Graph-Based Representation of Automata Cascades with an Application to Regular Decision Processes

Alessandro Ronca, Giuseppe De Giacomo

DIAG, Sapienza University of Rome, Italy
ronca@diag.uniroma1.it, degiacomo@diag.uniroma1.it

Abstract

Cascades allow for expressing any automaton as the composition of some basic automata of a restricted number of kinds. Cascades can be constructed in a completely incremental way, by adding one component at a time. Adding a new automaton amounts to a refinement of the previous cascade. The complexity of the phenomena described by a cascade increases greatly with the number of components because each new automaton has access to the states of the previous automata. Furthermore, components are taken from some well-identified classes of basic automata. We believe such characteristics are of great value. However, the potential of cascades is blocked by the lack of an appropriate representation formalism. We present a graph-based formalism to represent automata cascades, and we demonstrate its effectiveness through an application in Regular Decision Processes, where automata are employed to capture temporal dependencies in the dynamics of a domain.

Introduction

Cascades allow for expressing any automaton as a chain of simple automata where each automaton reads the state of the previous automata in addition to the input. Cascades are at the heart of algebraic automata theory, cf. (Ginzburg 1968), and in particular of the Krohn-Rhodes theory (Krohn and Rhodes 1965), which describes how all automata can be expressed as a cascade of some basic automata. The idea that an automaton should be realised in terms of components having specific functionalities, from whose interaction the behaviour of the overall automaton emerges, can also be found in (Minsky 1967). This is in contrast with the standard representation of automata as transition graphs.

We see cascades as a promising formalism. They allow for building automata out of simple components, taken from classes of basic automata, identified by the Krohn-Rhodes theory. Furthermore, cascades satisfy a valuable *incremental refinement* property. Namely, the range of phenomena described by a cascade can be extended by adding a new component, while leaving the meaning of the previous components unchanged. Finally, every new component can take advantage of what is already captured by the previous components; thus, even a simple component can add a complex meaning. We see the potential value of cascades in a number of applications such as Regular Decision Processes (RDPs) (Brafman and De Giacomo 2019; Abadi and Brafman 2020; Ronca and De Giacomo 2021), a class of non-Markov Decision processes where the dependency on the history is captured by finite-state automata.

Automata cascades are described in an algebraic form, that does not address representation issues. First, the algebraic form is inconvenient as a specification language. In particular, it is a linear structure where every automaton seemingly depends on all the previous ones. This makes it difficult to take advantage of the locality of the dependencies between the automata, in order to build the specification in a modular and incremental way. Second, in the algebraic form, the size of each component grows exponentially with its position in the cascade.

Our contribution is a graph-based representation formalism for *cascades of flip-flops*, a restricted but important class with the expressivity of star-free regular languages. Our representation achieves compactness and ease of specification by connecting flip-flops through Boolean functions, and arranging them into a directed acyclic graph according to their dependencies. In deep-learning parlance, our formalism is a feedforward architecture.

The class of star-free languages is also the expressivity of first-order logic on finite linear orders, as well as linear temporal logic on finite traces (LTLf) (De Giacomo and Vardi 2013). Thus, our formalism can represent all RDPs where the transition and reward functions are specified in LTLf. We present an application to an RDP consisting of a grid domain, showing how several kinds of temporal dependencies can be easily modelled in our formalism.

Preliminaries

Automata

This section introduces basic notions of automata theory, with some inspiration from (Ginzburg 1968; Maler 1990). Note that the perspective is rather different from the traditional one found in, e.g., (Hopcroft and Ullman 1979).

Consider a finite *alphabet* Σ. An *automaton* A is a tuple $\langle \Sigma, Q, \delta, q_{\text{init}} \rangle$ where Q is a finite set of *states*, $\delta : Q \times \Sigma \to Q$ is the (total) *transition function*, and $q_{\text{init}} \in Q$ is the *intial state*. The transition function is extended to strings as $\delta(q, w\sigma) = \delta(\delta(q, w), \sigma)$.

