Master thesis

Rasmus Bartholin

May 2023

husk at checke feedback fra sidste gang

Contents

1 Abstract

2 Introduction

3 Petri nets

Petri nets are a mathematical tool used for modelling many different types of problems. It has been used to analyze many different problems, such as concurrency problems, communication protocols and chemical pathways [7]. A Petri net can be viewed as a graph, with two distinct node types, transitions and places. The definition of a Petri net is:

Definition 1 *Petri net definition: [7]*

\begin{definition}[Petri net \cite{blah}]

A Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$ where:

$P = \{p_1, p_2, ..., p_m\}$ is a finite set of places

$T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions

$F = \in (P \times T) \cup (T \times P)$ is a set of arcs (flow relation)

$W : F \Rightarrow \{1, 2, 3, ...\}$ is a weight function

mærkelig spacing om kolon, hvis et kolon er til definition så brug \colon da : er en binær operator ift. spacing

$M_0 : P \Rightarrow \{0, 1, 2, 3, ...\}$ is a initial marking

ikke brug table

$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$

A Petri net structure $N = (P, T, F, W)$ without any specific initial marking is denoted N and is called a net.

A Petri net with the given initial marking is denoted by $PN = (N, M_0)$.

hov, du har lige defineret det anderledes, evt. start med en net def. og så byg Petri net på den

As the definition states, a Petri net consists of a finite set of places and transitions. The arcs between these node types can have weights ranging from 0 to $+\infty$. The places and transitions have different attributes. A place can contain 0 to $+\infty$ tokens. A token is some sort of abstract resource, that can be moved around the Petri net. This is done by firing transitions. Before we can talk about firing transition, we need a few more definitions:

Definition 2 *Postset and preset definitions [2]:*

For $x \in P \cup T$

$\cdot x := \{y | yFx\}$ is called a preset of x

$x\cdot := \{y | xFy\}$ is called a postset of x

underlig spacing, evt. brug \mid

In plain words, a preset of a transition or place, is the nodes that have arcs going into that place or transition. The postset is the reverse of a preset. It is the nodes that a place or transition have outgoing arcs to. The places and transition have a specific flow relation. A place can only have outgoing and ingoing arcs from transitions. The reverse is true for transitions. They can only have arcs to and from places. An example of a valid and non-valid Petri net, can be seen in figure 1. Figure 1a depicts how Petri nets are usually drawn. The circles are places, and the boxes are transitions. The small dots inside the places are the

stort F da "Figure" er en del af navnet

tokens of that place. The numbers above the edges are the weights. If an edge has a weight of 1, then that edge's weight is often not drawn. This will also be the case in this thesis. Now we are ready to define, what happens when a transition is *fired*.

(a) Valid Petri net (b) Non-valid Petri net

Figure 1: Example of a valid Petri net and one that is not valid. The Petri net that is not valid, has two places with outgoing arcs, connecting to another place. This is not allowed.

When a transition is fired, tokens from the transition's preset are removed, and tokens are added to its postset. How many tokens are taken and put in each place, is determined by the weights of the arcs. For the places in the transition's preset, the weight of the arc (p, t) determines how many tokens are removed from p. This action is often referred as the token from p has been *consumed*. For the places in the transition postset, the weight of the arcs from t to all places p in the postset determines how many tokens is put in those places. This is the reverse of consuming tokens from a place, and is often called *producing* tokens in place p [7]. This means, that the number of tokens that is consumed in a transition's preset, does not have to match the number of tokens produced in its postset when that transition is fired. Whether a transition can be fired or not depends on the number of tokens in each place in the preset of the transition, and the weight of the arcs going from those places to that transition. If every place in a transition preset has as many or more tokens than the weight of its arc to that transition, the transition can be fired and is said to be *enabled*. If a transition can not be fired, it is said to be *disabled* [7].

hvis hele afsnittet er baseret på den ref så skriv en sætning om det i starten, og så ellers udelad cite i resten
Dog, i defs. bør der stadig være en cite til deres def
\cite[Def. 42]{bibtexKey}

(a) (b)

Figure 2: A small example of firing a transition in a Petri net. The arcs (O_2, t) and (t, H_2O) does not have a depicted weight on them, because their weight is 1. In (a) the transition is enabled. (b) depicts the same Petri net as (a), but after t is fired. In (b), t is no longer enabled [7].

Thus, when a transition is fired, the Petri net changes state, as we can see in figure 2. Tokens are moved or disappear, and transitions can both be enabled or disabled. There is an important distinction in the definition of the Petri net, that was omitted until now. It is not one single entity, but consists of two objects. A *Net* denoted as N and a *marking* denoted as M. If we combine them, we get a Petri net $PN = (N, M_0)$. Here M_0 is the *initial marking*. A marking is something we place on top of a net and is the placement of the tokens. Note, that since the net N does not have a marking, it is not possible to talk about if a transition is enabled or not, since the notion of tokens is not yet introduced. This distinction becomes important later. A Petri net is thus comprised of two components. The structure of it in form of the net N, and its marking or *state* M. As an example, In figure 2a the Petri net is in the *initial marking*, denoted as M_0. Then t is fired, and we reach a new marking we can call M_1. Figure 3 is the Net N of 2a and 2b. A special case of a marking that will become important later is the *null marking*. This is the marking, where all places have 0 tokens[7].

"Figure"

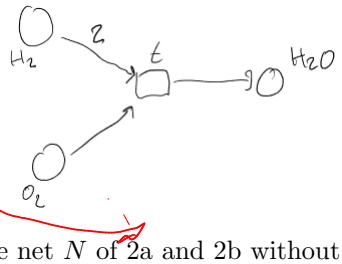


Figure 3: The net N of 2a and 2b without the marking.

From this, we can move on to the concept of a *reachabilty graph*. Since they will be at the core of this thesis, this subject will be covered in detail.

3.1 Reachabilty graphs

Figure 2a depicts a Petri net in a initial marking. From this initial marking, we can fire t and reach another marking, depicted in 2b. Since drawing the

"Figure"

inefficient?
efficient på dansk er effektivt, mens effective bedre oversættes til noget a la effektfuldt

entire Petri net for every firing is quite space ineffective, and is also hard to read if the Petri net is large, this process of going from one marking to another possible marking is often presented as a graph called a *reachability graph*. In a reachability graph, the nodes corresponds to the different markings and the directed arcs equates to a specific transition firing. The nodes can be depicted in a few different ways. One method is that each node is an array of numbers, where each entry corresponds to a specific place. The numbers on each entry, is the number of tokens in that place. This is the method used in figure 4. A reachability graph can thus be used, to get an overview of all reachable marking from some initial marking M_0 [7].

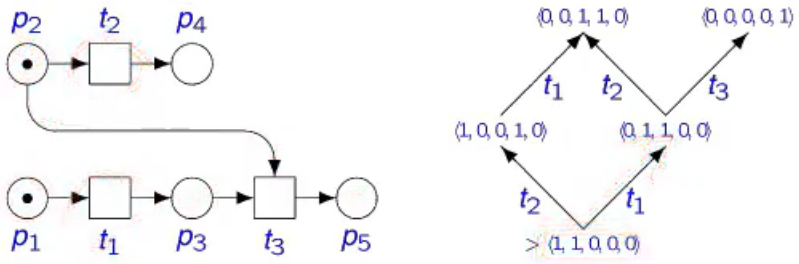


Figure 4: A Petri net and the corresponding reachability graph. The node in the bottom is the initial marking M_0. The order of the places in the nodes array are: p_1, p_2,p_3,p_4,p_5.

One important thing to note, is that a reachability graph can only be created, if the Petri net it is created from is *bounded* also called *k-bounded*. A Petri net is bounded, if it is impossible, from the initial marking M_0, to put more than k tokens in any place p in the Petri net, where k is some positive integer number. No matter what transitions is fired, or in what order they are fired in. Conversely, a Petri net is *unbounded* or *k-unbounded* if there exists a place p, where one can put an infinite number of tokens. If this is the case, then there cannot be created a reachability graph for that Petri net. Then a *coverability graph* can be produced instead [7]. The reason that a reachability graph cannot be produced if a place is unbounded is simple. Since there needs to be a node for each reachable marking, a Petri net with a place where it is possible to pump an infinite number of tokens into a place p, would obviously result in an infinitely large reachability graph which cannot be computed [7]. The smallest possible example of this, can be seen in figure 5.

Figure 5: A Petri net, that would result in an unbounded reachability graph. Since there t has no preset, it will always be enabled. We can thus always fire it, and create a new marking with one more token, then the previous one.

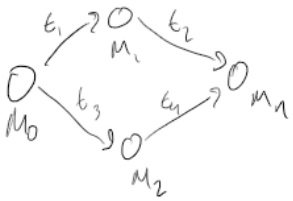
Now that we have defined what a reachability graph is, we will explore what aspects of the reachability graph we are interested in for this thesis.

