edge crossings, determining vertex positions, and lastly, drawing the graph [BM01; STT81]. In the following, we describe these steps.

5.2.2. Layer Assignment

Before the layer assignment, one must usually transform a digraph into a proper DAG by removing cycles [EX89], but since we only consider DAGs in this thesis, we can omit this step here.

Each vertex in the DAG is then assigned to a proper layer as defined in section 2.1. In this step, one usually temporarily inserts dummy vertices into the graph, used only to have edges between two consecutive layers. For example, one would add a dummy vertex in layer 2 when having an edge between layer 1 and 3. The edge would then be represented by the two edges between the source and dummy and dummy and the target vertex. This is done because only considering edges between consecutive layers simplifies the crossing reduction applied later. In addition, the dummy vertices are also used to find a proper layout (i.e., vertex position in the drawing) in the final drawing step [BM01; EW94]. The disadvantage of this approach is that one may have to add a lot of such dummy vertices [HN02].

We do not consider dummy vertices later and adapt the following approaches to not rely on them. The general reason why we do not use them is the unique dependency layout of ANDs, where dummy vertices are of no use in the layout step. We explain this in more detail in section 6.5.

We now present a simple method for applying a proper layering, the *Longest Path Algorithm (LPM)*. The LPM assigns a vertex to a layer based on the longest path from the vertices with no predecessors (called root vertices) to itself [TC08; HN02]. For example, assuming the root vertices are in L_1, a vertex v gets assigned to the layer L_n, where $n = \mathit{length}(\mathit{longestPathToRoot}(v)) + 1$. By doing so, we assign the first possible layer to each vertex which leads to a layering with the minimum height, which often leads to a dense clustering of vertices near the roots [TC08]. We can directly apply this to our ANDs since we only have a single root vertex, the start work package. For more layering algorithms, refer to [TC08].

5.2.3. Crossing Reduction

In this step, after getting a proper layering, one tries to find a vertex ordering that reduces the number of crossings between edges as much as possible. This problem is, in general, NP-complete; thus, there exists no efficient algorithm solving it [GJ83]. Therefore, some heuristics are used to minimize crossings while still being relatively efficient. However, it is essential to mention that the methods we introduce in the following are typically applied to normal DAGs (i.e., straight lines, edges leave vertices at any point, only edges between consecutive layers), and ANDs have a different structure. We address this by introducing an abstract view of the AND and introducing proper methods to handle edges between layers that are more than one layer apart.

The remainder of this section mostly considers the crossing improvement between two consecutive layers. Therefore, one often uses *bipartite graphs* to describe the approaches. A bipartite graph is an undirected graph $G = (V, E)$ where it is possible to partition L into two sets L_1, and L_2 in such a way that there are only edges from L_1 to L_2 or vice versa (i.e., if $(v, u) \in E$ then $v \in L_1$ and $u \in L_2$, or $v \in L_2$ and $u \in L_1$). The ordering of the vertices in L_1 and L_2 can be denoted as π_1 and π_2 [BM01]. We explain a method that uses the crossing reduction between two consecutive layers, and extend this approach to be able to handle graphs with more than two layers.

Optimal Assignment with Permutation Method

One can find the optimal ordering of the vertices in L_1 and L_2 by calculating the number of crossings for each possible permutation and then using the ordering that results in the fewest crossings. The problem is that the number of permutations is large even when considering only a few vertices; thus, this approach is not feasible for most graphs (i.e., it would be time-bound by n-factorial ($n!$) where n is the number of vertices). Even restricting it to find an ordering of one partition while the other is fixed is still very inefficient [JM96].

Layer-by-Layer Sweep

Due to the difficulty of reducing crossing between two consecutive layers where both can be permuted, the most common method always considers one layer mutable and all other layers fixed. The problem of this approach is that it is, in most cases, not possible to find an ordering with as few crossings as possible by just reordering vertices in one layer [JM96].

The *layer-by-layer sweep*, is used to handle the problem of just reordering a single layer. In the first step of the layer-by-layer sweep, one assigns each vertex a position within its layer (i.e., create an initial layer ordering). Then, in the forward sweep, one iterates from the second to the last layer, and for each layer $i \in \{2, 3, ..., h\}$ where h is the height of the graph, one fixes the layer L_{i-1}. One then uses a heuristic to reduce the number of crossings between two consecutive layers which produces a new ordering of layer L_i. In the backward sweep, one runs from the penultimate layer, L_{h-1}, to the first

layer, L_1, in each step fixing layer L_{i+1}. One then again applies the crossing reduction heuristic on the layer L_i. The forward and backward sweep is then repeated until the heuristic can no longer reduce the crossings within the graph [ESK05; EW94].

For example, when considering a bipartite graph, one would first fix the first layer in the forward sweep. One then applies a heuristic that tries to find an ordering of the second layer that produces fewer crossings than the current ordering. In the backward sweep, one would then fix the second layer and apply the heuristic to the first layer. The procedure then repeats until no improvement can be found anymore.

The results produced by the layer-by-layer sweep are sometimes not optimal. However, since considering all layers simultaneously is a challenging problem, it is still a reasonable approach. Nevertheless, one can extend the layer-by-layer sweep with a randomization of the ordering [BM01]. This randomization can be applied if the layer-by-layer sweep can not further improve the graph. In this case, one applies a new random ordering to each layer and starts again with the layer-by-layer sweep. By repeating this process several times, one may find a better result.

Barycenter Heuristic

This section is primarily based on [BM01].

One heuristic that can be used in the layer-by-layer sweep as crossing improvement is the *Barycenter Heuristic*. In general, it tries to assign neighboring vertices close to each other which is based on the intuition that placing adjacent vertices close to each other should only produce a few crossings.

This heuristic first calculates, for each vertex in a layer, the average position of their neighbors. We first define the set of neighboring vertices $N(u)$ where $N(u) = \{w \mid (u,w) \in E, \text{ or } (w,u) \in E\}$ and use it to calculate the barycenter value for vertex u as follows:

$$bary(u) = \frac{1}{deg^-(u) + deg^+(u)} \sum_{w \in N(u)} \pi_i(w), \text{ with } w \in L_i \tag{5.1}$$

After getting all the barycenter values for one layer, one can sort the vertices of that layer by their barycenter values and assign a new position based on the sorting.

Greedy Switch Heuristic

This section is based chiefly on [BM01].

The *Greedy Switch Heuristic* works quite similarly to the Bubblesort Algorithm. It always considers two neighboring pairs of vertices in one layer, calculates the initial number of crossings, and the number of crossings there would be if the vertices were switched. If swapping the two vertices would result in fewer crossings, one applies it; otherwise, nothing changes. This procedure is then applied to all consecutive pairs of vertices in the layer. The greedy switch heuristic is perfect for a post-processing step because, unlike other heuristics, it never wholly changes the order, but only adjusts it

when there is an enhancement. Thus, it can be helpful to use it in combination with other heuristics such as the barycenter heuristic [LMV97].

5.2.4. Position Assignment

The last step is used to assign each vertex an exact position within the graph based on the calculated layer and layer order. Here, one typically has to remove the dummy vertices and use them as bend positions. However, we handle this step differently since we did not use dummy vertices and ANDs have to be drawn differently due to the unique layout of dependencies.

6. Concept

The overall problem we want to solve in this chapter is to be able to generate ANDs with a well-readable output to be used as ex-tasks. Having a well-readable output reduces the number of manual changes one needs to apply after generation, since one does not have to restructure the whole AND to use it. We saw in section 5.2 how one can typically achieve such output for standard graphs. However, the ANDs we want to generate have a particular structure that deviates significantly from such standard graphs.

We start by introducing the elements of the ANDs structure and the problems they cause. Then we describe the AND abstraction we use to avoid these problems. This abstraction is then used throughout this chapter to create an AND and apply specific changes to it, so it can then be transformed into the lecture's AND structure.

6.1. Output Activity Network Structure

This section introduces the essential elements of the lecture's AND structure. We also explain the difficulties they cause, which are the main reason we introduce the AND abstraction. In the final step, we return to the lecture's AND structure and transform the AND abstraction into it.

6.1.1. Bends

The most significant difference between ANDs using the lecture structure and standard graphs is the unique dependency placement. When looking at Figure 6.1, one can see that we do not consider a dependency to be a straight line between the source and target work packages. For us, a dependency usually consists of at least two *bends* with an angle of 90 degrees. The problem is that, in contrast to just having a straight line, there are many possibilities to place these bends. For example, when considering the dependency AB, one can see that it has one 90 degree bend close to A and another one, further above, used to reach B. However, we could move both bends further to the left or right, resulting in a different drawing of the same dependency. Therefore, in contrast to a standard graph, we have many possibilities to draw a dependency, which differ by their bend positions.

6.1.2. Out-Going and In-Going Positions

Dependencies within the AND leave the source and enter the target work package on different sides depending on the actual dependency type. For example, in Figure 6.1 the dependency AC leaves and also enters on the right side; thus, it is a FF dependency.

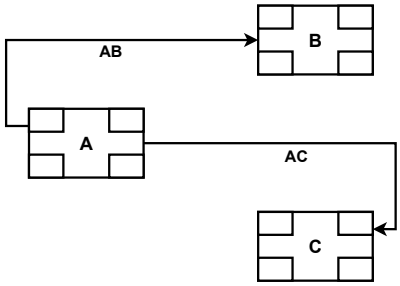


Figure 6.1.: Example snippet of an AND as defined in the lecture

However, the problem is that the actual position where a dependency leaves a work package on its designated side is not defined. Therefore, the dependency AC could just as well go out in the upper right corner of work package A. A human would quickly see that such a placement would be unreasonable, since the dependency target is below A. However, since we want to automate this process, we must define concrete rules for the exit and entry positions.

6.1.3. Conclusion

We see that the drawing step is based on many decisions that have to be made. For each dependency, one has to decide where it leaves and enters a work package and where its bends should be placed. Therefore, we only create the actual drawing structure discussed in this section in the final drawing step, and work with an AND abstraction in the preceding steps. This abstraction is important because it allows certain intermediate steps to be performed, in addition to also simplifying other intermediate steps.

6.2. Abstracted Activity Network Diagram

We introduce the following AND abstraction to make the steps until the actual drawing independent from drawing decisions, while still capturing as much information as possible. In general, we could also directly work with the introduced lecture AND structure, giving the best results. However, always considering the lecture's structure creates many problems and makes certain methods too complex and inefficient. We first introduce the abstracted work packages and then briefly explain the abstraction of dependencies.

6.2.1. Work Package Abstraction

To simplify the concept of work packages, we start by considering the previous AND within a grid (see Figure 6.2). The x-axis describes the layer of a work package, and the y-axis its position within the layer. All work packages that are on the same vertical line gets assigned the same *layer* (e.g., $L_1 = \{A\}$, and $L_2 = \{B, C\}$). Additionally, we introduce two row positions: the absolute row position and the relative position. We use the absolute row position and the layer to define the exact position of an abstracted work

package. This allows us to calculate crossings later, and helps during the transformation to the lecture's AND format in the final step. The relative position is mainly used to calculate the absolute row position, but is also needed for an algorithm used to reduce dependency crossings we later apply.

Relative Position

The *relative position* describes the position of a work package within its layer order, i.e., the number of work packages below itself. For example, in Figure 6.2 work package A has a relative position of zero as it is the only work package in the first layer. In contrast, in the second layer, work package C is below B, meaning B has a relative position of one.

Absolute Row Position

In contrast to the relative position, the *absolute row position* describes the position of each work package in the global context of the AND. Thus, if two work packages have the same absolute row position, they are placed on the same horizontal line within the AND.

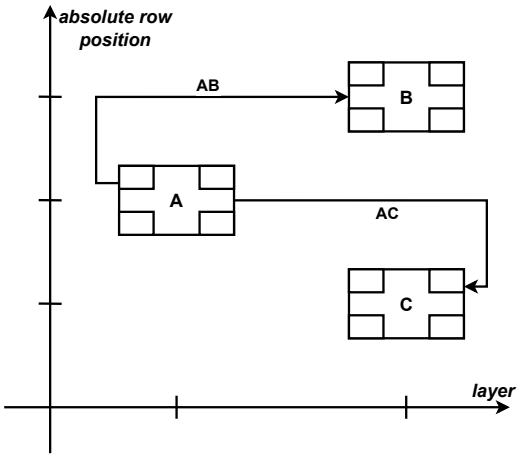


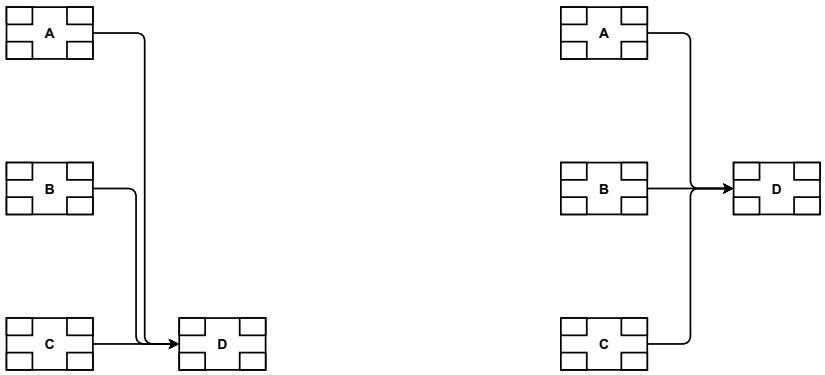
Figure 6.2.: Example snippet of an AND as defined in the lecture placed within a grid

Since we generate ANDs from scratch, we must define a proper way to calculate the absolute row position. In order to define how we calculate the absolute row position, we first introduce the two main concepts the calculation is based on. The main idea is to calculate the absolute row position, to be used during the intermediate steps, and also to help in the drawing step (e.g., we can efficiently transform the absolute row position into the actual work package position).

The first concept can be seen in Figure 6.2 where there is enough free space between two work packages within one layer. The primary usage of this free space is to allow us to apply an *offset* of one between layers with even and odd sizes. For example, in

Figure 6.2 both layers do not have a common absolute row position since the first layer contains an odd, and the second layer an even, number of work packages. This method increases the probability of placing a dependency with just a few bends. This is best shown by the dependency *AC*, where without such an offset, work package A would be at the absolute position of work package B or C. If this is the case, the dependency would need four bends since it first needs to go around work package B or C to reach its entry position.

Secondly, we want to include a *middle axis* in the grid, such that work packages are then evenly placed above and below that axis. Including this middle axis decreases the maximum dependency lengths which positively affects the number of crossings, and thus, the readability of the AND (as shown in Figure 6.3).



(a) Drawing without a Middle Axis (b) Drawing with a Middle Axis

Figure 6.3.: Comparison of drawing with and without a middle axis

The following formula defines the absolute row position of a work package u:

$$absRowPos(u) = (maxLayerSize - size(u.layer)) + 2 \cdot u.relativePosition \tag{6.1}$$

6.2.2. Dependency Abstraction

As stated in section 6.1.2 the exit and entry position of dependencies encode the dependency type in the AND's lecture structure. However, since we consider work packages only as a single point (i.e., layer and absolute row position), we simplify dependencies by only using a label to define their type. By doing so, we reduce the dependency properties to only the source and target work package and the dependency type. In addition, we now only consider a dependency to be a straight line between the source and target work packages to avoid handling the position of the bends during the intermediate steps.

6.2.3. Conclusion

We have discussed how to define an abstract AND that we can then later use to draw the actual AND with respect to the lecture layout. In Figure 6.4 one can see the abstract

representation of an AND having several work packages and dependencies. Unfortunately, due to the abstraction, we lose some information, especially about crossings that can occur within an AND, which we later discuss in detail. Nevertheless, both formats still contain the standard work package information described in section 2.2.4. It is important to keep in mind, that from now on, we use the AND abstraction and only come back to the lecture structure in the final transformation step.

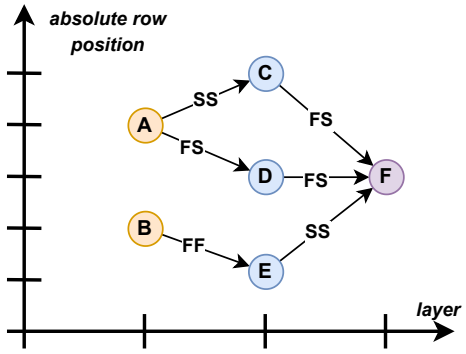


Figure 6.4.: Example AND abstraction

6.3. Steps Toward Generated Activity Network Diagrams

The following generation is designed as an iterative process which can be seen in Figure 6.5. In the first step, we create the initial AND structure. The subsequent crossing improvement step tries to order the randomly generated layers to have as few crossings as possible. However, since we only use the AND abstraction, it is unlikely that all crossings in the lecture's AND structure are removed. Therefore, we use the crossing avoidance step to further reduce crossings induced by the lecture's AND structure. We then calculate the schedule properties before transforming the AND abstraction to the lecture's structure in the final step.

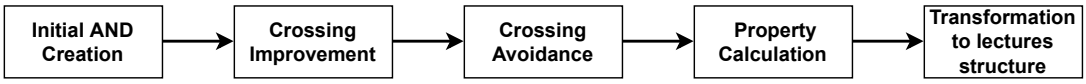


Figure 6.5.: Generation process leading to the output AND

6.4. Generating an Initial Activity Network Diagram

After introducing the AND abstraction we use throughout the following sections we now describe the first step of the generation process, the initial creation of the AND.

6.4.1. Properties of Suitable Activity Network Diagrams

Before generating ex-tasks, one must first define certain rules and restrictions that those ex-tasks should fulfill to be considered suitable. The generation method should therefore be configurable, so that one can later specify the concrete values one want to use for ex-tasks.

- The main goal of ex-tasks we generate is not to find out whether a student can calculate an AND with many work packages without an error, but rather to check the overall understanding of how ANDs work. Therefore, we consider a minimum and a maximum number of work packages to generate.
- It would not make sense to put all work packages in a single layer. If this were the case, we could not consider special dependencies or multiple incoming or out-going dependencies in one work package. Therefore, we also consider a minimum and a maximum number of layers.
- Calculating with large numbers is generally more complex and error-prone. Therefore, we want to be able to restrict the possible duration values, to ensure that the work packages schedule properties not get too extensive.
- As stated in section 2.2.4 ex-tasks should at least contain each dependency type once, so we include a number for each special type representing how often it occurs within the AND.

In the following, we introduce another concept in more detail, which doubles as the first method for crossing avoidance.

Dependency Length

Dependencies spanning many layers tend to lead to many crossings. Therefore, restricting the length of dependencies already reduces the number of crossings created during the generation. For this, we first introduce the two terms *jump value* and *jump distribution* which are crucial concept for our generation approach.

Definition 1. *Let $d = (d_s, d_t)$ be a dependency between the source d_s and target work package d_t, and l_{d_s}, l_{d_t} their layers with $l_{d_s} < l_{d_t}$. Then the **jump value** of d is defined by $jumpValue_d = l_{d_t} - l_{d_s}$*

Definition 2. *Let $dist = (p_1, p_2, ..., p_n)$ be an ordered set with $p_i \in [0, 1]$ and $\sum_{i=1}^{n} p_i = 1$. We call dist the **jump distribution**, and $p_i \in dist$ defines that a jump value i occurs with the probability p_i.*

The jump distribution allows us to define the probability for specific jump values. Therefore, we can also use it to specify approximately how many dependencies we want to have of a given length, thus, limiting the number of long dependencies.