The *transition graph* of A is a directed graph with one vertex for each state, with a distinguished vertex for q_{init}, an edge $\langle q, \delta(q,\sigma)\rangle$ with label σ for each $q \in Q$ and each $\sigma \in \Sigma$.

A *homomorphism* from $A_1 = \langle \Sigma, Q_1, \delta_1, q^1_{\text{init}}\rangle$ to $A_2 = \langle \Sigma, Q_2, \delta_2, q^2_{\text{init}}\rangle$ is a surjective function $\phi : Q_1 \to Q_2$ such that the two identities $\phi(q^1_{\text{init}}) = q^2_{\text{init}}$ and $\phi(\delta_1(q,\sigma)) = \delta_2(\phi(q),\sigma)$ are valid. If ϕ is bijective, then it is an *isomorphism* between A_1 and A_2.

Example 1. *Considering Figure 1, the two automata on the left admits the isomorphism ψ defined as $\psi(0) = 1$ and $\psi(0) = 1$. Intuitively, they are the same automaton, but their state names are swapped. Furthermore, the automata on the left both admit a homomorphism to the automaton on the right. Specifically, they admit the same homomorphism ξ defined $\xi(0) = \xi(1) = 0$. Finally, there is no homomorphism from the automaton on the right to any of the automata on the left, and hence no isomorphism either.*

When there is a homomorphism from A_1 to A_2, we say that A_2 is *captured* by A_1, written $A_2 \sqsubseteq A_1$—the notion may become more intuitive after seeing its implications on the expressivity of the acceptors built on A_1 and A_2, described below. Furthermore, A_2 is *precisely captured* by A_1, written $A_2 \equiv A_1$, when $A_2 \sqsubseteq A_1$ and $A_1 \sqsubseteq A_2$; we also say that the two automata are *homomorphically equivalent*. Finally, A_2 is *properly captured* by A_1, written $A_2 \sqsubset A_1$, when $A_2 \sqsubseteq A_1$ and $A_1 \not\sqsubseteq A_2$.

An automaton $\langle \Sigma, Q, \delta, q_{\text{init}}\rangle$ can be made into an *acceptor* by adding a function $\theta : Q \to \{0,1\}$ indicating the set of *accepting states*. A string x is accepted if $\theta(\delta(q_{\text{init}}, x)) = 1$, and the set of accepted strings is called the *language* recognised by the acceptor. Two acceptors are *equivalent* if they recognise the same language. An automaton A_1 is *at least as expressive as* an automaton A_2 if for every acceptor built on A_1 there is an equivalent acceptor built on A_2; if the converse holds as well, then they are *equally expressive*. The connection with homomorphisms is as follows:

1. $A_1 \sqsubseteq A_2$ implies that A_2 is at least as expressive as A_1;
2. $A_1 \equiv A_2$ implies that A_1 and A_2 are equally expressive.

The *expressivity* of an automaton is the set of languages recognised by acceptors built on the automaton, and the *expressivity of a class* of automata is the set of languages recognised by acceptors built on the automata in the class. The above notions can be easily generalised to *transducers* (see below) by allowing θ to be an output function over an alphabet Γ; namely $\theta : Q \to \Gamma$. Having in mind that such a generalisation is possible, we omit it, and we focus on the simpler case of acceptors.

Transducers. We follow (Moore 1956). A *transducer* is a tuple $\langle \Sigma, Q, \delta, q_{\text{init}}, \Gamma, \theta\rangle$ where: Σ, Q, $\delta : Q \times \Sigma \to Q$, and q_{init} are as above; Γ is the finite output alphabet; $\theta : Q \to \Gamma$ is the output function. We extend the use of δ to strings of length greater than one as $\delta(q, \sigma_1\sigma_2\dots\sigma_n) = \delta(\delta(q,\sigma_1), \sigma_2\dots\sigma_n)$, and to the empty string as $\delta(q,\varepsilon) = q$. We also extend the use of θ to arbitrary strings as $\theta(q, \sigma_1\dots\sigma_n) = \theta(q)\,\theta(\delta(q,\sigma_1), \sigma_2\dots\sigma_n)$ where the base case is $\theta(q,\varepsilon) = \theta(q)$. We call $\theta(q_{\text{init}}, \sigma_1\dots\sigma_n)$ *the output of the transducer* on $\sigma_1\dots\sigma_n$.