3.2 Firing sequences and paths

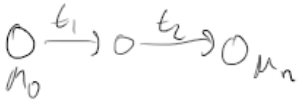
With a reachability graph created from a given Petri net, one can answer a lot of different questions about the Petri net, such as *liveness* and *reversibility*, which will not be explored in this project. The question that is relevant for this thesis, is if it is possible to reach a certain marking from M_0. We will call this marking the *end marking*, *goal marking* or the *target marking* in this thesis, and is denoted as M_n. This problem is known as the *reachability problem* [7]. This is written as $M_0[> \sigma M_n$. In this expression σ signifies a *firing sequence*. A firing sequence is as the name suggest a sequence of firings that can be performed from an initial marking to a target marking. What a firing sequence σ contains can vary depending on who you ask [7]. In this thesis, it will be a series a transitions $[t_1, t_2, t_3, ...t_n]$, that can be fired in the order of the sequence, to go from M_0 to M_n. Just for clarification, LoLA which we will explore more later, uses the term *reachability query* for the reachability problem, while σ is called a *witness* of the reachability query [8].

ser underligt ud, er det sådan de skriver det?

An important distinction needs to be made between a *path* in the reachability graph and a firing sequences. A path is a monomorphism of the reachability graph, and can be denoted as $M_0, t_1, M_1, t_2 ... t_n, M_n$ [7]. In other words, a path will be a subgraph of the reachability graph, where a firing sequence is just the edges on that path. An example of the differences between a path and a firing sequence is depicted in figure 6.



(a)



(b)

$[t_1, t_2]$

(c)

Figure 6: An example of the difference between a path of a reachability graph, and a firing sequence σ. (a) is a reachability graph, where the nodes are markings. The number of tokens on each of them are hidden, since it is not relevant for this example. (b) is an example of a path from M_0 to M_n. It contains both the markings and the transitions. The corresponding σ can be seen in (c).

The two have a bijective relation to each other. However, each individual firing sequence can be saved on less space since it only needs to store the transitions, while a path needs to store the transitions and nodes that represent the different markings. In section 6.5 we will discuss the advantages and disadvantages of storing either one while exploring the reachability graph. Just for completeness the strategy to go from a firing sequences to a path and vice versa is as follows.

If we have a firing sequences fire the transition from first to last using the initial marking as a starting point. After each transition is fired, record the marking that has just been reached. Now simply build the path. Retrieving the firing sequences from a path is even simpler. Start from the node that represents the initial marking, and for each edge encountered on the path, put the transition stored there in the back of a list. Continue this process until the end of the path has been reached.

Now that we have the definition of a path in a reachability graph and a firing sequences, we can finally discuss how Petri net will be used in detail in this thesis, and what will be a new entry to what can be done with Petri nets.

ikke så god en overskrift
det er præsenteres om PNs skulle gerne være hvad der er nødvendigt, så der er ikke brug for en afgrænsning af den grund. Måske det er mere er her det fra Sissels artikel er relevant? I.e., hvordan PNs kan bruges til kemi.

3.3 How Petri net will be used in this thesis

As discussed in the introduction, we want to use Petri nets to model chemical reaction networks, and find pathways between them. These networks can be expressed as Petri nets. How this transformation is done, won't be explored in this project. In practice, these will be produced by MØD.

Assuming we have such a Petri net that models a chemical reaction network, what question chemist we like answered, is if it is possible to go from an initial marking M_0 to a target marking M_n. Furthermore, they would like to know how they can reach M_n. As we have already discussed, there already exist programs that can do this such as LoLA [9]. What has not yet been created, is a program that can retrieve *all* possible firing sequences from M_0 too M_n. Why are we interested in this feature? Even though that it has been possible for a long time, to get chemical pathways from a chemical reaction network, there are a lot of questions about these pathways that are not known. Are there more than 1? If there are, how do they differ? Are some of them "better" in a chemical sense than others, or are they all equivalent? If there can be more than 1, are there just a few typically or are there many for a most chemical reaction networks? If it is possible for networks to have more than 1, are networks with only 1 path special in some way? This thesis won't answer these questions. I do not have the required background for it. But it will explore how such a tool to produce all of these paths can be designed and implement it, so that others will be able to.

However, before we go into how such a tool can be designed, we need some more background. Getting all firing sequences from M_0 to M_n is not all that the chemist will need. In section 4 we will cover the last parts we need.

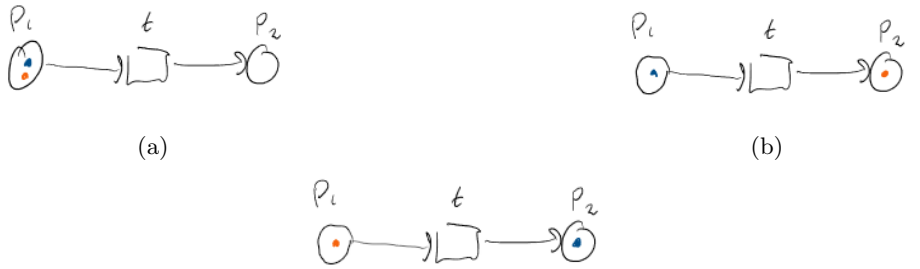
4 Individual vs collective token interpretation and Occurrence nets

As just stated, chemist will need a bit more, before they will have a solution they can use for their analysis. A Petri net as defined in section 3 can be used to model chemical reaction. We saw a small example of this in figure 2. However, they do not quite capture the whole story of a firing, because of the *collective token interpretation*. To explain this term, we need to look at what happens when a transition is fired, as a Petri net was defined in section 3.

When a transition is fired, of the transitions preset are consumed, and tokens in the postset are produced. But as these definitions indicate, it is not the same tokens in the preset and postset. When we say, that a token is consumed at a place, we literally mean, that they disappear from that place. When tokens are produced for a place, they are created for that place, and have no connection to the tokens that were removed from the preset. So for this model, there is no way to discuss where a specific token ended up after or during a firing sequence [5]. However, this can be important information for a chemist. What is needed, is an *individual token interpretation*.

In an individual token interpretation, each token is unique. When a transition is fired, a token is not consumed, but is moved from one place to another, though the arcs of the transition that was fired. This makes it possible to track a specific token, as transitions are fired in a firing sequence[5]. An example of this can be seen in figure 7. 7a depicts the initial marking of that Petri net.

The colors of the tokens, signifies that the two tokens are unique. Both 7b and 7c represents the two possible outcomes when firing t once. For contrast, figure 8 depicts the same Petri net, before and after firing t. Here, it is not possible to know which token now is in p_2.



(a) (b)

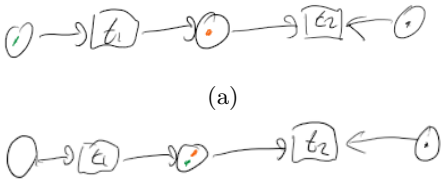
(c)

Figure 7: Individual token interpretation of a Petri net. (b) and (c) are two different outcomes that are possible from (a), when firing t once. The individuality of the nodes are represented by coloring the tokens.

(a) (b)

Figure 8: Collective token interpretation of the same Petri net as figure 7. Here there is only one interpretation after firing t, as we can see in (b).

This can make a huge difference, when talking about the casualty of a firing sequence. When the tokens are individualized, we can establish a casual link between the firings of a σ. This happens when a transition is using a token, that was moved to its preset by a previous firing. Ron van Glabbek has a brilliant little example, that I will borrow [5]. The figures can be seen in figure 9.



(a)

(b)

Figure 9: (a) is the initial marking of the Petri net. (b) is the Petri nets state after the transition t_1 was fired.

In figure 9b we can see the Petri net after the transition t_1 was fired. There are now two options, when we are at the state depicted in 9b. If the orange token is used to fire the transition t_2, then the firing of t_2 is independent of the firing of t_1. If the green token is used, then the firing of $t2$ is dependent on the firing of t_1. If we used the collective token interpretation, then this firing of t_2 would always be independent of the firing of t_1. This means, in the individualized token interpretation, we can establish a partial order of the firing of the transitions. We can in a meaningful way describe in all cases if a firing of a transition dependent on another transition [5]. This is often not the case for the collective token interpretation, as figure 9b is an example of.

We have now defined, what an individualized token interpretation is. We have yet to discuss how or if we are going to change our definition of a Petri net, in order to accommodate it. This will be discussed in the next section

4.1 Occurence net

It is possible to redefine the firing rule from section 3, so that it has an individual token interpretation [5]. This is not the approach that was used in this project.

Instead, we will derive the individualized token interpretation by creating what is called an *occurrence net*. The definition of an occurrence net, is sadly also dependent on which source you use to define it. However, they all reflect the same idea. Given an initial marking M_0 and a firing sequence σ we can create a Petri net like structure, that reflects what happened with the Petri net, as the transitions in σ were fired. What we need from it, is that we can use it to tell where each individual token was, after each firing. We will first present the definition that was used in this thesis, and then show a few examples of how one can draw an occurrence net from an M_0 and σ. However, before we can define what an occurrence net is, we need to revisit the definition of net.

