Weighted ω-automata with bounds

Ugo Thay[1], Philipp Schlehuber-Caissier[2], and Sven Dziadek[2]

[1]ENSIIE, Évry
{surname}.{name}@ensiie.eu
[2]Télécom SudParis, Évry
{surname}.{name}@telecom-sudparis.eu

Guillaume Burel[3]

[3]ENSIIE, Évry (internship tutor)
{surname}.{name}@ensiie.fr

Third year (Masters) internship
March 20, 2025 —September 19, 2025

Acknowledgements

First and foremost, I would like to thank Philipp Schlehuber-Caissier and Sven Dziadek for accepting to be my supervisor for these six months at Télécom SudParis, accompanying me through this internship, pointing out possible improvements in our implementations and helping me proofread this document.

Many thanks also to François Tessier, from the KERDATA team at Inria Rennes, who was also my internship supervisor during my time at KERDATA and without whom I would have probably not developed a strong interest in research.

I would also like to thank all the people I met at Télécom SudParis, Télécom Paris and during my brief visit at ENS Saclay, especially Dimitri, Zineddine, Montassar, Aoki, and all the people at the RST department at Télécom SudParis for the good times during lunch and coffee breaks, and Krzysztof Ziemański for his interesting (post-internship…) talk on presheaf automata and the discussions we had at ENS Saclay.

Last but not least, I want to thank my family for supporting me throughout this internship in every way they could.

Abstract

We implement existing algorithms from the literature to solve parity energy problems in weighted Büchi automata, and propose dedicated ones based on these algorithms to solve more efficiently specific types of ω-energy problems, such as in automata with Rabin or co-Büchi acceptance conditions. As a means to solve co-Büchi ω-energy problems, we also elaborate on the notion of *energy functions*, prove that it is possible to use the Floyd-Warshall algorithm on automata weighted with such functions to attain similar results as algorithms derived from the literature, and compare the performance of this algorithm with these other co-Büchi problem solvers.

Keywords: weighted automata, Büchi automata, ω-energy problems, energy functions, Floyd-Warshall algorithm

Contents

1 Introduction **3**

2 Solving for Büchi, parity and others **7**
2.1 Büchi automata and parity energy problems 7
2.2 Rabin condition 10
2.3 $\top$ condition (monitor) 12

3 Co-Büchi ω-automata **12**
3.1 A first algorithm 12
3.2 Refinements towards an optimized algorithm 13
3.3 Energy functions 18

4 Application of energy functions to energy problems **31**
4.1 Mutators 31
4.2 Floyd-Warshall algorithm with energy functions 36

5 Implementation of energy functions **37**
5.1 Structure 37
5.2 Performance and optimization 41

6 Perspectives **46**
6.1 Towards a generalization of $\mathrm{mut}_{EF(WU)}$? 47

Appendices **50**
A About Télécom SudParis 50
B Sustainable and Socially Responsible Development (DD&RS) 50
B.1 Green Transition and Social Transition Master Plan 50
B.2 A heavy reliance on AI technologies 50

List of Figures

1 State machine of a washing machine. 3
2 An acceptor on $\Sigma = \{a, b\}$ that accepts words starting with an a and ending with a b, i.e. the language $a\Sigma^*b$. 3
3 A weighted automaton that recognizes some city names equipped with probabilistic weights. For example, the word "Paris" has a higher weight than the word "Torcy". Labels have been condensed for readability (the label "oissy" would need to be separated into the letters "o", "i", "s", "s", "y"). 4
4 A Büchi automaton on $\Sigma = \{a, b\}$ that accepts infinite words that do not contain the letter b, i.e. the language a^ω. 5
5 An automaton accepting words that contain infinitely often the letter a with eventually some b between instances of a, i.e. the language $(b^*a)^\omega$). 5
6 A generalized Büchi automaton accepting words that contain an infinite amount of a and b, i.e. the language $b^*(a^+b^+)^\omega$). 5
7 On first iteration, the energy attained after traversing the (1,2,3) loop will be lower than when entering for the first time state 1 (from 0 with 10 energy) despite it being a non-negative loop. 15
8 An automaton with one energy-feasible loop and several energy-negative loops. This example is inspired from Figure 6 in [8]. 15

9 The loop $(1,2)^\omega$, is only accessible if the initial energy is between 5 and 10 because of the transition going from 0 to 1. . . . 18
10 Due to the $WU = 10$ constraint, the transition going from 3 to 0 is unusable. . . . 19
11 An automaton with two different positive loops. . . . 19
12 Examples of energy segments. . . . 20
13 F_{test}, a fully defined energy function. . . . 21
14 $\oplus$ operation on two non-intersecting energy segments. . . . 25
15 $\oplus$ on energy segments is commutative. . . . 25
16 $\oplus$ operation on two intersecting energy segments. . . . 25
17 $\times$ operation on an energy segment and an energy function. . . . 26
18 Equivalent energy function for a transition going from 1 to 2 with weight -3. . . . 33
19 Equivalent energy function for a transition going from 4 to 1 with weight 6. . . . 33
20 Energy function associated with transitions from 1 to 3. . . . 34
21 Energy function associated with transitions from 1 to 4. . . . 34
22 An automaton with an energy-neutral loop symbolized by an energy point in the energy function associated with the optimal energy path from 1 to 1. Here, $WU = 10$. 35
23 General form of the nested loops automaton with n loops. . . . 41
24 Automaton with five nested loops. . . . 42
25 General form of the "stairs" automaton with n loops. . . . 42
26 "Stairs" automaton with three loops. . . . 43
27 Execution times for various co-Büchi solving techniques in (top to bottom, left to right): the nested loops automaton, the non-feasible nested loops automaton, the "stairs" automaton and the circling automaton. . . . 44
28 Graphical representation of the `cProfile` profiler run on the Floyd-Warshall based co-Büchi solver on the nested loops automaton with 11 loops. . . . 45
29 Our modified Floyd-Warshall algorithm applied to a Büchi automaton. The energy function from 2 to itself is depicted. . . . 46
30 The Büchi accepting loop of this automaton is not accessible due to the transition between 0 and 1. . . . 47
31 An energy function that cannot be interpreted as a single integer-weighted transition. 47

