# Automating the creation of production chains in Factorio

Adam Bruun Faber - PKJ312
*DIKU, Department of Computer Science,*
*University of Copenhagen, Denmark*

June 9, 2024

# Abstract

A working implementation to automatically creating production chains in Factorio is demonstrated that interacts with the video game through its blueprint system. Techniques to design integrated circuits were applied in a similar matter when creating and designing the production chains via two abstract models, the N-ary tree and Non-deterministic Finite Automaton (NFA). The implementation was developed using the Factorio-Draftsman library for Python to provide a useful API to communicate with the blueprint system. The results that were generated by the implementation demonstrate how the production chain is first constructed using the abstract N-ary tree model, and then using that tree data structure to create an ingame variant. Lastly, a discussion goes through some of the benefits and drawbacks of using either the N-ary tree and NFA model, and certain case studies where one model might be more suitable than the other.

# Acknowledgements

I would like to express my greatest gratitude my supervisor David Gray Marchant for his exceptional guidance throughout this project, as well as great and fruitful discussions and sparrings regarding the development of this project.

# Contents

# 1 Introduction

Designing integrated circuits is a problem in and of itself, in which numerous techniques and strategies to aid in the design process of integrated circuits have been proposed. This project sets out to study how such techniques can be applied in a similar matter to a similar problem, namely automating the creation of production chains of Factorio programmatically. Any sort of hardware-system or pipeline could be expressed as a connected graph of processes, where each process is responsible for some part of producing the final output. By using a small subset of the available factory components in Factorio, each hardware component in a simple circuit can be mapped to an equivalent factory component in the video game, which means that a production chain can similarly be described as a connected graph of factory components, where each component is responsible for some small task in the factory. Akin to using a process-graph as an abstract model for describing the hardware-system, two abstract models, N-ary trees and Non-deterministic Finite Automaton (NFA), are introduced in this project. Each of these models present different ways of describing a production chain, and such differences are investigated.

# 2 Background

## 2.1 Problem statement

Automating the production chains in Factorio is an exercise in hardware design. More specifically, it is an exercise in how designing integrated circuits can be done automatically and if it is feasible to apply similar techniques to Factorio. By the phrase "be done automatically", I mean the ability of a program to automatically determine and select what parts to include in the integrated circuit, which is called a 'production chain' in Factorio.

## 2.2 Motivation

Tools to aid in the design of integrated circuits already exists such as OrCAD[5]. As such, this project does not set out to invent a new technology, but it does propose a solution to the problem of automatically generating production chains in Factorio programatically. Taking inspiration from one

of the subcategories within Electronic Design Automation (EDA)[31], namely Placement[34] and Routing[35] and combining it with my passion for video games is a motivating and interesting subject which I attempt to study in this project. The video game, Factorio, which will be introduced and discussed later on this report, provides the player with buildings that can be mapped to a subset of hardware components, such as logic-gates, busses and more. This gives the author, or however else that wishes to explore hardware design, a fun and interactive way to learn about and design integrated circuits.

## 2.3 Software

### 2.3.1 Programming language

When choosing a programming language, not only was I looking for something that is efficient performance or implementation wise, but also something that I was familiar with. One may consider C as a good language to use, as it is a general-purpose language which makes it feature rich, as opposed to a domain-specific language. It is however a compiled language, which means that for even the smallest changes in the implementation, it will have to compile again. Over a lot of compilations and iterations, this makes it difficult and slower to work with compared to a higher-level language.

A good middleground and what was initially decided on, was C#. It features object-oriented programming, JIT- or dynamic-compilation, and is a language that I am very familiar with via courses throughout the bachelors degree. A primitive version of the program was developed C#, before it was discovered that a library called "Factorio-Draftsman", which will be expanded upon in Section 2.3.3, had been developed for Python, which handles a lot of the boilerplate code to get a working implementation up and running.

Even though Python is not typically attributed with being a fast language performance wise, it was determined by that switching to Python would be a better choice, primarily because implementation of the program becomes easier due to the provided API from "Factorio-Draftsman", but also because generating blueprints is usually a one-and-done procedure, which means that the operation of running the program and how long that it takes to generate a blueprint is not of significant importance.

### 2.3.2 Factorio

Factorio is a video game, where the general goal of the game is to build and maintain factories[19]. The game world is situated in a two-dimensional grid of square tiles where the player is able to research different technologies, expand their factory by combining simple production buildings into more complex production chains, fight enemies and more, but for this research project, the focus point and usage of the game is the ability to convert a JSON string representation[11][12] of a complex series of buildings into a factory of connected production buildings, also called a production chain. This conversion system in Factorio is called the "Blueprint"-system[11]. What buildings in Factorio are and which buildings I will be using in this project is presented in Section 3.2.

Because each process in a process graph consists of a single, isolated process, similar to a building in the production chain, one can map each process/hardware component to its corresponding building in Factorio and thereby converting there entire process graph into a visual representation in Factorio.

### 2.3.3 Factorio-Draftsman

To manipulate and work with the blueprints in Factorio, I incorporate a Python library called "Factorio-Draftsman"[1]. This library sets out to give the user an API to interact with said blueprints in a number of ways, such as querying a specific building in the blueprint by its name, or simply by utilizing the 'getter' of a specific field of a building class instance, e.g. if we have an assembler class instance called "asm", then we can retrieve its tile-width with "asm.tile_width". As such, this library provides a set of programming features and tools that enables me to entirely focus on the core program functionality[1].

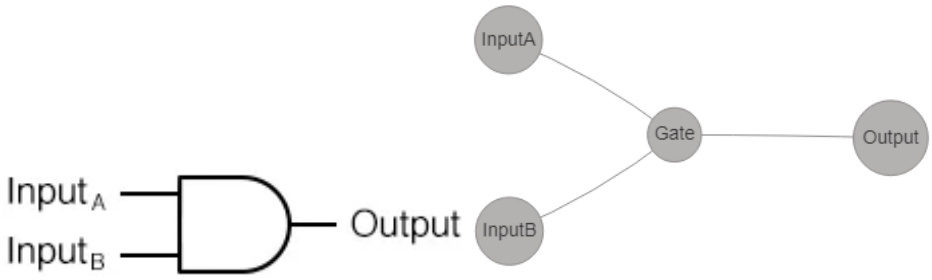
### 2.3.4 Process-graph

A 'process-graph' abstraction has been mentioned previously, and this section will expand on that idea and detail what it actually means.

When a system of hardware components in a series is presented, perhaps as a pipeline, it can be described as a connected graph of processes, where each process is a single hardware component. These processes will either have one or more inputs and an output. As an example, a logical 'AND' gate

takes two inputs and gives one output, if both inputs are active. Another example is the 'NOT' gate that only takes one input and gives one output, being the opposite or negated input. By connecting each of these logical gates with busses, which is a datastream highway that transfers data from one end to another, we can create a connected graph of processes, where the nodes in the graph is a process and an edge is a bus.

I will demonstrate this concept first with a simple example and then a more complex example afterwards:



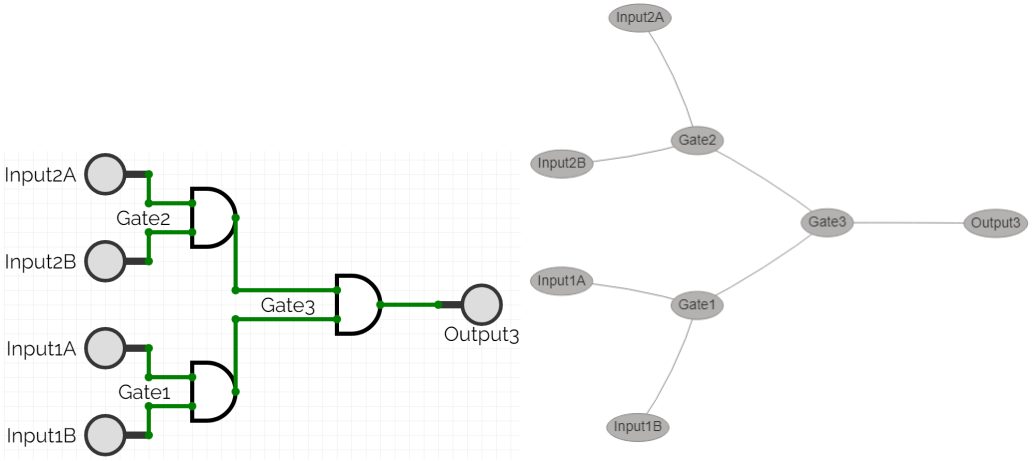

((a)) Logical 'AND' gate

((b)) Connected graph representation of 'AND' gate[29]

Figure 1: AND gate and its corresponding connected process-graph representation

In Figure 1 (a), a single 'AND' gate is shown. It takes two inputs, "InputA" and "InputB" and gives one output, "Output". This simple circuit can be split up into four distinct processes, each corresponding to a node in the graph shown in Figure 1 (b). The busses connecting the inputs to the gate and the gate to the output in Figure 1 (a) are implicitly constructed in Figure 1 (b) as edges connecting the nodes.