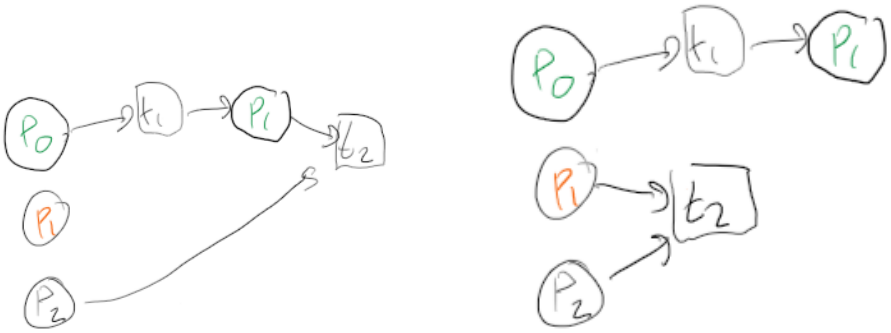
The definition of a net, is something we saw in section 3. As a reminder, a Net is the structure of the Petri net. It is the Petri net, without the marking. An occurrence net is a net, but with some restrictions on it:

Definition 3 *Occurrence net definition: [2]. A net K is an occurrence net iff*

(a) $\forall x, y \in K\ xF_K^+y \Rightarrow \neg(yF_K^+x)$ F_K^+ denoting the transitive closure of F_k.

(b) $\forall p \in P_K | \dot{p}| \leq 1 \wedge |p\dot{}| \leq 1$.

Let us examine the definition. The first restriction (a) simply states, that there shall be no cycles in an occurrence net. If there is a flow relation from x to y, then there is no flow relation to y to x. The second restriction (b) states that no arc going from or to a place can have a weight larger than 1. Another way to say this, is that no arcs have a weight larger than 1.



(a) (b)

Figure 10: The two possible occurrence nets that can be built from the Petri net in figure 9a after firing t_1 and t_2. The colors of the places indicates, what tokens they were built from. The names of the O-places indicates which place the token was taken from, when firing the transition it is has an outgoing arc to.

An example of an occurrence net can be seen in figure 10. Exactly how one would build an occurrence net from the Petri net (N, M_0) from figure 7a and the firings sequence $[t_1, t_2]$ will be explored later. For now, we just need to understand what an occurrence net is comprised off. Since an occurrence net is a net, it still has both places and transitions. These are drawn in the same manner as in a Petri net, with squares indicating that the node is a place, and a circle indication that the node is a transition. The arcs also have to comply to the same rules as a net. Places cannot have an arc's to and from another place, and a transition cannot have arcs going to or from another transition. Since an occurrence net does not have a marking, the places do not contain tokens, and is therefor never drawn with them either [2]. To avoid confusion going forward, when places and transitions are discussed, that are a part of an occurrence net, they will be called *O-places* and *O-transitions*.

The O-places and O-transitions of an occurrence net, is tied to a specific (N, M_0) and σ. The occurrence net that can be built from the Petri net in figure 9a and $\sigma = [t_1, t_2]$, is depicted in figure 10. The colors of the O-places,

indicates which token they are tied to. The names indicate what place the O-place was build from. Coloring the places is not normally done, and will not be done throughout the thesis. This is just to make the first few examples easier to understand. Each occurrence net that can be built from a Petri net $PN = (N, M_0)$ and a firing sequence σ corresponds to a choice, to what token to use for each firing. In figure 10a the green token was used for the firing of t_2, while in figure 10b, the orange token was used to fire t_2. We can thus use the different occurrence nets that can be generated from a (N, M_0) and a σ, to analyze how the individualized tokens can move though the Petri net. Note that given a (N, M_0) and a σ, one might get multiple "unique" and/or "different" occurrence nets. How occurrence net are defined to be either equal to or different to one another, will be discussed in section 4.3. The next section will explain how one can extract the different markings of an occurrence net.

4.2 Getting the markings from an occurrence net

We know that for a (N, M_0) and fire all the transitions of an σ in order, a number of different markings will be generated. These markings will also be present in the occurrence net(s) that can be generated from (N, M_0) and σ. However, how do we find the different markings in an occurrence net? Here we can use what Goltz and Reisig calls a *slice* of an occurrence net 3:

Definition 4 *Let K be an occurrence net.*

(a) $<_K := F_K^+$ is the *order relation of* K. The index K is omitted if it is obvious from the context.

(b) Let $\mathbf{li} \subseteq K \times K$ and $\mathbf{co} \subseteq K \times K$ be given by

$$x \mathbf{li} y :\Leftrightarrow x < y \vee y < x \vee x = y,$$

$$x \mathbf{co} y :\Leftrightarrow \neg(x \mathbf{li} y) \vee x = y.$$

(c) $M \subseteq K$ is a *cut* iff $\forall x, y \in M\ x \mathbf{co} y \wedge \forall z \in K \setminus M\ \exists x \in M\ \neg(x \mathbf{co} z)$.

(d) A cut $M \subseteq K$ is a *slice* iff $M \subseteq P_K$.

Definition 4 defines the order of the nodes in an occurrence net. Given two nodes x and y, they can either be ordered or the same element (denoted with $x \mathbf{li} y$), or we cannot compare the order of x and y or they are the same element (denoted with $x \mathbf{co} y$) [2]. A cut, is a subset of the occurrence net nodes, where order is not defined. A slice, which we are particularly interested in, is a cut that consists only of O-places. Figure 11 has some examples of slices of an occurrence net, and the difference between a cut and a slice.

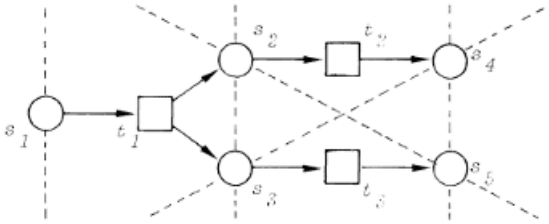
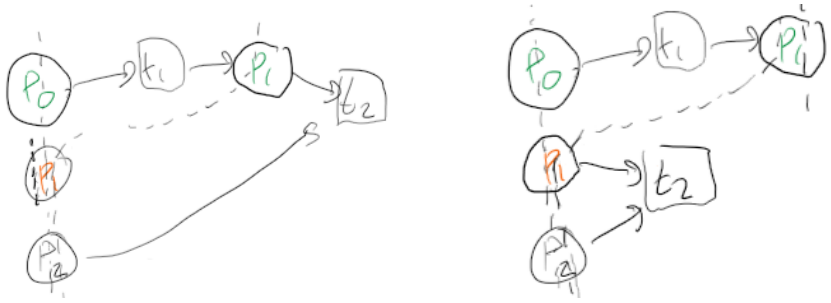


Figure 11: Example of slices from [2] The dotted lines are valid slices. $[p_2, t3]$ is a cut but not a slice, because of the presence of t_3.

Now we can finally discuss how one can find all the markings visited from firing the transition of σ from M_0 by examining the occurrence net. Let us revisit the occurrence nets of figure 10, but with some more detail that can be seen in figure 12.



(a) (b)

Figure 12: The two possible occurrence nets that can be built from the Petri net in figure 9a after firing t_1 and t_2. The dotted lines are the slices of the two occurrence nets.

The slices of the occurrence nets directly corresponds to the different markings that were reached from M_0 while firing $[t_1, t_2]$. There are in total 3 slices for both occurrence nets since there were 3 markings visited, M_0, the marking after firing t_1 and the marking after firing t_2. M_0 consists of the leftmost p_0, p_1 and p_2 in both 12a and 12b. After firing t_1, we will reach the marking M_1 that corresponds to the slice p_1, p_1, and p_2. The last marking M_n consists of the p_1 that was not used. This is orange p_1 for the occurrence net in figure 12a, and the green p_1 in figure 12b. Note that a special case is the null marking. Here, the slice of any occurrence net would be empty.

Note that neither the colors nor N is required to find the markings. However, it is not possible to always find only the markings that were reached from M_0 by firing σ in order, unless σ is provided. The initial marking, can always be created from the slice of the O-places that has no ingoing arcs. If we have an

arbitrary slice, that corresponds to the marking M_k, we can find a possible next marking M_{k+1} by using the following method.

We now have the required knowledge, to start building a solution for the initial problem. Given a Petri net (N, M_0) and a target marking M_n, compute all firing sequences from M_0 to M_n. Then for each σ and M_0 pair, compute all possible occurrence nets. One last piece we need, is related to the occurrence nets. As we mentioned early, for a M_0 and a σ, it is possible to produce many occurrence nets, that are either identical or unique. But we have not yet defined, what this means mathematically. This will be done in the next section. Then, we will start to build our solution.