6.4.2. Creation steps

Besides splitting the overall graph generation process into five steps as seen in Figure 6.5, we further divide the initial AND creation step into two sub-steps: the structural creation and the value assignment. The structural creation step generates work packages and dependencies between them. In this step, we do not consider the duration of work packages, and we only assign each dependency the standard FS type. In the value assignment step, we assign each work package a duration and exchange some dependency types to ensure that we have the number of special dependencies as defined. We use this separation because it allows us to extend both steps freely. For example, the duration assignment can be based on the duration of neighbors, meaning that we first need all work packages and dependencies to assign durations. The general idea of the following structural creation is based on the layer-by-layer method described in section 5.1.

In the following, we assume that we have already determined the number of layers and the number of work packages we want to create.

Filling the Layers with Work Packages

The structural creation starts with filling all layers with the minimum number of work packages, called *minimum layer size*. Since we do not want to have empty layers, we must put at least one work package in each layer.

In the next step, we randomly assign the remaining work packages onto the layers. Similarly to above, we use a maximum number of work packages, called *maximum layer size*, to further restrict the layer sizes. Therefore, we have to randomly assign each work package to a layer that is not yet full.

Generating out-going Dependencies

After each work package has been assigned to a layer, we create dependencies between those work packages. We have to ensure that each work package has at least one in-going and out-going dependency. The only exception to this are the start and the end work packages, where we only have to create out-going and in-going dependencies, respectively.

Therefore, we begin with creating FS dependencies between the start work package and all work packages within the first layer. Similarly, we create FS dependencies between all work packages in the last layer and the end work package. We use FS dependencies to ensure that each work package within the first layer has an ES of zero, and the ES of the end work package is the maximum EF of the last layer.

We now consider each work package from the first to the penultimate layer and generate exactly one out-going dependency for each. To do so, we first randomly choose a jump value with respect to the the jump distribution. We achieve this by choosing a uniformly random value $x \in [0, 1]$ and using the jump distribution to get the jump value it corresponds to. For example, in case of the jump distribution $[0.8, 0.15, 0.05]$, we would generate the jump values as follows: $x \in [0.0, 0.8] \rightarrow 1$, $x \in (0.8, 0.95] \rightarrow 2, x \in (0.95, 1.0] \rightarrow 3$.

A problem occurring here is the case where we choose a jump value resulting in a layer outside of the AND's boundaries (i.e., larger than the last layer). To solve this problem,

we map all jump values leading to an invalid layer to the last layer. For example, when we are in the third to last layer, each jump value greater than two would result in a dependency to the last layer.

After generating a jump value, we only need to consider all work packages within the layer corresponding to it, and randomly choose one work package from this layer as the dependency target. We randomly choose a target because this allows us to create any AND with the given restrictions (e.g., dependency length).

Generating in-going Dependencies

After the previous step, every work package has exactly one out-going dependency. However, since we randomly choose one work package from the target layer as a dependency target, there can still be work packages without in-going dependencies.

To handle this, we only have to consider every work package that has an in-degree of zero. We again calculate a jump value as described above and randomly choose a source work package from the layer corresponding to the jump value. Since we create in-going dependencies, we must consider the layer to the left. We again have the problem of choosing an invalid jump value which we handle the same as for the out-going dependencies.

After creating all in-going dependencies, we have a valid AND where each work package has at least one predecessor and at least one successor. Therefore, we can now continue with the second step, where each work package is assigned a duration, and we exchange dependency types.

Assigning Dependency Types and Work Package Information

After creating all work packages and dependencies, we start with assigning each work package a random duration with respect to the defined duration boundaries. However, we have to ensure that the start and end work packages get a duration of zero assigned.

Additionally, at the moment, all dependencies have the standard FS type. As mentioned above, we have defined how often each special type occurs within the AND. Therefore, we must change some dependency types to fulfill this requirement. We can do this by considering all FS dependencies, randomly choosing one dependency, and then changing the type to a special type. We repeat this until all special dependencies are assigned.

6.5. Crossing Improvement

Improving crossings within a graph is a critical step leading to a well-readable output. We already introduced the typical DAG drawing approach in section 5.2. We now further discuss how we handle the crossings improvement in our use case.

All improvement methods are typically based on counting crossings within a graph, applying changes to the graph, and then checking whether the changes improved the number of crossings or not. The problem is that calculating the number of crossings

is usually done only between dependencies with a jump value of one. To get a graph that only has edges with a jump value of one, one usually adds dummy vertices for edges spanning more than one layer, as intermediate points. In the end, the dummy vertices are removed, and the edge is drawn by connecting all consecutive vertices describing the edge. However, since dependencies in our use case only use 90-degree bends and non-standard exit and entry positions at work packages, including dummy vertices can increase the number of bends needed. For example, in Figure 6.6, we see an SF dependency with the dummy vertex X in between. When drawing the dependency with X as an intermediate point, we create eight bends while we only need four bends when drawing the dependency without considering X. Therefore, using these dummy vertices is not reasonable for us. However, not using dummy vertices means that we can have dependencies with a jump value greater than one; thus, we need to introduce a proper method to calculate crossings between dependencies that can also have larger jump values than one.

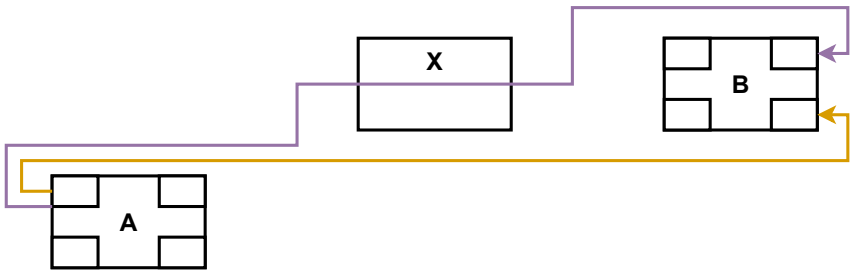


Figure 6.6.: Negative effect of dummy vertices when used in the drawing step

6.5.1. Calculating the Number of Crossings

To calculate the number of crossings, we use an approach described by Anton et al. [Ant+07] and extend it for our use case. The general idea of the approach is that one considers each dependency within the AND as a linear equation, and calculates the crossing points between them.

We divide the calculation of the number of crossings into three different steps. In the first step, we describe how to calculate the crossing point between two arbitrary dependencies. Then we explain how one can check whether the calculated crossing point produces an actual crossing within the AND. Lastly, we introduce a method to calculate the number of crossings of the whole AND using the first two steps.

Calculating the Crossing Point

We start by calculating the crossing point between two dependencies $A = (A_s, A_t)$ and $B = (B_s, B_t)$. To calculate the crossing point between these two dependencies, we first calculate the slope of both linear equations representing them. This can be done as

follows, where r is the absolute row position and l the layer:

$$m_Z = \frac{(r_{Z_t} - r_{Z_s})}{(l_{Z_t} - l_{Z_s})} \text{ for } Z \in \{A, B\} \tag{6.2}$$

When considering the example in Figure 6.7 this would result in $m_A = \frac{(1-3)}{(2-1)} = -2$ and $m_B = 1$. We can use the calculated slope to define the two linear equations $y_Z = m_Z \cdot x + b$. However, we still need to calculate the value of b as follows:

$$b_Z = r_{Z_s} - (l_{Z_s} * m_Z) \text{ for } Z \in \{A, B\} \tag{6.3}$$

In Figure 6.7 this would result in $b_A = 3 - (1 \cdot -2) = 5$ and $b_B = 0$. We now need to calculate the crossing point between the two linear equations by setting both equations equal and solving for x:

$$m_A \cdot x + b_A = m_B \cdot x + b_B \Rightarrow x = \frac{(b_B - b_A)}{(m_A - m_B)} \tag{6.4}$$

In our example this would lead to the crossing layer $x = \frac{(0-5)}{-2-1} = \frac{5}{3}$. We can now use either of the two linear equations to calculate the absolute row position of the crossing: $y = -2 \cdot \frac{5}{3} + 5 = \frac{5}{3}$. As a result we now have the crossing point $(\frac{5}{3}, \frac{5}{3})$ between the dependencies A and B.

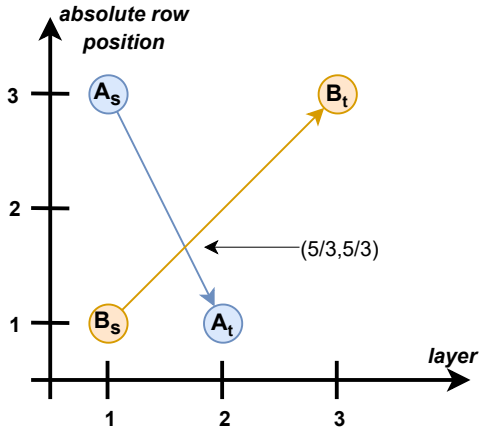


Figure 6.7.: Example for the crossing point calculation between two dependencies

Checking whether a Crossing Point produces a Crossing

After introducing how to calculate the crossing point between two dependencies, we must check whether it produces an actual crossing within the AND.

First, in the following l_{min}, l_{max} refers to the minimum and maximum layer, and r_{min}, r_{max} refers to the minimum and maximum absolute row position. We first check

whether the crossing point is within the ANDs boundaries by checking whether the crossing layer is between l_{min} and l_{max}, and the crossing row is between r_{min} and r_{max}. If the crossing point is not within the AND, we do not have to apply further checks. Otherwise, it could still not be an actual crossing. This can happen if the crossing point does not occur between the actual dependencies but somewhere further to the left or right but still in the AND's boundaries. To include these cases, one has to check the following four conditions [Ant+07]:

$$r_{A_s} > r_{B_s} + m_B \cdot (l_{A_s} - l_{B_s}) \tag{6.5}$$
$$r_{A_t} < r_{B_s} + m_B \cdot (l_{A_t} - l_{B_s}) \tag{6.6}$$
$$r_{B_s} < r_{A_s} + m_A \cdot (l_{B_s} - l_{A_s}) \tag{6.7}$$
$$r_{B_t} > r_{A_s} + m_A \cdot (l_{B_t} - l_{A_s}) \tag{6.8}$$

We know that the crossing point produces a crossing within the AND if these four conditions are true. In total, we have now calculate the crossing point for two dependencies and checked whether it actually produces a crossing within the AND.

Calculating the Number of Crossings

At this point, we can check whether there is a crossing between two dependencies. However, it is not feasible to apply this method to all possible dependency combinations to obtain the number of crossings. This is due to the fact that we have many combinations, even for a small number of dependencies; thus, the computational effort when considering all of them would be too great in practice. Fortunately, we can significantly reduce the number of combinations by only considering dependencies where a crossing could occur, making the method applicable.

The main idea of the following approach is that we first abstract from concrete dependencies to layer pairs (a, b) describing all dependencies between layer a and b. For example, the pair $(1, 3)$ would describe all dependencies starting in layer 1 and ending in layer 3. We then want to build pairs of the form $((a, b), (c, d))$ only when there could be a crossing between dependencies from layer a to b and dependencies from layer c to d. We then solely use these pairs and their corresponding dependencies to calculate the AND's crossings.

We define:

$$lp = \{(l_1, l_2) \mid startLayer \leq l_1 < l_2 \leq endLayer,\ |l_1 - l_2| \leq maxJumpValue\} \tag{6.9}$$

Intuitively, the lp set is the set of all pairs of layers where dependencies can occur between the left and the right layer. This is why, we limit the distance between the left and right layers to the maximum jump value because we know that no dependencies exist between layers with a greater distance. For example, when considering an AND with only three layers and a maximum jump value of one, we would get the following set: $lp = \{(1, 2), (2, 3)\}$.

Additionally, we define:

$$dp = \{(d_1, d_2) \mid d_1, d_2 \in lp\} \tag{6.10}$$

We use the lp set here to calculate all possible layer combinations between which there could be a crossing. Therefore, we start by considering all possible combinations of two pairs from the lp set (i.e., the Cartesian product of lp with itself) in the dp set (Equation 6.10). As the tuple $((a, b), (c, d))$ is equivalent to the tuple $((c, d), (a, b))$ in terms of crossings, we initially remove all such duplicates from the dp set. If we were to keep these duplicate combinations, we would count crossings twice. For example, the lp set from above would result in the set $dp = \{((1, 2),\ (1, 2)),\ ((1, 2), (2, 3)),\ ((2, 3), (2, 3))\}$.

Before we use this set for the crossing calculation, we want to filter it further to achieve the goal of greatly reducing the number of calculations needed. The only case left that still has to be handled is where the target layer of the left pair is less or equal to the source layer of the right pair (see Figure 6.8). In this case, we are not able to detect a crossing within the AND abstraction, so we exclude it with the following constraint:

$$((a, b), (c, d)) \notin dp \text{ if } b \leq c \tag{6.11}$$

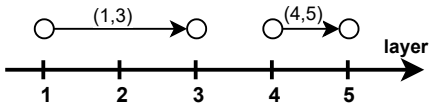


Figure 6.8.: Layer pair we do not consider since it is impossible to create a crossing in the AND abstraction

The primary purpose of the concepts introduced throughout this section is to calculate the number of crossings within our AND abstraction, which we define as follows:

Definition 3. *Let $dep(x, y)$ be the dependencies starting in layer x and ending in layer y and $checkCrossing(d_1, d_2) \rightarrow \{0, 1\}$ the function resulting in 1 if there is a crossing between d_1 and d_2, otherwise 0. Then the* ***number of crossings*** *of an AND is defined by*

$$numOfCros = \sum_{\substack{((a,b),(c,d)) \in dp \\ d_l \in dep(a,b) \\ d_r \in dep(c,d)}} checkCrossing(d_l, d_r)$$

Sliding Window to Decrease Computations

The primary goal within this section is to apply crossing improvement methods that are based on the calculation of the number of crossings and the application of heuristics. In section 5.2.3 we introduced two heuristics, the barycenter and the greedy switch heuristic. The greedy switch heuristic heavily depends on a method to efficiently calculate crossings, or more precisely, to check whether the number of crossings is reduced by

swapping two neighboring work packages. Therefore, always calculating the number of crossings of the whole AND is too complex, especially for larger ANDs. To still apply the heuristic, we introduce a method to efficiently check whether swapping two work packages improves the number of crossings.

Definition 4. *Let wp_1 and wp_2 be two work packages with $l_{wp_1} = l_{wp_2}$. Let a_x be the absolute row and r_x the relative position of $x \in \{wp_1, wp_2\}$. Then* ***swapping two work packages*** *is defined by swapping r_{wp_1} with r_{wp_2} and a_{wp_1} with a_{wp_2}*

Definition 5. *Let $L = (l_1, l_2, ..., l_n)$ be the ordered set of all layers within an AND. We define the* ***free layer*** *as the layer $l \in L$ such that we only change the ordering of work packages within this free layer, while all other layers $l' \in L \setminus \{l\}$ are considered immutable.*

We handle this problem by defining a window around the free layer and only calculating the crossing number within this window. Depending on the window size, this reduces the possible combinations where crossing can occur greatly.

Definition 6. *We define the* ***left window*** *(leftWin) as the largest jump value we consider to the left of the free layer and the* ***right window*** *(rightWin) as the largest jump value to the right of the free layer. We then define the* ***window*** *as $win = leftWin + rightWin + 1$.*

We divide the window into two parts, the left and right window, to be able to use different boundaries on each side, which can further decrease the complexity for larger ANDs. We now use the left and right window to adjust the lp set.

$$lp_{freeLayer} = \{(l_1, l_2) \mid (freeLayer - leftWin) \leq l_1 < l_2 \leq (freeLayer + rightWin)\} \quad (6.12)$$

Important to note is that the left bound is not smaller than the start layer, and the right bound is not larger than the end layer.

We use the maximum jump value for both left and right window, ensuring that we include all dependencies starting and ending in the free layer. Unfortunately, this does not include all dependencies that can produce crossings when swapping two work packages in the free layer. This can be seen in Figure 6.9 where the work packages on the right side are outside of the right window, but the dependency A that we currently do not consider produces a crossing with the dependency B.

We handle this problem by extending our lp set with only the necessary layer pairs. First, we do not have to consider dependencies that end in the left window layer or start in the right window layer since they can not produce crossings with the free layer dependencies. Additionally, we have already included all possible dependencies from the free layer. Therefore, we now only consider dependencies coming from outside the left window to the inside (Equation 6.13) or going from inside the right window to the outside (Equation 6.14). For example, in the case of Figure 6.9 we would include the pair $(4, 6)$ into the lp based on Equation 6.14.

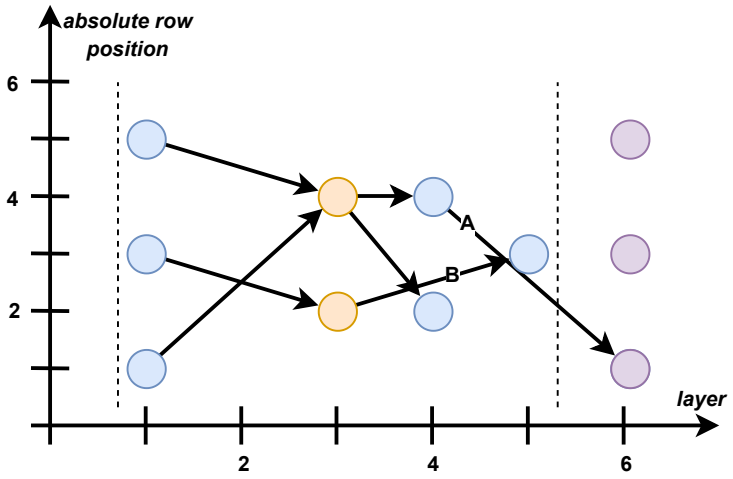


Figure 6.9.: Sliding window visualization. The dotted lines indicate the current window, orange dots the work packages in the free layer, and the violet dots the work packages outside of the window

$$(l_1, l_2) \in lp \text{ if } |dep(l_1, l_2)| \geq 1,\ l_1 < (freeLayer - leftWin), \\ \text{and } (freeLayer - leftWin) < l_2 < freeLayer \tag{6.13}$$

$$(l_1, l_2) \in lp \text{ if } |dep(l_1, l_2)| \geq 1,\ freeLayer < l_1 < (freeLayer + rightWin), \\ \text{and } (freeLayer + rightWin) < l_2 \tag{6.14}$$

We have now included all layer pairs into the $lp_{freeLayer}$ set that have dependencies that can produce crossing with the free layer. We now only have to calculate the dp set as defined in Equation 6.10 using the $lp_{freeLayer}$ set, and then calculate the number of crossings for the free layer using Definition 3.

6.5.2. Apply Improvement with Layer-by-Layer Sweep

We can now calculate the crossing number of the whole AND and only for a window around a free layer. Therefore, the only thing left to do is to apply the layer-by-layer sweep, introduced in section 5.2.3, using these two calculation methods. The main goal of the layer-by-layer sweep is to find an ordering for each layer that produces the fewest crossings within the AND. In the following, we go through the different steps of the layer-by-layer sweep, which can also be seen in the activity diagram in Figure 6.10.