Alternatively one may choose to construct the graph where busses would be their own separate nodes, however for the purpose of the method which, will be discussed in Section 3, and is used in this project, this approach is unnecessary and will only clutter the model with redundant nodes, that in any case will simply serve as a connection between two other nodes. I do mention this approach as it is relevant for certain optimizations of the process graph, and will be discussed later on, in section 6




((a)) Circuit of connected logical AND gates[6]

((b)) Connected graph representation of the circuit[29]

Figure 2: Series of hardware components and its corresponding connected process-graph representation

Now that we know how to construct a process-graph for a singular logic gate, an example of chaining multiple of such logic gates together and how a corresponding process-graph looks is shown in Figure 2.

The output from "Gate2" in the logic gate circuit in Figure 2 (a) is fed directly into one of the input slots in "Gate3". This is why, unlike the process-graph in Figure 1 (b) that has an explicit "Output" node, the output nodes from "Gate1" and "Gate2", along with the two input nodes leading into "Gate3" in the process-graph in Figure 2 (b) have been omitted. They would only serve as transmission nodes between two gates and are therefore not included to keep the process-graph concise. The "leaf" input and output nodes, i.e., the nodes that are not connected in between two different gates, are kept in the logical gate circuit and the process-graph to denote that an available input or output slot is present that other circuits could connect to, allowing for a series of different circuits to connect together and work as a single pipeline.

# 3 Method

This section starts by presenting two different methods of turning the abstract process-graph into a data structure that is more familiar in Section 3.1. Then in Section 3.2 a detailed introduction to a subset of buildings a player can use in Factorio to build production chains is presented. Once the building components have been introduced, in Section 3.3 I describe how the data structure from Section 3.1 can be applied to the Factorio buildings from Section 3.2 and Factorios blueprint system, such that circuits of hardware components can be brought ingame.

## 3.1 Modelling the circuit

### 3.1.1 NFAs

As presented in Section 2.3.4, any gate in the process-graph has an output that is either active, meaning it is transmitting or sending out a signal through the output slot, if all input slots are active, or not active if any of the input slots are not active. Depending on what type of logic gate is used, this input-to-output mechanism might be different, such as a 'NOT' gate where the output signal is the inverse of the input signal, but this section will focus on the behavior of the 'AND' gate from now on, as it is most analogous to buildings in Factorio.

This binary behavior of a gate either producing an output or not defines one of two states that the gate can be in:

- If the gate **is** producing an output, it means that it is receiving all of its inputs.
- If the gate **is not** producing an output, it means that it is missing at least one of its inputs.

Because I want to model the process-graph after a Non-deterministic Finite Automaton (NFA)[33], I shall also define a list of 'transitions' that will a process should follow depending on the state it is in:

- **P**: The state-machine follows this transition when all of the inputs at its current state(process) are active
- **R**: The state-machine follows the transition corresponding to the input at its current state(process), that is not active.

Using the process-graph from Figure 2 (b) as an example, every node can then be converted to a corresponding state, and every edge can be converted

to a list of transitions, which creates the following NFA:

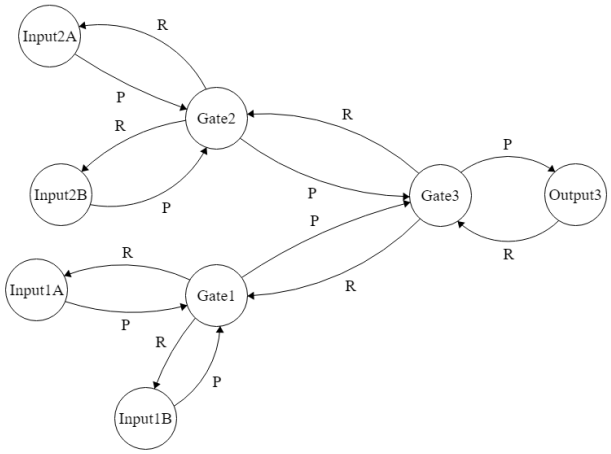



Figure 3: NFA conversion from the process-graph in Figure 2 (b)[30]

### 3.1.2 N-ary trees

The process-graph in Figure 2 (b) already highly resembles a tree data structure because it also uses nodes and edges to describe the relationships between the different nodes, and it follows the same hierarchical structure of having one or more input nodes (the tree equivalent of a 'child') and one output node (the tree equivalent of a 'parent'). The final output node in the process-graph ("Output3" in this case) would then be the root node of the tree.

As a tree typically grows downwards from its root, it has been rotated upright to have the root node highest up in the diagram.

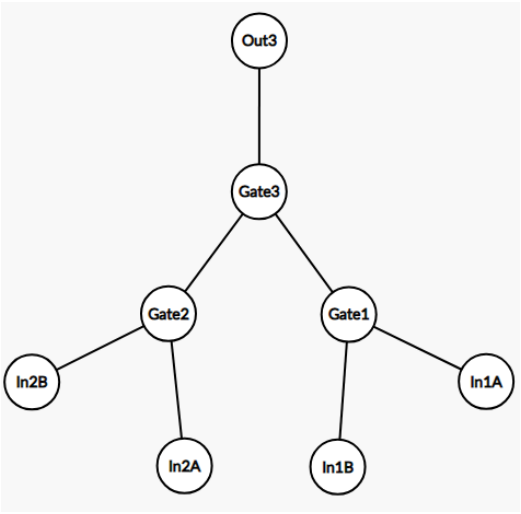



Figure 4: N-ary tree conversion from the process-graph in Figure 2 (b)[4]

Each node in Figure 4 has at most two child nodes, which makes it a binary tree, however in theory each logical 'AND' gate and its corresponding node could have an arbitrary number of inputs. As it is not always guaranteed that a logical gate has a static number of inputs, it is beneficial to use a data structure, such as an N-ary tree[32], that is not limited to such a restriction. In practice, only a discrete amount of hardware space is available, i.e., the finite area in which a hardware component can be placed on a printed circuit board, and because each hardware component such as logical gates or busses take up some discrete amount of space, this model is not a perfect solution, however in the spirit of this project, I am demonstrating how a theoretical hardware circuit can be brought into a video game and not onto a physical device and as such I will look past this theoretical-to-practical discrepancy.

The same can be said about the NFAs in Section 3.1.1, and the reasoning for using that model is the same.

## 3.2 Factorio Buildings

Buildings in Factorio are small simple factory components that all serve as part of a production chain. Some buildings produce items, other buildings transport items or store items in storage containers. Below is a list of the small subset of such buildings and components in Factorio that I will use as representations for the hardware components used in the example circuits

presented earlier in Section 2.3.4, and by extension the 'states' and 'transitions' in Section 3.1.1 and 'nodes' and 'edges' in Section 3.1.2.

- **Furnaces** - Figure 12: A furnace is a building that takes one or more items as input and produces a single output. As can be seen in Figure 12, a furnace has a tilesize of 2 by 2, and unless otherwise specified, every building in Factorio will have all of its orthogonally adjacent tiles as available inputs or outputs, which in this case is 8. Since I want to focus on the logical 'AND' gate as the primary hardware component to model after, a furnace could be anything in between a 1Input- to a 7Input-'AND' gate, with a single tile reserved to the output[20]. In Section 3.1.1 and Section 3.1.2 the furnace will model the state and node respectively.
- **Assemblers** - Figure 13: An assembler is a building that, just like the furnace, takes one or more items as inputs and produces a single output. Depending on what type of recipe the player has selected for the assembler, a specific combination of inputs are required to produce the desired output. As an example, if the player wishes to produce transport belts[26], they will need an input of an iron plate[25] and an iron gear wheel[24]. As can be seen in Figure 13, an assembler has a tilesize of 3 by 3, with all of its orthogonally adjacent tiles as available inputs or outputs, which in the case of the assembler is 12. Similarly to the furnace, I model it after an 'AND' gate, which means that an assembler could be anything between in between a 1Input- to a 11Input-'AND' gate, also with a single tile reserved for the output[9]. In Section 3.1.1 and Section 3.1.2 the assembler will model the state and node respectively.
- **Transport Belts** - Figure 14: A transport belt is a building component that moves up to 15 items per second in a given transport direction. From Figure 14, we can see that a transport belt has a tilesize of 1 by 1, one input and one output. The transport direction, illustrated by the blue arrow, denotes the location of the input and output respectively, and can be changed by turning the transport belt 90 degrees[26]. Just like how a hardware bus transfers data from one hardware component to another, a transport belt will transfer items from one building to another, which makes it a good anagolous representation, and is why I will use it in combination with inserters to model the transitions in Section 3.1.1 and edges in Section 3.1.2.
- **Inserters** - Figure 15 & 16: An inserter is another building component

that can move items in a given transport direction. Multiple different types of inserters exist in Factorio, some that can move a single item at a time while others can move multiple items at a time. Even though an inserter has the ability to select an arbitrary pickup position at any of the surrounding tiles, I will restrict it to be on the same tile as the input which is diametrically opposite of the output[23]. Regular (orange) inserters are used in the example production chains from Section 3.3 and Fast (blue) inserters are used in the test-cases, Section 4. and result, Section 5.

- **Chests** - Figure 17 & 18: A chest is a building that can store items within it. Depending on whether a chest is placed on the input- or the output-tile of an inserter, items can be taken out from the chest or be placed into it with the inserter. I will be using a regular wooden chest[14] to store the produced items of the final output of the production chain, and an infinity chest[22] to provide the basic resources, such as coal[15] or copper ore[17] as initial input to the production chain.