4.3 Occurrence nets and isomorphism

As vaguely discussed in section 4.1 the occurrence nets that can be produced by a firing sequence and a Petri net, can either be "identical" or "unique". We have a few good reasons to be able to tell, whether an occurrence net is "equal" to another. Since these occurrence nets will be implemented as a type of graph (more on this later), each can take up a significant amount of space. Therefor, we do not want to store several identical occurrence net. There are also runtime savings that can be achieved, by not storing them all. This will also be discussed in more detail later. Lastly, we do not want to waste the user's time, by making them look through a bunch of solutions, that are identical. With this motivation in place, we can define what an *isomorphism* is.

Isomorphism has it origin from group theory. As everything in group theory, it can be applied to a lot of settings. In general, if two objects are isomorphic, they are the equal to one another. Here we have to distinguish between if they are *representationally equal* or isomorphic. If two object a and b are representationally equal denoted $a \overset{r}{=} b$, they have the same structure, but are not the same mathematical object. If they are isomorphic, denoted with, $a \cong b$ then they are the same mathematical object [1].

An example could be the mathematical object or group of all real numbers $\mathbb{R}$. Here we can denote the same mathematical object $\frac{1}{2}$ in many ways such as, $\frac{2}{4}$, $\frac{3}{6}$ and $\frac{21}{42}$. Each of these objects are the isomorphic to each other, $\frac{1}{2} \cong \frac{2}{4}$ but they are not representationally equal $\frac{1}{2} \overset{r}{\neq} \frac{2}{4}$. One could then try to figure out, fractions are isomorphic to one another, by finding the greatest common divisor.

In the graphs, this question is more complicated. Here we can also distinguish between the representation of a graph, and the mathematical object. There are several ways a graph can be represented, such as adjacency list or adjacency matrix [3]. Here, the nodes are usally given ID's in the form of integers from 0 to n. However, these are **not** a part of the mathematical object. The ID's are simply attached to the nodes, so that we can reference the distinct nodes. You could take the exact same graph and label it completely differently, and it would still be the same graph. The example in figure 13 is taken from [1], and exemplifies this difference. In this figure, all the graphs are isomorphic

to one another. It is only their representation that differ. This is an important thing to note. The labels themselves are not a factor, when we try to compare 2 graphs for isomorphism. This form of isomorphism, is called *graph isomorphism without labels*[1].

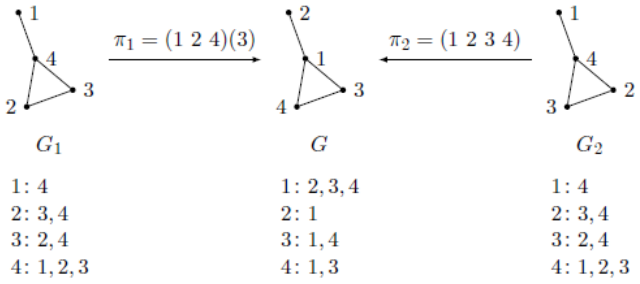


Figure 13: This example is taken from [1]. The graphs are represented as adjacency lists, and $G_1 \overset{r}{=} G_2$ while G is not representationally to the other two. All these graphs are isomorphic to each other.

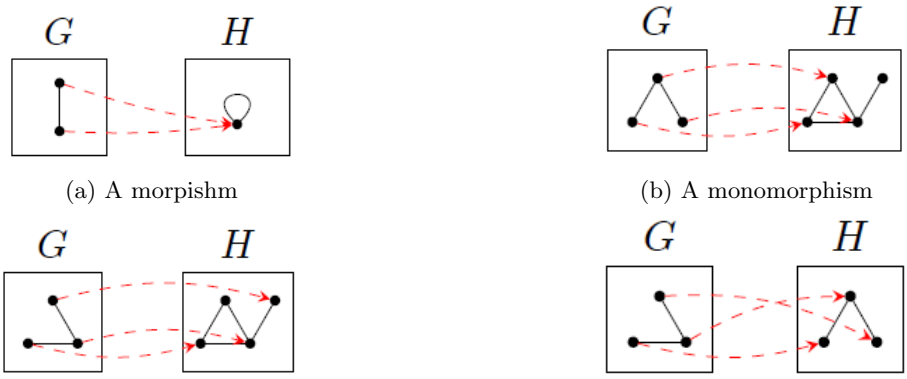
To move forward, we need a formal definition of what it means for two graph to share the same structure. We can use the following definitions [1]:

Definition 5 (Graph Morphisms). A graph (homo)morphism $m : G \to H$ is a structure-preserving mapping of vertices and edges. That is if $e = (u, v) \in E_G$ then $m(e) = (m(u), m(v)) \in E_H$.

- m is a *monomorphism* if it is injective: $\forall u, v \neq\!\!\to m(u) \neq m(v)$. When a monomorphism exists, we may simply write it as $G \subseteq H$ or in the reverse order $H \subseteq G$.
- m is a *subgraph isomorphism* if m is monomorphism and $(u, b) \in E_Q \Leftrightarrow (m(u), m(v)) \in E_H$.
- m is an *isomorphism* if it is a subgraph isomorphism, and it is a bijection of the vertices. When an isomorphism exists, we say G and H are *isomorphic* and write it as $G \cong H$.
- m is an *automorphism* if G and H refers to the same graph, and m is an isomorphism. We say that m is the *trivial* automorphism when the identity morphism id_G.

To help explain these definitions, the small graphical examples of [1] will be borrowed in figure 14. A morphism is just a mapping for all the nodes in one graph H to all the other edges in the graph G. This is also valid, if two nodes from H are mapped to the same node, as we can see in figure 14a. The only constraint is, that if there were an edge between two mapped nodes in H, there also need to be an edge between the nodes they are mapped to in G. A

monomorphism, is the same as a morphism with just one more constraint. It needs to be injective, so we cannot map several nodes in H to one node in G. Note, that since a monomorphism does not require a mapping to be surjective. We do not need to map for all edges and nodes. This is why, the mapping in figure 14b is a valid monomorphism. The restriction that the mapping of the edges must also be surjective comes with a subgraph monomorphism. If there is an edge between 2 nodes in H, there must also be an edge between those nodes in G. Lastly, an isomorphism is a subgraph isomorphism, with the added constraint that the mapping of the nodes must also be a bijection.



(a) A morpishm

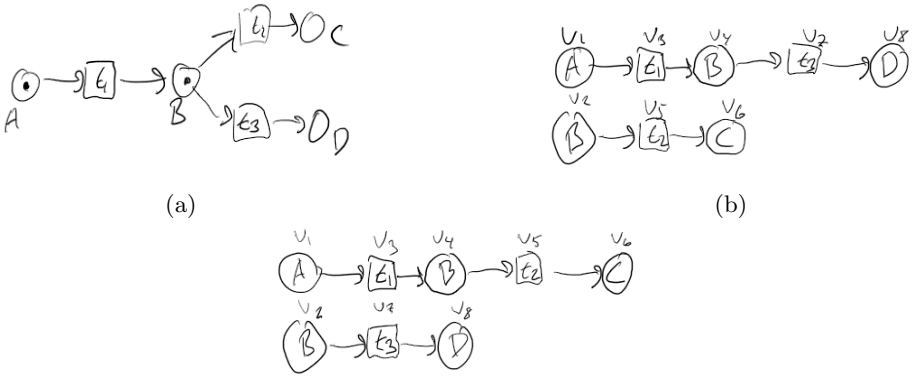
(b) A monomorphism

(c) A subgraph isomorphism

(d) An isomorphism

Figure 14: Example from [1] of the different morphism in definition 5. The red dotted lines, indicates the mappings, that makes the morphism, monomorphism, subgraph isomorphism and isomorphism valid.

This definition fits the case of an *unlabeled graph*. However, the graphs we will work with are not unlabeled. Let us take a look at an example, to see why the definition 5 is not enough on its own.



(a) (b)

(c)

Figure 15: An example, that the definition of isomorphism from 5 cannot cover the isomorphism question of occurrence net in a desirable way. The graphs in (b) and (c) are the 2 possible occurrence nets that can be created from (a) and the firing sequence $[t_1, t_2, t_3]$. The tags A, B, C, t_1 and t_2 inside the nodes of (b) and (c) is the auxiliary data of the nodes (The places that the O-places was build from, and the transition that was fired) and the tags v_n above the nodes of the occurrence nets of (b) and (c) are the labels. (b) $\cong$ (c) according to definition 5, but the two occurrence net are not equal.