Layer-by-Layer Sweep

The first two steps of the layer-by-layer sweep are the forward and the backward sweep as described in section 5.2.3. In both phases, we iterate over the AND's layers, once from

the start to the end layer and once from the end to the start layer. We then apply a heuristic for the layers in between by always considering one layer as the free layer. After both sweeps, we need to apply certain checks to verify whether we improved the number of crossings. We start by introducing how we applied the two heuristics introduced in section 5.2.3 and section 5.2.3, and then explain the steps that must be taken after the application of the two sweeps.

Barycenter Heuristic

In the case of the barycenter heuristic, we use the relative positions of each work package. So, we start by calculating the barycenter value for each work package within the free layer as described in section 5.2.3. We sort the barycenter values and assign each work package their new relative position within the free layer depending on the sorting. Thereby, the work package with the lowest value gets the relative position zero assigned, and the work package with the largest value the largest relative position. It is essential to mention that we also have to recompute the absolute row positions since we changed the ordering of a layer.

Greedy Switch Heuristic

When using the greedy switch heuristic, introduced in section 5.2.3, we use the window method to check whether we improve the crossing number by swapping two neighboring work packages. We keep the new ordering in case of an improvement; otherwise, we swap back the two work packages and continue with the next. The advantage of this approach is that it only changes the ordering if it actually detects an improvement within the AND. However, in contrast to the barycenter heuristic, we need to calculate if we achieve an improvement within the actual heuristic. So even though we defined the sliding window method to reduce the calculations, it is still inefficient for large ANDs.

Saving the Best Ordering

Since some heuristics occasionally produce a worse result after one step but then compensate for this again in a subsequent step [Pat04], we need to keep track of the best number of crossings, and the respective layer ordering. By doing so, we can reapply this ordering in case the subsequent steps can not achieve a better result. To ensure that we stop at some point, we need to define a maximum number of consecutive rounds that do not produce an improvement. A *round* refers to one complete execution of the layer-by-layer sweep.

Improvement checks

After applying one of the two heuristics in the forward and backward sweep, we first recalculate the number of crossings of the whole AND. If we have zero crossings, we have the best ordering possible and can stop. Otherwise, we must check whether the

last forward and backward sweep improved the number of crossings, in which case we act as follows:

- If we improved the crossings, we save this ordering and the number of crossings and reset the counter for rounds without an improvement. Then we continue with the next layer-by-layer sweep round.
- If we worsen the crossings, we increase the counter for rounds without improvements and check whether we reached its maximum. If this is the case, we reapply the best ordering and stop with the layer-by-layer sweep.

The resulting AND should now have an ordering with as few crossings as we can detect and reduce. However, we currently only consider the ANDs abstraction, which leads to specific problems in this step considering the lecture's AND structure, which we explain in the following.

6.5.3. Limitation of the Crossing Calculation

Unfortunately, the calculation of the number of crossings does not always calculate a correct result when considering the lecture's AND structure. So, as described in section 6.5.1 we use two linear equations and their crossing point to calculate if there is a crossing between two dependencies. This approach works well for graphs as shown in Figure 6.7 where vertices are dots, and edges are straight lines between them. This is however, not the case for the lecture's AND structure. As a result, there are cases where the abstraction has zero crossings, but we still have crossings within the lecture's AND structure. For example, in Figure 6.11 we see the AND after the final layout step and the equivalent AND abstraction. In this case, the number of crossings of the AND abstraction is zero, although we can see that there is still one crossing within the actual AND.

The problem is that it is practically impossible to calculate the correct number of crossings considering the lecture's AND structure. The reason for this is that the calculation depends on the layout decisions, and therefore one would have to redraw the complete AND every time one applies a change to it. In addition, instead of one linear equation, most dependencies would then consist of several linear equations since they are no longer a straight line. Furthermore, we would also not be able to reduce the combinations as much as we are now, because depending on the drawing, we can have a crossing between a dependency ending in a layer and another dependency starting in the same layer. We address some of these issues in the final transformation step, where we further try to reduce crossings by applying specific layout rules. In addition, we also perform a crossing avoidance step in the following, which is used to reduce the crossings further.

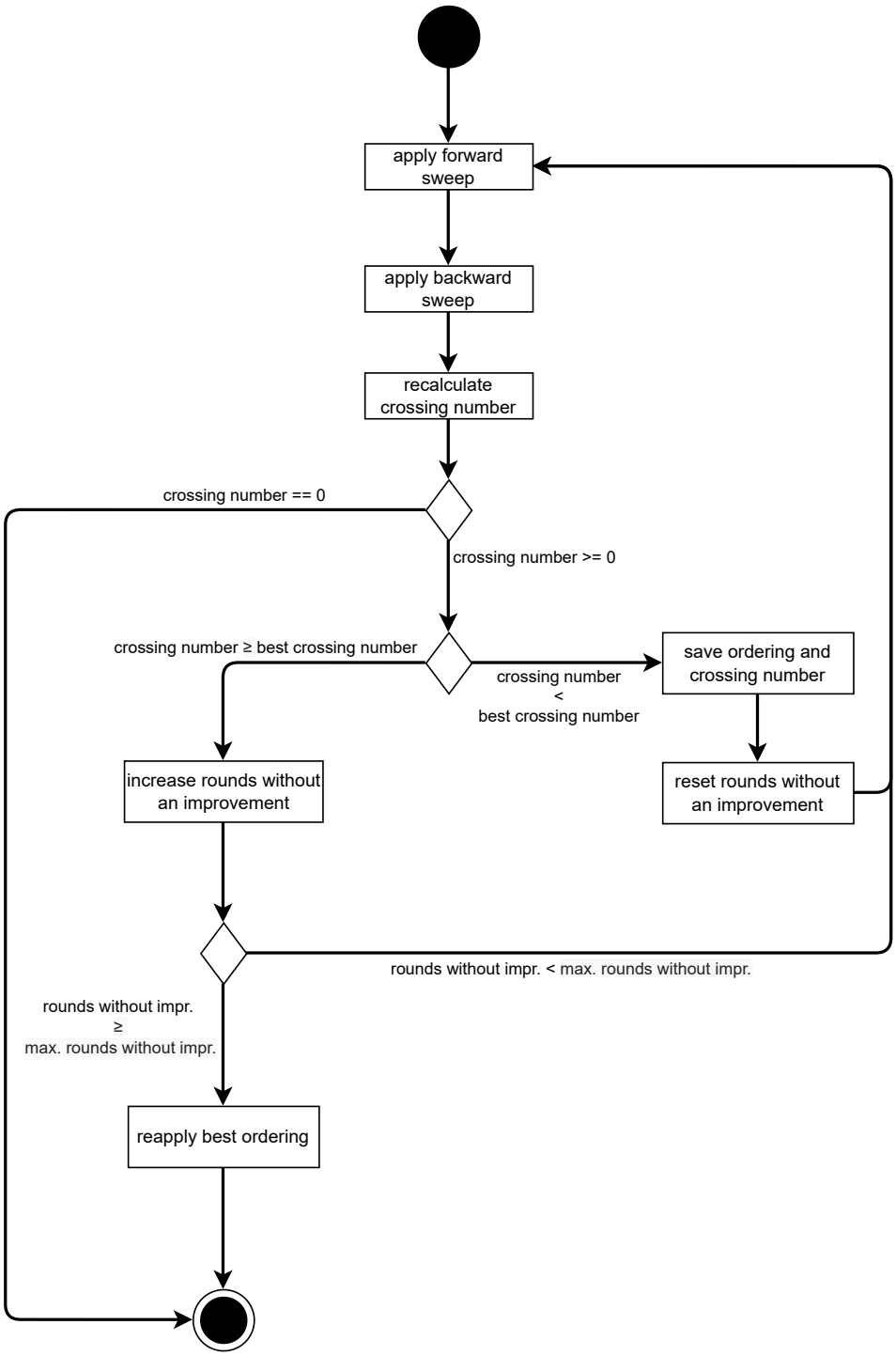
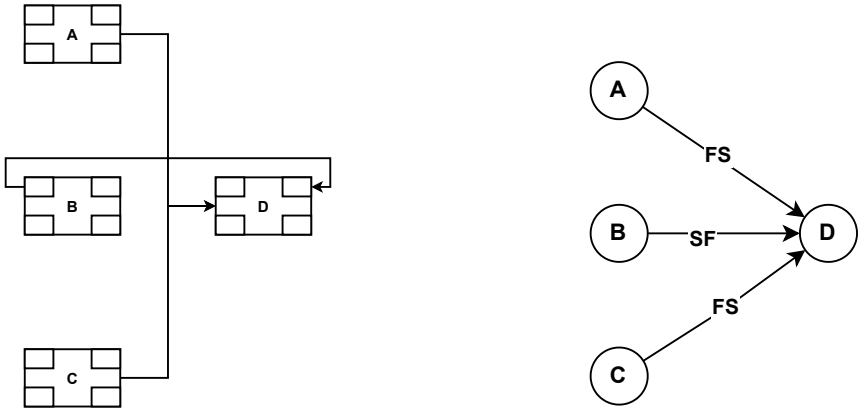


Figure 6.10.: Activity diagram of layer-by-layer sweep



(a) Drawing of the actual AND part (b) Calculation method view on the AND

Figure 6.11.: Crossing calculation limitations of the AND abstraction

6.6. Crossing Avoidance

In addition to creating ANDs, one of our main objectives is achieving a well-readable output. In the previous section we applied methods to find an ordering that reduces the crossings. However, due to the layout of ANDs in the lecture, and since we use an AND abstraction, the previous crossing improvement most likely not removes all crossings within the lecture's AND structure. Therefore, we apply a crossing avoidance step to reduce crossings further.

Definition 7. *Let $d = (d_s, d_t)$ be a dependency with d_s as source and d_t as target work package. We define the* ***absolute row difference*** *as $|absRowPos_{d_s} - absRowPos_{d_t}|$.*

Definition 8. *Let $A = (A_s, A_t)$ and $B = (B_s, B_t)$ be two dependencies. Then* ***swapping dependencies A and B*** *is defined by swapping their target work packages; thus, $A = (A_s, B_t)$ and $B = (B_s, A_t)$.*

After applying the crossing improvement, the two main reasons for the remaining crossings are dependencies with a sizeable absolute row difference and dependencies with a large jump value. This is because the longer a dependency is, the more crossings it can cause with other dependencies. Therefore, our primary goal in the crossing avoidance step is to reduce the length of dependencies.

However, as described in section 6.4, our goal is also to generate some dependencies that have a particular jump value. Hence, reducing the jump value of dependencies is not reasonable, which would ignore this constraint. Therefore, we will focus on reducing the absolute row difference of dependencies. For the sake of readability, we refer to the absolute row difference only as *row difference* and the absolute row position only as *row position*.

The general idea of the crossing avoidance is that we want to reduce the total row difference within the AND by reducing the row difference for dependencies that have

a significant row difference. We achieve this by swapping two dependencies when it reduces the total row difference in the AND. It is essential to mention that this can only be done because it is not needed to generate any possible AND. This is because, in contrast to the crossing improvement, which only changes work package positions, the crossing avoidance changes the meaning of the AND. We return to this problem in more detail at the end of this section.

6.6.1. Identifying Problematic Dependencies

Definition 9. *Let $D = \{d^1, d^2, ..., d^n\}$ be the set of all dependencies and th a predefined threshold. We then define the* ***problematic dependencies*** *as $probDep_{th} = \{d \mid d \in D,\ absRowDiff(d) \geq th\}$. We also refer to this set as* ***threshold dependencies****.*

We include a threshold when identifying problematic dependencies because it allows to decrease the complexity when considering larger ANDs, since by increasing the threshold we reduce the number of threshold dependencies. For the sake of understandability, we consider a threshold of two in the following. However, the presented concepts also hold for greater threshold values.

6.6.2. Reducing the Absolute Row Difference

After getting all dependencies having a row difference greater than or equal to the given threshold, we can now reduce their total row difference. For this, we consider each dependency independently and refer to a single dependency $d \in probDep_{th}$ simply as *threshold dependency*. We try to reduce the row difference for each threshold dependency by swapping them with any other dependency. By doing so, we have to make sure that the improvement for the threshold dependency is more significant than the worsening of the dependency we swap with. Since it might not be possible to improve all threshold dependencies, we only apply the following steps until we can not apply further changes to any threshold dependency.

Getting Possible Improvement Dependencies

Definition 10. *Let $D_k = \{d^1, d^2, ..., d^n\}$ be the set of dependencies ending in layer k. Let rd_d be the row difference, d_s be source and d_t the target of a dependency $d \in D_k$. We define the* ***possible improvement dependencies*** *for a given threshold dependency td ending in layer k as the set $posImpDep_{td} = \{d \mid d \in D_k,\ d \neq td,\ absRow(d_t) \in [absRow(td_s) - rd_{td} + 1,\ absRow(td_s) + rd_{td} - 1]\}$*

To ensure that we can not create dependencies that have an invalid jump value (i.e., greater than the maximum jump value allowed), we start by only considering dependencies ending in the same layer as the threshold dependency. Therefore, we get all dependencies that end in the target layer and at least reduce the row difference for the threshold dependency when swapping. For example, in Figure 6.12, the dashed area marks all dependency targets that would reduce the row difference for the threshold dependency AB.

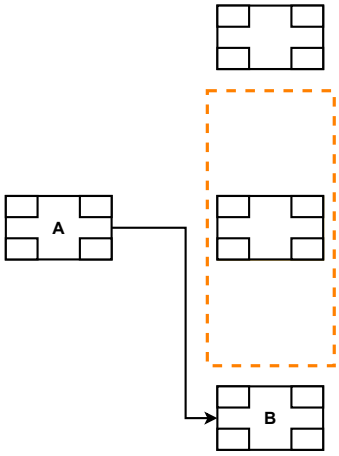


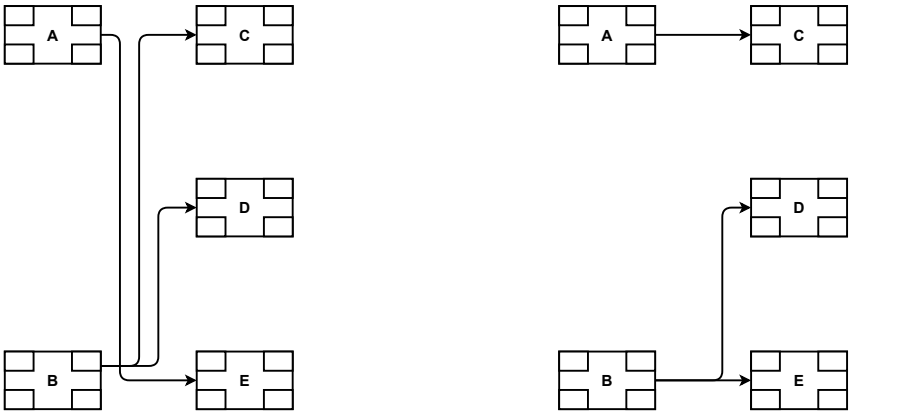
Figure 6.12.: Dependencies ending in the orange area would decrease the row difference for dependency AB

Finding the Dependency with the most significant Improvement

We now use the possible improvement dependencies to find the dependency resulting in the most significant improvement. We illustrate each of the following steps with the example in Figure 6.13.

1. We start by calculating the initial row difference of the threshold dependency. In our example, we try to reduce the row difference of the dependency AE; thus, the current row difference is two.
2. We then look at all possible improvement dependencies and calculate their current row difference. Since work packages C and D would decrease the row difference when used as targets, we now consider all dependencies having one of them as the target. This leads to the dependencies BC with a row difference of two and BD with a row difference of one.
3. Next, we calculate the row difference we get when swapping the threshold dependency with the possible improvement dependencies. Let us first consider swapping with the dependency BD. This results in the dependency AD with a row difference of one and BE with a row difference of zero; thus, a total row difference of one. When using dependency BC, we get the dependency AC with a row difference of zero, and BE also having a row difference of zero; thus, a total row difference of zero.
4. Lastly, we calculate the row difference improvement we achieve for each case, and use the dependency with the most significant improvement. Therefore, we again consider all possible improvement dependencies and calculate the row difference improvement when swapping. In both cases, we have an improvement of two for the possible swap dependency since both start at the same work package. However, the

improvement for the threshold dependency AE is one when using the dependency BD and two for the dependency BC; thus, swapping with the dependency BC gives us the most significant improvement.



(a) Dependencies which could result in an improvement

(b) Swapped target with the dependency with the largest improvement

Figure 6.13.: Reducing step visualization

It is crucial to mention that there is not necessarily a dependency that gives such an improvement. This is the case if all possible dependencies one could swap with would create a negative effect more significant than the positive effect the threshold dependency can achieve.

Post Checks after Swapping Two Dependencies

After swapping two dependencies, we first have to check whether the row difference of our threshold dependency is now less than the given threshold. If this is the case, we no longer consider it as a threshold dependency in subsequent steps; otherwise, we keep it because we might improve it further after other changes have been applied.

In addition, we also need to check whether the dependency we swapped with already had a row difference greater than the threshold. If this is the case, we apply the same checks as for the threshold dependency. Otherwise, we need to check whether its new row difference is greater or equal to the threshold and, if so, include it into the threshold dependencies.

6.6.3. Special Cases for the Crossing Avoidance

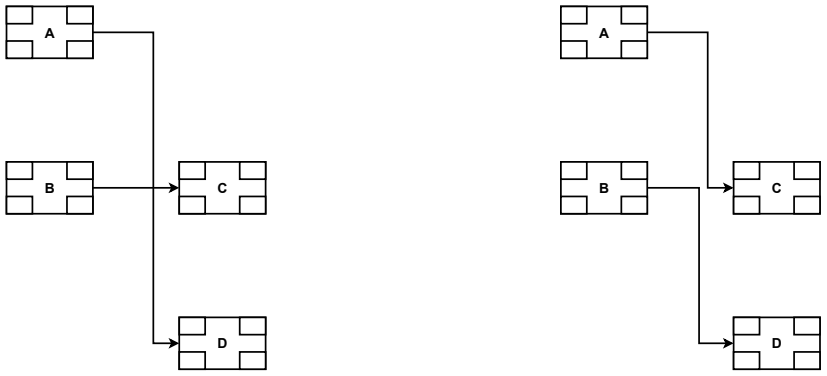
At the moment, we only consider possible swap dependencies that end in the same target layer, which ensures that we do not create invalid jump values and that the crossing avoidance does not change the number of dependencies with particular jump values (i.e., the jump value distribution). Additionally, we only swap two dependencies when the

total row difference is less than before, which is necessary not to swap two dependencies infinitely often. However, even though this method already yields good results, some cases are not covered, the handling of which would further improve the AND. Therefore, we now introduce the improvements we made to the method.

Reducing Maximum Absolute Row Difference

We currently do not consider the case where the total row difference remains the same, but we still remove a crossing when we swap two dependencies. One can see this in Figure 6.14a, where swapping both targets results in the same total row difference as before (i.e., $2 + 0$ vs. $1 + 1$), but still results in fewer crossings (Figure 6.14b).

To include this, we need to adjust the method to find the dependency that gives us the most significant improvement. We still consider the total row difference as the most decisive criteria. However, we use the maximum row difference as an improvement factor if we either achieve the same total row difference as before or have at least two dependencies yielding the same improvement. In this case, we compare the maximum row difference for all cases and then choose the one with the lowest maximum row difference. For example, the initial case (Figure 6.14a) has a maximum row difference of two for the dependency *AD*. When swapping both dependencies, we reduce it to a maximum row difference of one (Figure 6.14b); thus, we apply the depicted swap in this situation.