## 3.3 Bringing the model into Factorio

Items that furnaces or assemblers output each have a specific recipe to follow to produce exactly that item. The prerequisite items for such recipes vary greatly, meaning while some recipes only require basic items such as coal or copper ore, others require more complex items, which only come from production buildings such as furnaces or assemblers. If a player wants to produce one copper plate[18], they should provide the circuit one piece of copper ore[17] and one piece of coal[15], and the circuit would thereby consist of a two infinity chests to provide coal and copper ore respectively, one furnace to smelt the copper ore into copper plates and one wooden chest to contain the final output.

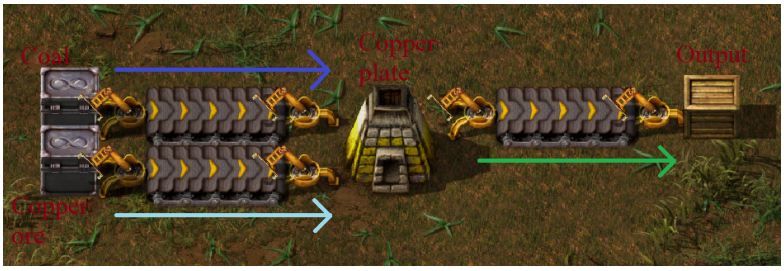
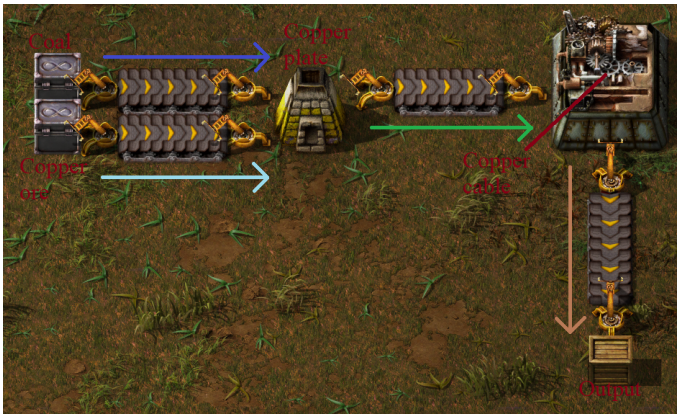



Figure 5: Ingame production chain representation of the example circuit explained above. The circuit above is the defined standalone circuit for producing copper plates, meaning that whenever a more complex building or recipe requires such an input, the above circuit will be built. For such standalone circuits that do not depend on items from other standalone circuits, unlike in Figure 6, the initial item inputs from the infinity chests are included

Suppose the player now wants to produce copper cables[16] with an assembler, which use copper plates as its only prerequisite item. We can then feed the item produced by the production chain shown in Figure 5 directly into the assembler.


((a))




((b)) Figure 5 and Figure 6 (a) circuits combined by chaining them together

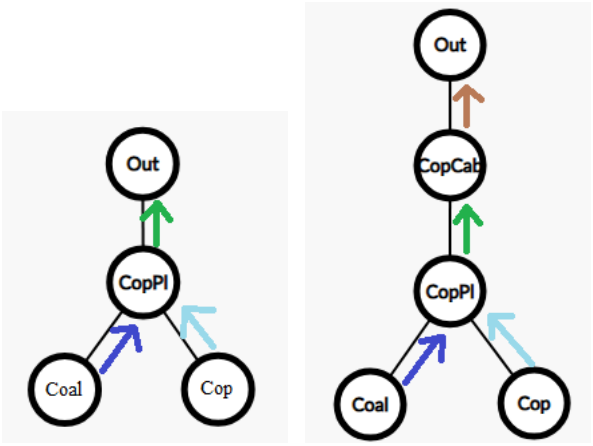
Figure 6: Ingame production chain representation of the example circuit explained above. ((a)) Standalone circuit of components for producing copper cables

By predetermining and defining each of these standalone circuits based

on the output that they produce, along with what prerequisite items they each require for a given recipe, a complete circuit can be constructed just from specifying what the final output/item should be, by recursively walking through each of the prerequisite/dependency branches and building every standalone circuit encountered.

Being able to statically determine which standalone circuits to build, along with the recursive walk through each prerequisite branch being a direct example of a recursive tree traversal, the N-ary tree presented in section 3.1.2 is a fitting model to use. That is not to say that it is strictly a superior model compared to the NFA presented in section 3.1.1, and some reasons for why that is will be discussed later in section 6, however for the purpose of demonstration and convenience, the properties of the N-ary tree will suffice.

The production chains from Figure 5 and 6 can be built from the following N-ary trees in Figure 7 by specifying its root node to be some desired final output item, and then recursively traversing the tree.




((a)) N-ary tree for the copper plate production chain/circuit

((b)) N-ary tree for the copper cable production chain/circuit

Figure 7: N-ary trees for production chains in Figure 5 and 6. For brevity, "Cop" refers to copper ore, "CopPl" refers to copper plate, "CopCab" refers to copper cable and "Out" refers to output. The arrows point in the direction of where the produced items go to, i.e. since copper plates need copper ores, the dark blue arrow in ((a)) points to the copper plate node from the copper ore node

# 4 Implementation and Testing

This section will begin with a short introduction to some of the design decisions for the implementation[3] of the program. Afterwards, a few different test cases are presented to demonstrate that the program produces the correct result.

## 4.1 Implementation

The program is intended to be ran every so often when a player wishes to quickly get a production chain blueprint to produce some ingame resource. This is done through a windows terminal or similar, by inputting the command along with some arguments: "python Main.py <outputAmount> <outputType> <outputRate> <runOrTest>". For an in-depth walkthrough of downloading and running the program, along with a list of all required dependencies, refer to the github repository[3] where the implementation is hosted.

Each command line argument and the intention behind it is listed below in the following list. Note that "python Main.py" is not discussed, as it is the standard method for running the provided file ("Main.py" in this case"). Furthermore, some of these arguments were not fully implemented in the implementation, but are still presented below for brevity and later in Section 6 when discussing future work for this project:

- **<outputAmount>**: This argument specifies how many of the <outputType> the production chain should produce. If the <outputRate> parameter is given, the production of the <outputType> should be based on the throughput of the production chain, rather than the total amount, denoted by <outputAmount>. This command line argument was not fully implemented, so it currently has no effect when provided to the program.
- **<outputType>**: This argument specifies what type of product the production chain should produce. Depending on which product is selected, an N-ary tree will be generated accordingly. This argument is fully implemented, and supports a small subset of the total products that can be produced in Factorio, all of which are listed on the github repository[3].
- **<outputRate>**: This argument specifies whether the production chain should have a specific lower-bound throughput and what that lower-

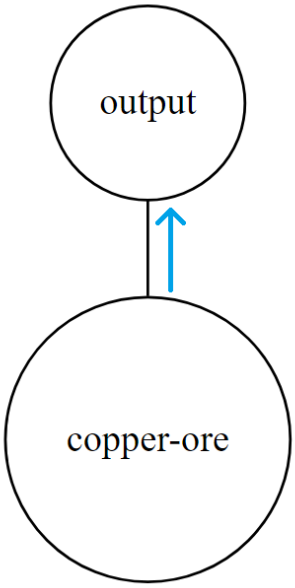
bound should be. Providing extreme throughput rates, such as some <outputType> per nanosecond or similar is infeasible because the Factorio buildings that transport items, namely Transport Belts from Section 3.2, have an upper-bound throughput of 45 items/second, or 2700 items/minute for the fastest transport belt variant[27]. Therefore, the player should provide a reasonable throughput that is within the bounds of the buildings described in Section 3.2. This command line argument was not fully implemented, so it currently has no effect when provided to the program.

- <**runOrTest**>: This argument specifies whether the player wishes to simply generate the production chain, by specifying "-r", or also get an overview of an average generation time over 100 iterations, by specifying "-t". Furthermore, the player can also use "-tverbose" if they wish to get wish to also get an individual generation time for each iteration, potentially to see of there are any major outliers affecting the average generation time. This argument is fully implemented, and supports three different modes of running the program.

## 4.2 Test cases

Based on what I have presented in Section 3 about how the N-ary trees should look and how the buildings in Factorio work and connect with each other, I will show three different test cases in this section, starting with a simple example and getting increasingly more complex, with more nodes in the tree. All of the test cases have been generated by the implemented program, by querying the desired final output, denoted by the <outputType> in each case header.