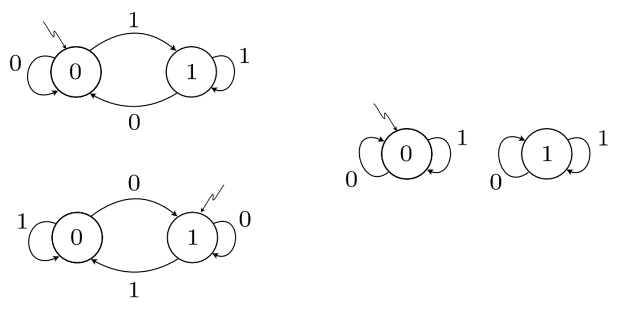

Figure 1: Three simple automata represented as transition graphs.

Automata Cascades

The notions in this section can be found in (Ginzburg 1968). However, here they are presented in a way that aims at keeping the terminology and notation from semigroup theory to a minimum. Given two automata

$$A_1 = \langle \Sigma, Q_1, \delta_1, q^1_{\text{init}}\rangle, \quad A_2 = \langle \Sigma \times Q_1, Q_2, \delta_2, q^2_{\text{init}}\rangle,$$

their *cascade product* $A_1 \ltimes A_2$ is $\langle \Sigma, Q_1 \times Q_2, \delta, q_{\text{init}}\rangle$ where $q_{\text{init}} = \langle q^1_{\text{init}}, q^2_{\text{init}}\rangle$ and the transition function is

$$\delta(\langle q_1, q_2\rangle, \sigma) = \big\langle \delta_1(q_1,\sigma), \delta_2(q_2, \langle \sigma, q_1\rangle)\big\rangle.$$

Note that the second automaton, reading input $\langle \sigma, q_1\rangle$, has access to the state q_1 of the first automaton, in addition to the input σ. Considering the cascade product as left-associative, an automaton *cascade* is

$$C = A_1 \ltimes \cdots \ltimes A_n,$$

where $A_i = \langle \Sigma \times Q_1 \times \cdots \times Q_{i-1}, Q_i, \delta_i, q^i_{\text{init}}\rangle$. Cascades allow for *incremental refinements*, since

$$C_1 \sqsubseteq C_2 \sqsubseteq \cdots \sqsubseteq C_n,$$

for C_i the cascade $A_1 \ltimes \cdots \ltimes A_i$.

We focus on cacades of flip-flops, or *f-cascades*. A *flip-flop* is a two-state automaton $\langle \Sigma, \{0,1\}, \delta, q_{\text{init}}\rangle$ where Σ is the union of three disjoint sets Σ_{read}, Σ_{set}, Σ_{reset} of inputs intuitively corresponding to *read*, *set*, and *reset* operations. Namely:

- $\delta(q,\sigma) = q$ for every $\sigma \in \Sigma_{\text{read}}$;
- $\delta(q,\sigma) = 1$ for every $\sigma \in \Sigma_{\text{set}}$;
- $\delta(q,\sigma) = 0$ for every $\sigma \in \Sigma_{\text{reset}}$.

A flip-flop is *trivial* if $\Sigma = \Sigma_{\text{read}}$. Intuitively, a flip-flop is a device capable of storing one bit of information. The automata in Figure 1 are flip-flops. The top-left one has an intuitive meaning of its inputs, since $\Sigma_{\text{set}} = \{1\}$ and $\Sigma_{\text{reset}} = \{0\}$. In the bottom-left one the meaning of the inputs is swapped, i.e., $\Sigma_{\text{set}} = \{0\}$ and $\Sigma_{\text{reset}} = \{1\}$. The one on the right is a trivial flip-flop, since $\Sigma_{\text{read}} = \{0,1\}$.

