1 View Examples

In this chapter we will introduce and discuss some examples created by the author. Their purpose is to understand the idea and concept of a get to know some views that might be useful in real world applications.
When considering we only want into account those that utilize properties of MDPs or that do computations that are also feasible on normal graphs but are of explicit relevance MDPs.

1.1 Views Utilizing MDP Components

In this subsection we will introduce some views the are purely based on the components of an . Their will neither be computations on the graph-structure of an MDP nor computations using the result vector.

1.1.1 Atomic Propositions

One of the least involved approaches to create a view is to base it on the atomic propositions that are assigned to each state by the labeling function. The notion is to group states that were assigned the same set of atomic propositions. Why useful?, currently no "has AP" -¿ maybe should be added
Let = be . The view is defined by its grouping function with $:\to, \mapsto ()$.
The grouping function is exactly the labeling function i.e. for all ∈ it is () := ().
So it is $(_1) = (_2) \iff (_1) = (_2)$. According to Definition ?? for it is $[] = \{\in| () = ()\}$.
By this we obtain the for a given where: $' = \bigcup_{\in}\{\} = \bigcup_{a\in}\{\{\in| () = a\}\}$. All other components are constructed as in Definition ??.
Although this view might seem rather simple because essentially it only performs : it is the most powerful one. This is because every view presented in the following is reducible to this one. That is because a essentially asserts an atomic proposition to every state, namely the value in respect to the considered property of a given view. The reduction of the labeling function can be realized by replacing the labeling function with the grouping function of the respective view. That is := and := () for some . While this works it alters the underlying .

1.1.2 Initial States

An a little more involved idea than directly using a given function is to utilize the set of initial states. We can group states that have a probability greater zero, that they are started from. In practice this might be useful to quickly find all initial states.
Let = be and := {∈|∈}. The view is defined by its grouping function with :→ with

$$\mapsto \{ \emptyset, if \in, \quad , otherwise$$

and := {∅}∪.
For $_{1,2}\in$ it is $(_1) = (_2)$ $_{1,2}\in$ or $_1 =_2$. According to Definition ?? it is
{∈| () = ∅} for ∈ and
= {∈| () = {}} ∧ {} for ∉ .
By this we obtain the for a given where: $' = \bigcup_{\in}\{\} = \{\in|\in\} \cup \bigcup_{\in\setminus}\{\{\}\}$.
All other components are constructed as in Definition ??.

1.1.3 Outgoing Actions

define outgoing transition and outgoing action The *OutAction View* groups states that share some property regarding their actions of the outgoing transitions. Several variants are feasible. In the following we will use the expression "outgoing action " equivalent with "transition with outgoing action ".
The most obvious variant to group states is to group states that *have* a given outgoing action.
Let = be and ∈. The view is defined by its :→ with

$$\mapsto \{ , if \exists' \in: (, , ') \in \quad , otherwise$$

and :=.
For $_{1,2}\in$ it is $(_1) = (_2)$ there exist $_{a,b}\in$ with $(_1, _a), (_2, _b) \in$ (i.e. they have the same outgoing action) or $_1 =_2$. In accordance with Definition ?? it is
= {∈| ∃′ ∈: (, , ′) ∈}
= {∈| () = } = {s} otherwise
Thereby we obtain the for a given where $' = \bigcup_{\in}\{\} =:_1 \cup _2$ where $_1 := \{\in| \exists' \in: (, , ') \in \} = \{\in| has outgoing action\}$ and
$_2 :=$.
Since actions are a very important part of as well as of its more powerful siblings MDPs and MCs it seems useful to further enhance this view and look at variants of it. Instead of only grouping states that only *have* outgoing actions we could also quantify the amount of times that an outgoing action should be outgoing.
For example we could require that a given action has to be outgoing a minimum amount of times.
Let = be and ∈. The view is defined by its :→ with

$$\mapsto \{ , if \exists_1, \dots, _{\in} : \quad , otherwise$$

where :=, ∈ is the minimum amount of times a transition with action has to be outgoing in order to be grouped with the other states and