### 4.2.1 Case 1 - Copper ore

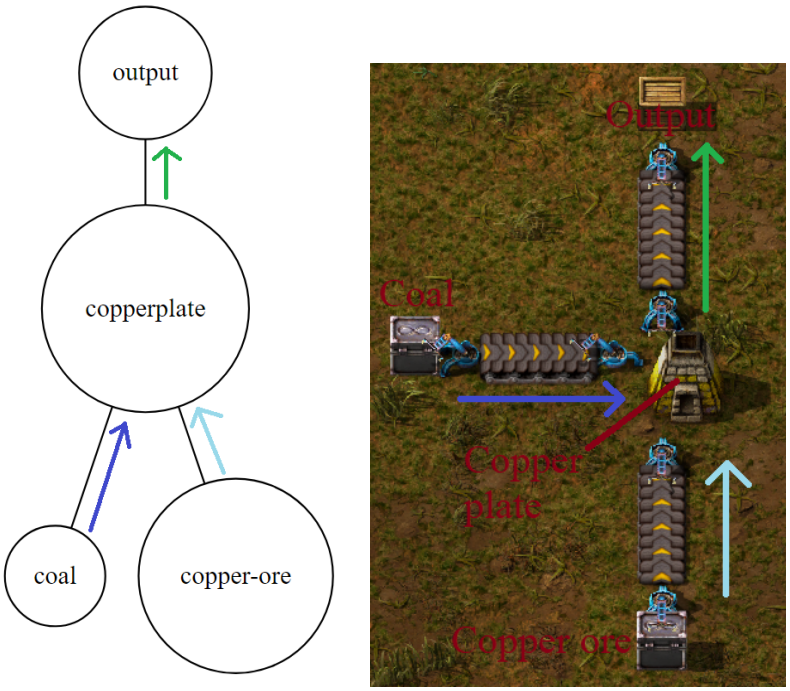



((a)) N-ary tree for the copper ore production chain


((b)) Ingame copper ore production chain

Figure 8: Simple production chain that only has a single input node that feeds directly into the final output node

### 4.2.2 Case 2 - Copper plate

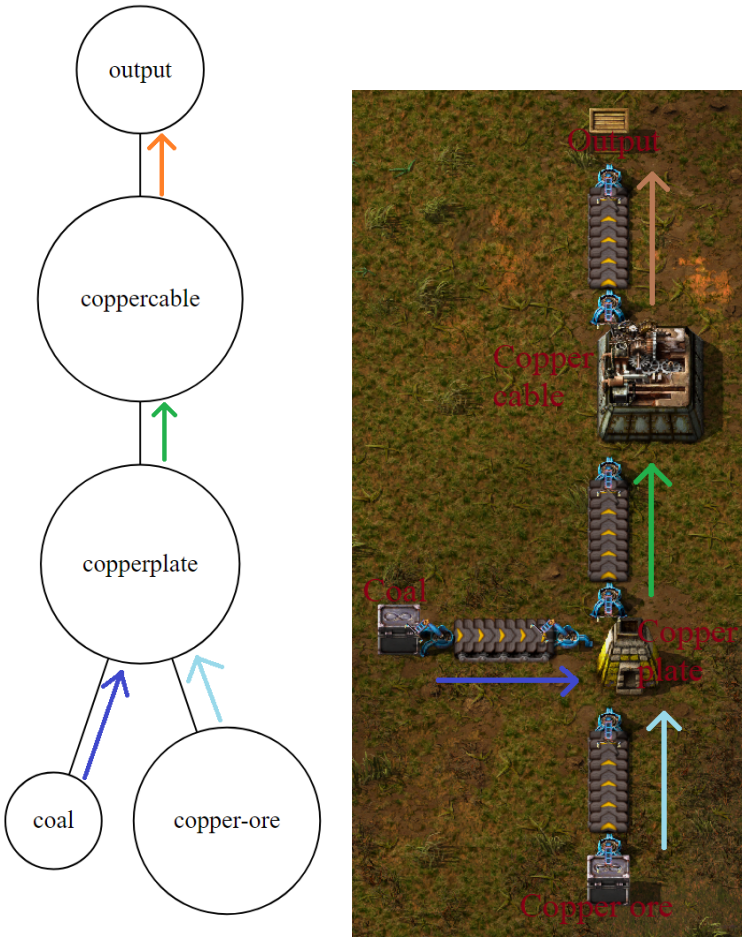



((a)) N-ary tree for the copper plate production chain

((b)) Ingame copper plate production chain

Figure 9: A more complex production chain. This was an attempt to replicate the N-ary tree from Figure 7 (a), by only querying only the output type of the production chain, which was successful as can be seen in Figure 9 (a). Although some buildings are not facing the same direction, the corresponding ingame production chain in Figure 5, also matches the newly generated production chain in Figure 9 (b) with respect to the output that it produces

### 4.2.3 Case 3 - Copper cable




((a)) N-ary tree for the copper cable production chain

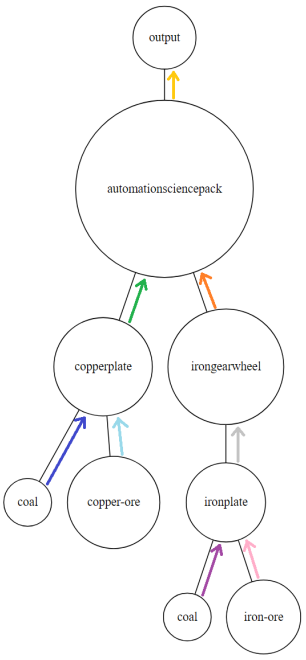
((b)) Ingame copper cable production chain

Figure 10: An even more complex production chain. This was similarly an attempt to replicate the N-ary tree from Figure 7 (b), by only querying only the output type of the production chain, which was successful as can be seen in Figure 10 (a). Although some buildings are not facing the same direction, the corresponding ingame production chain in Figure 6, also matches the newly generated production chain in Figure 10 (b) with respect to the output that it produces

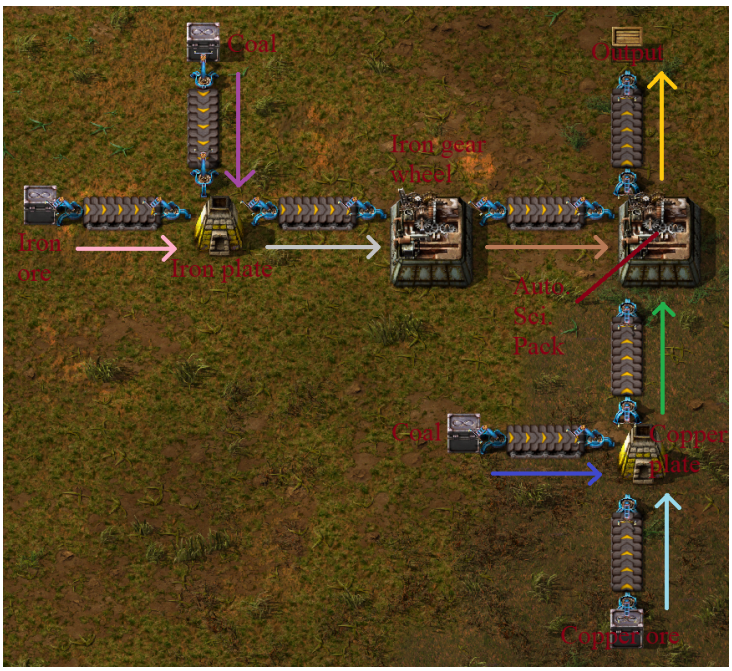
As demonstrated, the program is capable of generating the entire production chain, so long as each standalone circuit has been defined. In other words, the program is open for extension, which means that more and more standalone circuits could be added to the program and it would still be able to generate the production chain.

# 5 Results

From the previous section, we saw that the program could successfully generate the correct production chain, based only from the N-ary tree provided. Furthermore N-ary trees were correctly generated, by only querying the final output of the production chain that the player wanted. This section will conclude on both the Implementation and Testing Section, and the Results section by presenting the largest and most complex production chain supported by the program, as well as some benchmarks for all four production chains generated in Section 4 & 5.




((a)) N-ary tree for the automation science pack production chain, see Figure 19 for a larger figure




((b)) Ingame automation science pack production chain

Figure 11

By querying the automation science pack[10] output type, the N-ary tree and corresponding production chain shown in Figure 11 is produced. The observant reader might have also noticed that each subtree connected to the "automationsciencepack" node in Figure 11 (a) are themselves full N-ary trees that were presented earlier in Section 4, Figure 9 & 10 and the same can be said for the ingame production chains. Adding more output types that takes the automation science pack as an input would not a problem, as the entire N-ary tree would simply become a subtree for the full N-ary tree of the newly added output type.

| Output type | Runtime Benchmark |
|---|---|
| Copper ore[17] | 7.75 ms |
| Copper plate[18] | 12.60 ms |
| Copper cable[16] | 13.98 ms |
| Automation science pack[10] | 24.91 ms |

Table 1

As Table 1 suggests and as expected, the generation runtime sum of all individual production chains used is roughly equal to the generation runtime of the final automation science pack production chain. Generating an even bigger N-ary tree and production chain should therefore yield a runtime generation that is also approximately equal to the runtime generation of each of its subtrees and individual production chains. Table 1 also demonstrates that even though I switched programming langauge from C# to Python, the runtime of generating a production chain is still very reasonable.

# 6 Discussion and Future work

The results presented in Section 5 show that the program can generate a given production chain in only a couple of milliseconds and that an increasingly larger N-ary tree could be constructed, if more and more standalone circuits were added to the program. It was also argued that with the current implementation, a corresponding ingame production chain could be constructed. Furthermore, each production chain presented in Section 4 and 5 have been closely inspected and observed to be correct in the sense, that they produce the correct output. There are however some limitations with the current implementation, as well as Factorio itself, that limit how large the ingame production chain can grow. Unlike how one can keep on adding more and more nodes to the N-ary tree without precaution, the blueprint system in Factorio has a size limit of 10000 by 10000 tiles[11]. Even though nearly all factories in Factorio will never reach such an absurdly large size, it means that for a sufficiently large N-ary trees, it is impossible to generate the corresponding ingame production chain. One may then raise the question of how optimal has the production chain been generated or packed into the finite-sized blueprint grid with respect to the total area or amount of tiles it is spanning over? Such a problem can be described as a packing problem

and more concretely as a 'two-dimensional orthogonal packing problem'[7], because all buildings in Factorio are rectangular shapes and are required to be placed orthogonally on the tilegrid. Said problem has already been studied to some degree in Factorio in Patterson et al.[8], and could be combined, or at least inspirationally be used, with this project as a potential area of future work.