(a) Swapping both dependencies would result in the same total row difference

(b) Swapping both dependencies still reduces the number of crossings

Figure 6.14.: Reducing step also considering the maximum row difference

Dependencies with larger jump value

Furthermore, the initial approach only considers dependencies that end in the same layer as the threshold dependency. This is because our primary goal is to reduce the total row difference while maintaining the jump value distribution. Due to this constraint, however, dependencies do not consider all of their possible improvement dependencies.

One can see this in Figure 6.15 where we would not consider the dependency BC as possible swap dependency despite the fact that using it results in a smaller row difference while still maintaining the jump value distribution.

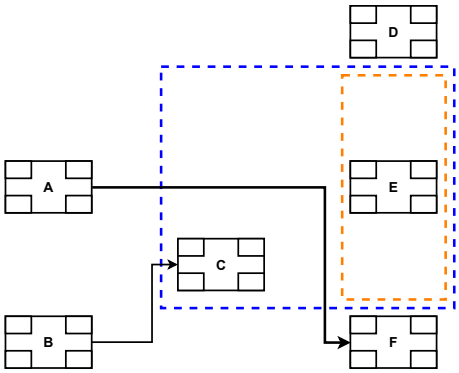


Figure 6.15.: The orange area marks the area where possible improvement dependencies are allowed to end as defined in Definition 10 and visualized in Figure 6.12. The blue area marks the extended area when considering the dependency AF and a maximum jump value of two

To handle this, we extend the possible swap dependencies to look at all work packages that are reachable with the maximal jump value from the source layer, since setting one of these work packages as the new target results in a valid jump value.

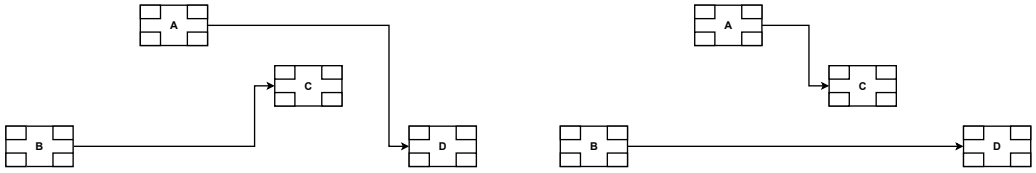
Definition 11. *Let td be a possible threshold dependency. Let $D_{td} = \{d \mid l_{d_t} \in [l_{td_s} + 1, l_{td_s} + maxJumpValue]\}$. We then define the* ***extended possible improvement dependencies*** *for td as the set $extPosImpDep_{td} = \{d \mid d \in D_{td},\ d \neq td,\ absRow(d_t) \in [absRow(td_s) - rd_{td} + 1,\ absRow(td_s) + rd_{td} - 1]\}$*

However, we have to adjust the procedure that gives us the swap dependency with the most significant improvement to avoid getting invalid jump values. One case where this can happen is shown in Figure 6.16a where swapping both dependencies leads to a row difference improvement but creates an invalid jump value of three (Figure 6.16b).

To include this, we only have to check if the jump values stay the same or have simply been swapped. By doing so, we ensure that we do not produce invalid dependencies and maintain the jump value distribution.

6.6.4. Number of Producible Activity Network Diagrams

As mentioned at the start of this section, one problem of this approach is the number of producible ANDs. In contrast to the crossing improvement, we change the targets of dependencies in the crossing avoidance, which changes the meaning of the AND. This can be seen when looking back at Figure 6.13b. If we only see this AND we are not able to decide whether it was produced by Figure 6.13a with crossing avoidance, or if it got created like this. This means that if we want to produce any possible AND, it is not



(a) Before swapping (b) After swapping

Figure 6.16.: Problem when considering all dependencies going into work packages between the source and target layer as potential swap dependencies

reasonable to use this crossing avoidance step. However, we apply this crossing avoidance step because it is better, in our use case, to generate ANDs with fewer crossings, which are more readable.

6.7. Work Package Property Calculation

At this point, we have created the AND and applied various changes to it. Since the crossing avoidance step applies changes to dependencies, we can only calculate the schedule properties after the crossing avoidance step. In the following, we introduce the calculation's prerequisites and then explain the calculation itself.

6.7.1. Prerequisites

The following calculation method is based on the forward and backward pass, introduced in section 2.2.3. Additionally, it depends on a well-layered AND, which means that each dependency goes from a left to a right layer. The generation method we introduced in section 6.4 is based on the layer-by-layer method; thus, we only create well-layered ANDs. However, we also want to be able to input modified ANDs to calculate their properties. Therefore, before applying the calculation, we need to check whether the AND is correctly layered and, if not, apply a correct layering with one of the methods described in section 5.2.2.

6.7.2. Calculation

The forward and backward passes now make use of the layered AND. In general, we can only calculate the values of a work package in the forward pass when we have already considered all of its predecessors. Since we have a layered AND where each work package only has predecessors in previous layers, this statement is equivalent to that we already calculated the ES and EF in all previous layers. Therefore, in the forward pass, we run from the start to the end layer and always calculate the ES and EF for each work package within a layer before continuing with the next layer. This means that when we reach the *kth* layer, we know that all work packages up to this layer have their correct ES and EF assigned. We calculate the ES and EF using the calculation rules defined in section 2.2.3.

In the backward pass, we again uses the fact that the AND is correctly layered, but this time we go from the end to the start layer and calculate the LS and LF.

After applying both phases, we have an AND with correctly calculated properties, and the only thing left to do now is to transform the AND abstraction into the lecture's AND structure.

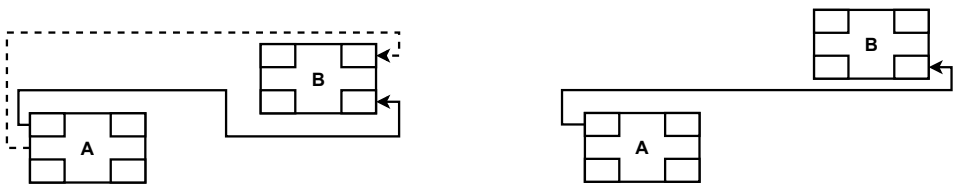
6.8. Drawing the Activity Network Diagram

Until now, we have only considered our AND abstraction. In this final transformation step, we now transform this abstraction into the lecturer's AND structure. Here we focus on the essential concepts that the transformation is based on.

6.8.1. Drawing the Work Packages

The position assignment of work packages is based on two concepts. We first have the *x-spacing* we include between two neighboring layers. This x-spacing is later used to arrange dependencies between two layers.

Furthermore, we also include the *y-spacing* between two absolute row positions, allowing us to create dependencies with more direct paths. For example, in Figure 6.17 we consider the SF dependency AB with an absolute row difference of one. Without additional y-spacing, we would need to create more bends than needed or a long dependency (see Figure 6.17a); thus, the y-spacing allows us to more directly arrange dependencies to the target work package (Figure 6.17b).



(a) Activity network diagram without y-spacing

(b) Activity network diagram with appropriate y-spacing included

Figure 6.17.: Visualization of the y-spacing concept

6.8.2. Drawing the Dependencies

As already discussed at the beginning of this chapter, a dependency within the lecture's AND structure is not considered to be a straight line between two work packages, but rather contains some 90-degree bends. Therefore, there are many possibilities for how we can arrange those dependencies. As mentioned in section 5.2.1 having few bends is one criterion for a well-readable graph drawing; thus, we decided to create dependencies with the fewest bends needed (at most four bends).

Since the exit and entry sides at a work package are well defined by the dependency type, as stated in section 2.2.4, we omit the sides here and only focus on where, and in which direction (i.e., upwards or downwards), a dependency leaves and enters a work package. We introduce several special rules to decide whether a dependency leaves a work package towards the top, as seen in Figure 6.17b, or towards the bottom. If none of the rules we introduce are applicable, we place a dependency towards the top when the target is above the source and towards the bottom when the target is below the source. In addition, we always draw a dependency directly to its target layer through the nearest y-spacing, so we only go upwards or downwards in the actual target layer. This reduces the case complexity, and the following rules are based on this assumption.

Dependencies to the same Absolute Row Position

We consider the case of a dependency going into a work package with the same absolute row position. A problem of this case can be seen in Figure 6.18, where placing the dependency towards the top or bottom of the source work package influences the total number of crossings. Therefore, we want to specify a rule that chooses the best of the two variants.

The main idea of this rule is that we determine how many crossings are exclusively produced when placing the dependency at the top and at the bottom, and then use the placement yielding fewer crossings.

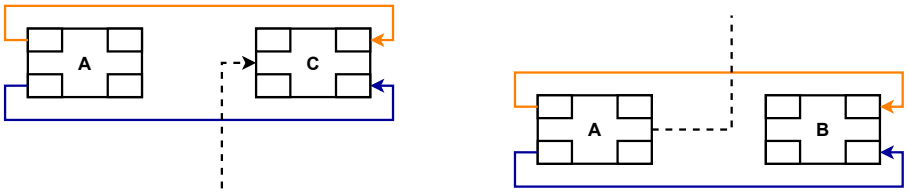


Figure 6.18.: Depending on the situation it might be better to place the start-to-x dependency above or below

Definition 12. *Let wp be a work package. Let $predWP(wp)$ be all predecessor work packages and $succWP(wp)$ all successor work packages from wp. We then define $xAbv(wp) = \sum_{a \in xWP(wp)}(absRow(a) > absRow(wp))$ for $x \in \{pred, succ\}$ to be the number of predecessor/successor dependencies going to a work package above wp. Similarly $xBelow(wp) = \sum_{a \in xWP(wp)}(absRow(a) < absRow(wp))$ is the number of predecessor/successor dependencies going to a work package below wp.*

Definition 13. *Let $d = (d_s, d_t)$ be a dependency, and $absRow(d_s) = absRow(d_t)$. Let $bt(d) = \{wp \mid absRow(wp) = absRow(d_s),\ l_{d_s} < l_{wp_i} < l_{d_t}\}$. We then define $crosTop(d) = succAbv(d_s) + \sum_{wp \in bt(d)}(succAbv(wp) + predAbv(wp)) + predAbv(d_t)$ and $crosBot(d) = succBelow(d_s) + \sum_{wp \in bt(d)}(succBelow(wp) + predBelow(wp)) + predBelow(d_t)$*

When having a dependency where the source and target work packages are on the same absolute row position, we calculate *crosTop* and *crosBot*. Then, we can handle

this case by choosing the arrangement producing the fewest crossings depending on these two calculated values.

Finish-to-Start Dependency Offset Assignment

Since most of our dependencies are FS dependencies, we must take special care to handle cases that can occur for them. The most significant problem for them is placing all dependencies on the same line within the x-spacing. When doing so, it can be impossible to see where a dependency ends. For example, in Figure 6.19 we can arrange all dependencies on the same line, causing no problem. However, when considering a dependency starting above A and ending below B that uses the same line, we are no longer able to identify which dependencies end in B or further down. Therefore, our goal is to arrange FS dependencies in the x-spacing in such a way that they do not use the same line and do not create avoidable crossings.

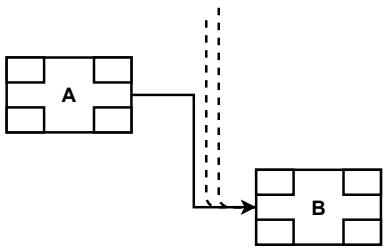


Figure 6.19.: Shifting dependencies depending on the number of other dependencies passing by

The general idea is that we want to assign each FS dependency an offset describing how far we shift the dependency to the left in the target layer. For example, in Figure 6.19 dependency AB has an offset of two. As mentioned above, we assume that each dependency first directly goes into its target layer before going upwards or downwards to the target work package.

Definition 14. *Let $d = (A_s, A_t)$ be a FS dependency with $absRow(A_s) > absRow(A_t)$. Let $D = \{d \mid d_{type} = FS,\ l_{d_t} = l_{A_t},\ d_s \neq A_s\}$. We then define the **offset** of A as $offset_A = |\{dp \mid dp \in D,\ absRow(dp_s) > absRow(A_s),\ absRow(dp_t) \leq absRow(A_t\}|$.*

We only consider an FS dependency that goes downwards (i.e., the target is below the source) in Definition 14 but it works analogously for the upward case. When again considering Figure 6.19, we can see that dependency AB has the offset of two because two dependencies come from above and go into the same target work package.

The problem is that we currently do not consider dependencies that start in the same absolute row positions, in different layers, and go to the same target layer and in the same vertical direction (see Figure 6.20). We handle this case by increasing the offset for a specific dependency depending on how many such dependencies start to the right of it. For example, in Figure 6.20 we only increase the offset for dependency AC.

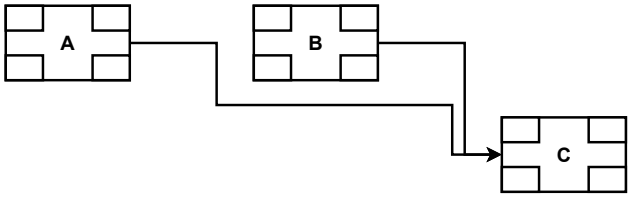


Figure 6.20.: Problem when considering two dependencies starting in the same row, going into the same target layer with different jump values

Finish-to-Start Multiple Out-Going Dependencies

We also want to consider the case of two or more dependencies from the same work package going into the same layer and in the same direction, to further increases the readability of the AND (see Figure 6.21). The goal is to keep the offset for the closest target and increase it the most for the target the furthest away.

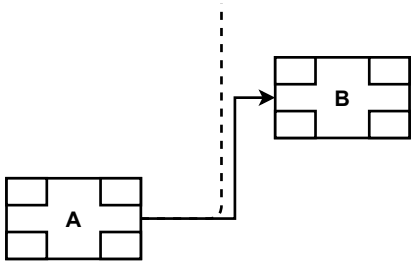


Figure 6.21.: Shifting dependencies that start at the same source and going to the same direction

To achieve this, we consider all dependencies starting at the same source and going in the same direction. We can then order these work packages by their absolute row position. The offset of a dependency is then increased by its positioning in that ordering. So, the dependency ending in the nearest target work package can keep its offset while we increase it the most for the dependency where the target is the furthest away. For example, in Figure 6.21, we keep the offset of dependency AB and increase the offset for the dashed dependency by one.

Special Dependencies at Activity Network Boundary

Definition 15. *Let $A = (A_s, A_t)$ be a special dependency and $bt(A) = \{wp \mid l_{A_s} \leq l_{wp} \leq l_{A_t},\ absRow(wp) > absRow(A_s)\}$. We then define dependency A to be in a* ***local maximum*** *if $|bt(A)| = 0$.*

Definition 16. *Let $A = (A_s, A_t)$ be a special dependency and $bt(A) = \{wp \mid l_{A_s} \leq l_{wp} \leq l_{A_t},\ absRow(wp) < absRow(A_s)\}$. We then define dependency A to be in a* ***local minimum*** *if $|bt(A)| = 0$.*

As mentioned above, whether a dependency goes out at the top or at the bottom of the source work package usually depends on whether the target is above or below the source. However, there are cases where arranging a dependency at the bottom for a dependency where the target is above can yield good improvements (analogously for downward dependencies). These cases are those where the dependency is in a local minimum or maximum, because, by definition, we know that there are no work packages that can produce crossings above a local maximum or below a local minimum. For example, in Figure 6.22, assuming work package B is a local maximum, we can, instead of leaving at the bottom, leave at the top, which avoids the crossing with the dependency AB.

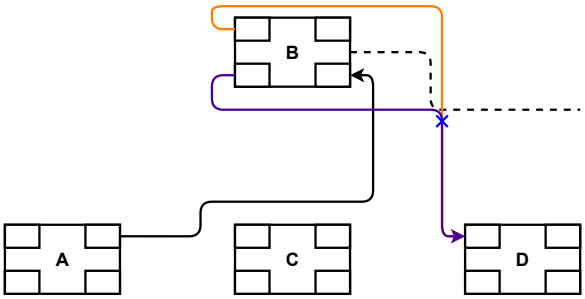


Figure 6.22.: Arranging the dependency below the source would result in fewer crossings within the AND

Placing the dependency on the outside of a local minimum or maximum can sometimes also have the effect of increasing the number of crossings. Furthermore, when both arrangements yield the same number of crossings, one should always choose the more direct variant since the dependency length is much smaller in that case.

We only consider a special dependency being in a local maximum, but the local minimum case works analogously. Again, it is essential that each dependency directly goes into its target layer before going upwards or downwards. This is needed, because the main idea of the following rule is based on calculating the crossings produced exclusively by the different drawing possibilities up to the point where they are drawn similarly (e.g., up to the blue cross in Figure 6.22).

Definition 17. *Let $A = (A_s, A_t)$ be a special dependency in a local maximum. Let $sa(A) = \{wp \mid absRow(wp) = absRow(A_s),\ l_{A_s} \leq l_{wp} < l_{A_t}\}$ (in case of SF or FF dependency $l_{A_s} \leq l_{wp} \leq l_{A_t}$). Then we can define the* ***outside number of crossings*** *as $outsideCros(A) = |\{D \mid D \neq A,\ D_s \in sa(A),\ l_{D_t} > l_{A_t}\}|$.*

We start by calculating the number of crossings produced when placing the dependency on the outer side in Definition 17. For example, in Figure 6.22 we need to calculate the number of out-going dependencies from B that go into a layer to the right of D, indicated by the dashed dependency.

Definition 18. *Let $A = (A_s, A_t)$ be a special dependency in a local maximum. Let $sa(A) = \{wp \mid absRow(wp) = absRow(A_s),\ l_{A_s} \leq l_{wp} < l_{A_t}\}$. Then we can define the* ***normal number of crossings*** *as $normalCros(A) = |\{D \mid D \neq A,\ D_t \in sa(A)$ or $D_s \in sa(A)$ and $l_{D_t} < l_{A_t}\}|$.*

In the second step, we now calculate the number of crossings we produce up to the target layer when placing the dependency normally (see Definition 18). For example, in Figure 6.22 we consider the dependency AB since it only produces a crossing when placing the dependency normally.

As a result, we have the number of crossings produced exclusively by placing the dependency normally or to the outside. Therefore, it only remains to choose the drawing variant producing fewer crossings.

6.8.3. Placing the Bends

We have introduced several special rules that we apply to create a well-readable layout. On the one hand, we calculated the offsets for FS dependencies and, on the other hand, whether we place a dependency at the top or at the bottom in specific cases.

Furthermore, we decided to shift the exit and entry position at the work packages for special dependencies further to the top or bottom depending on the dependency arrangement (e.g., in Figure 6.23 the SF dependency AB goes out at the top of A and goes in at the bottom of B). We do this because we only have relatively few special dependencies, so it is easier to distinguish between special and standard dependencies. In contrast, a FS dependency always leaves and enters at the middle of the source and target.

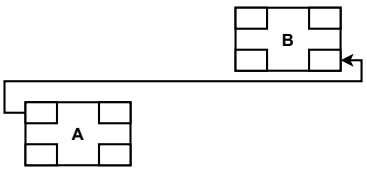


Figure 6.23.: Exit and entry positions at a work package shifted depending on the relationship between source and target work package

Finish-to-Start Bend Points

In addition to the x-spacing between two neighboring layers, we create a margin between the work packages and the x-spacing, to have the space to arrange special dependencies and draw the dependencys' arrows. We also use the right margin's boundary as starting point to shift FS dependencies to the left based on their calculated offset (i.e., an FS dependency with an offset of zero is placed onto the right boundary and the greater the offset, the further the dependency is shifted to the left).