$$:= ((, , _1), \dots, (, ,) \in) \wedge |\{_1, \dots, \}| =$$

is a first order logic predicate.
The number and The predicate requires that there are transitions with action to distinct states.
For $_{1,2}\in$ it is $(_1) = (_2)$ there exist distinct $a_1, \dots a_{\in}$ and distinct $b_1, \dots b_{\in}$ so that $(_1, _{a_1}), \dots, (_1, _{a_{\in}})$ and $(_2, _{b_1}), \dots, (_2, _{b_{\in}}) \in$ or $_1 =_2$. According to Definition ?? it is {∈| () =} if
= {∈| () = } = {} otherwise
By this we obtain the for a given where $' = \bigcup_{\in}\{\} =:_1 \cup _2$ where
$_1 := \{\in| \exists_1, \dots \in :\} = \{\in| the action is outgoing at least times\}$ and
$_2 :=$.
In a similar fashion we define view that groups states at most a certain number of times a given action is outgoing.
Let = be and ∈. The view is defined by its :→ with

$$\mapsto \{ , if \forall_1, \dots, _{+1} \in: \quad , otherwise$$

where := ∪, ∈ is the maximal number of times a transition with action may be outgoing and

$$:= ((, , _1), \dots, (, , _{+1}) \in) \bigvee_{i,j\in\{1,\dots,+1\}:i<j} {}_i =_j$$

is a first order logic predicate.
It ensures that if there are more than outgoing transitions with an action at least two of the states where the transitions end are in fact the same. Since this is required for all possible combinations of +1 states by the grouping function, only states that have at most outgoing actions will be assigned with by the grouping function. The reasoning that the equality of the values, the obtained equivalence classes and the resulting set of states ′ of the view is analogous to .
Since we already defined and hence views for a required minimal and maximal amount of times an action has to be outgoing it is now easily possible to define a view that groups states where the amount of outgoing actions is at least and at most .
Let = be and ∈. The view is defined by its :→ with

$$\mapsto \{ , if \exists_1, \dots, _{\in} : [] and \forall_1, \dots, _{+1} \in: \quad , otherwise$$

where := ∪ and , ∈ are the minimal and maximal number of transitions with action in order for state to be grouped. The predicates and are the predicates from Definition 4.4definition.4.4 and Definition 4.5definition.4.5 respectively.
We already know that for a given ∈ the expressions $\exists_1, \dots, _{\in} : []$ and $\forall_1, \dots, _{+1} \in: []$ from Definition 4.4definition.4.4 and Definition 4.5definition.4.5 require that has minimal and maximal amount of outgoing transitions with an action respectively. Hence the conjunction will be true for states where the amount of outgoing transitions with action is element of the set {, +1, . . . , −1, }. We will write for this that the number of outgoing actions is *in the span.*
For a given state and action we set

$$:= \exists_1, \dots, _{\in} : [] \wedge \forall_1, \dots, _{+1} \in:$$

for convenience. is true the number of outgoing actions is in the span. For $_{1,2}\in$ it is $(_1) = (_2)$ $[_1] \wedge [_2]$ or $_1 =_2$. Then its equivalence classes are
= {∈| () =} true
= {∈| () = } = {} otherwise
The new set of states ′ of the view is the union of the equivalence classes of the equivalence relation on the set of states of the original . Hence it is $' = \bigcup_{\in} =:_1 \cup_2$ where
$_1 := \{\in| () =\}$
= {∈| true}
= {∈| the action is outgoing to times} and
$_2 :=$.
The above can be combined with . The thereby obtained requires that the respective conditions of all the combined are met. In this sense it is a conjunctive combination.
Instead of making requirements about states and group them based on whether they meet these requirements it also possible to group states that are very similar or even identical in regard to their outaction. We consider this idea with the in two variants: and (idententiy). Firstly we will consider the variant of strong identity.
Let = be and ∈. The is defined by its :→ with

$$\mapsto \{(,) |\in, is the number of times that is outgoing from\}$$

and $:= \times_0$.
The asserts to each state a set of pairs. Note that a pair is contained int the set for each action ∈. In case there is no outgoing transition from state with an action it is (, 0) ∈. For $_{1,2}\in$ it is $(_1) = (_2)$ $_1$ and $_2$ are mapped to the same set of pairs. By Definition ?? the obtained equivalence classes of are