As mentioned in Section 4.1 some of the arguments that the player should provide to the program were not implemented. More concretely, the <outputAmount> and <outputRate> which describe how many of the desired output the player wants, and at what rate the production chain should produce that amount. Being able to measure and ultimately decide how many of a particular item the factory produces would be a welcome addition, as it would give the player even more control over the production chain that the implementation generates. Before implementing the ability to denote the throughput of the generated factory, one should consider the following cases first:

## 6.1 Case Studies

### 6.1.1 Sufficient throughput already

The current implementation will not selectively pick what variant of a building to choose, such as if it should use a normal or fast inserter, but rather already has a selection of such buildings predefined. Just like with all other Factorio buildings, these predefined buildings each have a specific maximum throughput and it might therefore be entirely possible that the desired throughput, which a player would provide the program, is already satisfied in the generated production chain, by being within the throughput bounds. It might be that the player would want a throughput that was lower than the maximum possible throughput of the entire production chain, in which case the program could then just limit how much of basic resources such as coal or copper ore it receives.

### 6.1.2 Multiple of the same input sub tree

Note that the current implementation only uses one input tile for each prerequisite item in the recipe, set for a particular production building. For certain cases this is sufficient to continuously provide enough items for the

production building to consistently produce its output and never sit idle. For other cases the production building may, as an example, need 3 times as many copper ores as it needs coal, which means that for the current implementation the production building can not consistently produce its output. The input that provides the copper ore will have to work 3 times as long as the coal input and in the mean time the coal input will sit idle until it can input another piece of coal. To accommodate for this bottleneck, the program could be improved to use additional input tiles such that for every input of coal, 3 separate copper-ore inputs each input one copper-ore. This of course assumes that there are available input tiles to begin with. See Appendix, Figure 22 for an example of how that could look.

### 6.1.3 Filling up all input tiles and still needing more input

Suppose a production chain has an furnace, with one input tile and one output tile available and all other surrounding input/output tiles are occupied with other production buildings or similar, like in Appendix, Figure 20. Then imagine a scenario similar to that in Section 6.1.2 where the furnace needs 3 times as many copper ore as coal.

The program can not just add an additional input sub tree, because there are now not enough available input tiles. Looking back at Section 2.3.4, it was briefly mentioned that for certain cases, it could be beneficial to represent the busses or transport belt as their own separate node in the N-ary tree, and this is one of such cases. Without such a transport belt node, the program cannot generate the production chain, as can be seen in Appendix, Figure 20, however by adding the transport belt as a separate node, each input sub tree can be connected to that instead, which in turn then transports all of the items to the production building. An example of how such an N-ary tree and corresponding production chain could look like is shown in Appendix, Figure 21

### 6.1.4 Exceeding the maximum throughput of a singular production building

As indicated in Section 6.1.1 each production building has an upper limit for how many items it can physically produce per unit of time. As an example, a stone furnace can produce 0.3125 copper plates per second[21]. Now suppose that a player wants 1 copper plate per second, and the stone furnace is the

only furnace variant the program can utilize. No matter how many copper ores is supplied to a singular furnace, it can never satisfy the 1 copper plate per second throughput, so the program would have to generate additional stone furnaces in the resulting production chain to accommodate. In this case, a total of 4 stone furnaces, each with their own inputs of coal and copper-ore, is needed to satisfy 1 copper plate per second, which is shown in Appendix, Figure 23

## 6.2 Static and dynamic routing of transport belts

Another interesting aspect of generating the production chains is how the routing between production buildings is done. Currently for every standalone circuit/production chain, a singular straight segment of transport belts is placed in one of the four cardinal directions, leading to the next production building. As demonstrated in Section 4 & 5, this strategy works fine, as long as the whole production chain is relatively simple, but it does not allow for placing a production building in an arbitrary location within the blueprint. The range of freedom as well as the capability of optimizing the spanning area of the factory is greatly reduced if the routing is done statically.

Now suppose we have a more complex factory, where the route of transport belts between a furnace and an assembler is missing, as shown in Appendix, Figure 24. With the previously described static routing, it is impossible to find a connecting route, between the two production buildings shown with red squares. I thereby propose another area of future work, where a pathfinding algorithm would be a suitable and effective solution to solving this problem. With a pathfinding algorithm, a route such as the one highlighted with a green arrow in Figure 24 (b), could be found.

## 6.3 Non-deterministic item production

Up until now, all buildings in Factorio that have been presented so far, have been able to produce guaranteed deterministic amounts of items. This was one of the arguments for choosing the N-ary tree model in Section 3.3, however as I shall discuss shortly this is not always the case, which means that for certain buildings in Factorio, an N-ary tree will not be as fitting of a model.

Uranium processing in Factorio is the process of turning uranium ore in a centrifuge[13] into either "uranium-235" or uranium-238"[28]. These two

outcomes are not produced deterministically, but rather based on probability. Producing one uranium-238 has a 99.3% probability of happening, while producing one uranium-235 has a 0.3% probability of happening. Having more than one singular outcome does not coincide with nodes in the N-ary tree, because such nodes only have a single parent node, also known as a single outcome. The same cannot be said for NFAs as they are not confined to such a hierarchical structure. From the NFA shown in Appendix, Figure 25, the program could construct a standalone circuit for producing uranium-235 and uranium-238 respectively, both leading out from the centrifuge.

For brevity, if we still use the N-ary trees to model the production chain, such an N-ary tree will look like shown in Appendix, Figure 26. This N-ary tree may look nothing out of the ordinary, but remember that it should have two distinct outcomes. From this figure, only one of the outcomes, in this case uranium-235, is taken as output, whereas uranium-238 will still be produced, but never moved or used. A heap of uranium-238 will therefore continue to grow inside the finite-sized inventory of the Centrifuge until it can no longer hold anymore items, and the production idles. Furthermore, because the model only represents one outcome and the outcomes are based on probability, depending on which outcomes is querying or used in further production, it may result in having to provide multiple rounds of uranium ore to the centrifuge before it outputs the desired output.

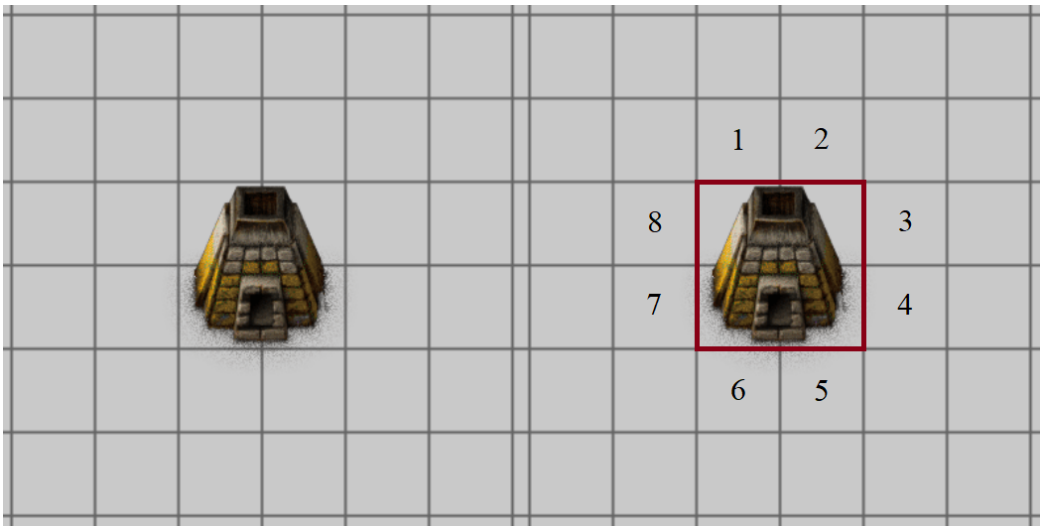
# 7 Conclusion

This project set out to investigate and demonstrate how a working solution to the problem of automatically creating production chains in Factorio could look like. Mappings between hardware components and equivalent production buildings in Factorio are defined, as well as abstract theoretical models for representing integrated circuits and production chains respectively. The project also presents the benefits and drawbacks of both the N-ary tree and the NFA, when using it as a model and data structure to generate the production chains from. By closely observing the different test cases and results demonstrated in the report via checking if the ingame production chain produces the correct result, I can conclude that the proposed theoretical models are suitable and the practical implementation is correct.

By working with this project, I can also conclude that I have gained insight, even if on a basic level, into integrated circuit design and how certain

well known models and data structures can be applied in a reasonable way on problems that resemble that of designing integrated circuits.