A fundamental result for cascades is the Krohn-Rhodes decomposition theorem (Krohn and Rhodes 1965)—see also (Ginzburg 1968; Maler 1990; Dömösi and Nehaniv 2005). Here we focus on its restricted version that relates f-cascades with counter-free (or aperiodic, or group-free) automata, whose expressivity has been established by (Schützenberger 1965).

Theorem 1 (Krohn-Rhodes for Counter-free Automata)**.** *Every counter-free automaton is captured by a cascade of flip-flops. The converse holds as well.*

Theorem 2 (Schützenberger)**.** *The expressivity of the class of counter-free automata is the star-free regular languages.*

Thus, f-cascades have the expressivity of star-free regular languages. *Star-free regular languages* are the ones expressed by star-free regular expressions, cf. (Ginzburg 1968). It is a central class, also because it is the expressivity of both monadic first-order logic on finite linearly-ordered domains, and linear temporal logic on finite traces (De Giacomo and Vardi 2013).

The following example shows how the size of the components of a cascade grows exponentially.

Example 2. *Consider n flip-flops where the i-th flip-flop is $F_i = \langle \{0,1\}^{n+1}, \{0,1\}, \delta_i, 0\rangle$ with transition function*

$$\delta_i(q, \langle r, b_1, \dots, b_n\rangle) = q, \text{ if } r = 1, \qquad \text{(read)}$$

$$\delta_i(q, \langle r, b_1, \dots, b_n\rangle) = b_i, \text{ if } r = 0. \qquad \text{(set/reset)}$$

Their direct product $F_1 \times \cdots \times F_n$ is a device to store and read words of n bits; in particular, all flip-flops keep the stored bit if $r = 1$, and each G_i stores the current i-th input bit otherwise. Note that each F_i is independent of the others. In any homomorphically-equivalent f-cascade $F'_1 \ltimes \cdots \ltimes F'_n$, every flip-flop F'_i is required to read the state of every preceding flip-flop, making the input alphabet of F'_i exponentially larger than the original input alphabet. Furthermore, there is no canonical form, since every permutation of the n flip-flops is equivalent.

Decision Processes

A *Non-Markov Decision Process* (NMDP), cf. (Brafman and De Giacomo 2019), is a tuple $\mathcal{P} = \langle A, O, R, \mathbf{T}, \mathbf{R}, \gamma\rangle$ where: A is a finite set of *actions*; O is a finite set of *observations*; $R \subseteq \mathbb{R}$ is a finite set of *reward values*; $\mathbf{T} : O^* \times A \times O \to [0,1]$ is the *transition function* which defines a probability distribution $\mathbf{T}(\cdot|h,a)$ over O for every $h \in O^*$ and every $a \in A$; $\mathbf{R} : O^* \times A \times O \to R$ is the *reward function*; $\gamma \in (0,1)$ is the *discount factor*. The transition and reward functions can be combined into the *dynamics function* $\mathbf{D} : O^* \times A \times O \times R \to [0,1]$ which defines a probability distribution $\mathbf{D}(\cdot|h,a)$ over $O \times R$ for every $h \in O^*$ and every $a \in A$. Namely, $\mathbf{D}(o,r|h,a)$ is $\mathbf{T}(o|h,a)$ if $r = \mathbf{R}(h,a,o)$, and zero otherwise. Every element of O^* is called a *history*. A *policy* is a function $\pi : O^* \times A \to [0,1]$ that, for every history h, defines a probability distribution $\pi(\cdot|h)$ over the actions A. Every element of $(AOR)^*$ is called a *trace*. The *dynamics of $\mathcal{P}$ under a policy* π describe the probability of an upcoming trace, given the history so far, when actions are chosen according to a policy π; it can be recursively computed as $\mathbf{D}_\pi(aort|h) = \pi(a|h) \cdot \mathbf{D}(o,r|h) \cdot \mathbf{D}_\pi(t|ho)$, with base case $\mathbf{D}_\pi(\varepsilon|h) = 1$ for ε the empty trace. The *value of a policy π on a history* h, written $\mathbf{v}_\pi(h)$, is the expected discounted sum of future rewards when actions are chosen according to π given that the history so far is h; it can be recursively computed as $\mathbf{v}_\pi(h) = \sum_{aor} \pi(a|h) \cdot \mathbf{D}(o,r|h,a) \cdot (r + \gamma \cdot \mathbf{v}_\pi(ho))$.