List of Algorithms

1 Büchi accepting lassos search in WBAs . . . 8
2 Modified Bellman-Ford algorithm . . . 8
3 Optimal energy prefixes calculation . . . 9
4 Algorithm for solving parity energy problems . . . 11
5 Solving for Rabin . . . 11
6 "Naive" solving for co-Büchi . . . 12
7 Co-Büchi solving using cycle storage . . . 14
8 Co-Büchi solving using backtracking . . . 17
9 Standard Floyd-Warshall algorithm . . . 18
10 Cleaning algorithm . . . 29
11 Floyd-Warshall algorithm for arbitrary semirings . . . 36
12 Additive operation ($\oplus$) on energy functions . . . 40
13 Multiplicative operation ($\otimes$) on two energy functions . . . 40

1 ▷ Introduction

Finite-state machines or automata are elementary structures of automata theory which consist of a finite number of states, including one initial state, and transitions between states equipped with a criterion that must be fulfilled to use them.

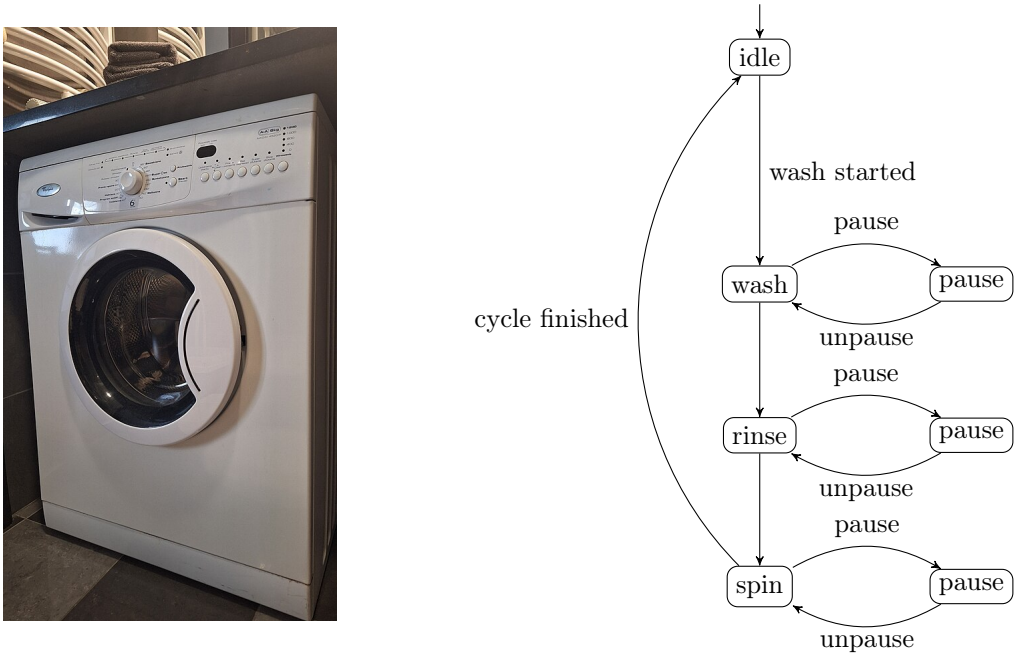


Figure 1: State machine of a washing machine.

Figure 1 illustrates an example of a (simple) state machine.

An acceptor, such as the one presented in Figure 2, is a type of finite-state machine that returns a boolean output depending on its input, which is a finite word $\omega \in \Sigma^*$ on an alphabet Σ (we say the word is accepted or rejected). As such, transitions in this kind of automaton are often labeled with letters of Σ. A common application of such automata is in formal language theory, where a language (a set of words) is *regular* (expressible by a regular expression) iff there exists an acceptor that recognizes exactly that language (i.e. there exists an automaton that accepts exactly all the words of the language).

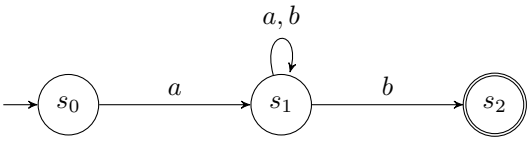


Figure 2: An acceptor on $\Sigma = \{a, b\}$ that accepts words starting with an a and ending with a b, i.e. the language $a\Sigma^*b$.

States in an acceptor automaton are divided into accepting and non-accepting states, which determine at the end of the execution, or *run* (the consumption of a word's letters and the use of the automaton's transitions according to these letters), if a word is accepted or not. In the following, we only focus on acceptor automata.

Some problems where we want to attribute some cost or weight to transitions cannot be represented by finite-state machines. Weighted automata aim to solve this issue by complementing transitions (that already have letters from Σ equipped to them, without which the automaton becomes a weighted directed graph) with weights valuated on a semiring.