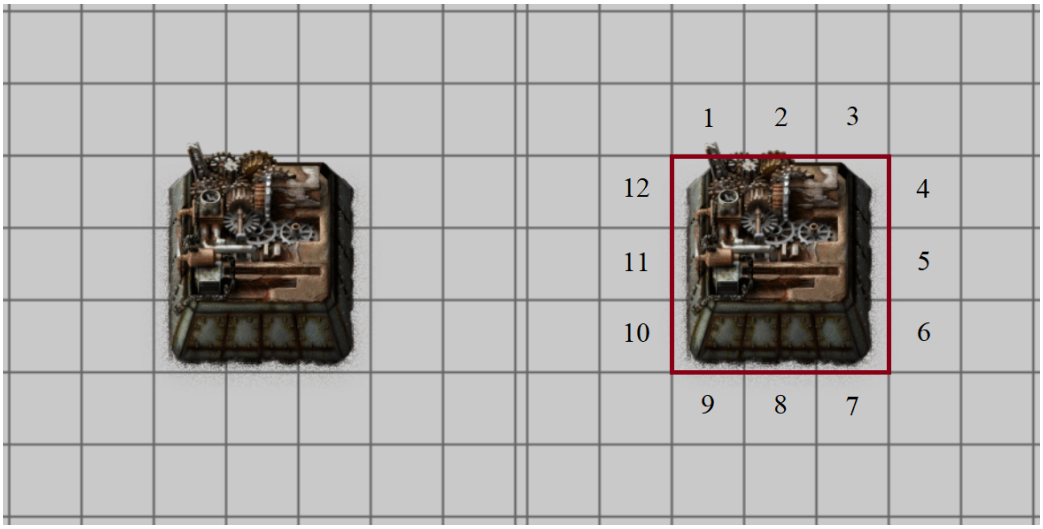
# 8 Appendix




((a)) The furnace building from Factorio

((b)) The furnace building with marked tilesize and inputs/outputs marked with numbers

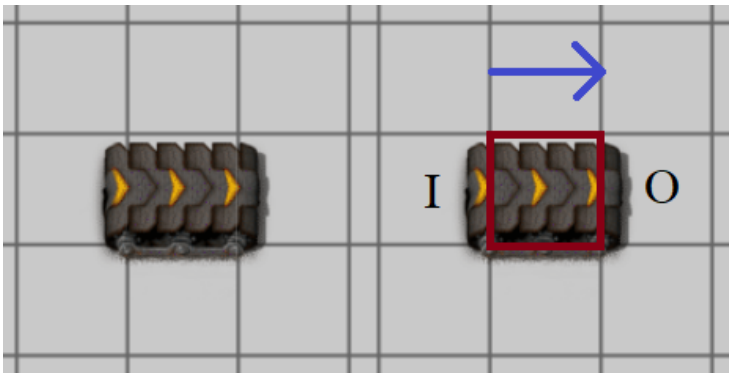
Figure 12: An ingame furnace with an illustration of its tilesize and inputs/outputs[2]




((a)) The assembler building from Factorio

((b)) The assembler building with marked tilesize and inputs/outputs marked with numbers

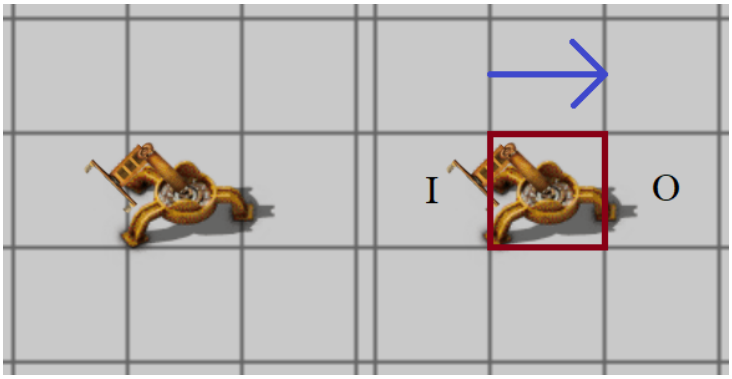
Figure 13: An ingame assembler with an illustration of its tilesize and inputs/outputs[2]




((a)) The transport belt building from Factorio

((b)) The transport belt building with marked tilesize, input/output and transport direction

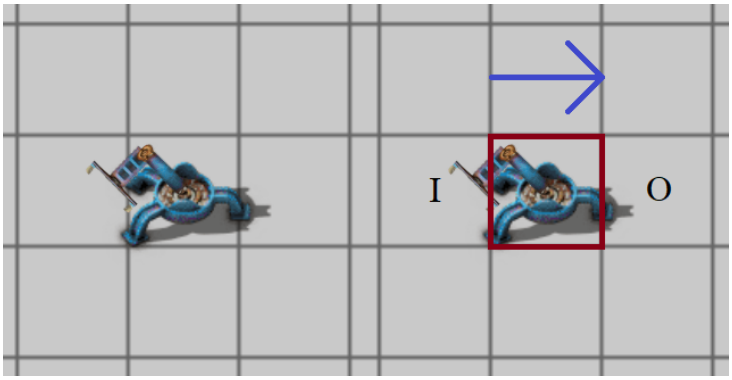
Figure 14: An ingame transport belt with an illustration of its tilesize, input/output and the transport direction[2]




((a)) The inserter building from Factorio

((b)) The inserter building with marked tilesize, input/output and transport direction

Figure 15: An ingame inserter with an illustration of its tilesize, input/output and the transport direction[2]




((a)) The fast inserter building from Factorio

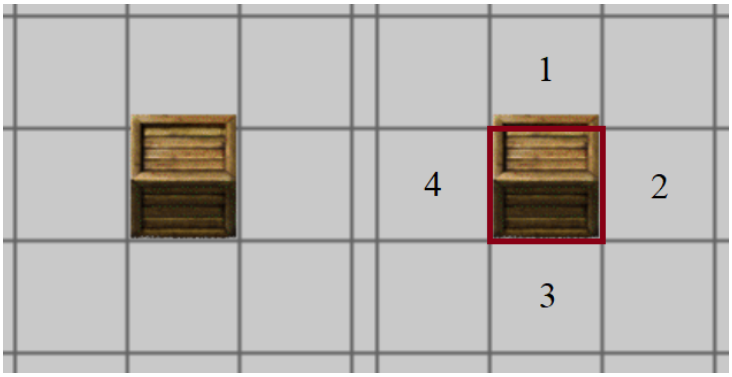
((b)) The fast inserter building with marked tilesize, input/output and transport direction

Figure 16: An ingame fast inserter with an illustration of its tilesize, input/output and the transport direction[2]


((a)) The infinity chest building from Factorio

((b)) The infinity chest building with marked tilesize and outputs

Figure 17: An ingame infinity chest with an illustration of its tilesize and outputs[2]




((a)) The wooden chest building from Factorio

((b)) The wooden chest building with marked tilesize and inputs

Figure 18: An ingame wooden chest with an illustration of its tilesize and inputs[2]

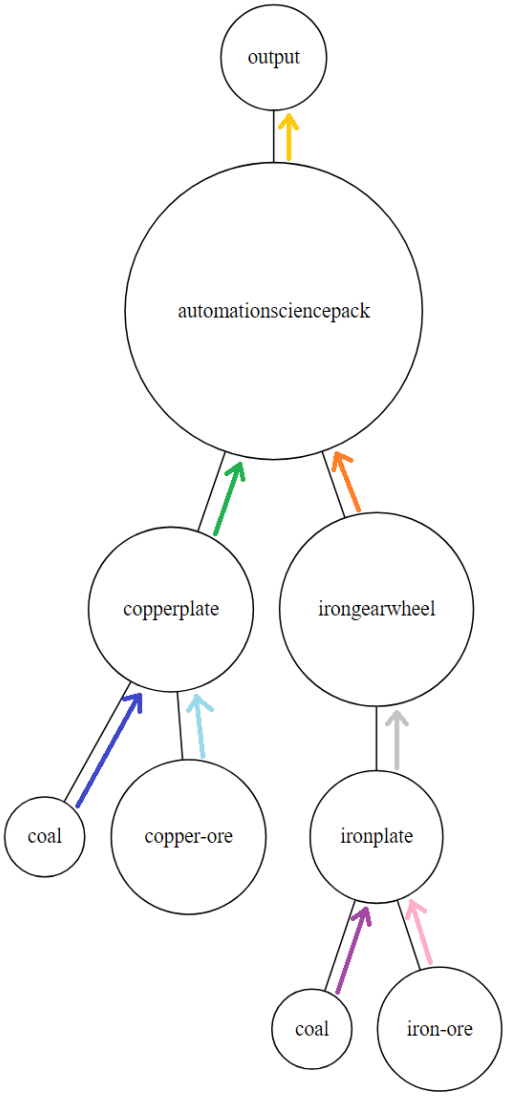
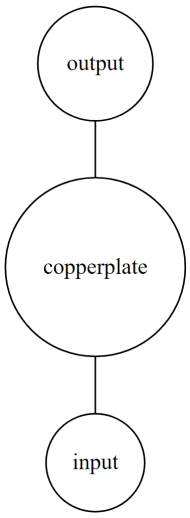
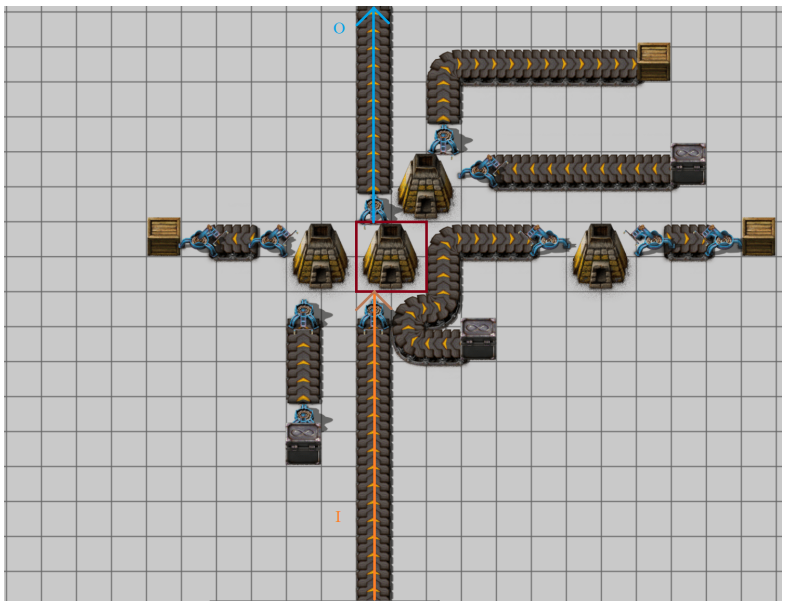