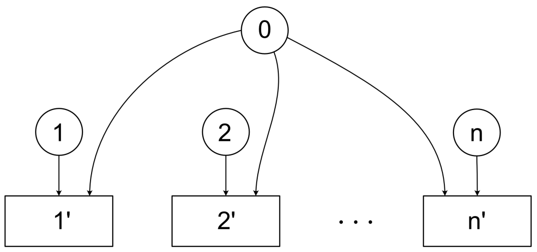

Figure 2: F-graph for a bank of n flip-flops.

The *optimal value on a history* h is $\mathbf{v}_*(h) = \max_\pi \mathbf{v}_\pi(h)$, which can be expressed without reference to any policy as $\mathbf{v}_*(h) = \max_a \left(\sum_{or} \mathbf{D}(o,r|h,a) \cdot (r + \gamma \cdot \mathbf{v}_*(ho))\right)$. A policy π is *optimal on a history* h if $\mathbf{v}_\pi(h) = \mathbf{v}_*(h)$. For $\epsilon > 0$, a policy π is *ϵ-optimal on* h if $\mathbf{v}_\pi(h) \geq \mathbf{v}_*(h) - \epsilon$. A policy is *optimal* (resp., *ϵ-optimal*) if it is so on every history.

A *Regular Decision Process (RDP)* (Brafman and De Giacomo 2019)[1] is an NMDP $\mathcal{P} = \langle A, O, R, \mathbf{T}, \mathbf{R}, \gamma\rangle$ whose transition and reward functions can be represented by a *finite transducer*. Specifically, there is a finite transducer that, on every history h, outputs the function $\mathbf{T}_h : A \times O \to [0,1]$ induced by $\mathbf{T}$ when its first argument is h; and there is a finite transducer that, on every history h, outputs the function $\mathbf{R}_h : A \times O \times R \to [0,1]$ induced by $\mathbf{R}$ when its first argument is h. Note that the cross-product of such transducers yields a finite transducer for the dynamics function $\mathbf{D}$ of $\mathcal{P}$. Optimal behaviour in RDPs can be achieved by acting according the states of the dynamics transducer (Ronca and De Giacomo 2021). Namely, for every $\epsilon \geq 0$, there is a Markov policy on the states of the dynamics transducer such that its composition with the tranisition function of the dynamics transducer is an ϵ-optimal policy for the RDP. The Markov policy can be computed by solving the MDP induced by the dynamics transducer.

Boolean Functions

The Boolean domain is $\{0,1\}$. An n-ary *Boolean function* is $f : \{0,1\}^n \to \{0,1\}$. We will abstract from the representation of Boolean functions. They can be represented using, e.g., Ordered Binary Decision Diagrams (OBDDs) (Bryant 1986, 1992), that are succinct for most practical functions given an appropriate choice of the variable order.

Graph-Based Representation of Cascades

We introduce graph-based representations of f-cascades.

Definition 1 (Syntax)**.** *A* flip-flop graph *(or f-graph) is a directed acyclic graph with vertices V of two kinds. An* input vertex *is labelled by an index and has no parents. A* flip-flop vertex *v (or f-vertex) is labelled by an n-ary Boolean function and has $n-1$ parents.*

[1]In (Brafman and De Giacomo 2019) the functions $\mathbf{T}$ and $\mathbf{R}$ are represented using the temporal logics on finite traces LDL_f. Here instead we use directly finite transducers to express them. Note that all $\mathbf{T}$ and $\mathbf{R}$ representable in LDL_f are indeed expressible through finite transducers.