In figure 15 we can see a Petri net, and two occurrence net build from the firing sequence $[t_1, t_2, t_3]$. If we follow the definition of isomorphism that we saw from definition 5, then the occurrence net from figure 15b and 15c would be isomorphic. However, we do not want these two occurrence net to be categorized as isomorphic. The reason is, that the casualty of the firing of the transitions are not the same for the two occurrence net. In the occurrence net of figure 15b, the firing of t_3 depends on the firing of t_1, and t_2 can be fired independently off the other two transitions. In figure 15c t_2 depends on the firing of t_1 and t_3 is independent of the other 2. Since the point of the individualized token is to capture these dependencies, we need further restrictions on our definition of isomorphism by including *auxiliary data* on the nodes.

Auxiliary data, is some extra data that will be attached to each node. Note, that this is different from the graph's representation. The auxiliary data is part of the mathematical object of the graph, while its representation is not. In the figures 15b and 15c the difference between the auxiliary data and labels are displayed. The auxiliary data are the tags A, B, C, t_1, t_2 inside the nodes, while the labels are the v_n tags above them. We need the auxiliary data to be taken into account, when comparing the two occurrence nets to capture the dependencies of the firings. However, we could change the v_n labels to anything, and it would not change the occurrence net itself, only the representation of it. In our case, the auxiliary data can simply be a number that represents what place the O-place was build from or the transitions ID from the Petri net. Given

this, we can create a new definition that takes inspiration from [6]:

Definition 6 m is a *labeled isomorphism* if it is an isomorphism, and $\forall v \in V$, $l(v) = l(m(v))$.

Here I have chosen to follow convention, which is sadly to call a graph with auxiliary data a *labeled graph* and an isomorphism with labeled graphs a labeled isomorphism [6]. Definition 6 simply states, that a labeled isomorphism is an isomorphism, where the labels from H to G for all vertices also match. Under this definition, the graphs of 15b and 15c are not isomorphic. An example of an occurrence net, with a labeled graph that is labeled isomorphic to the occurrence net in figure 15b can be seen in figure 16. This occurrence net was also created from the Petri net in 15a, but from the firing sequence $[t_2, t_1, t_3]$.

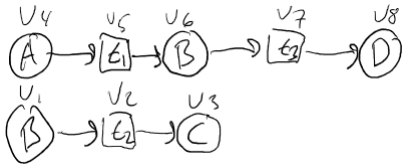


Figure 16: The only valid occurrence net that can be created from the Petri net, and the firing sequence $[t_2, t_1, t_3]$. The graph of this occurrence net, is labeled isomorphic to the graph of the occurrence net of figure 15b.

Going forward, we will never use isomorphism on graphs that are not labeled again. Definition 5 was only introduced, as a building block to labeled isomorphism. Since this is the case, I will always refer to labeled isomorphism as isomorphism from now on. If 2 occurrence nets H and G are compared for labeled isomorphism, then I will similarly use $H \cong G$ to indicate that they are labeled isomorphic.

In principle, we are ready to start to build our solution. However, as we have already discussed shortly, we will need a method for telling if 2 occurence nets are isomorphic. Before we start implementing Petri nets and occurrence nets, let us take a short detour to discuss how this can be done.

4.4 Finding isomorphism with VF2

5 Implementing Petri nets

5.1 Checking if a transition is enabled after a fire and backfire

Some of the functions that will be called often are `fire()` and `backfire()`. Profiling has revealed, that up to 80% of the time could be used on these two functions while exploring the entire reachability graph, and when a straight forward implementation without any optimizations was used. It is thus important that any unnecessary work is removed from them.

The same profiling results showed that both `fire()` and `backfire()` used around 60% of their time calling and executing the function `is_transition_enabled()`. So if this function could be optimized or called less often, it could lead to a speedup. As the name indicates, the job `is_transition_enabled()` is to check whether a transition is enabled or not. It is called in the instances, when a transition is fired or backfired, and other transitions that shares places with it could be affected. Figure 17 exemplifies this.

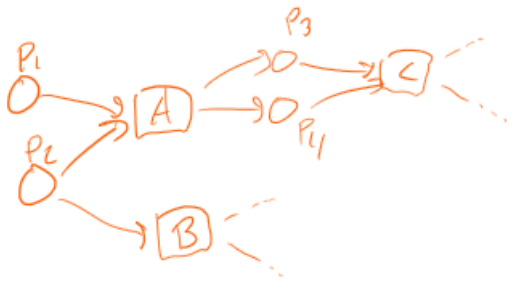


Figure 17: A is the transition we fire, and that can affect B and C's enabledness.

If A is fired, then it will take tokens from P_1 and P_2 and place some tokens in P_3 and P_4. Let us pretend that the transition B was enabled before A was fired. Depending on the number of tokens in P_2 and the weight of the edge from P_2 too B, this could disable B. Likewise, if C was disabled before A was fired, the firing of A could enable C. Thus, we need `is_transition_enabled()` to check if the states of B and C when A is fired. `is_transition_enabled()` needs to check for all places in B and C's pre-set, the weight of the edge going from the place to the transition, is higher than the number of tokens in that place. If this is true for any place/edge combination, then the transition needs to be disabled. If it is false for all of them, the transition should be enabled. However, there are cases where we can omit these checks. If B was already disabled before A was fired, we do not need to check if it should be enabled or not. The reason for this is, that firing A removes tokens from B's pre-set. If it is already is disabled, then removing tokens from the pre-set of B cannot enable it. The reverse is true for C. If C was already enabled before A was fired, then it will also be enabled after A put tokens in the pre-set of C. Thus,

we do not need to check the pre-set of C either in this case. The same logic can be applied for backfire but in reverse. C is now getting tokens removed from the places of the pre-set, so if C was disabled before A was fired, then the call to `is_transition_enabled()` can be skipped. The same logic can be used for B.

There is one last call to `is_transition_enabled()` that we need to consider, and that is to the transition that is given to `fire()` and `backfire()` themselves. As we know, after a transition is fired, it might get disabled. So we need a call to `is_transition_enabled()`, or check it as we remove tokens from the transition's pre-set. The amount of work done is the same, just at two different times. So there is anything to gain here. The same cannot be said for `backfire()`. We do not need to check, if the transition backfired is enabled after it has been backfired, since we can deduce that it must be enabled at that point. A transition we backfire in `explore_state_space()` must have been enabled when it was first fired, since it could not have been fired otherwise. Thus, it must become enabled again after the call to `backfire()`. Thus, we never have to call `is_transition_enabled()` to check the backfired transition.

6 Getting all paths

6.1 Answering the reachability problem

The simplest and most naive way to answer a reachability problem, is to produce the reachability graph from the petri net, and then check if the target marking is in the reachability graph.

6.2 Explore state space

Algorithm 1: Explore state space

Input : `marking`, The initial marking
`end_marking`, The target marking

```
1  stack = [null,(enabled(Marking))]
2  while stack is not empty do
3      (t_from,enabled) = stack.pop()
4      if enabled is not empty then
5          transition = enabled.pop()
6          stack.push((t_from,enabled))
7          marking.fire(transition)
8          stack.push(transition,enabled(marking))
9      else
10         if t_from is equal to null then
11             continue
12         marking.backfire(t_from)
13         path.pop()
```

6.3 Is reachable

6.4 Find a path

6.5 Find all paths v1

Once `find path` is defined, it is trivial to create the algorithm to find all firing sequences from the initial marking to the target marking. This algorithm will be referred to as `Get_all_paths`. A trivial solution is to use the same pseudocode from `get path` and make two small changes. The first is to add some container that will hold all of the paths. The second is to save a copy of the *path* in that container every time the target marking is found, and not stop the exploration of the state space before the entire state space has been explored.

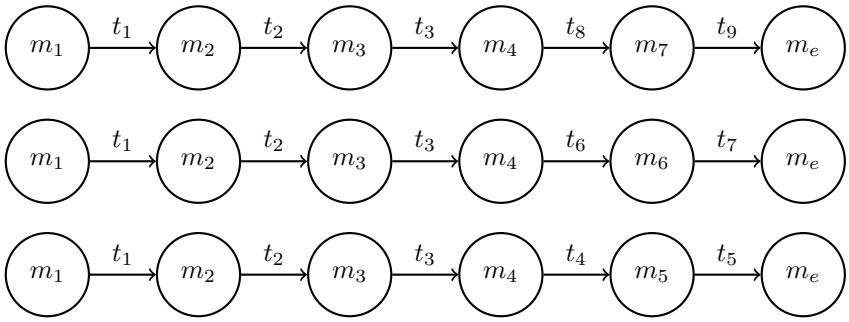
Algorithm 2: Get all paths v1

Input : `marking`, The initial marking
`end_marking`, The target marking
Output: `all_paths`, a container with all the paths

```
1  Initialize the path and the all_paths
2  if marking is equal to end_marking then
3  |  return All_paths
4  stack = [null,(enabled(Marking))]
5  while stack is not empty do
6  |  (t_from,enabled) = stack.pop()
7  |  if enabled is not empty then
8  |  |  transition = enabled.pop()
9  |  |  stack.push((t_from,enabled))
10 |  |  marking.fire(transition)
11 |  |  path.push(transition)
12 |  |  if marking is equal to end_marking then
13 |  |  |  all_paths.push_back(path)
14 |  |  |  marking.backfire(transition)
15 |  |  |  continue
16 |  |  stack.push(transition,enabled(marking))
17 |  else
18 |  |  if t_from is equal to null then
19 |  |  |  continue
20 |  |  marking.backfire(t_from)
21 |  |  path.pop()
22 return all_paths
```

Algorithm 2 will return all firing sequences, but it has some major flaws. The first one is that any paths that goes through the same n markings, will have firing sequences that have the same $n-1$ transitions in the same order, and thus wasting space in this scenario. In the worst case, all paths to the target marking will be the same until the last two transitions. The memory consumption will thus be $\Theta(n \times m)$ where n is the number of paths, and m is the length of them measured in the number of arcs in them. In this case $\Theta(n \times (m-2))$ of the arcs are the same for all paths. This scenario is depicted in figure 18 for a small state space graph.