$$:= \{\in| () = \{(_1, _1), \dots, (_l, _l)\}, l = ||\}$$

According to Definition ?? the set $' := \bigcup_{\in}$ is the set of states of . All other components of are as usual structured in accordance with the Definition ??. As mentioned earlier a variant of the is also conceivable.
Let = be and ∈. The is defined by its :→ with

$$\mapsto \{\in| \exists' \in: (, , ') \in \}$$

and :=. condition of image-set written in inconsistent style to strong identity. Swap strong and weak (order)?
The grouping function maps to the set of outgoing actions of a state and thereby discards information about the number of times an actions is outgoing. If an action is not outgoing from a state it is not contained in the set.
For $_{1,2}\in$ it is $(_1) = (_2)$ they are mapped to the same set of actions. Hence the equivalence classes of are

$$[] = \{\in| () = () =: \{_1, \dots, _l\}, l \in\}.$$

According to Definition ?? the set $' := \bigcup_{\in}$ is the set of states of .

1.1.4 Ingoing Actions

Analogously to Outgoing Actions views of utilizing ingoing actions are feasable. Since there is no difference apart from the definitions itself, we only provide the definitions.
Let = be and ∈. The view is defined by its :→ with

$$\mapsto \{ , if \exists' \in: (', ,) \in \quad , otherwise$$

and := ∪.
Let = be and ∈. The view is defined by its :→ with

$$\mapsto \{ , if \exists_1, \dots, _{\in} : \quad , otherwise$$

where := ∪, ∈ is the minimum amount of times a transition with action has to be ingoing in order to be grouped with the other states and

$$:= ((_1, ,), \dots, (, ,) \in) \wedge |\{_1, \dots, \}| =$$

is a first order logic predicate.
Let = be and ∈. The view is defined by its :→ with

$$\mapsto \{ , if \forall_1, \dots, _{+1} \in: \quad , otherwise$$

where := ∪, ∈ is the maximal number of times a transition with action may be ingoing and

$$:= ((_1, ,), \dots, (_{+1}, ,) \in) \bigvee_{i,j\in\{1,\dots,+1\}:i<j} {}_i =_j$$

is a first order logic predicate.
Let = be and ∈. The view is defined by its :→ with

$$\mapsto \{ , if \exists_1, \dots, _{\in} : [] and \forall_1, \dots, _{+1} \in: \quad , otherwise$$

where := ∪ and , ∈ are the minimal and maximal number of transitions with action in order for state to be grouped. The predicates and are the predicates from Definition 4.9definition.4.9 and Definition 4.10definition.4.10 respectively.
Let = be and ∈. The is defined by its :→ with

$$\mapsto \{(,) |\in, is the number of times that is ingoing from\}$$

and $:= \times_0$.
Let = be and ∈. The is defined by its :→ with

$$\mapsto \{\in| \exists' \in: (', ,) \in \}$$

and :=.

1.1.5 Variables

The concept of variables is not part of the definitions of neither , MCs or even though, it is of great importance in practice. We will introduce and discuss this concept before utilizing it for .
Since states describe some information about a system at a certain moment of its behavior the information carried is usually not atomic but rather consists of several pieces. For instance when considering a computer program at a given moment during execution all its currently available variables will have some value, the stack will have a certain structure and the program counter points to a specific instruction. Systems in general at a certain moment of their behavior have several properties that in total pose the current state of the system. That is in practice each state is actually derived from a possible variable assertion. Choosing the respective variable assertion state representation would result in rather complex state objects. Therefore the values of the variables are usually stored in a separate data structure and a simple identifier like an integer represents the actual state. When formalizing this in practice used approach two options using the already existing components of an come into consideration. Firstly there is the option of declaring the states as complex objects that contain this information. This would be possible because the set of states is not further specified. Accessing this information stored within a state would be rather tedious because there is no already existing entity to properly accomplishing the access. Hence this option was not further explored. The second option is to require that all possible assertions of all variables are included in the atomic propositions. The labeling function would then assert each every atomic proposition for each variable for each state, that specifies what the current value of that variable is. Although this would be possible, for simplicity we chose that variables and its values are provided in addition to the . Obviously they are to be understood as derived and therefore dependent on the system, that the has been derived from. By no means the set of variables and its evaluation are arbitrary.
The set is called *variables.*
Let be an arbitrary set. The function $: \times \to$ is called *variable evaluation function.*
Most of the time we will use to refer to the variable evaluation function. When we speak about the value of a variable in a state we refer to the image of for that state and variable. The set is arbitrary so that arbitrary values can be assigned to a variable. Speaking in terms of computer science and programming this loosens as an example the restriction of only being able to assign numbers and no booleans.
The most apparent idea for a utilizing variables is to group states that meet some requirement regarding the values of the variables.
state has variable not here until now because probably not used
Let