A semiring (also sometimes called *rig*) is defined as a ring-like structure, that is a set R with the usual properties of a ring:

- R is equipped with an additive operation $\oplus$ and a multiplicative operation $\otimes$;
- $(R, \oplus)$ is a commutative monoid;
- $(R, \otimes)$ is a monoid;
- $\otimes$ is distributive to the left and to the right over $\oplus$;

but where some elements of R may not have an inverse for the additive operation.

Weighted automata are commonly used in domains where probabilistic approaches are needed, such as language models and speech recognition in [12]. Figure 3 represents an example of a possible application of weighted automata.

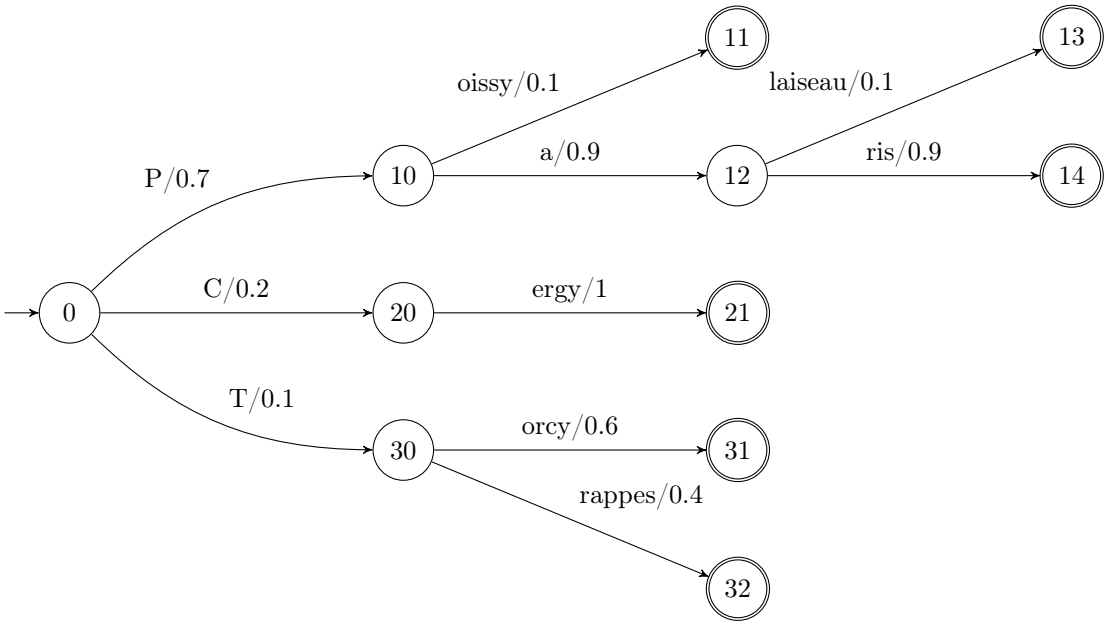


Figure 3: A weighted automaton that recognizes some city names equipped with probabilistic weights. For example, the word "Paris" has a higher weight than the word "Torcy". Labels have been condensed for readability (the label "oissy" would need to be separated into the letters "o", "i", "s", "s", "y").

Finite-state acceptors may also only take finite words (elements of Σ^*) as input: they cannot be used to model problems that do not halt, such as operating systems. ω-automata are a solution to this problem, by letting standard finite-state automata accept infinite words, or elements of Σ^ω (thus the name of this type of automaton) as input. Since a run in such automata is now infinite, there is no notion of final state or state at the end of the execution; as such, new conditions for accepting words must be defined.

In 1962, Büchi, in [20], is the first to develop a type of ω-automaton called Büchi automata, which are composed of the usual elements of a finite-state machine (set of states, alphabet, set of

transitions, initial state) as well as a set of *accepting states*. A run is considered accepting if there is at least one accepting state occuring infinitely often in it. An example of such an automaton is illustrated in Figure 4.

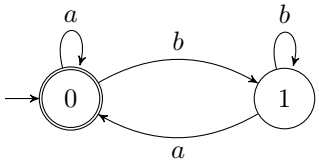


Figure 4: A Büchi automaton on $\Sigma = \{a, b\}$ that accepts infinite words that do not contain the letter b, i.e. the language a^ω.

Such Büchi automata are also called *state-based Büchi automata*, since they rely on accepting states to determine if a run is accepting or not. *Transition-based Büchi automata* are another type of Büchi automata in which the set of accepting states is replaced with a set of accepting transitions, with a similar criterion for determining accepting runs. There exists a bijection between state-based and transition-based Büchi automata according to [7]. An example of transition-based Büchi automaton is provided in Figure 5.

ω-automata are not limited to Büchi automata: other acceptance conditions have been proposed, such as the Muller condition in [14] where the set of accepting states is replaced with a set F of sets of states, i.e. a run is accepting if the set of all states that occur infinitely often is an element of F; or the generalized Büchi condition, such as in Figure 6, where there exist n sets of accepting states $\{F_i\}_{1 \leq i \leq n}$, i.e. a run is accepting if, for each $i \in [1, n]$, there exists a state $s_i \in F_i$ so that s_i occurs infinitely often in the run. Such sets of accepting states are also called *acceptance sets* or *colors* and are usually numbered using positive non-zero integers.