Figure 18: All paths memory consumption worst case



The second flaw is more troublesome, since it is not case dependent and will in many cases prevent us from computing a solution in a reasonable amount of time. If we only collect firing sequences and no paths while exploring the state space, we cannot make use of any `Collection`. The concept of a `Collection` will be fully explained in section 6.8. For now, it is enough for the reader to know that a `Collection` is some sort of container that stores all of the seen markings. When a new marking is found while exploring the reachability graph, it is saved in the `Collection`. It can then be used for look ups after a transition is fired. If we have seen the marking before, then we do not have to explore it again and we can simply backtrack. This gives a major speedup of the cost of memory consumption. This idea comes from this LoLA article [9]. As tests will later reveal, exploring the state space without a `Collection` is in most cases meaningless since the algorithm will slow down to an unacceptable degree. Thus, using a `Collection` is required.

However, if a `Collection` is used then allot of desirable guarantees will be lost about the paths that can be recreated from the firing sequences, without any post processing. Such as if all paths are represented by a firing sequence, or if the shortest path or the longest path is present in the returned firing sequences. All that can be guaranteed is that at least one path will be returned if there is a path from the initial marking to the target marking. The figures 19 and 20 highlights the problem. In both figures the green node is the initial marking, and the target marking is the yellow one, while the blue path is the first that is discovered. If we remember the markings 2,3 and 4 with a `Collection` we will have two choices when we discover that our current path leads us to a marking that have already been explored that is not the target marking. It can either be stored as is or it can be forgotten.

If we don't store it, then we don't store any firing sequences that goes though the same markings. This means that only the blue path in figure 19 and 20 will be stored, and not all paths will be returned. The algorithm is therefore not correct.

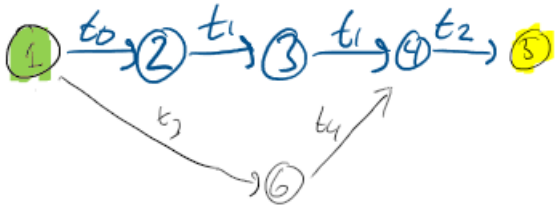


Figure 19: Shortest path will not be saved by *All paths* with `Collection`

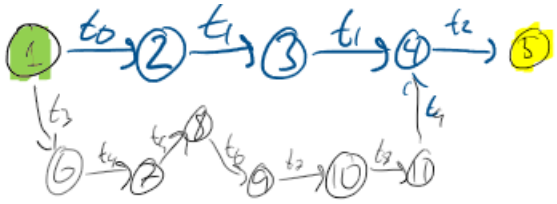


Figure 20: Longest path will not be saved by *All paths* with `Collection`

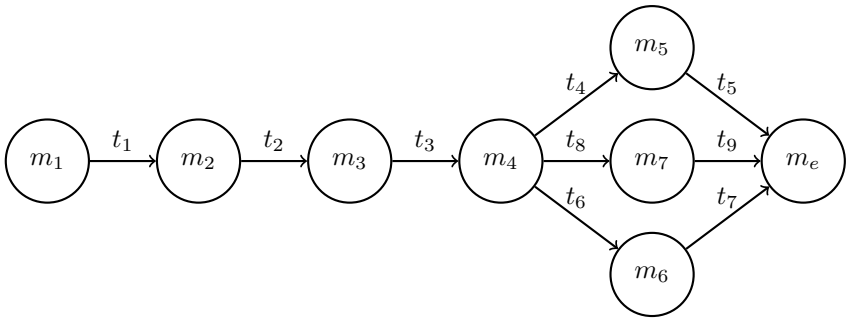
If we do store the firing sequences, when a marking that has been visited before is encountered then the firing sequences themselves will loose many desirable properties. Except for the first, it cannot be guaranteed that the firing sequences leads to the target marking. So they will not be useful, before they have been post processed. The natural thing to do is to create the state space graph from the firing sequences and retrieve the paths from it. These paths can later be used to produce all firing sequences. This is not a very elegant or efficient solution.

The other option is to build the state space graph during the state space exploration. The downside of this strategy is, that the graph itself can become quite memory intensive. However, this is also true if we only stored the firing sequences. Which one is the best memory wise, depends on the state space graph. If it has a lot of paths that share the same edges, then it will be optimal to create the graph during the state space exploration. If the state space graph consist of many isolated paths, then it will be optimal to store it as a list of firing sequences. And this strategy has the advantage of only having one post process step after the state space exploration is complete, namely getting the firing sequences from the graph. For this reason, the later strategy has been chosen in this project.

In the next few sections, we will arrive at what will be the template for the `state_space_exploration` algorithm. We will add the optimization of a `Collection` to it, and make the algorithm more generic. So far, we have seen three different versions of basically the same algorithm. `is_reachable`, `get_path` and `get_all_paths` all explore the reachability graph in the exact same way. The only thing that differs between them, is what information is

picked up along the way and when the algorithms terminates. This introduces duplicate code, which is bad for many reasons, which will be explored later. To avoid this, we can use the strategy of `visitor patterns`, that is also famously used for algorithms such as `depth first search` [3]. This and `Collections` will be explored in the next two sections

Figure 21: All paths memory consumption using a state space graph



I have already discussed some of the problem with `is_reachable`, `get_path` and `get_all_paths` in section 6.5. If we knew beforehand that these three algorithms would never change, and that it made no sense to try different strategies to build the state space graph, then it may be possible to argue that this is okay. However, this is not the case. We want to try different strategies of building the state space graph, and test them against each other. As an example, we might want to only add paths without cycles in them. Right now, a entirely new and mostly identical algorithm would have to be added. Just like with `is_reachable`, `get_path` and `get_all_paths` the only difference between the mentioned algorithm and the others is the information gathered while exploring the state space graph. The cost of innovating and trying new things is thus quite high, since for each new algorithm we would add allot of duplicate code that has to be maintained. This is also bad for benchmarking purposes. The reason for this is, that we want these algorithms to be directly comparable when benchmarking them against each other. We want everything to be the same, execpt for the information gathered along the way. However, the chance of this not being the case, is higher with many mostly identical algorithms. What if one of the algorithms did not get an optimization that the others did, simply because it was overlook or wrongly implemented?

Thus, the state space exploration part of the algorithm and the information collecting about the reachability graph needs to be separated. If an optimization to the state space exploration part of the algorithm is implemented, we want it to make that optimization in one place, so that it is applied to all algorithms. And we want to be able to try a new algorithm, without having to rewrite much or ideally any code. Thankfully, others have already encountered these problems before and have created design patterns to address them. One suitable for this case is as the title of the section suggest *visitor patterns*.

The idea of visitor patterns is to make adding operations to one or more types easy [4]. In our case, the type would be the reachability graph we explore, and the operations are the different functions that pick up information about the graph as it is explored.

6.6 Visitor patterns

What visitor pattern does, is to isolate one aspect of a function or class that changes and isolates it. In our case, we want to isolate the gathering of information, and when the algorithm should stop, from how the reachability graph is explored.

6.7 State space exploration with visitor patterns

6.8 Collections

As shortly discussed in section 2 section, `Collection` is a container that stores markings. However, we have not yet covered them in detail and why we can simply backtrack, if we encounter an already explored marking. This is done in this section. First a definition:

Definition 7 *A collection is a container that holds markings. It has a single operation,* `try_insert()` *that takes a marking and tries to insert it into the collection. If the marking is not in the collection, it is inserted into it, and is given a unique number as an ID. This number is returned along with the boolean value false. If the marking is already in the container, the boolean value true is returned along with that marking's unique ID number.*

As mentioned, the idea of a Collection comes from In LoLA. As they describe it, a collection has two operations, `search()` and `insert()`[9]. These operations are used to update the collection when a new marking is encountered, and to check, after each firing of a transition, if the marking that has just been reached is one that has already been explored earlier. If this is the case, then they claim that it is not necessary to explore that marking again. We can simply backfire and go to the next entry on the stack. [9]. Our definition follows their description, except that the two operations have been comprised into `try_insert()`, and that we return a number that identifies that marking. These two choices will be explained later. For now, there is a more pressing concern, is their claim true? Can we backtrack when we encounter a marking that has been seen before?