Figure 19: N-ary tree for the automation science pack production chain. Larger figure of the same tree as in Figure 11 (a)

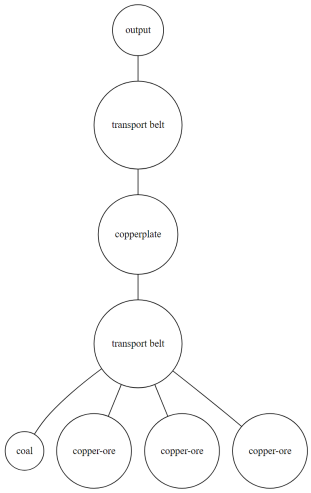



((a)) N-ary tree for the example circuit.

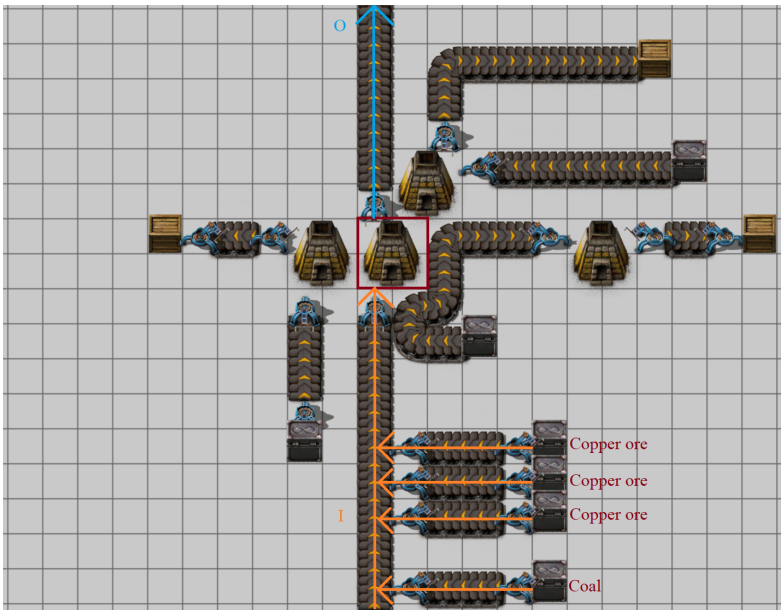



((b)) Ingame example production chain

Figure 20: The N-ary tree and production chain shown in Figure 20 are just example figures that help to understand the concept, but are not actual products of running the program. The orange and blue arrows denote the Input and Output of the production chain, and the red square denotes the production building in question. Note that the current implementation cannot build the production chain corresponding to the example given in the paragraph, so the node "Input" is simply a placeholder node in the N-ary tree

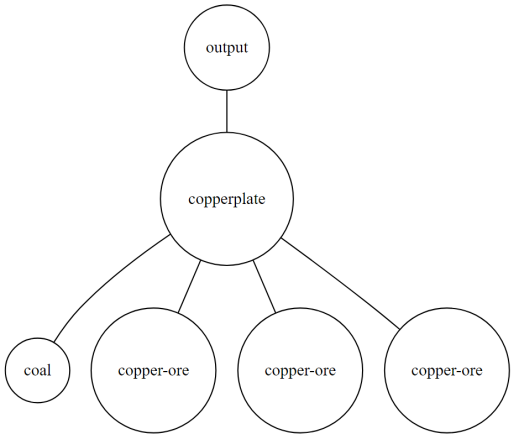



((a)) N-ary tree with new transport belt nodes

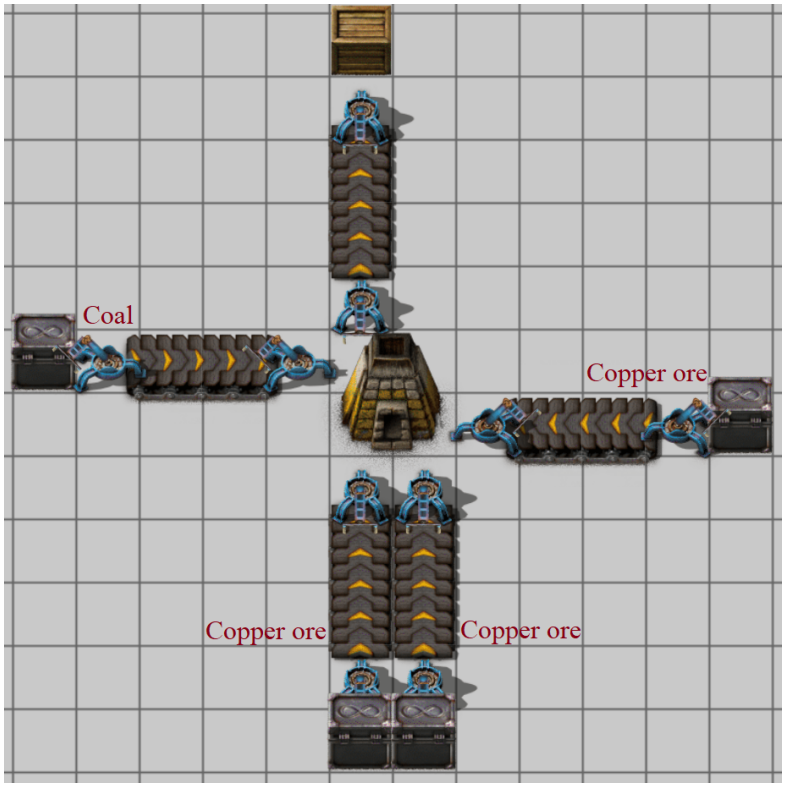



((b)) Ingame example production chain

Figure 21: The N-ary tree and production chain shown in Figure 21 are just example figures that help to understand the concept, but are not actual products of running the program. The orange and blue arrows denote the Input and Output of the production chain, and the red square denotes the production building in question

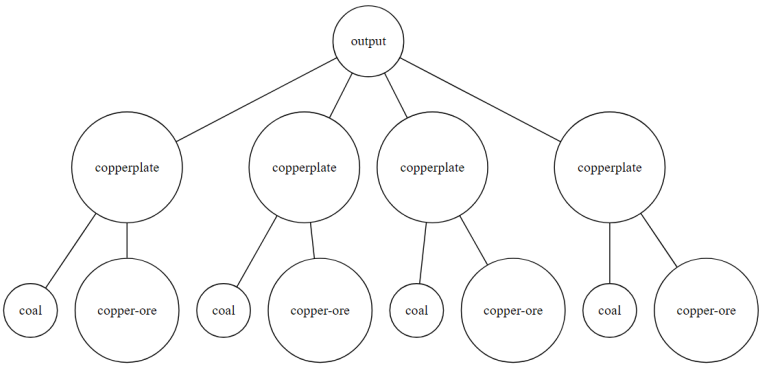



((a)) N-ary tree with multiple of the same input type

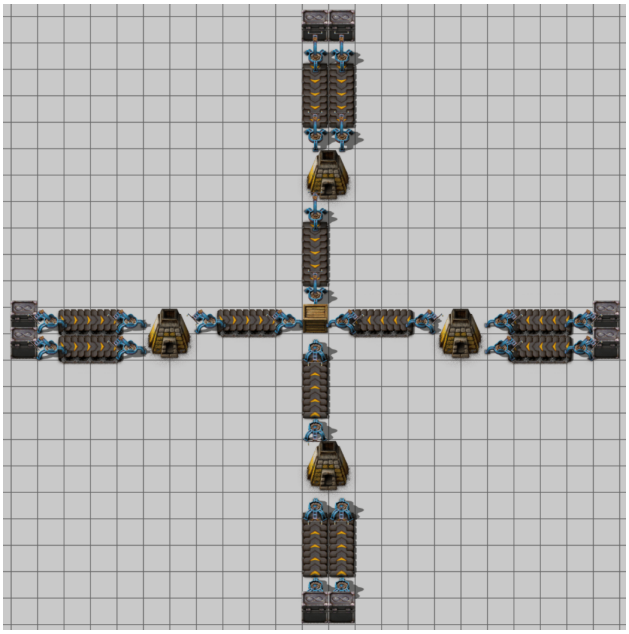



((b)) Ingame example production chain

Figure 22: The N-ary tree and production chain shown in Figure 22 are just example figures that help to understand the concept, but are not actual products of running the program.