The parity condition, introduced in [13], is another type of acceptance condition equippable to a WWA, where the acceptability of a run depends on the highest or lowest color that appears infinitely often during the run (in parity conditions, colors are sometimes called *priorities*). The case where a run is considered accepting if the highest priority is even is called the *max-even* case, similar cases exist when the appropriate priority must be odd, or when we consider the lowest priority instead (*min-even*, *min-odd*). More generally, an ω-automaton with an arbitrary acceptance condition is also called an *Emerson-Lei automaton*, as introduced in [1].

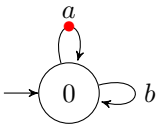


Figure 5: An automaton accepting words that contain infinitely often the letter a with eventually some b between instances of a, i.e. the language $(b^*a)^\omega$).

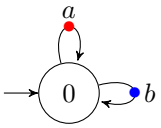


Figure 6: A generalized Büchi automaton accepting words that contain an infinite amount of a and b, i.e. the language $b^*(a^+b^+)^\omega$).

An interesting extension of these two types of automaton (weighted and ω) is that they can be combined to form *weighted ω-automata*, or WWA. A particular case of WWA is when the initial

non-weighted automaton has a Büchi condition, yielding a *weighted Büchi automaton* (WBA).

First introduced in [5] on classic weighted automata (without acceptance conditions), *energy problems* on WWAs consist in determining if there exists a run in an transition-based WWA that respects the WWA's acceptance condition but also an additional quantitative condition.

This quantitative condition often depends on a fixed quantity, called *upper bound*, that is analogous to the maximum capacity of a battery or industrial tank. It also depends on a *lower bound* (usually 0). Thus, the quantitative condition in an energy problem is that at any moment during the run, the accumulated energy (which varies during the run depending on the transitions taken) must remain within these two bounds. This allows us to model problems where a certain quantity has lower and upper bounds, such as autonomous systems with a limited energy supply or more abstract systems that require the use of a bounded accumulator. However, in such problems, the semiring property of weights is lost.

In [4], energy problems are divided into three types:

- the lower-bound case: there is no upper bound, the accumulated energy can be as high as we want. This case can model systems where the energy supply is large enough to be considered infinite, such as dams;
- the lower-upper-bound (or upper-bound) case: it is not possible to use transitions that would result in an energy superior to the upper bound. This case can be used to model systems where there exist physical restrictions on the energy supply, for example industrial tanks with a limited capacity and a continuous supply that may not spill out;
- the lower-weak-upper-bound (or weak-upper-bound) case: it is possible to use transitions that would result in an energy superior to the upper bound, in which case the final energy is capped to the upper bound (if the starting energy is a and the energy associated with the transition to be used is b, then the resulting energy after using the transition will be the minimum between the upper bound and $a + b$). This case is commonly used to model batteries or systems that require electricity.

As a WWA remains an ω-automaton, it is still equipped with an acceptance condition, which represents a qualitative condition in an energy problem. This qualitative condition does not depend on the weight and instead keeps the properties from the unweighted ω-automaton it comes from ((generalized) Büchi, Muller, etc.).

[8] presents algorithms for solving weak-upper-bound energy problems in WBAs, as well as in WWAs equipped with a parity condition via a reduction to multiple Büchi energy problems. In a first time, we implement the algorithm for solving parity energy problems in Python using (and extending) the `wspot` library [16], a Python set of bindings for the C++ library Spot [21].

In a second time, we pick up the work presented in [8] and extend it to automata with other acceptance conditions than Büchi and parity. While it has been proven in [19] that Emerson-Lei automata *can* be translated to parity automata, this transformation (which is implemented in Spot) does not preserve weights. As such, one of the main problems we try to address here is the existence of efficient algorithms for solving energy problems on co-Büchi automata.

In the following, we use the conventions adopted in [8] regarding :

- the alphabet Σ is supposed already defined and finite in all definitions;
- automata are also considered finite (i.e. have a finite number of states);
- a *weighted Büchi automaton* (WBA) $\mathcal{A} = (\mathcal{M}, S, s_0, T)$ is composed of a finite set of integer-labeled colors $\mathcal{M}$, a set of integer-labeled states S with $s_0 \in S$ being the initial states, and a set of transitions $T \subseteq S \times 2^{\mathcal{M}} \times \mathbb{Z} \times S$;

- an *acceptance condition* α, as defined in [3], is a boolean formula following the grammar:

$$\alpha ::= \bot \mid \top \mid \texttt{Inf}(col) \mid \texttt{Fin}(col) \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid (\alpha)$$

  where $\bot$ and $\top$ represent the usual associated booleans, $\wedge$ is the logical `AND`, $\vee$ is the logical `OR`, and $col \in \mathcal{M}$ is a color. The acceptance condition $\texttt{Inf}(col)$ can be interpreted as "an accepting run sees an infinite number of times the color *col*", while as $\texttt{Fin}(col)$ can be interpreted as "an accepting run sees a finite number of times the color *col*";

- a *weighted ω-automaton* (WWA) $\mathcal{A} = (\mathcal{M}, S, s_0, T, \alpha)$ is a WBA equipped with an acceptance formula α, where the grammar of acceptance formulas is defined in [3] (a WBA can also be considered as a WWA with $\alpha = \texttt{Inf}(0)$);

- as $\mathcal{M} = \{m_i\}_{1 \leq i \leq n}$ is a set of integers, where $n = |\mathcal{M}|$, it is possible to reindex the m_i so that they are numbered from 0 to $n-1$;

- a *transition* $t = (s, M, w, s') \in T$ in $\mathcal{A}$ going from s to s' is annotated by a set of colors $M \subseteq \mathcal{M}$ and a weight $w \in \mathbb{Z}$.