- = be ,
- be the respective set of variables with ∈ and
- (,) = where ∈.

The view is defined by its :→ with

$$\mapsto \{ , if (,) = \quad , otherwise$$

where := ∪{a}.
The view groups states that share the same value for a given variable. For $_{1,2}$ it is $(_1) = (_2)$ $(_1,) = (_2,)$ or $_1 =_2$. The obtained equivalence classes are = {∈| (,) = }
= {∈| () = } = {}
The set of states ′ of the view is the union of the equivalence classes of . It is $' = \bigcup_{\in} =:_1 \cup_2$ where
$_1 := \{\in| () =\}$
= {∈| (,) = } and
$_2 :=$.
Analogously a view that requires inequality instead of equality is feasible.
Let

- = be ,
- be the respective set of variables with ∈ and
- (,) ≠ where ∈.

The view is defined by its :→ with

$$\mapsto \{ , if (,) \neq \quad , otherwise$$

where := ∪{a}.
If states are to be grouped with the requirement of several variables equaling or not equaling specified values this can be achieved by using .
To allow even more flexibility a view can be used that also allows a combination of requirements on variables in a disjunctive manner. To extend this idea to its full potential we will define that a allows requirements using a disjunctive normal formal (DNF). To formalize this view more easily we will write [, kl] = short for $(_{kl}) =$ where k, l ∈ and x_{kl} ∈. In the same sense we write [, kl] ≠. It may be that $x_{ab} = x_{mn}$ for a, b, m, n ∈ NOT HAPPY WITH THIS. We define the symbol to be an element of the set {=, ≠}. That is, whenever it is used with this each time written it is a representative for either the symbol = or ≠. It allows to write one symbol whenever = and ≠ could or should be possible. Moreover for this context we consider ([, kl] =) as a literal and ([, kl] ≠) as its negation. We write ([, kl]) for a literal that could be negated or not negated.
Let

- = be ,
- the respective set of variables
- the respective variable evaluation function and

$$d(s) = (([, 11]_{11}) \wedge \dots \wedge ([, 1l]_{1l_1})) \vee \dots \vee (([, k1]_{k1}) \wedge \dots \wedge ([, kl]_{kl_k}))$$

proposition logical formulae in disjunctive normal form where

- $\{_{kl}| k \in, l \in \{l_1, \dots l_k\} \subseteq\} \subseteq$ and $[, kl] = (_{kl})$
- $\{_{kl}| k \in, l \in \{l_1, \dots l_k\} \subseteq\} \subseteq (,)$

The is defined by its :→ with

$$\mapsto \{ true, if d(s) is true, \quad otherwise$$

where := {true, }.
The DNF allows us to specify a requirement in disjunctive normal form about variables. States are mapped to the same value depending on whether or not they meet this requirement.
DISCUSSION OF EQUALITY, EQ CLASSES AND RESULTING STATES MISSING
Analogously a view based on a conjunctive normal formal can be defined that may be more convenient, depending on the query.
Let = be and

$$c(s) = (([, 11]_{11}) \vee \dots \vee ([, 1l]_{1l_1})) \wedge \dots \wedge (([, k1]_{k1}) \vee \dots \vee ([, kl]_{kl_k}))$$

proposition logical formulae in disjunctive normal form where

- $\{_{kl}| k \in, l \in \{l_1, \dots l_k\} \subseteq\} \subseteq$ and $[, kl] = (_{kl})$
- $\{_{kl}| k \in, l \in \{l_1, \dots l_k\} \subseteq\} \subseteq (,)$