We have three different cases for the bend point assignment. First, when having a FS dependency to a work package into its own absolute row position with no work package in between, we do not have to include any bends (e.g., dependency AC in Figure 6.24).

In the second case, we consider a dependency where we can directly go into the target layer, but the target is not on the same absolute row position. In this case we directly go into the target layer and then upwards or downwards to the target work package (e.g., dashed dependency starting from D in Figure 6.24).

In the last case, we consider a dependency where one can not directly go into the target layer (e.g., dependency AF in Figure 6.24). In this case we first check in which direction we leave the source, then use the nearest y-spacing to get into the target layer and lastly go upwards or downwards to the target.

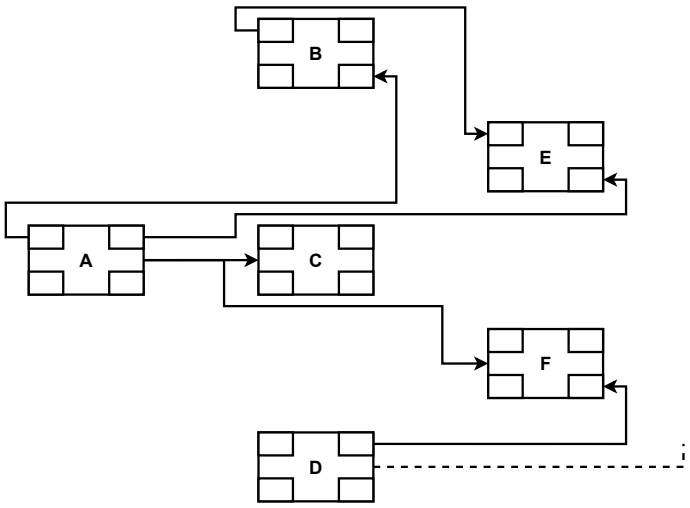


Figure 6.24.: Visualization of the bend point placement

Special Dependency Bend Points

The FF dependency is the only special dependency that might also directly go into the target layer without using the y-spacing (e.g., dependency DF in Figure 6.24). This is because both other special types always first have to go around their own source work package to reach the target layer.

However, if we are not in the case above, we also first check in which direction the special dependency leaves, then go to the target layer (i.e., left side of target for SS and right side for FF and SF dependencies) through the nearest y-spacing to then go upwards or downwards to the target work package (e.g., SF dependency AB, SS dependency BE, and FF dependency AE in Figure 6.24).

6.8.4. Conclusion

In this section, we first described specific rules to define the position of work packages, and decision rules on which the bend placement is based. In the last step, we used these decision rules and the spacings we introduced during the work package placement to describe how to assign each dependency its bend points. The main focus was to not create dependency crossings and to only use at most four bends for each dependency. This last step is crucial for achieving a well-readable output, because if certain cases are not considered carefully, the previous steps we applied to the AND abstraction would be mostly useless. This is due to the first observation we made in this chapter, where we discussed that by wrongly assigning positions and bend points, the resulting AND can be hard to understand.

To sum up, we look back at the initial example AND abstraction in Figure 6.4. Throughout this chapter, we mainly worked with such an abstract view of the AND, and only in the final step did we transformed it to the lecturer's AND structure. Figure 6.25 shows what the initial abstract AND example looks like after having applied the final transformation step to it.

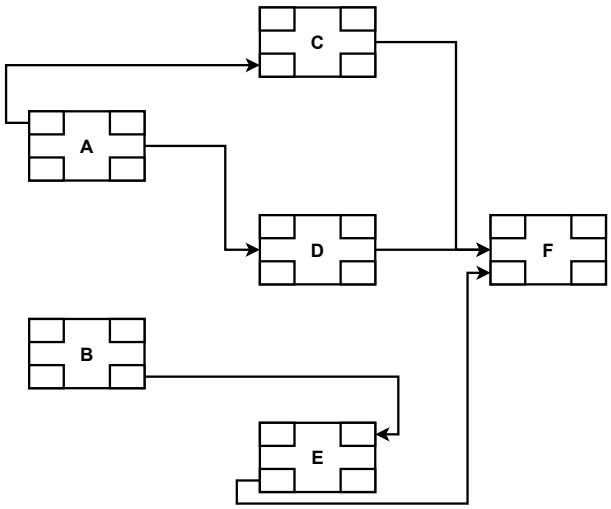


Figure 6.25.: Initial abstract activity network diagram after applying the final transformation step to it

7. Design

In the previous chapter, we have shown the different steps a generated AND is processed with before it gets drawn in the final step. We mainly used our AND abstraction throughout these steps and only transformed it to the lecture's AND structure in the last step. We now introduce our data model, which includes the internal AND model used to represent the AND abstraction. In the end, we introduce the different microservices we use and how they interact with each other.

7.1. Data model

7.1.1. Activity Network Diagram Design

We will start by explaining the structure of the internal AND model, which can be seen in Figure 7.1, that corresponds to the AND abstraction described in section 6.2.

First of all, we have a class describing the dependencies, consisting of the identifier of its source and target vertices and its dependency type. The vertex object directly contains the structural properties we introduced for the AND abstraction in section 6.2, so we can quickly access them, since we need them more frequently. The actual work package information is then included as a separate object to isolate this information from the structural information.

The actual AND model contains a list of dependencies and a vertex map with the id as key and the vertex as value. This vertex map allows us to quickly access the vertices corresponding to a dependency and to check whether the AND already contains a vertex. In addition, we also included a map having the layer number as key and the layer size as value. This allows us to quickly access a layer's size and get the maximum layer size, both of which are needed to calculate the absolute row position of vertices.

Business Logic

Interactive software systems often consist of three distinct tiers: the presentation, application, and data tier [LL13]. The presentation tier is used to interact with a user, and its primary purpose is to provide information to or get data from the user. We use a *Command Line Interface (CLI)* for this. The application tier contains the logic to process the presentation tier's information and modify the data in the data tier. The data tier is solely used to manage the information given by the application tier [LL13].

We do not use such a strict division between the application and data tier, since we want to avoid creating too many abstractions, as we currently do not persistently store the AND model. Therefore, the *ActivityNetworkDAO* also contains some functionality

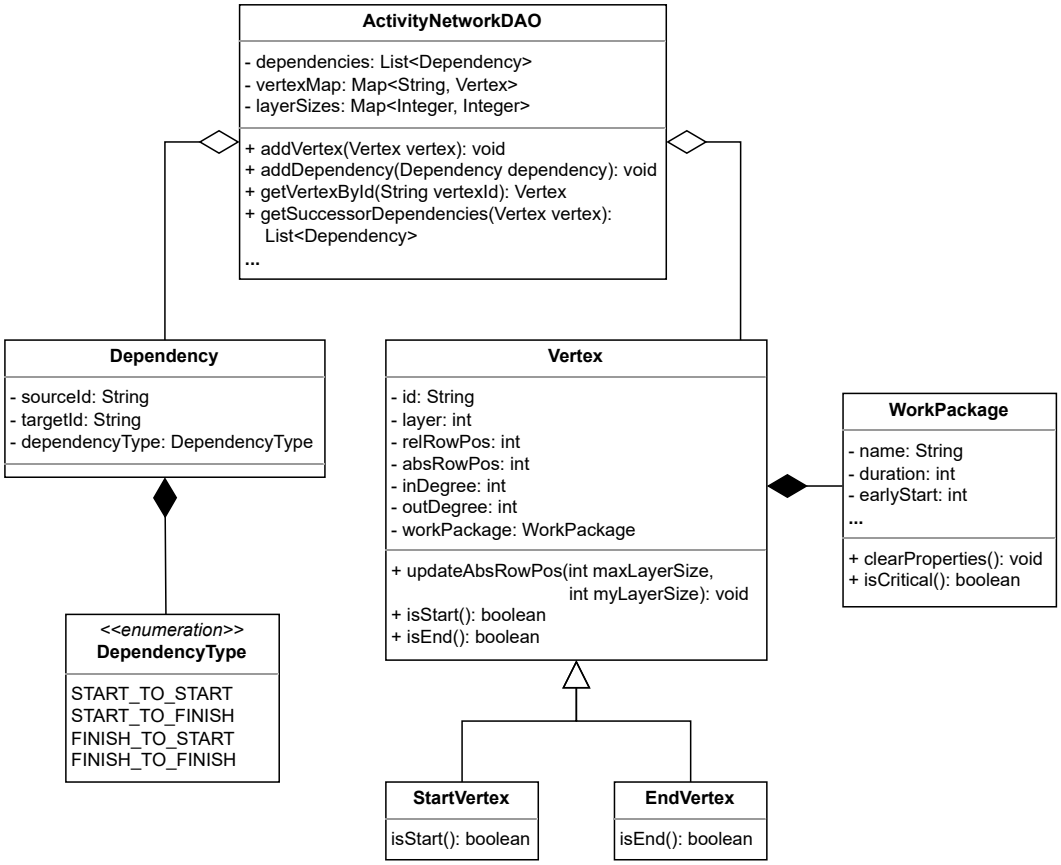


Figure 7.1.: Design of then internal AND model

to modify its properties, e.g., to add dependencies or vertices. Nevertheless, we only included the basic functionality and still used business logic objects to provide more extensive functionalities. This reduces the effort required if we want to make the internal data model persistent in the future. We briefly introduce the main business logic objects and their usage in the following.

- The *ActivityNetworkOrderer* provides functionality to check whether an AND is correctly layered, and in case it is not, it provides a method to create a new valid layering. Additionally, since the absolute row position of a vertex is based on the maximum and its own layer size, the object allows recalculating the absolute row position for each work package.
- The *ActivityNetworkModifier* provides safe methods for modifying the AND. Safe, in this context, means that we ensure that all properties are still correct after changing anything within the AND. This is needed since the base AND model provides methods to add vertices or dependencies but does not check whether the AND is correctly layered or each absolute row position is correct.

7.1.2. Transformed Activity Network Diagram

To make the transformation described in section 6.8 independent of the final output format, we include a transformation from the internal AND model into a data model that includes all the properties needed for the AND drawing. This data model can be seen in Figure 7.2 showing that we still have a similar structure to the internal AND model, but rather with actual drawing properties. For example, the dependency type is replaced by the *exitX* and *entryX* values describing the work package sides a dependency leaves its source and enters at the target work package.

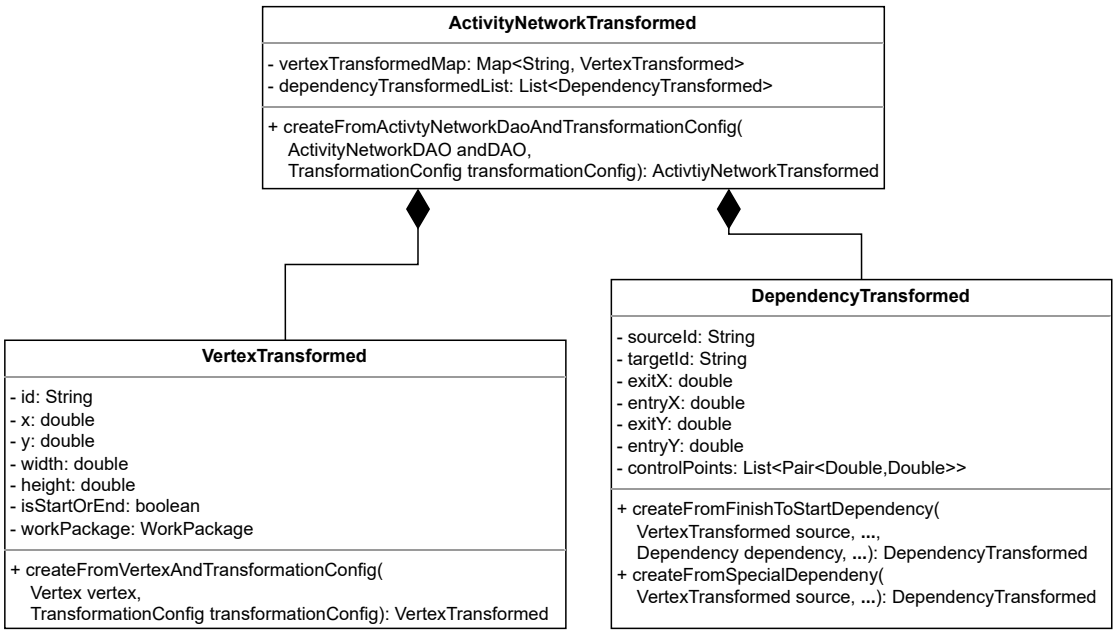


Figure 7.2.: UML class diagram of abstracted lectures activity network structure

7.1.3. Design of the Crossing Improvement - Strategy

The essential step to get a well-readable AND as output is the crossing improvement. This is because the generation step randomly creates dependencies between work packages without considering any positioning aspects like creating dependencies between work packages that are close to each other. Therefore, we must reorder each layer afterward to bring all work packages, and thereby the dependencies into a good structure (see section 6.5).

We use the strategy pattern, which is one of the well-known Gang Of Four [Gam+95] design patterns, to model the crossing improvement. The main idea of the pattern is to create an abstract view of a class of algorithms by introducing an interface describing the structure of the algorithm, called *Strategy*. A *Concrete Strategy* then implements this interface and can be assigned to the strategy used throughout the application. Therefore,

changing the concrete strategy usually only has to be done in a single place.

We first applied the strategy pattern to the crossing heuristics. These heuristics have in common that they consider one layer as free and the remaining layers as fixed. Therefore, we introduce the abstract class *CrossingHeuristic*, including a *permute* method, as one strategy. The two concrete strategies, the greedy switch and barycenter heuristics extend this class, so both must implement the permute method. The strategy pattern is particularly well suited here, as there are several other such heuristics [ML03].

Furthermore, we also applied it for the two crossing counter approaches where one approach calculates crossings in the whole AND and the other approach only around a specific layer (see section 6.5.1).

As a result, we have a highly extensible crossing improvement. However, we deviated from the original strategy pattern described in [Gam+95] since we did not use interfaces but rather abstract classes. This simplifies the concrete strategies since they all must depend on the AND. The explained design can be seen in the UML class diagram in Figure 7.3.

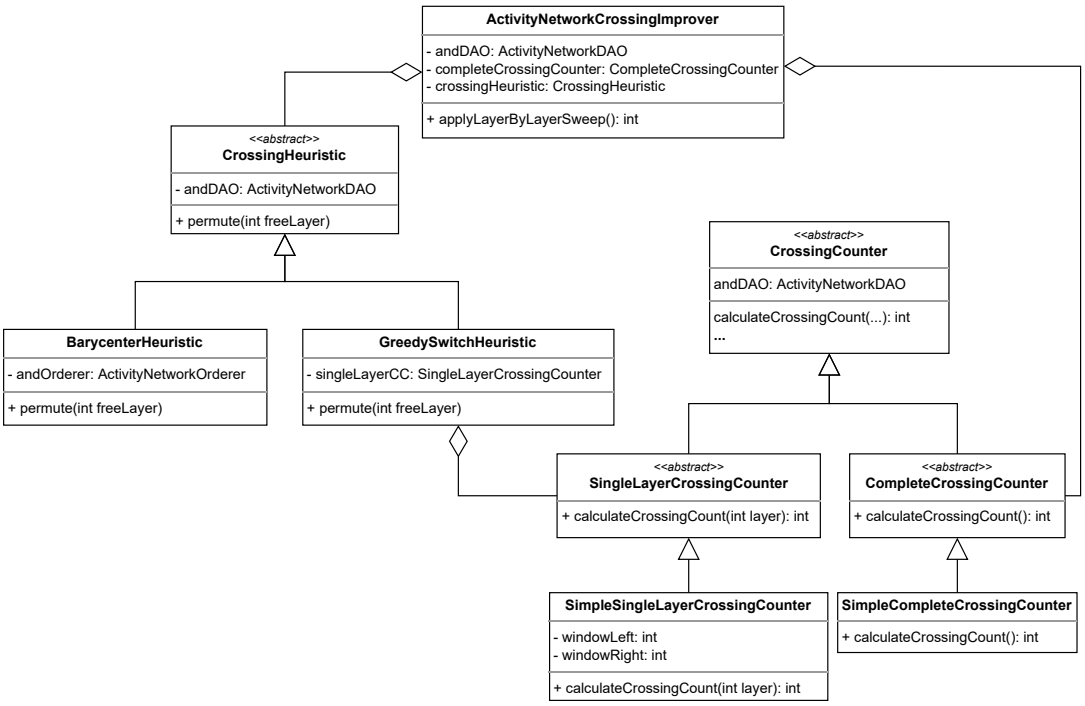


Figure 7.3.: UML class diagram for the crossing improvement

7.2. Microservice Architecture

One of our requirements was to use a microservice architecture to increase the extensibility of the application. In the following section, we first give a static view of the

microservices by explaining the functionality of each service. Then we will look at the dynamic view, i.e., the relationship and communication, between services.

7.2.1. Microservices - Static View

In the following, we introduce the functionality of the five services we use. In doing so, we sometimes refer to a configuration, which we introduce in section 8.2. However, it is only essential for this section to know that such a configuration exists.

API Gateway

The API gateway is the entry point to the system, and we use it to provide all the functionality to the user. Its main task is to provide user-friendly interfaces so that the user does not have to rely on the communication technologies used between the services [Ric19]. In addition, a gateway is often used to aggregate results from different services. However, we primarily use the gateway to transform user calls into an equivalent call utilized by the underlying services, forwarding them, and in the end, returning the result to the user.

Generation Service

The generation service is used to create the initial internal AND model. For this purpose, it must be provided with a suitable configuration that is to be used during the creation. Furthermore, it is only reasonable to apply the crossing avoidance step after the generation since it does not only reorder work packages but changes dependencies which should not happen with an input AND. Therefore, we have also included the crossing avoidance step in this service.

Calculation Service

The calculation service provides the functionality to calculate all AND properties. Since the calculation method requires an AND with proper layering, it also applies a new layering if the AND is not well layered.

Transformation Service

The transformation service is always the last service an AND runs through before being returned to the user. In doing so, it first transforms the internal AND model into the AND model containing the actual drawing information. The AND model is then further transformed into a proper graph format for a specific graph modeling tool such as MS Visio[1] or Draw IO[2]. We first transform the internal AND model to the AND model containing the drawing information because this allows us to relatively quickly add new graph formats.

[1] https://www.microsoft.com/de-de/microsoft-365/visio/flowchart-software
[2] https://app.diagrams.net/

The service can also get specific graph formats as input and transform them into the internal AND model. This is needed to get a modified AND, or an AND created from scratch, as input to calculate its properties and output it again. The AND structure and the properties of the work packages must be preserved when transforming it. This is because the output graph should have the same structure as the input, and all values are needed for possible functional extensions.

Configuration Service

The configuration service is used to store and modify the configurations we use within the generation and transformation service. It provides interfaces that encapsulate the typical *CRUD* (Create, Read, Update, and Delete) functionalities.

7.2.2. Microservices - Dynamic View

The previous subsection introduced the different microservices and their functionality, but only provided a static view of the architecture. We now explain how these services interact and communicate with each other.