We also consider $c \in \mathbb{N}$ the *initial credit* or *initial energy*, which is the accumulated energy from the initial state; as well as $b \in \mathbb{N}$ the upper bound. In the following, we use WU to designate the upper bound.

In a first time, we consider classic energy problems in integer-weighted WWAs, then we later extend the definition of a WWA to a new structure inspired of the results of [4] called **energy functions**, which can be assimilated to functions that associate an input energy (when entering a transition or group or transitions) to an output energy (when leaving a transition or group of transitions).

2 ▷ Solving for Büchi, parity and others

In this section, we recall known algorithms to solve energy problems in Büchi automata as well as parity ones, and build new algorithms derived from Büchi and parity solvers to solve energy problems in other types of WWAs.

2.1 ▷ Büchi automata and parity energy problems

Continuing the work already done in [8], we work on automata respecting the Hanoi-Omega Automaton format defined in [3] augmented to support weights on transitions. The version of Spot supporting weights can be found on the `sven/weighted` branch at [21].

The Python code for our implementation of the algorithms in [8] can be found on GitHub at `https://github.com/PhilippSchlehuberCaissier/wspot/tree/thay`, in the `WBA_solvers` module.

We recall the already implemented algorithm for solving ω-energy problems in weighted Büchi automata, presented in Algorithm 1, and the algorithm that already appeared informally in [8] which allows solving parity problems in WWAs which will be presented in Algorithm 4.

It is demonstrated in [8] that any accepting path in $\mathcal{A}$ is of the form $\gamma_1 \gamma_2^\omega$, called *lasso*, where γ_1 is a path (a sequence of transitions) that designates the finite prefix of the lasso which is only traversed once, and γ_2 is a path that designates the cycle of the lasso which is repeated an infinite number of times.

In a WBA, the acceptance condition $\alpha = \texttt{Inf}(0)$ may be interpreted as "use an infinite number of times transitions that are equipped with the color 0". It follows that the cycle part of an accepting run must contain at least one transition that is equipped with the color 0 for it to satisfy this qualitative condition.

Algorithm 1: Büchi accepting lassos search in WBAs

```
Data: a WBA A = (M, S, s_0, T), an initial credit c, a WU
Result: ⊤ if there is a Büchi accepting loop in A, ⊥ else
1  E ⟵ optimal energy prefixes in A from its initial state
2  SCCs ⟵ list of SCCs in A
3  for scc ∈ SCCs do
4      GS, back-edges ⟵ degeneralize(scc)
5      for be = src → dst with weight w ∈ back-edges do
6          E′ ⟵ optimal energy prefixes in GS from dst with initial energy E at dst
7          e′ ⟵ min(b, E′[src] + w)
8          if E[dst] ≤ e′ then
9              return ⊤
10         else
11             E″ ⟵ optimal energy prefixes in GS from dst
12             e″ ⟵ min(b, E″[src] + w)
13             if e′ ≤ e″ then
14                 return ⊤
15             else
16                 for s_M where E″[s_M] = WU do
17                     E_→ ⟵ optimal energy prefixes in GS from s_M
18                     e_dst ⟵ min(b, E_→[src] + w
19                     E_← ⟵ optimal energy prefixes in GS from dst
20                     if E_←[s_M] = b then
21                         return ⊤
22 return ⊥
```

Algorithm 2: Modified Bellman-Ford algorithm

```
Data: a WBA A = (M, S, s_0, T) with n states, an initial credit c, a WU, an array E of
      size n
Result: an array with n elements with the optimal energy for each state, an array with n
        lists of optimal predecessors for each state
1  Pred ⟵ empty array of size n filled with []
2  for s ∈ states of A do
3      for t from s to s′ with weight w ∈ transitions of A do
4          e′ ⟵ min(E[s]) + w, WU)
5          if e′ ≥ 0 and E[s′] <e′ then
6              E[s′] ⟵ e′
7              if s′ has less than 2 predecessors or t is not in the 2 last seen predecessors of s′
                then
8                  append t to P[s′]
```

Algorithm 3 calculates optimal energy paths for every state of $\mathcal{A}$, that is paths that maximize the final energy attained at that state (this is the γ_1 part of an accepting lasso). In this algorithm, a modified version of the standard Bellman-Ford algorithm on weighted graph structures, which is traditionally used to calculate shortest paths from a single state to every state while supporting negatively weighted cycles, is used while E has not reached a fixed point, i.e. while there exist

Algorithm 3: Optimal energy prefixes calculation

Data: a WBA $A = (\mathcal{M}, S, s_0, T)$ with n states, an initial credit c, a WU
Result: an array with n elements with the optimal energy for each state, an array with n lists of optimal predecessors for each state

```
E ⟵ empty array of size n filled with −∞
E[s_0] ⟵ c
while no fixed point in E do
    apply Algorithm 2 to A using the array E
    pump all loops in A
return E
```

states which optimal energy can be improved.

The modified Bellman-Ford algorithm is recalled in Algorithm 2. It is an extension of this classic algorithm, which stores in an array E the optimal energy attained for each state, as well as the transitions that have been used to reach an energy optimal state at one point in an array of transition lists called Pred (we have direct access to a predecessor given a transition) to be able to later reconstitute the energy optimal prefix.

The loops pumping procedure, which will not be detailed here, consists of finding every state that changed energy during the application of the modified Bellman-Ford algorithm. If such a state s has a positive change, this could mean that a positive loop was used to increase its optimal energy: it is then possible to use that positive loop again.