To demonstrate that this will go well, we can look at the figures below. In figure 22 the marking that is found again is the end marking. The question is now, why we can omit to search beyond the end marking for more paths. Since we want all paths, including the ones that go through the end marking, we need to establish that the markings that are past the end marking have already been explored, or will be explored later. We can thus divide this up into those two cases.

The first is easy. If the marking we find already has been fully explore, there is no reason to explore it again. We can then just backtrack and go to the next iteration in the while loop.

The other case is not quite as trivial. When a marking that has been encountered before is found, that has not been explored yet. This implies, that the enabled transitions from this marking already is somewhere on the stack. If we backtrack now, eventually that marking will be reached again, and then all reachable markings from it will be searched by firing those enabled transitions. Thus, there is no reason to explore that marking now, and it can be omitted.

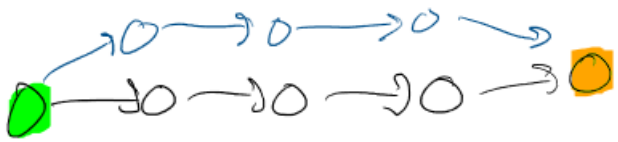


Figure 22: New path to add,

There is another very important observation related to Collections that the authors of LoLA makes. The markings stored in the collection is only used for checking if the current marking has already been seen. They are never altered in any way, and not used for anything else. We can thus compress the markings stored in the collection to save memory [9]. The only requirement for this compression is that it is injective. So for every marking m, the compression $c(m)$ should be unique, and the inverse function $c^{-1}(m)$ does not need to exist. The compression function is allowed to be bijective. It is just not required. This fact gives the option to compress a given marking in a way, where it will not be usable as a Marking-container after it is compressed. Compressing a Marking-container in such a way, will create a new entity, we will call a *collection-container*. The definition can be seen below:

Definition 8 *A collection-container is created from a Marking-container. For every Marking-container, a unique Collection-container exists. It is not required but allowed, that the inverse function exists.*

Note that this definition allows to use the identity function (do nothing) on a Marking-container to "create" a collection-container. This might be beneficial to do, if memory is not an issue for the specific instance. The reason is, no matter what function (except for the identity function) is used, it will cost some time to do. Since a `search` in the Collection is needed for every time a transition is fired, and an `insert` for each new marking found, this could save a substantial number of computations. This tradeoff will be tested later.

It is clear from the definition, that a collection works slightly differently than the corresponding construct from LoLA. Our collection has taken the two functions `search()` and `search()` and combined it into one function, `try_insert()`. The reason for this, is that it is more efficient to combine them. After each fire,

we need to insert the marking if it is not there. If it is, then we can immediately return true along with that marking's ID, instead of searching for it just after. The same is true, if the marking is not there. There is no reason to search for it just after, since we just had the correct values. However, LoLA probably does something similar, even though they don't write it [9].

The ID's that the markings are given are not necessary if we were only trying to implement `is_reachable()` and `get_path()`. But they are extremely useful when building the state space graph. How they are used, will be covered in 7.

6.9 Collection-containers

7 Building the state space graph

7.1 Building the state space graph with cycles

Call this APC

7.2 Building the state space graph as a DAG

We will use the same strategy, of using `build_first_path()` and `attach_path()` for creating the graph. No changes are needed for `build_first_path()`. It has to be called for the exact same cases, since creating the first path can never create a cycle. `attach_path()` does not need to change either, but when it is called does since this function can create a cycle. We need to either know beforehand if a cycle is created and not add the path, or add it and check if a cycle has been created and then remove it. Let's explore the former option.

ikke start en sætning med ikke-tekst

pas på det ikke bliver for wall-of-texty, kan der tilføjes en figur? pseudokode? eller andet?

It is not enough to just look at the current path, and check if that creates a cycle with itself. The DAG itself has to be checked. Given a DAG, if an edge (u,v) is added to it, then that edge will only create a cycle, if there already exists a path from (v,u) in the DAG. This can be checked with a DFS or BFS. Since memory is a general concern in this setting, it is best to use DFS in this case. Fortunately, we do not need to check for each edge added if a cycle will be created. This comes from the fact, that `attach_path()` adds the paths from the top to the bottom. As we have already seen, this gives us an invariant that states, that we add our path as soon as the node in the top, is equal to a node in the graph, we know that from that node, to the same node in the chain, they are not already in the graph. We thus just have to find that node, and check if there already exist a path from it, to the node we want to link to. So for each path added, we need to do one DFS search on the DAG, to check if we create a cycle.

There are two cases, where the DFS search is not necessary before we attach a path to the DAG. The first is, when the path that leads to the initial marking. This will always create a cycle, meaning that we never have to consider adding this path. The second case is, when the path leads to the end marking.

The cost is one linear search, from the top of the path, to where it connects to the DAG plus a DFS on the current DAG. Then in the worst case, we need to

add the path, given us the same linear cost. This is quite expensive. Compared to APC, we need one more linear search and a DFS search on the DAG for each path added in the graph APC produces. So even though we might add fewer nodes and arcs in total needs to be created in the state space graph, we except the running time to be slower.

8 Creating the occurrence nets

9 Implementation

9.1 Getting the enabled transition stack

As we have seen the `explore_state_space()` algorithm needs to retrieve a stack of all enabled transition, every time a new marking is reached. How this stack retrieved could greatly impact performance. The naive approach is to iterate though all transitions and put them on a stack, if they are enabled. The problem with this approach is, that the number of enabled transition will often be fewer, sometimes allot fewer than the number of transitions in the Petri net. A better approach is to keep some sort of data structure updated that contains the enabled transitions, and use that to return the stack. Note that no matter what data structure we choose for this job, `explore_state_space()` can never use references to that data structure, to save on the stack. The inner stacks on the transition stack can never change, unless they are popped. But if they are references, they could change every time a `fire()` or a `backfire()` is called when exploring the state space. Therefor, it has to be either a copy of the data structure we chose (if it itself is a stack of the enabled transitions) or it has to be able to construct the stack and return it `explore_state_space()`.

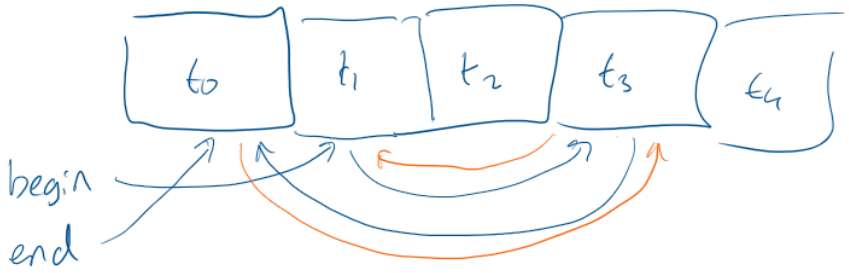
The designers of LoLA also encountered this problem, and they solved it with some kind of linked list. I have not followed their example fully, since they seem to maintain an in-order of the transitions in their linked list, since they have implemented the ability to delete and insert transitions to any position in their linked list. I have not been able to figure why this could be helpful, so that detail is not present in my solution [9]. The data structure that is inspired by LoLA's solution is called `Linked_list_alike`.

`Linked_list_alike` is an array with an internal double linked list inside of it. It contains all transitions in nodes, that also have a `forward` and a `backward` index, which are both set to -1 if the transition is not enabled. These functions as the pointers of the nodes. Lastly, `Linked_list_alike` has a `begin` and `end` index. When the linked list is empty (no transition is enabled) then `begin` and `end` are both set to -1. The position of the node's in the array, is determined by the transitions id's. Since they go from 0 to n (number of transitions) we can use this to jump directly to a transition's node when we want to update it, and not do a linear search as is typically done with linked lists [3].

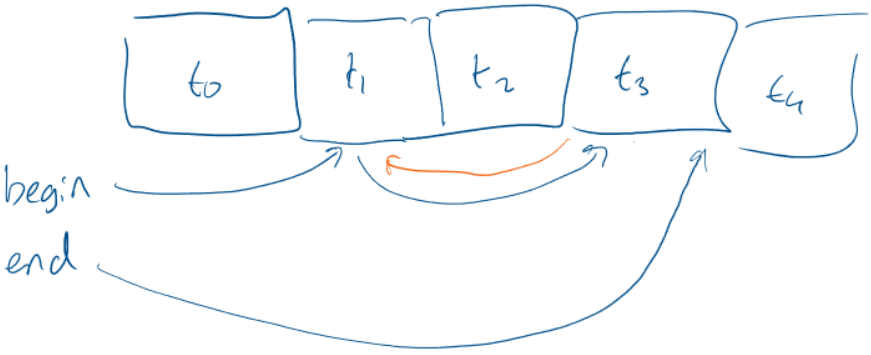
A transition is added to the linked list by passing it to the method `insert()` when it is enabled. When a new element is added, it is always put in the back of the linked list. It is thus inserted by following the end index setting the new nodes pointers. The forward pointer will be set to -1 and the backward pointer is set to the old end of the linked list. Then the old end is reset, by adjusting the node's forward index to point to the new node. `insert()` running time is thus a constant function. Note that the backward pointers of the nodes, are not used at all in this method. However, we still need them, when we want to delete a node.