The services communicate with each other by using a message broker. In Figure 7.4 we can see that the API Gateway at the top is the entry point to the system, and all communication goes through the message broker. This means that each service queues messages at the message broker that the designated target services can then consume. However, since Figure 7.4 still does not visualize which services interact with each other via the message broker, Figure 7.5 includes the logical flow between the services.

We now look at the sequence diagram in Figure 7.6 visualizing the generation process. For the sake of readability, we omit the message broker. First, the user has to provide a profile name correlated to an existing configuration which is to be used for the generation and transformation. The gateway then gets the corresponding profile configuration from the configuration service and forwards it to the generation service. The generation service performs all steps prior to the calculation as introduced in section 6.3 and forwards the generated AND to the calculation service. After the calculation, the transformation service produces the output AND, and sends it to the gateway, which returns it to the user.

In Figure 7.7 we see the sequence diagram for inputting an AND to calculate its properties. This process is very similar to the generation process, except that the generation step is replaced with the transformation of the input AND into the internal AND model.

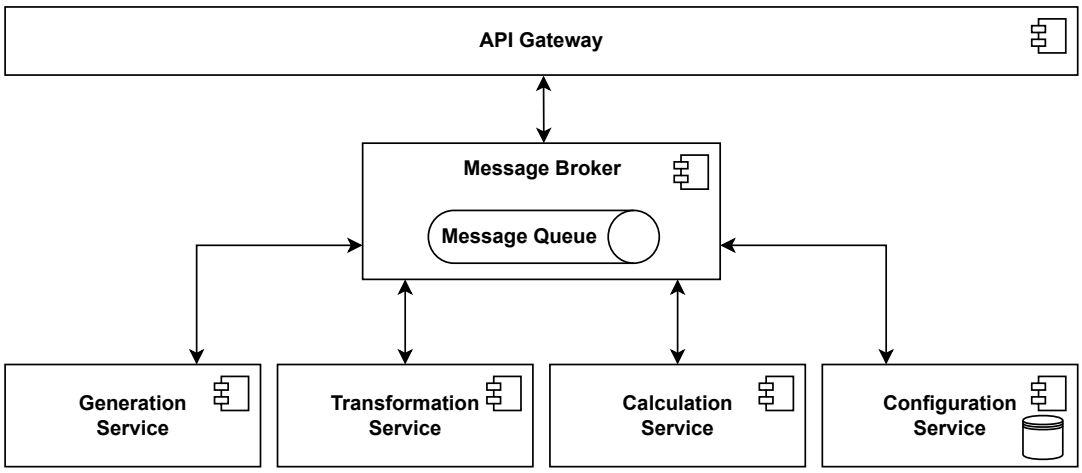


Figure 7.4.: Data flow between microservices

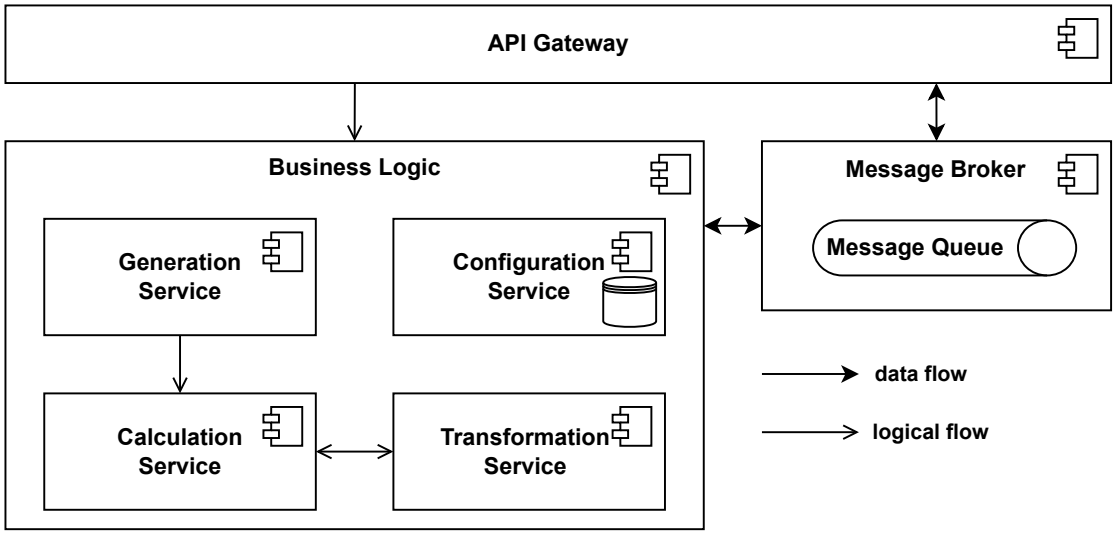


Figure 7.5.: Logical flow between microservices

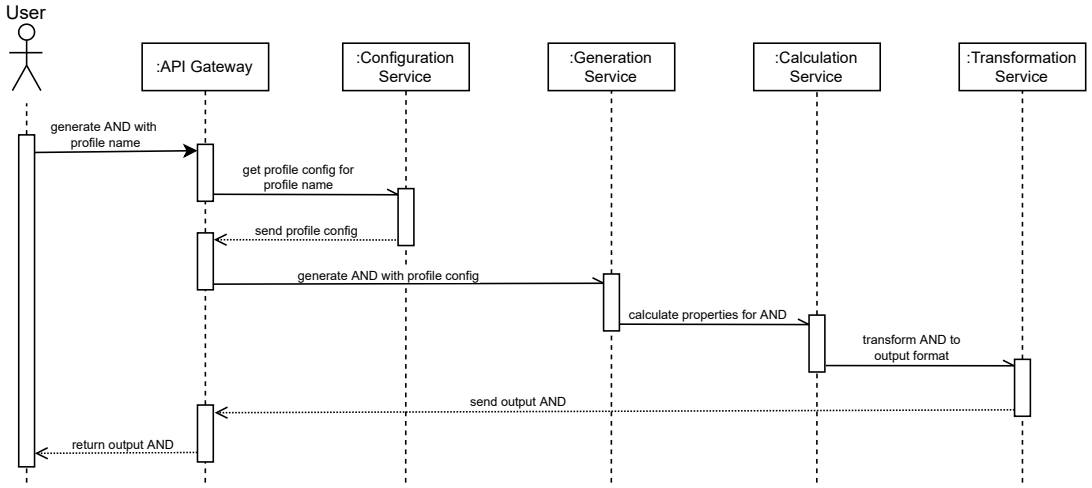


Figure 7.6.: Sequence diagram for the generation process

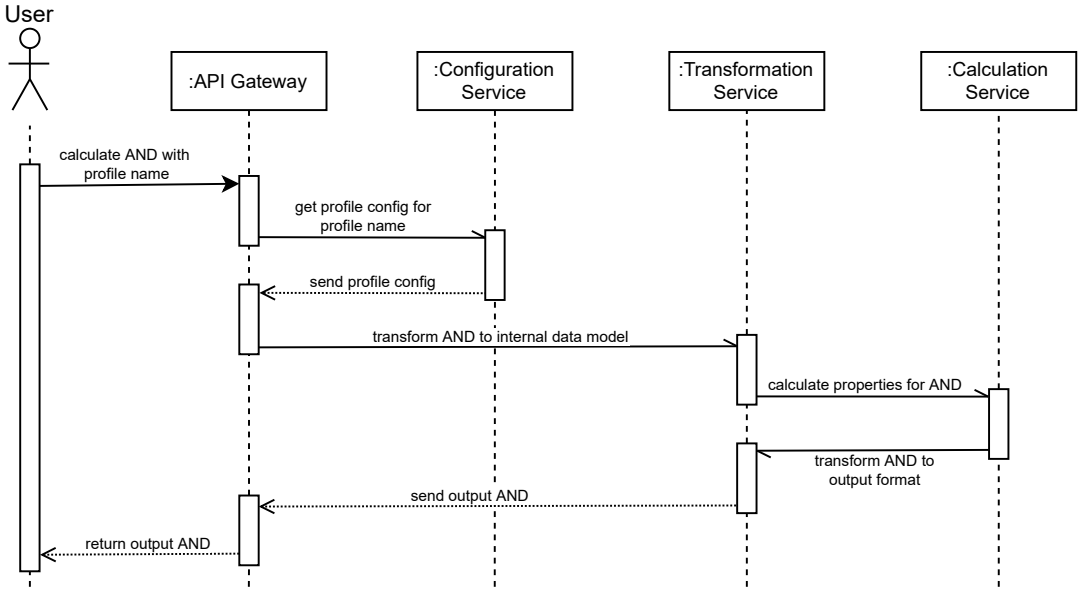


Figure 7.7.: Sequence diagram for calculating an input AND

8. Implementation

In this chapter we introduce the technologies used to implement the services introduced in the previous chapter. Then we describe the configuration we use for the generation and transformation process. Additionally, we introduce the graph modeling tool we use to visualize, modify, and create ANDs.

8.1. Technologies

For the implementation of the different microservices, we use *Java Spring Boot*[1]. We use *Maven*[2] to build each service, and to manage the dependencies to modules of other projects. We use *Docker*[3] to create docker containers for the microservices which allows to deploy services simply and quickly. To simplify the workflow with the several docker containers, we use *Docker Compose*[4], allowing us to start all containers with a simple command and configure them within a single file. We use the *NoSQL* database *MongoDB*[5] to persistently store the profile configurations because of the document-like structure of the configurations. We use *Apache Kafka*[6] as message broker because it provides efficient methods to exchange messages. Additionally, we use *Jackson*[7] to be able to serialize and deserialize Java Objects, to exchange them between services. To provide the different functionalities to users, we defined a simple *REST API* at the gateway service. Lastly, we use *Draw IO* as the graph modeling tool, which we will introduce in detail in section 8.3.

8.2. Configuration

Our first prototype had the generation and transformation properties statically set within the respective services. The problem with this approach is that it does not allow us to adjust the configuration depending on the desired output. The ability to change which configuration is used is especially useful for the drawing step, as we can use a configuration specifying a compact printing for the final version of ex-tasks, and a configuration for draft ex-tasks, where space is not an issue.

[1] https://spring.io/projects/spring-boot
[2] https://maven.apache.org/
[3] https://www.docker.com/
[4] https://docs.docker.com/compose/
[5] https://www.mongodb.com/
[6] https://kafka.apache.org/
[7] https://github.com/FasterXML/jackson

We use two configuration formats to handle this, one for the generation and one for the transformation. Both of these combined then create a configuration profile that is persistently stored in a MongoDB.

8.2.1. Generation Configuration

Even though, we only want to create ex-tasks in this thesis, we designed the generation step in such a way that it can also generated larger ANDs. We now introduce the generation configuration properties and then explain a problem they have. Each generation configuration consists of the following:

- a minimum and maximum *number of layers*
- a minimum and maximum *number of work packages*
- a minimum and maximum *size of the first, intermediate, and last layer*
- a minimum and maximum *duration*
- a *jump distribution*
- a *number of finish-to-finish* dependencies
- a *number of start-to-finish* dependencies
- a *number of start-to-start* dependencies
- a *boolean* defining if the post-processing should be applied after the generation

Minimum and Maximum size of Layers

The only exceptions where the above values deviate from the ones used in section 6.4.2 are the minimum and maximum layer sizes. Here, we divided these two values further into six distinct properties. These properties are used to more finely define the layer sizes of the first layer, all intermediate layers, and the last layer, which is helpful for the creation of ex-tasks. The reason for this is that the first and last layers are uninteresting for ex-tasks since their in-going (for the first layer) and out-going (for the last layer) dependencies are easy to calculate. This is due to the fact that the start and end work packages have a duration of zero and only FS dependencies.

Generation Configuration Problem

The generation configuration has the problem that the minimum and the maximum number of work packages we can generate do not depend only on the corresponding values defined in the configuration. Equation 8.1 and Equation 8.2 define the additional conditions for the minimum and maximum number of generatable work packages. These conditions show that it might not be possible to generate ANDs with the minimum or the maximum number of work packages as specified in the configuration.

$$minNumWP \geq minSizeFLay + (minNumLay - 2) * minSizeILay + minSizeLLay \quad (8.1)$$

$$maxNumWP \leq maxSizeFLay + (maxNumLay - 2) * maxSizeILay + maxSizeLLay \quad (8.2)$$

We handle this problem by using the maximum value of the two lower bounds (i.e., the defined value and the value in Equation 8.1) and the minimum value of the two upper bounds. By doing so, we ensure that we only use correct boundaries within the generation.

8.2.2. Transformation Configuration

Having presented the generation configuration, we will now provide the transformation configuration, which consists of the following elements:

- a *critical path color*
- an *x-zero* and *y-zero* position
- an *x-spacing* and *y-spacing*
- a work package *width* and *height*
- a start/end work package *width* and *height*
- a *boolean* defining whether the name and duration should be separated by a line break or white space
- a *boolean* defining whether to use rounded bends

8.3. Draw IO

Draw IO is a tool that allows creating just about any diagram one can think of. It follows a simple drag and drop system where one can move and combine shapes in any way. We use Draw IO, since one of our requirements was to be able to modify ANDs and build ANDs from scratch, which is possible with Draw IO. In addition, the graph representation language of Draw IO allows the importation and exportation of ANDs in a format that can be transformed into the internal data model. However, using such a comprehensive tool also has some disadvantages because we have to define certain restrictions and assumptions to transform a created or modified AND into our internal AND model.

8.3.1. Diagram Representation with XML structure

We begin with introducing the Draw IO diagram representation which is based on an XML structure, which is generally a reasonable decision, as diagrams often follow a hierarchical structure. So, considering the work package representation, we could think

of a group element as the parent with five children. These children would then describe the actual properties of a work package, such as the name and duration, or the ES. Additionally, one could introduce several tags describing the different elements, such as vertices and edges. Unfortunately, the Draw IO XML representation does not do any of these things.

The Draw IO XML structure first always consist of an outer *mxGraphModel* followed by a *root* tag. Then the root tag contains all the visible diagram elements as direct children. This means that the XML representation does not make use of the hierarchical structure of XML but instead tries to avoid it by using other techniques. For example, the root tag only contains *mxCell* elements as children. Here, each of these elements has either an *edge = "1"* or *vertex = "1"* attribute. In other words, instead of working with different tags (e.g., *mxVertex* and *mxEdge*), they define the type using an attribute.

As mentioned, each visible element is a direct child of the root tag. Therefore, the model does not include a typical parent-children relationship of elements by placing the children between the parent tags. This means that in Draw IO, each element contains an attribute *parent = "xy"* referring to the actual parent element. To do so each *mxCell* element contains a unique identifier as an attribute. Only the essential structural information, such as the position or width of an element, are contained in a child *mxGeometry* element. In Source Code 8.1 we see the Draw IO XML representation of Figure 8.1.

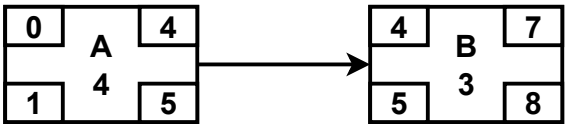


Figure 8.1.: Diagram visualization of the mxGraphModel in Source Code 8.1

Fortunately, there is the *JGraphX*[8] package that provides some functionality to work with the mxGraphModel in Java. Although the primary use case of this package is to visualize and work with the mxGraphModel within Java Swing (i.e., an old GUI-Toolkit for Java), it is sufficient for us. It provides functionality to create a mxGraphModel programmatically and then transform it into the XML structure. In addition, one can also transform the XML structure into a mxGraphModel needed to get a modified or entirely new AND as input.

8.3.2. Mapping the Internal AND to mxGraphModel

The JGraphX package allows us to create a mxGraphModel and fill it with mxCells. For the sake of readability, we refer to an mxCell only as *cell* from now on. When creating an AND drawing, the first step is to transform the internal AND used within the services into the AND representation that contains the actual drawing information. Since the transformed vertices and dependencies already contain all of the properties we need for the mxGraphModel, we only have to create each cell and output the resulting

[8] https://github.com/jgraph/jgraphx

```
<mxGraphModel ...>
  <root>
    <mxCell id="0" />
    <mxCell id="1" parent="0" />

    <mxCell id="14" style="...;exitX=1;exitY=0.5;entryX=0;entryY=0.5" parent="1"
        source="2" target="9" edge="1">
      <mxGeometry relative="1" as="geometry" />
    </mxCell>

    <mxCell id="2" value="" style="group" parent="1" vertex="1">
      <mxGeometry x="220" y="320" width="100" height="60" as="geometry" />
    </mxCell>
    <mxCell id="3" value="A&lt;br&gt;4" style="..." parent="2" vertex="1">
      <mxGeometry width="100" height="60" as="geometry" />
    </mxCell>
    <mxCell id="4" value="0" style="..." parent="2" vertex="1">
      <mxGeometry width="30" height="20" as="geometry" />
    </mxCell>
    <mxCell id="5" value="4" style="..." parent="2" vertex="1">
      <mxGeometry x="70" width="30" height="20" as="geometry" />
    </mxCell>
    <mxCell id="6" value="1" style="..." parent="2" vertex="1">
      <mxGeometry y="40" width="30" height="20" as="geometry" />
    </mxCell>
    <mxCell id="7" value="5" style="..." parent="2" vertex="1">
      <mxGeometry x="70" y="40" width="30" height="20" as="geometry" />
    </mxCell>

    <mxCell id="8" value="" style="group" parent="1" vertex="1">
      <mxGeometry x="410" y="320" width="100" height="60" as="geometry" />
    </mxCell>
    <mxCell id="9" value="B&lt;br&gt;3" style="..." parent="8" vertex="1">
      <mxGeometry width="100" height="60" as="geometry" />
    </mxCell>
    <mxCell id="10" value="4" style="..." parent="8" vertex="1">
      <mxGeometry width="30" height="20" as="geometry" />
    </mxCell>
    <mxCell id="11" value="7" style="..." parent="8" vertex="1">
      <mxGeometry x="70" width="30" height="20" as="geometry" />
    </mxCell>
    <mxCell id="12" value="5" style="..." parent="8" vertex="1">
      <mxGeometry y="40" width="30" height="20" as="geometry" />
    </mxCell>
    <mxCell id="13" value="8" style="..." parent="8" vertex="1">
      <mxGeometry x="70" y="40" width="30" height="20" as="geometry" />
    </mxCell>
  </root>
</mxGraphModel>
```

Source Code 8.1: mxGraphModel describing Figure 8.1

mxGraphModel. Here, a work package consists of six cells, an outer group cell containing five child cells, one for each of the work package properties. The cell for the name and duration is the same size as the group, while the four cells for the schedule properties are smaller and placed in their respective corners, as described in section 2.2.3.

8.3.3. Mapping mxGraphModel to the Internal AND

The first version of our prototype did not read in all work package values but only the name and duration. In general, this is sufficient when only considering importing ANDs to calculate their schedule properties. However, as we want to maintain the extensibility of our architecture, we make sure that there is no information loss during the importation of an AND. We now explain how we transform a mxGraphModel into the internal AND model that takes all properties into account.

First, in contrast to producing the output mxGraphModel, parsing mxGraphModel is more challenging. The most prominent problem is identifying which property a vertex cell describes, since the group cell, the name and duration cell, and each schedule property cell are all vertex cells. Therefore, we now describe how we identify what part of an AND a given vertex cell describes, which we then use to explain the transformation.