In that case, the single loop pumping procedure (which is not detailed here either) is used. This procedure uses the Pred array that appears in the modified Bellman-Ford algorithm to find states belonging to the loop that contains s, and maximises the optimal energy of all these states.

The loops pumping procedure allows to reduce the number of Bellman-Ford iterations to only one, which is useful if the number of times the loop must be taken is high (for example, if the WU is high).

The procedure to find an accepting lasso in a WBA $\mathcal{A}$ is decomposed into two steps, as explained in [8]:

- first, energy-optimal paths to each state are computed in $\mathcal{A}$ using Algorithm 3. These paths give us the maximum energy attained when entering the automaton with an initial energy of c. They will be the prefixes of potential accepting lassos;
- then, the algorithm searches for accepting cycles in each of the automaton's strongly connected components or SCCs. By definition of a SCC, that is a set of states where there exists a path between every pair of states, it is not possible to find a cycle that spans over multiple SCCs (if a state s is in a SCC, s' is in another and there exists a cycle that spans over both SCCs, it would mean there exists a path between s and s' and vice versa by using that cycle, which is absurd). This step uses a modified version of the Couvreur algorithm used to find SCCs in a generic graph that is bundled with Spot and that preserves weights.

The search for accepting cycles in a SCC (which is itself a WBA as it is extracted from the WBA $\mathcal{A}$) is further divided in several parts;

- the SCC is degeneralized. This process, also explained in [8], transform a WBA with multiple colors into an equivalent WBA with a single color (i.e. $\alpha = \texttt{Inf}(0)$ by using a layering method: $k = |\mathcal{M}|$ copies of $\mathcal{A}$, called *levels* and indexed from 1 to k, are created (as such, the resulting automaton has $O(k|S|)$ states). Each level $i \in [1, k]$ contains the same transitions as $\mathcal{A}$ except for the ones that are colored with the color i: these transitions do not lead to their original

destination state in the current level, but to the equivalent destination state in the level $i+1$ or 1 if $i = k$. Transitions that allow returning to the first level are called *back-edges* and are the only ones colored in the new automaton with the color 0;

- since an accepting loop must contain at least one transition that accepts 0, we check for each back-edge *be* going from *src* to *dst* if there exists an energy-accepting cycle that includes *be*.

  To do so, the initial step of computing the energy-optimal paths from *dst* to each state is repeated, but with an initial energy equal to the optimal energy at *dst* and with *be* as the last used transition. This yields a final attained energy at *dst*, which is then compared with the initial energy at *dst*:

  - if the final energy is greater than the initial energy, this means that we did not lose energy when traversing the cycle. This means we have found our lasso, by combining this cycle with the energy-optimal path used to reach *dst*;
  - if this is not the case, then this step is repeated but with an initial energy at *dst* equal to the final energy reached prior to this comparison. This step must be repeated to account for some cases where, for example, the final energy would be less than the initial one due to the WU being reached early in the cycle.

In [8], an "onion" method that reuses the Büchi solving algorithm is used to find accepting paths in WWAs equipped with a parity condition. In the following, we place ourselves in a *max-even* parity problem.

We recall how the "onion" method works:

- we initially calculate the optimal energy prefixes just like in the Büchi case;
- for the same reasons as in WBAs, we only need to find accepting loops in the SCCs of $\mathcal{A}$;
- if the resulting automaton (SCC) is empty then there are no accepting loops, else we examine the parity of the highest priority:
  - if it is odd, then we do not want it to appear in the final cycle, so we remove it entirely from the automaton and re-run a parity solve on the resulting stripped automaton;
  - if it is even, then we search if there exist accepting loops that include this color. To do so, colors are removed from all transitions except for the highest one, reducing the problem to a Büchi problem. We then return a positive result if there is indeed an accepting loop in this new WBA or re-run a parity solve on the SCC without the highest priority otherwise.

Algorithm 4 is a formalization of this parity solving algorithm.

We can also use the Büchi and parity solvers as basis for new algorithms that allow us to solve energy problems in automata using other acceptance conditions.

2.2 ▷ Rabin condition

In an automaton equipped with a Rabin condition, an even number of colors is considered. Colors are divided into (odd color, even color) color pairs: for an run to be accepting under this condition, there must exist a color pair where the even color occurs finitely often and the odd color occurs infinitely often. As such, a Rabin condition is of the form $\alpha = \bigvee_{i=0}^{k\in\mathbb{N}^*} \texttt{Fin}(2i) \wedge \texttt{Inf}(2i+1)$.

To solve an energy problem in a WWA $\mathcal{A}$ under a Rabin acceptance condition, we can try for each pair of acceptance sets $(2i, 2i+1)$ to strip $\mathcal{A}$ from transitions accepting $2i$, and run a Büchi solver on an automaton with the same states and transitions as $\mathcal{A}$ but with no colors except for $2i+1$ (which is renumbered to 0 in the scope of the Büchi solver). We present in Algorithm 5 an algorithm to solve such problems.

In the following, when we say we *remove the color col* from $\mathcal{A}$, where $col \in \mathcal{M}$ is a color, this means that we remove every transition that accepts *col*.