The is defined by its :→ with

$$\mapsto \{ true, if c(s) is true, \quad otherwise$$

where := {true, }.
The only difference from the to the is whether the respective formulae is in conjunctive or disjunctive normal form.CITATION Since each formulae in conjunctive normal form can be transformed to a formulae in disjunctive normal form and vice versa neither one of the can perform an action the other can not. Hence there is no difference in expressivity, but there may be one in size. Therefore both views have been implemented and formalized.
The views discussed before reduce the in a very precise but also manual manner, because it not only dictates the variable but also its value. A more general approach is to stipulate only the variable but not its value. This way states will be grouped that have the same value for that variable with no regard to the actual value of that variable. This idea could be achieved with a view based on the [1]
Let = be . The view is defined by its :→ with

$$\mapsto (,)$$

and := (,).
With this grouping function we directly map to the value of the variable. Hence for $_{1,2}\in$ it is $(_1) = (_2)$ they are mapped to the value ∈. Hence the equivalence classes of are

$$[] = \{\in| () = () = \{\}\}.$$

According to Definition ?? the set $' := \bigcup_{\in}$ is the set of states of .

1.2 Utilizing the MDP Graphstructure

1.2.1 Distance

Implementation Algorithm Let = be and ⊆ arbitrary. IMPLEMENTATION IN PSEUDOCODE? DEFINITION BEFORE? The is defined by its :→ with

$$\mapsto \quad where (,) \in$$

and $M = \times(\cup\{inf\})$.
Due to the implementation of for every state ∈ exists a pair (,) in the returned set of . That is it is there for every state ∈ there exists (,) with (,) ∈. The view groups states that have the same distance to the set measured with the amounts of transitions necessary reach the the closest considering the granularity .

1.2.2 Double Directed Edges with same Action

NOT SURE IF SHOULD BE INCLUDED Let = be .

1.2.3 Cycles

A cycle is a structure that can exist in every graph. The concept of cycles is not specific to or any of its more specialized variants. The purpose of this thesis is to discuss views that utilize domain specific knowledge or if a general concept is of special relevance when exploring . The former and the latter apply on cycles. Formalizing based on cycles requires some formalization of the concept cycle. We will use a domain specific definition that will serve us the most.
Let = be . A (simple) *cycle* in is any sequence alternating between states and actions where $n \in$, $\{_0, \dots, _{n-1}\} \subseteq$ a set of distinct states, $\{_0, \dots, _{n-1}\} \subseteq$ and for all $i \in \{0, \dots, n-1\}$ it is $(_i \, _{i+1 \bmod n}) \in$.
When the actions in the cycle are of no further importance, we will omit them only writing a sequence of states. In the following let = be a cycle. For convenience we will write ∈ if the state is contained in the and ∈ if the action is contained in in the cycle . only if?? In words we will write a state or action is *on* or *in* the cycle. Let $_1$ and $_2$ be cycles. $_1\cup_2 := \{\in|\in_1 or \in_2\}$.
Let be an cycle and be the states set of . The number $n = |\{\in|\in\}|$ is called the *length* of a cycle.
Apart from the formalization of the term cycle and the length of a cycle we will also have to formalize a way to find cycles.
Let = be and n ∈. The function (, n) is called *cycle finder function* (CFF). It returns a set of all cycles in where for all ∈ it is $|\{\in|\in\}| \geq n$. That is there are at least n states contained in each cycle.
In a programming like manner we will write (, n) and thereby referring to its returned set. This will be clear from the context.
In practice there exist several cycle finding algorithms. The function (, n) is an abstraction for one of these algorithms being used. The actual implementation relies on algorithms of the jave library namely the Algorithm Szwarcfiter and Lauer - $O(V + EC)$ and Tiernan - $O(V.constV)$ CITATION!!
With the formalization done we will introduce some views utilizing the concept cycles. For one we will combine the notion cycle with domain specific knowledge for the other we will consider how and why cycles in MDPs are in general of relevance. We will begin with the latter. Cycles in general are of interest because they pose the risk of getting stuck in endless loops, when performing actions on the . Model checking is one of the most relevant actions on that are vulnerable to loops. They are performed commonly on only on them. Therefore a that simply finds existing cycles is feasible.
Let = be and n ∈. The is defined by its :→ with