Identifying the Start and End Cell

We begin with the start and end work package cells. The first approach we considered to identify them was distinguishing them by their different width and height since these properties usually differ from normal work package cells. However, this would only be possible when using the width and height values from the transformation configuration provided, but we want to handle the input independently from the configuration. Therefore, we decided that the start cell needs to have the name *start* and the end cell the name *end*, since that is the most simple and convenient method to differentiate them from normal work package cells.

Identifying the Group Cell

As stated above, a work package always consists of a group cell containing five child cells, one for each of the five properties of an AND (the name and duration make up one property). Therefore, we first check if a cell contains five child cells. However, to avoid accidentally identifying any cell with five children as a work package group cell, we also check whether the cell contains four children with numeric or empty values, which must be the case for the four schedule properties.

Identifying the Schedule Property Cell

To identify schedule property cells, we first check if the parent cell is a group cell and if it has no children of its own. Finally, to ensure that it is a schedule property and not the name and duration cell, we check whether the value is either numerical or blank.

We identify which of the four schedule properties it is by checking in which corner of the group cell it is located.

Identifying the Name and Duration Cell

To identify the name and duration cell, we first check whether the parent cell is a group cell and the cell itself has no children. If this is the case, we only need to check whether it has the same width and height as its parent cell (i.e., the group cell).

Creating Work Packages Vertices

To create work package vertices from vertex cells, we consider each vertex cell individually, check whether it is a start, end, or regular work package group cell, and then create the individual AND parts. We can directly create start and end vertices since they always have a duration of zero and no visible schedule properties. For normal work package group cells, we need to consider each child and extract their values. In addition, each internal vertex gets the id of the group cell assigned, since we can be sure that it is unique.

Additionally, we want to preserve the input positions, since we always want to keep the work packages in the same layer for the resulting output. The absolute row position and relative position should also be preserved but might change if the crossing improvement finds a more suitable ordering. To achieve this, we need to save cells' x- and y-positions when transforming them. After transforming all work package cells to AND vertices, we first group them by their corresponding x-positions and assign them a layer based on the group's x position. We then consider each group separately and sort them by their corresponding y-positions to assign each vertex a relative position.

In the end, we have a list of work packages, each with the correct layer and relative position assigned. Therefore, we only need to calculate the absolute row positions to set all properties correctly.

Creating dependencies

Creating dependencies is easier than creating work package vertices, since the dependency type is the only thing we must transform. This is because each edge cell already contains its source and target cell id. To transform the dependency type, we need to extract the *exitX* and *entryX* positions and assign the dependency type based on these two values (e.g., $exitX = 0$ and $entryX = 1$ means leaving on left side of the source and entering on the right side of the target, thus an SF dependency).

9. Evaluation

We now discuss the results of this thesis. We begin by considering the generation of ex-tasks since this was the primary goal of this thesis. In doing so, we compare the reference AND for this thesis (see Figure 9.2) with a generated AND having a similar number of work packages. We then use quantitative analysis to verify that the generated ANDs meet the requirements and to compare the performance of the crossing improvement heuristics we introduced. Lastly, we consider the generation of larger ANDs, and how well our generator performs in this case.

We always specify the random seeds which were used to generate our sample and example ANDs, to provide the possibility to replicate the ANDs we used.

9.1. Generating Activity Network Diagram for Ex-Tasks

The following section is mostly based on ANDs that were generated with the default ex-task profile configuration in Source Code A.1.

9.1.1. Qualitative Analysis

We will begin by reconsidering the reference AND we have seen in Figure 9.2 and compare it with the generated AND in Figure 9.1 having the same number of work packages and layers.

Property Comparison

The reference AND has five different special dependencies compared to only four in the generated AND. The reason for this is that we have specified a value for each special dependency type, that describes how often it occurs in a generated AND. For our ex-tasks, this value is set to one for each special dependency type. If we increase a single special dependency type by one, we get the same number as in the reference AND, but it is not reasonable because we always create the same special type twice. This shows that the current generation configuration is missing a parameter that specifies how many random additional special dependency types should be generated. However, this is not a big problem because one can manually add another special dependency type after the generation.

If we look at the in-degree and out-degree of work packages, we see that we have a maximum in-degree and out-degree of two in the reference AND. In contrast, we have work package E in the generated AND with three in-going and three out-going dependencies. This is because the generation method favors a jump value of one for

ex-tasks. Therefore, a layer with only a single work package has a high probability of having many in-going and out-going dependencies. Furthermore, we can see that we have more work packages with only a single in-going and out-going dependency in the generated AND (eight versus four). This shows that it might be helpful to include a method that moves dependencies from work packages with a high in-degree or out-degree to work packages having only a single in-going or out-going dependency.

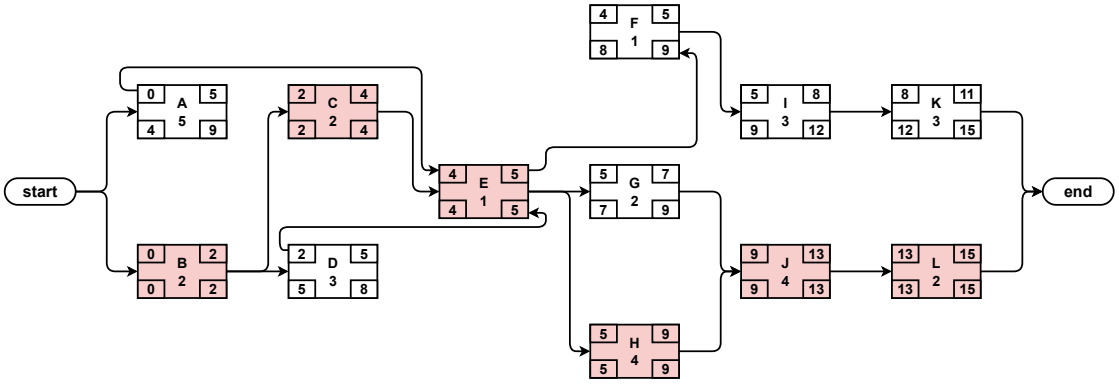


Figure 9.1.: Example for a generated ex-tasks activity network diagram generated with Source Code A.1 (seed: 1645629446882)

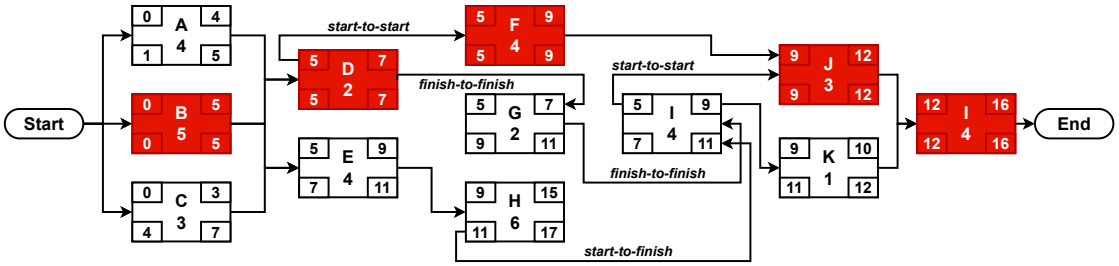


Figure 9.2.: Reference AND taken from a previous exam task (Figure 2.7)

Layout Comparison

We can see that the reference AND is more compact than the generated AND. However, this is mainly due to the empty space we introduced between neighboring work packages within a layer, affecting only the AND drawing height. This is essential because it is more important to have a compact width, which is the case for the generated AND, in order to be able to place the AND on an exercise or exam sheet.

Furthermore, we can see that the generated AND does not contain labels for the special dependency types. However, the dependency type is well defined by their out-going and in-going sides, and the label is only included within the reference AND to avoid misunderstandings by students. Therefore, not including the label is not a problem,

especially since automatically placing the label without overlapping with other work packages or dependencies is much more complicated than manually placing the label before completing the ex-task.

When further looking at the dependency arrangement in the reference AND, we can see similar decisions as we have included in our drawing. For example, the SF dependency *HI* in the reference AND is placed to the bottom even though the target work package is above *H*. When now considering the generated AND, we have similar behavior for the dependency *AE* which is the result of one of our introduced drawing rules. In addition, we decided to have a maximum of four bends to increase the readability, which is also the case in the reference AND.

9.1.2. Quantitative Analysis

We conducted a quantitative analysis to evaluate the different methods and approaches we applied. In doing so, we generated 100 ANDs with the profile configuration Source Code A.1 and seeds 1000 to 1099. We analyzed structural information such as the number of work packages and layers, and three crossing improvement heuristic approaches, the barycenter heuristic, the greedy switch heuristic, and combining both by first applying the barycenter and then the greedy switch heuristic, to evaluate which approach performs the best. In the following, we describe the results we obtained.

Structural Evaluation

	Min	Median	Max	Mean
Number of work packages	11	12	13	12.0
Number of layers	5	5	6	5.49
Dependencies of length one	12	16	20	15.51
Dependencies of length two	0	2	7	2.04
Maximum in-degree	2	3	5	2,78
Maximum out-degree	2	2	4	2,38

Table 9.1.: Table showing the statistical information from the 100 generated ANDs

We now discuss whether the generated ex-tasks fulfill the requirements we defined in section 2.2.4 and the limits defined in the generation configuration. Table 9.1 summarizes the most important metrics we considered.

We can see that most of the values are in line with the expected values, for example in our sample, the average number of layers is 5.49 and the number of work packages is 12, where we would expect an average number of layers of 5.5 and an average number of work packages of 12. The only deviation is the number of dependencies with length one and two, where we expected to get around 80% dependencies of length one and 20% of length two but 88.5% have length one and only 11.5% a length of two. The reason for this is that we only have a small number of work packages in the start, penultimate, and

last layer, where we can only create dependencies of length one; thus, dependencies of length two can only be generated originating in the first layer to the third to last layer. To validate that this is the reason for this behavior, we considered some large ANDs with only a single work package in the first and last layer, which showed that we then approximately get 80% dependencies of length one and 20% of length two. However, since we only have one dependency of length two in the reference AND, and the generated AND already has two of them on average, we should not increase the probability for a jump value of two.

We also considered the maximum in-degree and out-degree, but excluded the start work package's out-degree and end work package's in-degree. We do this because the schedule properties resulting from their dependencies are not challenging to determine. In doing so, we see that each AND contains at least one work package with two in-going dependencies and one work package with two out-going dependencies, which is needed for an ex-task. In addition, we see that we often have a work package with an in-degree of three, which we might want to consider decreasing, as discussed in section 9.1.1.

Lastly, each AND contains each special dependency type once. Hence, the profile configuration in Source Code A.1 produces ANDs that fulfill the minimum ex-tasks requirements defined in section 2.2.4.

Crossing Heuristic Performance

We will now look at which of the three crossing improvement approaches stated above works the best and also evaluate the effect of the crossing avoidance step we included.

We start by looking at Figure 9.3, which shows the number of crossings we had before, and Figure 9.4, showing the number of crossings we had after, we applied the crossing avoidance step in the AND abstraction. Here we can see that the crossing avoidance step greatly reduced the number of crossings in the AND abstraction. Thus, it is very likely that we also have such a reduction in the actual AND drawing, but, at the very least, this should not have made it worse.

Furthermore, we also compared the number of crossings before and after the crossing avoidance for each generated AND, which confirmed that the crossing avoidance step never increases the number of crossings in the AND abstraction. In addition, when excluding the cases where we already had zero crossings before the crossing avoidance step, the crossing avoidance achieved a reduction in the number of crossings in about 59% of cases.

When now looking at how well the three different approaches perform, we can see that the greedy switch and barycenter heuristics produce quite similar results after the crossing avoidance step (Figure 9.4). Nevertheless, the barycenter heuristic produces fewer crossings in the resulting AND drawing (Figure 9.5), indicating that completely reordering the layer instead of just swapping work packages tends to produce better results for the actual AND, even though the result in the AND abstraction are quite similar.

However, we can see that the combined approach outperforms both other methods in all three stages. The reason for this is that we first get the results of the barycenter

heuristic in the first step but then use the greedy heuristic to find further improvements; thus, we improve and already acceptable results. To further validate this, we generated some large ANDs and compared the number of crossings before the crossing avoidance step. We saw that the barycenter heuristic performs considerably better than the greedy switch heuristic with respect to the runtime and the resulting number of crossings. Applying bot approaches sequentially reduces the resulting number of crossings even more.

In summary, we showed that using the barycenter followed by the greedy switch heuristic gives the best results, so this is the approach that should be used.

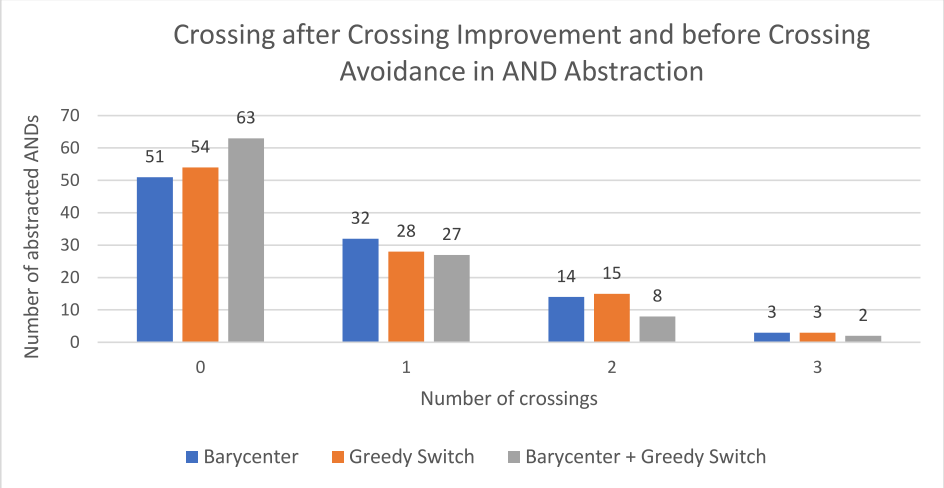


Figure 9.3.: Number of crossings in the AND abstraction after applying the crossing improvement step and before the crossing avoidance step

9.1.3. Activity Network Diagram Generation Problems

The generated AND we discussed in section 9.1.1 is only a single example AND having a well-readable structure. We can see in Figure 9.5 that only in 39% of cases are there no crossings in the AND, and there are even ANDs with a total of six crossings. One such AND can be seen in Figure 9.6, and we can identify several reasons for this problem. First, we did not consider the special dependency combination at work package K in our decision when a work package is in a local minimum or maximum, since it is improbable. In addition, we can see that we have three dependencies with a high absolute row difference (EK, GK, and HL) where the crossing avoidance step can not further reduce their absolute row difference. Additionally, we also have five dependencies with a jump value of two, which is considerable more than the average of two such dependencies, we see in Table 9.1. This clearly shows that the intuition that long dependencies tend to lead to many crossings is correct.

Nonetheless, we see in Figure 9.5 that the probability of having at least two crossings is only about 31% in our 100 generated ANDs. This shows that we can produce well-

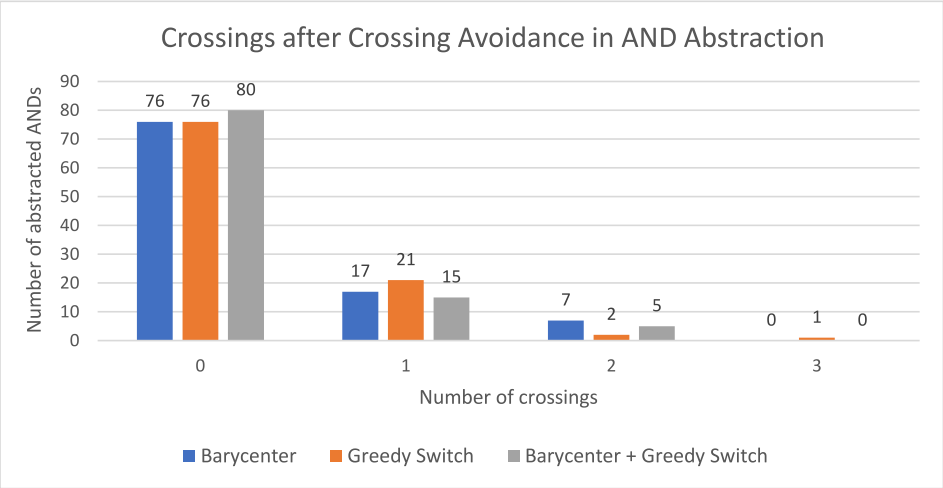


Figure 9.4.: Number of crossings in the AND abstraction after applying the crossing improvement step and crossing avoidance step

readable output ex-task ANDs most of the time, and an AND as seen in Figure 9.6 is only an exception. In addition, since the generation process for ANDs of this small size is fast, we can simply generate new ANDs until we get a more preferable output. If doing so, it usually only requires a few tries to get a practical AND as output.

9.2. Generation of Larger Activity Network Diagrams

In addition to generating ANDs for ex-tasks, the generation also supports the generation of larger ANDs. We already mentioned large ANDs in the previous section, and used them to validate our results. Now we want to look at a large example AND in Figure 9.7 to visualize, and briefly discuss, the possibilities of our generation method.

The initial large AND, without any changes, had a total of 359 crossings. The crossing improvement method reduces the crossings to only 77 with both heuristics combined, 79 with only the barycenter, and 240 with only the greedy switch heuristic. When considering the runtime, the barycenter heuristic is the fastest with 1.5 seconds, the combined approach needed 23 seconds, and the greedy switch heuristic 35 seconds. These values illustrate the runtime difference between the three approaches and show how efficient the barycenter heuristic is while still providing a good result.

Lastly, for such large ANDs, the crossing avoidance step can greatly reduce the crossings, in this case reducing them from 77 to only 12 for the combined approach. Since the generation of such ANDs is not our focus, we will not discuss this further.

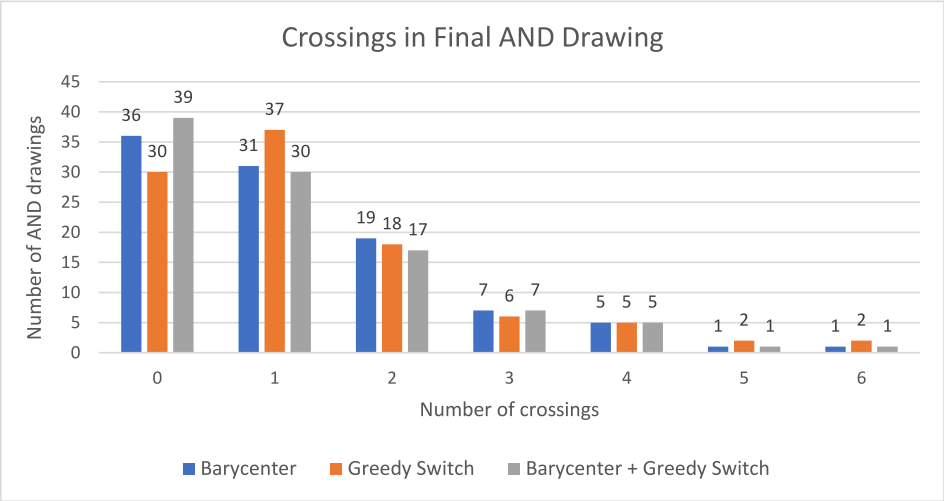


Figure 9.5.: Number of crossings in the AND drawing after applying the crossing improvement step and crossing avoidance step

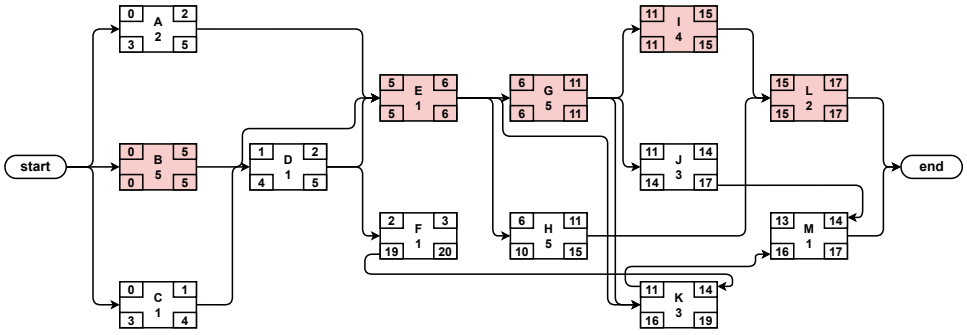


Figure 9.6.: Example of a bad ex-task AND containing six crossings (seed: 1089)

9.3. Discussion

After looking at some generated ex-task ANDs and the analysis of them in the previous sections, we now discuss these results and revisit the research questions we defined in chapter 3.