Algorithm 4: Algorithm for solving parity energy problems

Data: a parity automaton A
Result: $\top$ if there is an accepting loop in A, $\bot$ else

$E \longleftarrow$ optimal energy prefixes in A from its initial state
for *scc* $\in$ *SCCs in* A **do**
  if *scc has no transitions* **then**
    return $\bot$
  if *the highest priority is even* **then**
    $A' \longleftarrow$ *scc* with the acceptance condition set to Büchi
    for *transition* $\in A'$ **do**
      if *transition accepts the highest priority* **then**
        set acceptance set to 0
      else
        remove all acceptance sets
      res $\longleftarrow$ Büchi solving on A'
      if *res* **then**
        return $\top$
      else
        return parity solving on *scc*
  if *highest priority is odd* **then**
    remove all transitions accepting the highest priority from *scc*
    return parity solving on *scc*

Algorithm 5: Solving for Rabin

Data: a Rabin ω-automaton $A = (M, S, s_0, T)$ with $|M| = 2p$ colors, an initial state s_0, a weak upper bound WU, an initial credit c
Result: a `BuechiResult`

begin
  for $k \in [0, p-1]$ **do**
    $f \longleftarrow 2k$
    $i \longleftarrow 2k+1$
    $buchiHoa \longleftarrow A$ with the color f removed
    set $buchiHoa$ acceptance to Büchi
    for $e \in buchiHoa$ *edges* **do**
      if e *accepts* i **then**
        e now accepts 0
      else
        e now accepts nothing
      $br \longleftarrow$ solve for Büchi in $buchiHoa$
      if br **then**
        return br
  return no loop

2.3 ▷ $\top$ condition (monitor)

When $\alpha = \top$, every infinite run that is accepting under the energy condition is also accepting under the qualitative condition. We will see in Section 3 that we can use the same method as in co-Büchi automata to solve energy problems in this kind of automaton.

3 ▷ Co-Büchi ω-automata

A *co-Büchi ω-automaton* is a WWA where $\alpha = \texttt{Fin}(0)$. A *generalized co-Büchi ω-automaton* is a generalization of co-Büchi ω-automata to multiple acceptance sets, i.e. $\alpha = \bigvee_{i=1}^{k \in \mathbb{N}^*} \texttt{Fin}(k)$. Note that a co-Büchi automaton is also a generalized co-Büchi automaton ($k = 1$).

We can interpret the co-Büchi acceptance condition as "there exists an energy-feasible run that doesn't traverse an infinite amount of times a transition labeled 0". By [8], this is equivalent to finding a lasso $\rho = \gamma_1 \gamma_2^\omega$ where $\gamma_{1,2}$ are finite energy-feasible runs and γ_2 is a cycle in which the acceptance set 0 may not appear.

In this section, we examine different approaches for solving energy problems in co-Büchi automata. We first design algorithms inspired by Büchi solving techniques, then propose a new semiring called semiring of energy functions that allows us to use a different approach to solve energy problems.

3.1 ▷ A first algorithm

We first use a "naive" algorithm, presented in Algorithm 6, to find energy-feasible runs derived from the Büchi solving algorithm: given that the first step of this algorithm already calculates all energy-optimal paths starting from s_0, we can search for energy-feasible cycles that do not contain transitions equipped with the color 0 by removing every such transition in a new, reduced automaton $\mathcal{A}'$.

Since the second step of the Büchi solving algorithm (already presented in Algorithm 1) searches for cycles traversing a back-edge, that is a transition that accepts the color 0 in a WBA, we can create an equivalent Büchi automaton of $\mathcal{A}$ by promoting every transition in $\mathcal{A}'$ to back-edge, setting the acceptance condition of this new automaton to $\texttt{Inf}(0)$ and solve a Büchi problem on it.

Algorithm 6: "Naive" solving for co-Büchi

Data: a co-Büchi (generalized) ω-automaton $A = (M, S, s_0, T)$, an initial state $s0$, a weak upper bound WU, an initial credit $c0$
Result: a `BuechiResult`

```
1  begin
2      en, pred = the optimal energy predecessors in A'
3      for col ∈ M do
4          A' ⟵ A with the color col removed
5          set acceptance condition of A' to Büchi
6          for e ∈ T do
7              set acceptance of e to {0}
8          res ⟵ BuechiEnergy(A', s0, WU, c0, en, pred)
9          if res is not null then
10             return res
```

Before proceeding to the time complexity analysis of this algorithm, we first need to determine the complexity of the Büchi solving algorithm. Let $k = |\mathcal{M}|$ the number of colors, $n = |S|$ the number of states and $t = |T|$ the number of transitions of $\mathcal{A}$. In the case $\mathcal{A}$ is sparse, we have $t = O(n)$.

It is shown in [8] that Algorithm 3 is in polynomial time, but no estimate of its complexity appears in this paper. As such, we need first to calculate the complexity of this algorithm that appears several times in the Büchi solving algorithm. We already know that the modified Bellman-Ford algorithm keeps its original complexity of $O(nt)$.

To calculate the time complexity of the optimal energy prefixes calculation algorithm, we must determine when the while loop is exited, that is when E reaches a fixed point. Unfortunately, there is no easy way of knowing this for an arbitrary Büchi automaton $\mathcal{A}$. However, we can estimate that in the worst case, we would need to pump each one of the n states once (this is an $O(n)$ operation). Since a loop in $\mathcal{A}$ cannot contain more than n states, we can deduce the complexity of Algorithm 3, which is $O(n^3)$ if $\mathcal{A}$ is sparse, or $O(n^2 t)$ if $\mathcal{A}$ is particularly dense. In the following, we consider $\mathcal{A}$ to be a dense automaton.