The structure of an f-graph is shown in Figure 2. In the following definition, all tuples are sorted according to an arbitrary (but fixed) total order on vertices.

Definition 2 (Semantics). *The automaton $\langle \Sigma, Q, \delta, q_{\text{init}} \rangle$ represented by an f-graph is defined as follows.*

- *Alphabet $\Sigma = \{0,1\}^n$ where n is the maximum index of an input vertex.*
- *States $Q = \{0,1\}^m$ where m is the number of f-vertices.*
- *Initial state $q_{\text{init}} = \langle 0, \ldots, 0 \rangle$.*
- *Transition function*

$$\delta(\langle q_{v_1}, \ldots, q_{v_m} \rangle, \langle x_1, \ldots, x_n \rangle) = \langle q'_{v_1}, \ldots, q'_{v_m} \rangle,$$

where each q'_{v_i} is recursively defined as follows. First, for v an input vertex with index j, let $q_v = x_j$. Then, for v_i an f-vertex with label f,

$$q'_{v_i} = f(q_{v_i}, q_{v_{i_1}}, \ldots, q_{v_{i_r}}),$$

where $v_{i_1}, \ldots, v_{i_r}$ are its parents.

An f-graph G homomorphically represents *an automaton A if the automaton represented by G precisely-captures A.*

Example 3. *Consider the f-graph in Figure 2 where f-vertex i' has the label f_i defined as $f_i(q, x_0, x_i) = q$ if $x_0 = 1$, and $f(q, x_0, x_i) = x_i$ if $x_0 = 0$. The f-graph homomorphically represents the f-cascade of Example 2.*

F-graphs are a compact representation of f-cascades.

Theorem 3. *Any f-cascade C is homomorphically-represented by an f-graph whose size is the logarithm of the size of C plus the size of the labels.*

Proof. Consider an f-cascade $C = F_1 \ltimes \cdots \ltimes F_n$ on alphabet $\{0,1\}^m$. We assume w.l.o.g. that the initial state of each F_i is 0—renaming states always yields an equivalent automaton. Let δ_i be the transition function of F_i. The f-cascade is precisely-captured by the f-graph G consisting of m input vertices and an f-vertex v_i for each F_i, with label

$$f_i(x_0, x_1, \ldots, x_k) = \delta_i(x_0, \langle x_1, \ldots, x_k \rangle),$$

and with parents all input vertices and all f-vertices v_j for $j < i$. Finally, observe that C has size exponential in n since the input alphabet of each F_i is exponential in i. On the contrary, the f-graph G has size polynomial in the number n of components of the cascade. □

Application to Regular Decision Processes

In this section we show how f-graphs can be used to represent regular decision processes. We do it through a running example. We first present the basic features of a grid domain, and then we introduce more features incrementally. The grid domain is shown in Figure 3. The corresponding f-graph is shown in Figure 4.

Basic grid. Consider a 8×4 grid where an agent has to move from the bottom left corner, position $(0,0)$, to the top right corner, position $(7,3)$. The agent's actions are $A = \{\text{up}, \text{down}, \text{left}, \text{right}\}$. They have their intuitive meaning, but the agent may fail to move into the chosen direction with probability 0.1. When an action fails, the agent moves to its previous position. The agent can observe its current position. Specifically, at every step, it receives an observation consisting of the 5-bit encoding of the current coordinates. The agent is rewarded when it reaches the destination. Note that for this basic setting, the colours shown in the grid figure are not relevant.

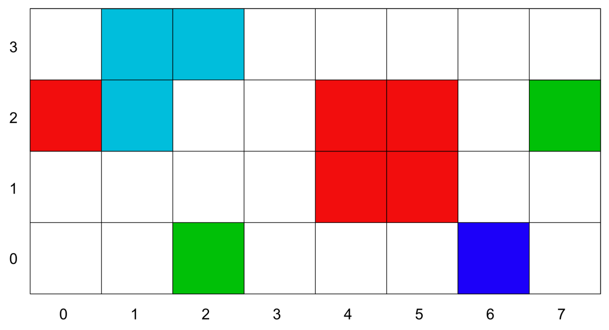

Figure 3: Grid environment.