The results in the previous section have shown that we can generate ex-tasks with good properties. We have shown that we satisfy all the minimum requirements we defined for a practical ex-task in section 2.2.4. However, we have also found some weaknesses compared to the reference AND (Figure 9.2). The first problem is that the current generation only considers a number for each special dependency type describing how often it should be generated, so currently, one can not create a dependency of a random special type. Moreover, the generated ANDs often contain at least one work package with three in-going dependencies, which is not necessarily desirable. However, this is not a significant problem considering that we can easily modify a generated AND manually,

such as adding a fifth special dependency type, or removing or replacing a dependency at a work package. So, in the end, we can generate practical ex-tasks that only need minor adjustments to be used, which answers our first research question.

The second research question is about the resulting AND drawing of a generated graph, i.e., whether we can obtain a similar result for our generated ANDs as for the reference AND. We have shown that the drawing rules we defined work well for most generated ANDs when applying the crossing improvement and crossing avoidance steps beforehand. In total, only a small number of generated ANDs have a significant number of crossings in the resulting AND, which are caused by improbable structural properties (e.g., many dependencies with a jump value of two). Therefore, we can say that the several steps an initially generated AND runs through, and our drawing rules in the final steps, lead to a well-readable output being relatively similar to the reference AND.

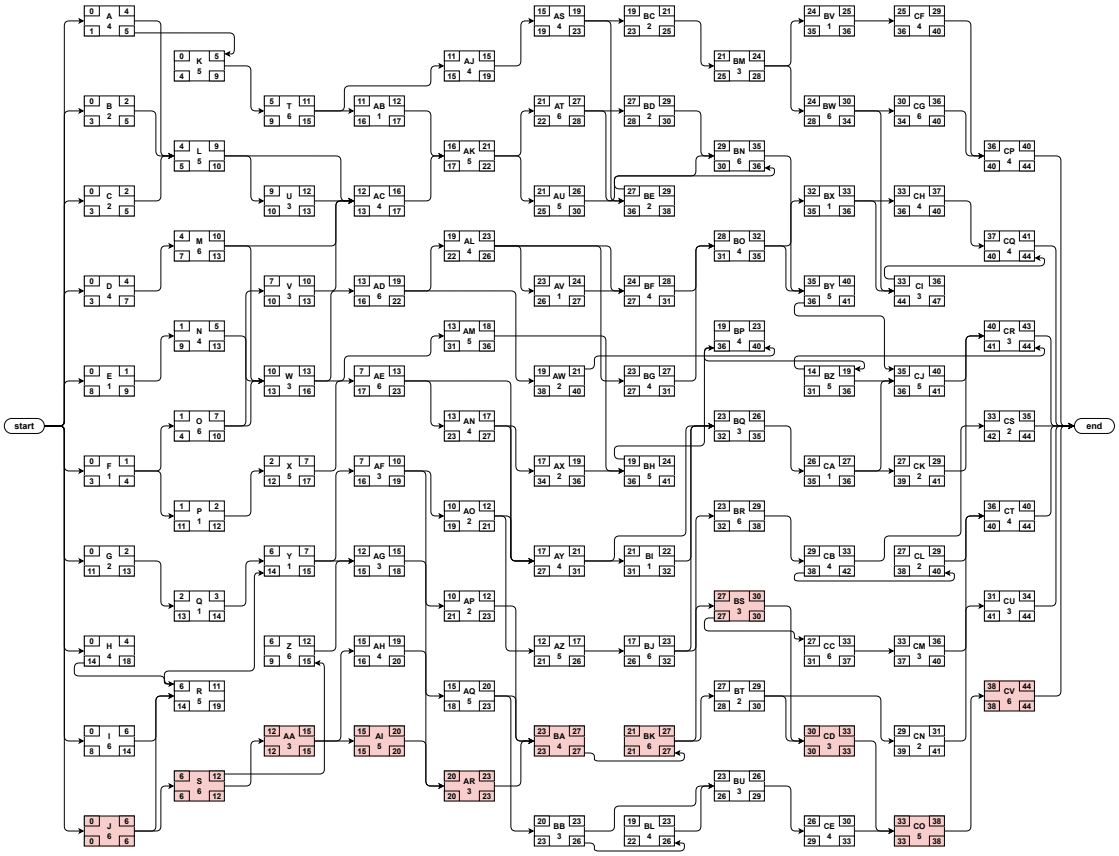


Figure 9.7.: Example for a large AND generated with Source Code A.2 and both heuristics combined (seed: 1645713753621)

10. Conclusion

10.1. Summary

This thesis presented an automated approach to generating ANDs with a well-readable output. Our primary goal was to generate ANDs suitable for ex-tasks to support the time-consuming and error-prone process of creating such tasks. A critical requirement, here, is getting well-readable output, otherwise one must apply many manual changes to make a generated AND understandable.

In the first step, we introduced an AND abstraction to isolate the intermediate steps from the lecture's AND structure. This abstraction was needed since including layout decisions such as the actual work package and dependency placement makes the intermediate steps too complex, some to the point where they become inapplicable. We defined the AND abstraction so that it still contained layout information, such as the position, but removed anything drawing related such as the unique dependency structure.

The generation process consists of several different steps. We start by creating the initial AND abstraction and then applying certain changes to it. First, we try to find the best work package ordering in each layer to reduce the crossings as much as possible. After the crossing improvement, the ANDs may still have crossings, so we apply a crossing avoidance step to reduce them further. After completing the crossing improvement and crossing avoidance steps, we have the final AND structure and proceed by calculate its schedule properties. In the final transformation step, we transform the AND abstraction into the lecture's AND structure. To implement these steps, we use five microservices, and Draw IO as the graph modeling tool. Draw IO allows modifying a generated AND and building a new AND from scratch. This is needed to be able to easily modify a generated AND without the need to manually recalculate the schedule properties.

In the end, we evaluated how practical and readable the generated ex-task ANDs are. In doing so, we compared a generated AND with the reference AND for this thesis, which showed that the generated AND only needs slight adjustments to have similar properties as the reference AND. In addition, we conducted a quantitative analysis to validate that generated ex-task ANDs fulfill their minimum requirements, which they do. We also used the quantitative analysis to compare two heuristics, and the combination of the two, used for the crossing improvement, which showed that the combined approach works the best. Nevertheless, we also saw that we sometimes get ANDs with many crossings, which are difficult to read. However, since the generation process for the ex-tasks is fast, one can generate new ANDs until one gets a proper output, without expending significant time.

Overall, we conclude that our generator for ex-tasks produces good results, with a well-readable output, that only need slight adjustments to be used as ex-tasks.

10.2. Future Work

To end, we discuss some ideas for future work for further improving the framework developed in this thesis.

Extending Generation Configuration

We discussed certain problems with the current generation method that could be solved by extending the generator. It might be useful to add a fifth property specifying how many special dependencies of a random type should be generated in addition to the base number of special dependencies. One could also introduce a limit for the in-degree and the out-degree to avoid work packages with high in- or out-degrees. Furthermore, the duration assignment is currently completely random, which could be extended to more systematically assign durations, to make it more difficult to directly identify which dependency yields the minimum or maximum values needed for the forward pass and backward pass.

Implementation of an Evaluation Service

The main focus of our prototype was to reduce the time needed to create ex-tasks. However, not only the creation but also the evaluation of such tasks is time-consuming if one does not want to punish students for cascading errors. At the moment, in case of a wrong value, one has to manually calculate the parts affected by this error since the following calculations could be correct using the wrong value. Therefore, the evaluation process can be time-consuming depending on the number of students who handed in a solution containing a wrong value. In addition, manually calculating with the wrong values makes the evaluation process itself error-prone.

This problem could be addressed by including an evaluation service into the microservice architecture that takes an AND as input and automatically evaluates it by considering cascading errors. In doing so, one must introduce a proper evaluation metric or a proper output that visualizes the effect of errors. Ren et al. [RZC20; RZJ21] have developed such an automated grading approach for AON networks by using Excel as submission format and Excel formulas to grade them. However, their AON networks only contain FS dependencies; thus, one must check whether their approach is applicable for the lecture's ANDs.

Implementation of an User Interface

Currently, the generation and calculation of ANDs, and the operations for the profile configurations are only accessible using a command line interface. Therefore, extending the microservice architecture with a UI that provides suitable methods to access the functionalities is reasonable. In doing so, one might also include a more convenient graph modeling tool that can be directly included in the UI.

A. Appendix

```
{
  "profileName": "defaultProfileConfig",
  "generationConfig": {
    "minNumberOfLayers": 5,
    "maxNumberOfLayers": 6,
    "minNumberOfWorkPackages": 11,
    "maxNumberOfWorkPackages": 13,
    "minSizeFirstLayer": 2,
    "maxSizeFirstLayer": 3,
    "minSizeIntermediateLayers": 1,
    "maxSizeIntermediateLayers": 3,
    "minSizeLastLayer": 1,
    "maxSizeLastLayer": 2,
    "minDuration": 1,
    "maxDuration": 5,
    "jumpDistribution": [
      0.8,
      0.2
    ],
    "numberOfFinishToFinish": 1,
    "numberOfStartToFinish": 1,
    "numberOfStartToStart": 1,
    "applyPostProcessing": true
  },
  "transformationConfig": {
    "criticalPathColor": {
      "red": 248,
      "green": 206,
      "blue": 204
    },
    "workPackageWidth": 100,
    "workPackageHeight": 60,
    "startEndWidth": 80,
    "startEndHeight": 30,
    "hasRoundedEdges": true,
    "hasLineBreakSeparator": true,
    "xzero": 20,
    "xspacing": 70,
    "yzero": 500,
    "yspacing": 30
  }
}
```

Source Code A.1: Default profile configuration used to generate ex-tasks

```
{
  "profileName": "largeANDProfileConfiguration",
  "generationConfig": {
    "minNumberOfLayers": 10,
    "maxNumberOfLayers": 12,
    "minNumberOfWorkPackages": 0,
    "maxNumberOfWorkPackages": 0,
    "minSizeFirstLayer": 8,
    "maxSizeFirstLayer": 10,
    "minSizeIntermediateLayers": 7,
    "maxSizeIntermediateLayers": 12,
    "minSizeLastLayer": 5,
    "maxSizeLastLayer": 7,
    "minDuration": 1,
    "maxDuration": 6,
    "jumpDistribution": [
      0.82,
      0.18
    ],
    "numberOfFinishToFinish": 5,
    "numberOfStartToFinish": 5,
    "numberOfStartToStart": 5,
    "applyPostProcessing": true
  },
  "transformationConfig": {
    "criticalPathColor": {
      "red": 248,
      "green": 206,
      "blue": 204
    },
    "workPackageWidth": 100,
    "workPackageHeight": 60,
    "startEndWidth": 80,
    "startEndHeight": 30,
    "hasRoundedEdges": true,
    "hasLineBreakSeparator": true,
    "xspacing": 80,
    "yzero": 500,
    "yspacing": 30,
    "xzero": 20
  }
}
```

Source Code A.2: Profile configuration that generates large activity network diagrams

Bibliography

[AEH96] M. Agrawal, S. E. Elmaghraby, and W. S. Herroelen. "DAGEN: A generator of testsets for project activity nets." In: *European journal of operational research* 90.2 (1996), pp. 376–382. DOI: 10.1016/0377-2217(95)00361-4 (cit. on p. 14).

[Ant+07] G. A. Anton et al. "Optimizing the graphical arrangement of network construction schedules." In: *Proceedings of the Construction Research Congress, eds. PS Chinowsky, AD Songer, PM Carrillo. Reston, Virginia: American Society of Civil Engineers*. Citeseer. 2007 (cit. on pp. 29, 31).

[BM01] O. Bastert and C. Matuszewski. "Layered drawings of digraphs." In: *Drawing graphs*. Springer, 2001, pp. 87–120 (cit. on pp. 15–18).

[Cor+10] D. Cordeiro et al. "Random graph generation for scheduling simulations." In: *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. Malaga, Spain: ICST, Mar. 2010, p. 10 (cit. on p. 15).

[CSH19] L.-C. Canon, M. E. Sayah, and P.-C. Héam. "A Comparison of Random Task Graph Generation Methods for Scheduling Problems." In: *Euro-Par 2019: Parallel Processing*. Ed. by R. Yahyapour. Cham: Springer International Publishing, 2019, pp. 61–73. DOI: 10.48550/arXiv.1902.05808 (cit. on p. 15).

[Deo17] N. Deo. *Graph theory with applications to engineering and computer science*. Courier Dover Publications, 2017. ISBN: 9780486807935 (cit. on p. 3).

[DRW98] R. Dick, D. Rhodes, and W. Wolf. "TGFF: task graphs for free." In: *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE'98)*. 1998, pp. 97–101. DOI: 10.1109/HSC.1998.666245 (cit. on pp. 13, 14).

[DVH03] E. Demeulemeester, M. Vanhoucke, and W. Herroelen. "RanGen: A Random Network Generator for Activity-on-the-Node Networks." In: *Journal of scheduling* 6.1 (2003), pp. 17–38. DOI: 10.1023/A:1022283403119 (cit. on p. 14).

[ESK05] M. Eiglsperger, M. Siebenhaller, and M. Kaufmann. "An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing." In: *Journal of Graph Algorithms and Applications* (2005), pp. 305–325. DOI: 10.7155/jgaa.00111 (cit. on p. 18).

[EW94] P. Eades and N. C. Wormald. "Edge crossings in drawings of bipartite graphs." In: *Algorithmica* 11.4 (1994), pp. 379–403. DOI: 10.1007/BF01187020 (cit. on pp. 16, 18).

[EX89] P. Eades and L. Xuemin. "How to draw a directed graph." In: *1989 IEEE Workshop on Visual Languages*. IEEE Computer Society, 1989, pp. 13–17. DOI: 10.1109/WVL.1989.77035 (cit. on p. 16).

[Gam+95] E. Gamma et al. "Elements of Reusable Object-Oriented Software." In: *Design Patterns. massachusetts: Addison-Wesley Publishing Company* (1995) (cit. on pp. 55, 56).

[GJ83] M. R. Garey and D. S. Johnson. "Crossing Number is NP-Complete." In: *SIAM Journal on Algebraic Discrete Methods* 4.3 (1983), pp. 312–316. DOI: 10.1137/0604033 (cit. on p. 17).

[Han10] E. Hanser. *Agile Prozesse: Von XP über Scrum bis MAP*. Springer-Verlag, 2010. ISBN: 978-3642123122 (cit. on p. 9).

[Hea16] J. Heagney. *Fundamentals of project management*. Amacom, 2016. ISBN: 9780814437360 (cit. on pp. 4–7).

[HL05] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research, 9th*. Ninth. 2005 (cit. on pp. 4, 7).

[HN02] P. Healy and N. S. Nikolov. "How to Layer a Directed Acyclic Graph." In: *Graph Drawing*. Ed. by P. Mutzel, M. Jünger, and S. Leipert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 16–30. DOI: 10.1007/3-540-45848-4_2 (cit. on pp. 3, 16).

[JM96] M. Jünger and P. Mutzel. "Exact and heuristic algorithms for 2-layer straight-line crossing minimization." In: *Graph Drawing*. Ed. by F. J. Brandenburg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 337–348. DOI: 10.1142/9789812777638_0001 (cit. on p. 17).

[Lar+18] X. Larrucea et al. "Microservices." In: *IEEE Software* 35.3 (2018), pp. 96–100. DOI: 10.1109/MS.2018.2141030 (cit. on p. 8).

[LG96] K. G. Lockyer and J. Gordon. *Project management and project network techniques*. Sixth. Pearson Education, 1996. ISBN: 9780273614548 (cit. on pp. 4–7).

[LL13] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013. ISBN: 9783864900921 (cit. on p. 53).

[LMV97] M. Laguna, R. Martí, and V. Valls. "Arc crossing minimization in hierarchical digraphs with tabu search." In: *Computers & operations research* 24.12 (1997), pp. 1175–1186. DOI: 10.1016/S0305-0548(96)00083-4 (cit. on p. 19).

[ML03] R. Martí and M. Laguna. "Heuristics and meta-heuristics for 2-layer straight line crossing minimization." In: *Discrete Applied Mathematics* 127.3 (2003), pp. 665–678. DOI: 10.1016/S0166-218X(02)00397-9 (cit. on p. 56).

[New15] S. Newman. *Building microservices - Designing Fine-Grained Systems*. OReilly Media, Inc., 2015. ISBN: 9781491950357 (cit. on p. 8).

[Pat04] P. Patarasuk. "Crossing reduction for layered hierarchical graph drawing." In: (2004) (cit. on p. 35).

[Ric19] C. Richardson. *Microservice Patterns. With Examples in Java*. 2019 (cit. on pp. 8, 9, 57).

[RZC20] R. Ren, J. Zhang, and Y. Chen. "An Automated Grading Method for Activity-on-Node Calculations to Support Construction Management Education." In: *Construction Research Congress 2020*. 2020, pp. 733–742. DOI: 10.1061/9780784482872.080 (cit. on p. 78).

[RZJ21] R. Ren, J. Zhang, and Y. Jiang. "New Automated Activity-on-Node Calculation Grading Method for Construction Management Education Innovation." In: *Journal of Civil Engineering Education* 147.3 (2021). DOI: 10.1061/(ASCE)EI.2643-9115.0000043 (cit. on p. 78).

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. "Methods for Visual Understanding of Hierarchical System Structures." In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (1981), pp. 109–125. DOI: 10.1109/TSMC.1981.4308636 (cit. on p. 16).

[SW21] J. Shore and S. Warden. *The art of agile development*. O'Reilly Media, Inc., 2021. ISBN: 9781492080695 (cit. on p. 9).

[TC08] H. Tang and S. Chen. "Research on Layering Algorithm of DAG." In: *2008 International Conference on Computer Science and Software Engineering*. Vol. 2. 2008, pp. 271–274. DOI: 10.1109/CSSE.2008.803 (cit. on p. 16).

[Val08] K. Vallerio. "Task graphs for free (tgff v3. 0)." In: (2008) (cit. on p. 13).

[WG04] W. L. Winston and J. B. Goldberg. *Operations Research: Applications and Algorithms*. Vol. 3. Thomson Brooks/Cole Belmont, 2004. ISBN: 978-0534520205 (cit. on p. 4).