- forsiden er side 1
- forsiden har ikke vist sidetal
- sidetal til og uden indholdsfortegnelse skrives med romertal
- indholdsfortengelsen resetter sidetal til side 1, og der bruges alm. arabiske tal til visning derfra
- abstract kommer før indholdsfortegnelse, brug \begin{abstract} til det
- der er et formelt krav til at have et dansk "Resumé" når du skriver på engelsk
  forslag: skriv et engelsk abstract og oversæt det til sidst (og hav begge i rapporten)

Master thesis

Rasmus Bartholin

May 2023

ang. struktur: det skal gerne flyde fra "ting andre har lavet, men jeg gengiver" over i "ting jeg har lavet, baseret på andres" og ende i "ting jeg har lavet"
tommelfingerregel: en ekspert (vejleder/censor) skal kunne få et godt overblik over hvad du har lavet, blot ved at skimme indholdsfortegnelsen

Contents

1 Abstract

2 Introduction

der bør ikke være overskrifter lige efter hinanden, hav som minimum et par sætninger der sætter "stemningen" for hvad der skal til at ske

3 Theory

3.1 Firing sequences vs paths

An important distinction needs be made between a path in the state space graph and a firing sequences. A path is a monomorphism of the state space graph while a firing sequences, is a sequences of firings from some initial marking to a target marking. The two have a bijective relation to each other. However, each individual firing sequence is much smaller since it only needs to store the transitions, while a path needs to store the transitions and nodes that represents the different markings. In section 3.5 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.

3.2 Explore state space

Algorithm 1: Explore state space

```
Input  : marking, The initial marking
         end_marking, The target marking
stack = [null,(enabled(Marking))]
while stack is not empty do
    (t_from,enabled) = stack.pop()
    if enabled is not empty then
        transition = enabled.pop()
        stack.push((t_from,enabled))
        marking.fire(transition)
        stack.push(transition,enabled(marking))
    else
        if t_from is equal to null then
            continue
        marking.backfire(t_from)
        path.pop()
```

pas på med for meget konkret kode-notation i pseudo-kode oftere kan det være pænere med mere matematisk notation

der er noget med spacing efter kommaerne der ser underlig ud

3.3 Is reachable

3.4 Find a path

3.5 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 1 for a small state space graph.

Figure 1: All paths memory consumption worst case

$$\begin{tikzcd}
m_1 \arrow[r, "t_1"] & m_2 \arrow[r, "t_2"] & m_3 \arrow[r, "t_3"] & m_4 \arrow[r, "t_8"] & m_7 \arrow[r, "t_9"] & m_e \\
m_1 \arrow[r, "t_1"] & m_2 \arrow[r, "t_2"] & m_3 \arrow[r, "t_3"] & m_4 \arrow[r, "t_6"] & m_6 \arrow[r, "t_7"] & m_e \\
m_1 \arrow[r, "t_1"] & m_2 \arrow[r, "t_2"] & m_3 \arrow[r, "t_3"] & m_4 \arrow[r, "t_4"] & m_5 \arrow[r, "t_5"] & m_e
\end{tikzcd}$$

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 3.8. 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 [3]. 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 2 and 3 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 2 and 3 will be stored, and not all paths will be returned. The algorithm is therefore not correct.