$$\mapsto \{ inCycle, if (, n) \neq \emptyset, \quad otherwise$$

and = {inCycle} ∪ .
This groups all states that are contained in cycle with a length of at least n. It is to note that the affinity of a state to one or several respective cycles is lost.
DISCUSSION OF EQUALITY, EQ CLASSES AND RESULTING STATES MISSING
Let = be and n ∈. The is defined by its :→ with

$$\mapsto \{\in (, n) |\in\}$$

and = (, n)
This groups states that have the same set of cycles they are contained in. Thus if C_1 and C_2 are distinct cycles and $_{1,2} \in C_1$ and $_1 \in C_2$ but $_2 \notin C_2$ they are not grouped. It suffices that a state is on one cycle that the other one is not, in order for the states not being grouped. In graphs with many cycles this can lead to little grouping.
DISCUSSION OF EQUALITY, EQ CLASSES AND RESULTING STATES MISSING
GREEDY APPROACHES (STATE ON ONLY ONE SINGLE CYCLE), AND SET APPROACH s -¿ $C_1 \cup C_2$ omitted because not sure if needed
The view above might be useful when having found cycles to see what cycles specifically exist. Often it is interesting to find cycles that consist only of transitions of the same action. The following view accomplishes that.
Let = be and n ∈. The is defined by its :→ with

$$\mapsto \{\in (, n) |\in, \tilde{} \in, \forall \in: := \tilde{}\}$$

and = (, n)
The view specializes the view from Definition 4.28definition.4.28 in the sense that it additionally requires for all ∈ that all actions occurring in the cycle are the same. doubling to above definition?
NO DISCUSSION IF PREVIOUSLY

1.2.4 Strongly Connected Components

Strongly connect components () are of major importance in model checking I think so because I found the terms many times in the book. Remaining question: really? why? Therefore it is feasible to consider a view utilizing strongly connected components. Deviating from most definitions we will define not as a subgraph but only as the set of its nodes. Moreover the definition is written in the terms of .
Let = be . A set of the from $= \{_1 \in| \forall' \in: \exists n \in, \exists(_1 \, _1 \, _2 \, _2 \dots, _n, _n, ')$ where $(_n, _n, ') \in \wedge \forall i \in \{1, \dots, n-1\} : (_i \, _i \, _{i+1}) \in\}$ is called *strongly connected component* of . That is for every state in the set there has to exist a sequence of transitions to every other state of the set. The set of strongly connected components of is denoted with .
Since the strong connection is an equivalence relation the are the equivalence classes and hence disjoint. To find Tarjans algorithm is the classic. In the implementation an improved variant from Gabow is used supplied by the library. CITATION
Let = be . The is defined by its :→ with

$$\mapsto \quad where \in \in$$

and =.
This view groups all states together that are in the same . Because are disjoint each will be mapped to its one and only . This is because... DISCUSSION OF EQUALITY, EQ CLASSES AND RESULTING STATES MISSING
A special kind of strongly connected components is the bottom strongly connected component.
Let be a . A *bottom strongly connected component* () is a where it holds that:
concurrent versions

$$\forall \in: \forall(, , ') \in: ' \in$$

$$\forall \in: t \in (, t) = 1$$

That is from there is no state reachable outside of . The set of bottom strongly connected components of is denoted with .
These are of special relevance because..
Let = be . The is defined by its :→ with

$$\mapsto \quad where \in \in$$

and =.
In the implementation strongly connected components are determined using the algorithm of Gabow, afterwards filtering those that have only transitions to states within the . Equality, equivalence classes and the new state set are constructed analogously to the view .

1.3 Utilizing the MDP Result Table

In this section we will discuss views utilizing the result table. The actual implementation is relies on one powerful that can set to perform arbitrary actions using model checking results.
needhelp Understanding Cluster