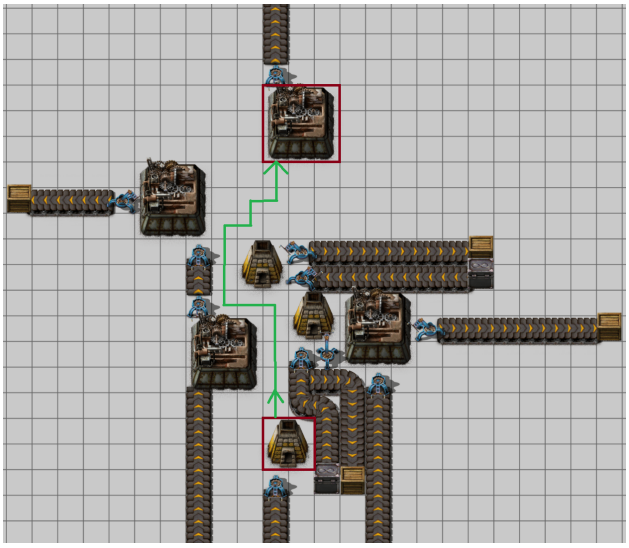



((a)) N-ary tree with multiple of the same input sub tree

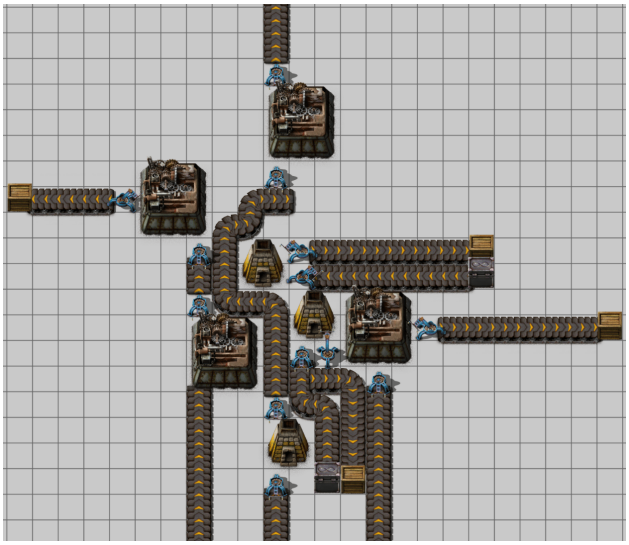


((b)) Ingame example production chain

Figure 23: The N-ary tree and production chain shown in Figure 22 are just example figures that help to understand the concept, but are not actual products of running the program. Each input sub tree produces a copper plate, similarly to 4.2.2



((a)) Complex factory, with a production chain that is missing a transport belt route between two (red-square) buildings



((b)) The same complex factory with a route find via a pathfinding algorithm

Figure 24: The production chain shown in Figure 24 is just an example figure that help to understand the concept, but are not actual products of running the program.

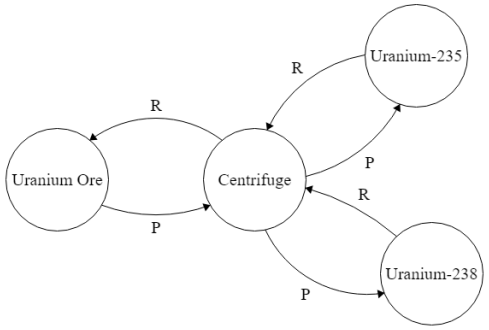



Figure 25: NFA for describing a uranium-235 and uranium-238 production chain

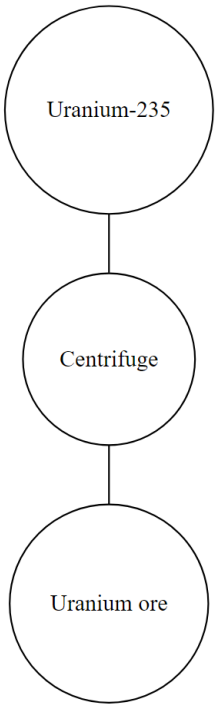



Figure 26: N-ary tree for describing a uranium-235 production

# References


[1] redruin1 (github). *Factorio-draftsman, version 1.1.1*. URL: https://github.com/redruin1/factorio-draftsman and https://factorio-draftsman.readthedocs.io/en/latest/index.html. 2024.

[2] teoxoy (github). *Factorio Blueprint Editor*. URL: https://teoxoy.github.io/factorio-blueprint-editor/ and https://fbe.teoxoy.com/. 2024.

[3] PARRYH0TTEr (myself). *AutoProductionChains$_{P}$ython*. 2024. URL: https://github.com/PARRYH0TTEr/AutoProductionChains_Python.

[4] CS Academy. *Graph Editor*. 2024. URL: https://csacademy.com/app/graph_editor/.

[5] Cadence. *OrCAD X Platform*. 2024. URL: https://www.cadence.com/en_US/home/tools/pcb-design-and-analysis/orcad.html.

[6] CircuitVerse. *Digital Circuit Simulator*. 2024. URL: https://circuitverse.org/simulator.

[7] Andrea Lodi, Silvano Martello, and Michele Monaci. "Two-dimensional packing problems: A survey". In: *European Journal of Operational Research* 141.2 (2002), pp. 241–252. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(02)00123-6. URL: https://www.sciencedirect.com/science/article/pii/S0377221702001236.

[8] Sean Patterson et al. *Towards Automatic Design of Factorio Blueprints*. 2023. arXiv: 2310.01505 [cs.AI].

[9] Wube Software. *Assembling machine 1*. 2022. URL: https://wiki.factorio.com/Assembling_machine_1.

[10] Wube Software. *Automation science pack*. 2019. URL: https://wiki.factorio.com/Automation_science_pack.

[11] Wube Software. *Blueprint*. 2023. URL: https://wiki.factorio.com/Blueprint.

[12] Wube Software. *Blueprint*. 2023. URL: https://wiki.factorio.com/Blueprint_string_format.

[13] Wube Software. *Centrifuge*. 2021. URL: https://wiki.factorio.com/Centrifuge.

[14] Wube Software. *Chests*. 2022. URL: https://wiki.factorio.com/Chests.

[15] Wube Software. *Coal*. 2022. URL: https://wiki.factorio.com/Coal.

[16] Wube Software. *Copper cable*. 2020. URL: https://wiki.factorio.com/Copper_cable.

[17] Wube Software. *Copper ore*. 2020. URL: https://wiki.factorio.com/Copper_ore.

[18] Wube Software. *Copper plate*. 2023. URL: https://wiki.factorio.com/Copper_plate.

[19] Wube Software. *Factorio, version 1.1.107*. 2020. URL: https://store.steampowered.com/app/427520/Factorio/.

[20] Wube Software. *Furnace*. 2020. URL: https://wiki.factorio.com/Furnace.

[21] Wube Software. *Furnace*. 2023. URL: https://wiki.factorio.com/Stone_furnace.

[22] Wube Software. *InfinityContainerPrototype*. 2024. URL: https://lua-api.factorio.com/latest/prototypes/InfinityContainerPrototype.html.

[23] Wube Software. *Inserters*. 2024. URL: https://wiki.factorio.com/Inserters.

[24] Wube Software. *Iron gear wheel*. 2021. URL: https://wiki.factorio.com/Iron_gear_wheel.

[25] Wube Software. *Iron plate*. 2023. URL: https://wiki.factorio.com/Iron_plate.

[26] Wube Software. *Transport belt*. 2022. URL: https://wiki.factorio.com/Transport_belt.

[27] Wube Software. *Transport Belts (physics)*. 2023. URL: https://wiki.factorio.com/Transport_belts/Physics.

[28] Wube software. *Uranium Processing*. 2024. URL: https://wiki.factorio.com/Uranium_processing.

[29] TreeConverter. *Binary Tree and Graph Visualizer*. 2024. URL: https://treeconverter.com/.

[30] Evan Wallace. *Finite State Machine Designer*. 2010. URL: https://www.cs.unc.edu/~otternes/comp455/fsm_designer/.

[31] Wikipedia. *Electronic design automation*. 2024. URL: https://en.wikipedia.org/wiki/Electronic_design_automation. (accessed: 30/05/2024).

[32] Wikipedia. *M-ary tree*. 2024. URL: https://en.wikipedia.org/wiki/M-ary_tree. (accessed: 09/06/2024).

[33] Wikipedia. *Nondeterministic Finite Automaton*. 2024. URL: https://en.wikipedia.org/wiki/Nondeterministic_finite_automaton. (accessed: 09/06/2024).

[34] Wikipedia. *Placement (EDA)*. 2024. URL: https://en.wikipedia.org/wiki/Placement_(electronic_design_automation). (accessed: 30/05/2024).

[35] Wikipedia. *Routing (EDA)*. 2024. URL: https://en.wikipedia.org/wiki/Routing_(electronic_design_automation). (accessed: 30/05/2024).