The dynamics transducer has a state $q_{(i,j),(i',j')}$ for each pair of neighbouring grid positions (i,j), (i',j'). From state $q_{(i,j),(i',j')}$, it transitions to $q_{(i',j'),(i'',j'')}$ when it reads an input that encodes (i'',j''). Indeed, observe that knowing the current and previous positions suffices to describe the conditional probability of the next observation-reward for any given action. Now, consider the black subgraph of the f-graph in Figure 4, consisting of vertices 1–5 and 10–14. Vertices 1–5 the input vertices for the 5-bit encoding of the current position. Vertices 10–14 are f-vertices, all having the same label f_{store} defined as $f_{\text{store}}(q, x) = x$, that simply stores the state of the parent. Then, the automaton represented by the black f-graph precisely-captures the automaton of the dynamics transducer. In particular, the input vertices store the current position, and the f-vertices store the previous position.

Position sequences. We make it more difficult for the agent to obtain a reward. It is still rewarded when it reaches position $(7,3)$, but only if it has visited positions $(2,3), (1,3), (1,2)$ exactly in the specified order—note that they are the cyan cells in the figure. It is not valid to interleave the sequence with other positions. For instance, $(2,3), (1,3), (0,3), (1,3), (1,2)$ is not a valid sequence. Consider the f-graph for the basic grid, extended with f-vertices 15, 16, and 17, having labels f_{15}, f_{16}, f_{17} defined as follows:

- $f_{15}(q, x_1, \ldots, x_5) = 1$ if $x_1, \ldots, x_5$ encode $(2,3)$, and $f_{15}(q, x_1, \ldots, x_5) = 0$ otherwise;
- $f_{16}(q, x_1, \ldots, x_5, x_{15}) = 1$ if $x_1, \ldots, x_5$ encode $(1,3)$ and $x_{15} = 1$, and $f_{16}(q, x_1, \ldots, x_5, x_{15}) = 0$ otherwise;
- $f_{17}(q, x_1, \ldots, x_5, x_{16}) = 1$ if $x_1, \ldots, x_5$ encode $(1,2)$ and $x_{16} = 1$, and $f_{17}(q, x_1, \ldots, x_5, x_{16}) = q$ otherwise (i.e., propagate current state).

Then, the state of vertex 17 is 1 iff the agent has completed the sequence.

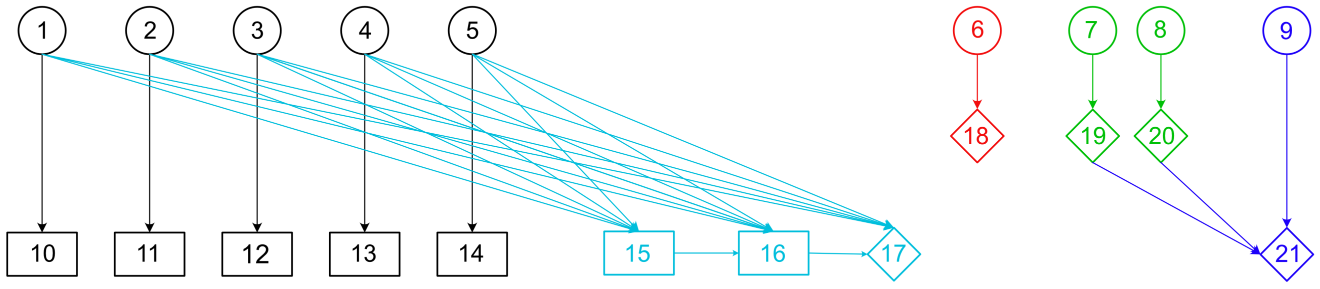

Figure 4: F-graph of the dynamics transducer for the grid environment.