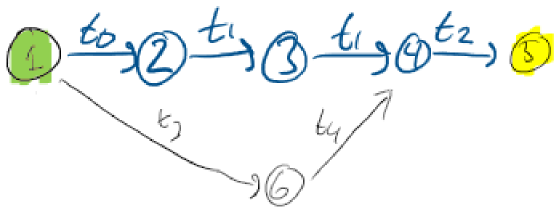


Figure 2: Shortest path will not be saved by *All paths* with `Collection`

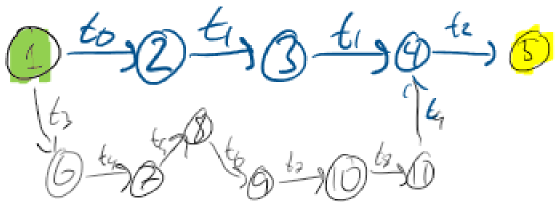


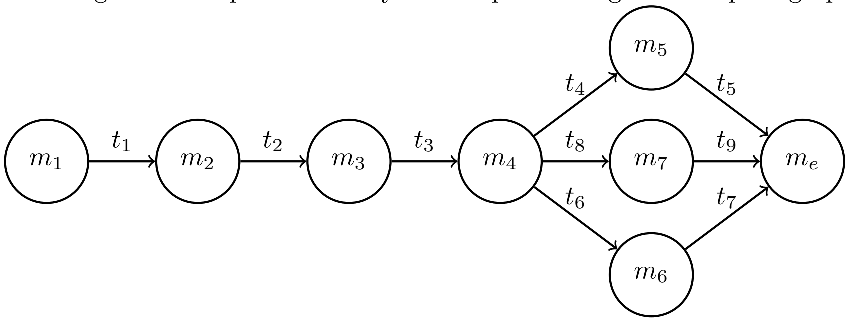
Figure 3: Longest path will not be saved by *All paths* with `Collection`

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` [1]. This and `Collections` will be explored in the next two sections

Figure 4: 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 3.5. 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 [2]. 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.

3.6 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.

3.7 State space exploration with visitor patterns

3.8 Collections

"Section 2", "Section" er en del af navnet, så det skal være med stort, samme med "Figure", "Table", "Algorithm", osv.

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 ~~will be~~ done in this section.

"is", der er der allerede

In LoLA a collection has two operations, `search()` and `insert()`[3]. 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 we do not explore that marking again. We can simply backfire and go to the next entry on the stack. [3].

hvad betyder det?

figurer er floats i Latex, brug \ref ikke relative lokationer

To demonstrate that this will go well, we can look at the figures below. In figure 5 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.

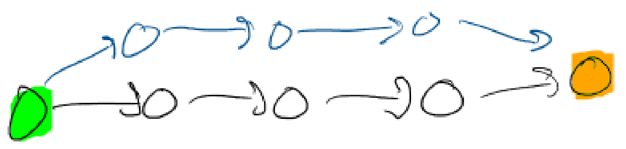
\ref

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. which loop?

d

evt. del Fig. 5 op i to figurer en for hver case

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.



generel note om figurer:
dens tekst (caption) skal være forklarende, ikke blot en overskrift. Fra teksten skal man ikke være i tvivl om hvad figuren forestiller, men det kan godt være at man ikke får en dybere analyse med. Hovedteksten hvor figuren refereres skal have denne dybere mening. Det er ok hvis man logisk set sige lidt af det samme begge steder.

Figure 5: New path to add, shares nothing with the current state space graph

vær konsitent med font

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 [3]. 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:

pas på med terminologi, vi snakkede om på et tidspunkt at bruge "Marking", og så noget a la "ReadableMarking" for dem man kan læse tokens ud fra. "Marking-container" kan være svær at forstå hvad betyder.

"CollectionMarking"? lad os snakke om hvilke typer markings der er

måske lidt i overkanten at bruge en definition

Definition 1 *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.*

start med to backticks `, slut med to single-quotes '

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.

men størrelsen på en marking påvirker tiden for search, så det er ikke trivielt

Now that we have defined what a Collection needs to hold, let's define the Collection itself.

undgå alle contractions

nu hedder den try_insert?

font

igen måske lidt meget med en definition

Definition 2 *A collection is a container that holds collection-containers. It has a single operation* `try_insert()` *that takes a marking container 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.*

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, generelt et ganske godt afsnit

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~~ describe it [3].

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 4. Section~\ref{sec:blah}

3.9 Collection-containers

4 Building the state space graph

4.1 Building the state space graph with cycles

Call this APC

4.2 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.

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.

5 Implementation

5.1 Finding all the path to an end marking

5.2 Building the state space graph

5.3 Using type traits, to choose the correct Transition stack

6 Testing

7 Discussion

8 Conclusion

References

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

[2] 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.

[3] 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.