Removing a node, with the method `erase()` when the transition is no longer enabled, has a slightly different behavior than the typical linked list. Instead of

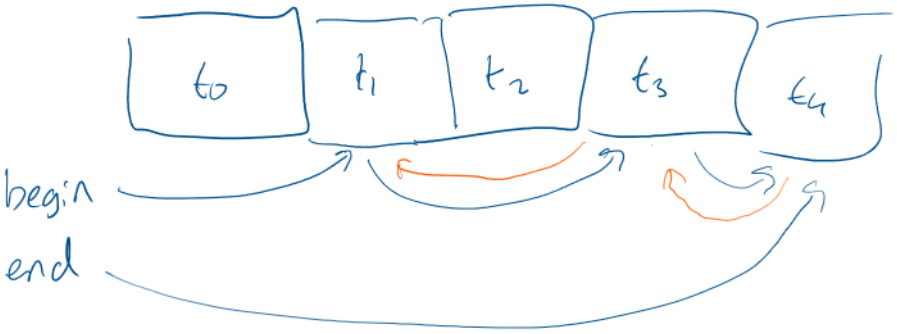
searching from the `begin` index, we can simply jump directly to where the node is stored. The forward and backward index of the node is simply set to -1, and the predecessor and successor of that node has their forward and backward index adjust in the typical manner [3]. Note that we need the backward pointers of the nodes in this case. If we want to adjust the predecessor of a node's forward pointer, we need to do know what node is the predecessor. If the nodes do not have a backward pointer, then the only way to find it, is to do a linear search from `begin`. With backward pointers, this update can be done in constant time. Thus, `erase()` has a constant running time as well.



(a) Initial state.



(b) Remove t_0 from the enabled transition.



(c) Add t_4 to the enabled transitions.

Figure 23: A small example of a `linked_list_alike`. It has 5 transitions where t_0, t_1 and t_3 are enabled. The blue lines are the "pointers" or indexes that goes forward, while the orange are the "pointers" or indexes that go backwards for a given node.

The third method of `Linked_list_alike` called `get_enabled_transitions_stack`, is to create the stack of enabled transitions. It does this by following the point-

ers of the internal linked list. The process is started from the `begin` index, and then the forward indexes are followed until the current node's forward pointer is equal to -1. The method returns a `std::vector<Transition>` which size is fitted to the number of enabled transitions. This is important to do, since there will be a lot of these `vector`'s in memory. Adding this information to the `Linked_list_alike` and keeping it updated with each `insert()` and `erase()`, means that the `std::vector<Transition>` can be initialized to the correct size, and there is no waited time with memory allocations either, making the creating of `vector` as fast and memory efficient as possible.

One last observation, is the order of the transitions is not preserved. Normally, that is not a concern. However, when developing and debugging the `explore_state_space()`, it is helpful to be able to enforce a certain order of firing the transitions as the reachability graph is explored. It is also useful, when writing unit tests. For this purpose, another data structure has also been implemented called `Use_set`. It can be given either a `std::set<Transition>` or a `std::unordered_set<Transition>` as a template argument. Then it will use these containers, to keep track of what transition is enabled, with their `insert()` and `erase()` method, while its iterators is used to create the same `std::vector<Transition>`. `Use_set` uses the same interface as `Linked_list_alike`, so they can be used interchangeably in the `Marking` class. These will also later be used for benchmarking `Linked_list_alike` against two other reasonable solutions, to see how effective it is.

9.2 Finding all the path to an end marking

9.3 Building the state space graph

9.4 Using type traits, to choose the correct Transition stack

9.5 Optimize `attach_path()`

10 benchmarking

In this section, we will benchmark the code that has been implemented. There will be 3 overall sections. The first section, will benchmark the implementation of Petri net, and state space exploration. As we have discussed, there were a lot of different choices that could be made throughout. Which token containers do we use, how do we retrieve the transition stack, what collection was used and so on. The goal of this first section, is to find the best combination of those choices in the general case.

The next section, will be a comparison with LoLA. Our software can solve problems that LoLA can not, and vice versa. But how do we compare when it comes to the reachability problem, and retrieving one firing sequence from an initial marking to a target marking? Most likely, our solution will be a lot slower than LoLA in most cases. However, we do not know in which cases that we have a comparable performance to LoLA and where we do not. Finding those cases, could reveal where our software has weakness, and what optimizations could be added in future works to improve it.

The last section, will benchmark the generation of occurrence nets. We will try to figure out, what type of Petri nets are expensive to produce occurrence nets for and which are not. We will also run some benchmark for instances from MØD.

10.1 Token container

10.2 Benchmarking Collections

As we know, the collections have 2 jobs. Store the different markings as they are seen for the first time, and allow for lookup of all previous seen markings. Given this fact, both running time and space used is interesting from a benchmarking perspective. What we would like to learn, is which collection is the fastest and which is the most memory efficient and if there is a tradeoff. We will not be examining the `Dory_collection` in benchmarks of this section, since that collection is not useful in most cases, as we have seen. This leaves us with the `Collection_map` as our collection to benchmark. As a reminder, it stores the given marking as in either a `map` or `unordered_map`. It has 2 template parameters, `Input_container` and `Stored_container`. The two types can either be the same, which means that no compression of the `Token_containers` that is given as input will be taking place. If the `Input_container` is the same as the `Token_container`, then we can choose to compress it to a `Sparse_Vector`. This gives us a bunch of options to test for. The goal of these benchmarks is to figure out what the best combination is in general.

The first benchmark, will be comparing storing the `Input_container` in a `map` vs `unordered_map`. Here we want to learn, which of the two is faster at storing and retrieving the stored `Token_containers`, since there is some disagreement depending on the specific setting. Since we have already established that only the `Vector_marking` is relevant going forward, we will only

use it as the `Input_container`, and we will not compress in this test, so the `Stored_container` will be the same. Just to justify this decision further, how fast `Collection_map` will be, is heavily dependent on how fast the `Stored_container` can be either hashed or sorted. Both rely on the iterators of the `Stored_container`'s. Since both `map` and `unordered_map` has a slower iterator than a `vector`, there is not much reason to believe that `Map_marking` will ever outperform `Vector_marking`.

After we have found the fastest of the two, we would like to know if it is better to store our markings as `Vector_marking` or as a `Sparse_marking`.

There is no reason to believe, that

11 Discussion

12 Conclusion

References

[1] Jakob Lykke Andersen. "Analysis of Generative Chemistries". In: (2016).

[2] Roberto Bruni and Hernán Melgratti. "Non-sequential Behaviour of Dynamic Nets". In: *Petri Nets and Other Models of Concurrency - ICATPN 2006*. Ed. by Susanna Donatelli and P. S. Thiagarajan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 105–124. ISBN: 978-3-540-34700-2.

[3] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[4] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st ed., Reprint. Addison-Wesley Longman, Amsterdam, 1994. ISBN: 0201633612.

[5] Robert Jan van Glabbeek. "The Individual and Collective Token Interpretations of Petri Nets". In: *CONCUR 2005 – Concurrency Theory*. Ed. by Martín Abadi and Luca de Alfaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 323–337. ISBN: 978-3-540-31934-4.

[6] Shu-Ming Hsieh, Chiun-Chieh Hsu, and Li-Fu Hsu. "Efficient Method to Perform Isomorphism Testing of Labeled Graphs". In: *Proceedings of the 2006 International Conference on Computational Science and Its Applications - Volume Part V*. ICCSA'06. Glasgow, UK: Springer-Verlag, 2006, pp. 422–431. ISBN: 3540340793. DOI: 10.1007/11751649_46. URL: https://doi.org/10.1007/11751649_46.

[7] T. Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580. DOI: 10.1109/5.24143.

[8] Karsten Schmidt. "LoLA: A low level analyser". In: vol. 1825. June 2000, pp. 465–474. ISBN: 978-3-540-67693-5. DOI: 10.1007/3-540-44988-4_27.

[9] Karsten Wolf. "Generating Petri Net State Spaces". In: *Petri Nets and Other Models of Concurrency – ICATPN 2007*. Ed. by Jetty Kleijn and Alex Yakovlev. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 29–42. ISBN: 978-3-540-73094-1.