Positions to avoid. Let us extend the observations with an additional bit, that takes value 1 for grid positions that the agent must avoid. These positions are the red ones in the figure. If the agent visits any of the red cells, then it is no longer rewarded. Consider the red subgraph in Figure 4, consisting of vertices 6 and 18. Vertex 6 is the input vertex for the additional bit. Let the label f_{18} of vertex 18 be defined as $f_{18}(q, x_6) = 1$ if $x_6 = 1$, and $f_{18}(q, x_6) = q$ otherwise. Then, the state of vertex 18 indicates whether the agent has ever visited a red cell.

Positions to visit. Let us further extend the observations with two additional bits. Each of them indicates whether the agent is in one of the two green cells, respectively. The agent is rewarded only after having visited both cells, in any order. Consider the green f-graph, consisting of input vertices 7 and 8 for the two additional bits, and f-vertices 19 and 20. Let the label f_{19} of vertex 19 be defined as $f_{19}(q, x_7) = 1$ if $x_7 = 1$, and $f_{19}(q, x_7) = q$ otherwise; similarly, let the label f_{20} of vertex 19 be defined as $f_{20}(q, x_8) = 1$ if $x_8 = 1$, and $f_{20}(q, x_8) = q$ otherwise. Then, the states of 19 and 20 can be used to check whether the agent has visited the two green cells.

Positions to visit in order. Finally, let us extend the observations with one bit that indicates whether the agent is in the purple cell. The agent is rewarded only if it visits the purple cell after having visited both green cells. Consider the green f-graph together with the purple one. Let the label f_{21} of vertex 21 be defined as $f_{21}(q, x_{19}, x_{20}, x_9) = 1$ if $x_{19} = x_{20} = x_9 = 1$, and $f_{21}(q, x_{19}, x_{20}, x_9) = 1$ otherwise. Then, the state of vertex 21 can be used to check the required condition and issue the reward accordingly.

Conclusion

We see f-graphs as enabling the incremental construction of representations of complex automata. We note that f-graphs are in line with 'symbolic' representations of automata (Coudert, Madre, and Berthet 1990; Burch et al. 1992; McMillan 1992). In particular, they can be seen as an approach to building such representations.

References

Abadi, E.; and Brafman, R. I. 2020. Learning and Solving Regular Decision Processes. In *IJCAI*.

Brafman, R. I.; and De Giacomo, G. 2019. Regular Decision Processes: A Model for Non-Markovian Domains. In *IJCAI*.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8): 677–691.

Bryant, R. E. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3): 293–318.

Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; and Hwang, L. J. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2): 142–170.

Coudert, O.; Madre, J. C.; and Berthet, C. 1990. Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams. In *CAV*, 23–32.

De Giacomo, G.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI*, 854–860.

Dömösi, P.; and Nehaniv, C. L. 2005. *Algebraic Theory of Automata Networks: An Introduction*. SIAM.

Ginzburg, A. 1968. *Algebraic Theory of Automata*. Academic Press.

Hopcroft, J.; and Ullman, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Krohn, K.; and Rhodes, J. 1965. Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines. *Trans. Am. Math. Soc.*, 116: 450–64.

Maler, O. 1990. *Finite Automata: Infinite Behaviour, Learnability and Decomposition*. Ph.D. thesis, The Weizmann Institute of Science.

McMillan, K. L. 1992. *Symbolic Model Checking: An Approach to The State Explosion Problem*. Ph.D. thesis, Carnegie Mellon University.

Minsky, M. L. 1967. *Computation: Finite and Infinite Machines*. Prentice-Hall.

Moore, E. F. 1956. Gedanken-experiments on sequential machines. *Automata Studies*, 34.

Ronca, A.; and De Giacomo, G. 2021. Efficient PAC Reinforcement Learning in Regular Decision Processes. In Zhou, Z., ed., *IJCAI*.

Schützenberger, M. P. 1965. On Finite Monoids Having Only Trivial Subgroups. *Inf. Control.*, 8(2): 190–194.