Algorithm 1 iterates over each SCC, however, in the worst case, the automaton is composed of one single SCC composed of n states and t transitions. Such an automaton does not allow returning early as it is the case when traversing an automaton with multiple SCCs: as the algorithm sequentially analyzes the SCCs of the automaton, it doesn't need to traverse next SCCs if it manages to find an accepting lasso in a SCC early on.

The equivalent degeneralized version of the automaton is used in case it has a generalized acceptance condition (actually, this is also the case in standard Büchi acceptance, with $k = 1$). This new automaton has k levels with $O(n)$ states and $O(t)$ transitions for each level: for this process, we only need to iterate over the k colors and the t transitions for each color. As such, the degeneralization process is in $O(kt)$, creating an equivalent automaton with $O(kn)$ states and $O(kt)$ transitions, including $O(t)$ back-edges.

In the degeneralized automaton, we need to look at its back-edges; there may be by definition at most t back-edges. Optimal energy prefixes are calculated, possibly for each one of the kn states in the degeneralized automaton, for each one of the $O(t)$ back-edges (this operation is in $O((kn)^2(kt)) = O(k^3 n^2 t)$). As such, the final complexity of the Büchi solving algorithm is $O(k^4 n^3 t^2)$.

3.2 ▷ Refinements towards an optimized algorithm

As stated in Section 3.1, the naive algorithm might not be optimized: as such, we propose other algorithms to find cycles in $\mathcal{A}$ and compare them with the existing naive one.

Algorithm 7 can be interpreted as a variation of the classic depth-first search (DFS) algorithm on directed graphs: for every color *col* that appears in the acceptance condition,

- we consider $\mathcal{A}'$ the automaton obtained from removing the color *col* from $\mathcal{A}$;
- as in a classical DFS, we keep the list of successors and already discovered states. However, our approach differs by storing couples (state, inbound energy) composed of a state and the energy accumulated when entering the state instead of storing only the state. To be able to reconstruct loops, an element of the stack of successors also contains additional information about the path from `s0` to the current state, which is a list of transitions. This stack starts with the state `s0`, also called the closing state, coupled with the initial credit. This state was not reached by using any transition, being the starting state;
- we also keep a list of every visited loop;

Algorithm 7: Co-Büchi solving using cycle storage

Data: a co-Büchi (generalized) ω-automaton $A = (M, S, s_0, T)$, an initial state $s0$, a weak upper bound WU, an initial credit $c0$
Result: a `BuechiResult`

```
1  en ⟵ optimal energy prefixes in A from its initial state
2  for col ∈ M do
3      A' ⟵ A with color col removed
4      succ ⟵ [([], None, (s0,c0))]
5      discovered ⟵ []
6      examinedLoops ⟵ []
7      while succ contains elements do
8          (path, currentEdge, (currentState, currentEnergy)) = succ.pop()
           /* Verify loop acceptance if they exist                          */
9          if path.length ≠ 0 then
10             closingState ⟵ path.first
11             if closingState occurs twice in the path then
12                 loopEdges ⟵ edges composing the loop
13                 while the first state to appear in the loop is not the smallest do
14                     shift loopEdges
15                 add loopEdges to examinedLoops
16                 energy ⟵en[closingState]
17                 for _ ∈ [0, path.length] do
18                     energy ⟵ sum of the energies of the path
19                     if energy is lower than at the start or energy < 0 then
20                         shift the loop
21                     else
22                         return the loop
           /* Continue the DFS if there is no loop                          */
23         if (currentState, currentEnergy) is not discovered and the detected loop (if
             applicable) was not already seen then
24             discover (currentState, currentEnergy)
25             for e ∈ edges departing from currentState do
26                 if energy after traversing e ≥ 0 then
27                     add (path + [e], e, (e.destination, energy after traversing e)) to succ
       /* We reached the end of the DFS without finding an accepting loop   */
28     return no loop
```

- we start the DFS by getting the (state, energy) couple, its associated path and previous used edge at the top of the stack of successors;
- if the popped path is not empty then we check if the path contains a loop, i.e. if a state (ignoring the inbound energy) appears twice in the path:
  - if the loop was already seen (if it exists in the list of already visited loops), we can ignore it as we know this is an energy-negative loop (this loop can eventually be shifted, so we must check for each visited loop if the shifted version of that loop correlates with the loop that's being processed);

– else, we have found a potential loop. We delete the prefix (the part that doesn't loop) from the path. Starting from its closing state with the initial energy `c0`, we can check if the final energy reached when using all transitions in the loop is greater or equal to the initial energy. If this is the case, then we have found an accepting loop.
There are cases when calculating the final energy for the closing state only is not sufficient. Indeed, the loop may need to be traversed a second time, for example in Figure 7 where the initial energy at the closing state was already $WU = 10$:

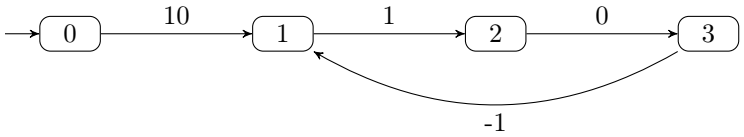


Figure 7: On first iteration, the energy attained after traversing the (1,2,3) loop will be lower than when entering for the first time state 1 (from 0 with 10 energy) despite it being a non-negative loop.

- we then continue the DFS by pushing to the stack the successors of the current state, as well as the energy attained when reaching them, by updating the path used to reach them.

We detail an execution of this algorithm, where $\alpha = \texttt{Fin}(0)$, $WU = 5$ and the initial credit is 0, on the automaton depicted in Figure 8 which has an energy-feasible cycle:

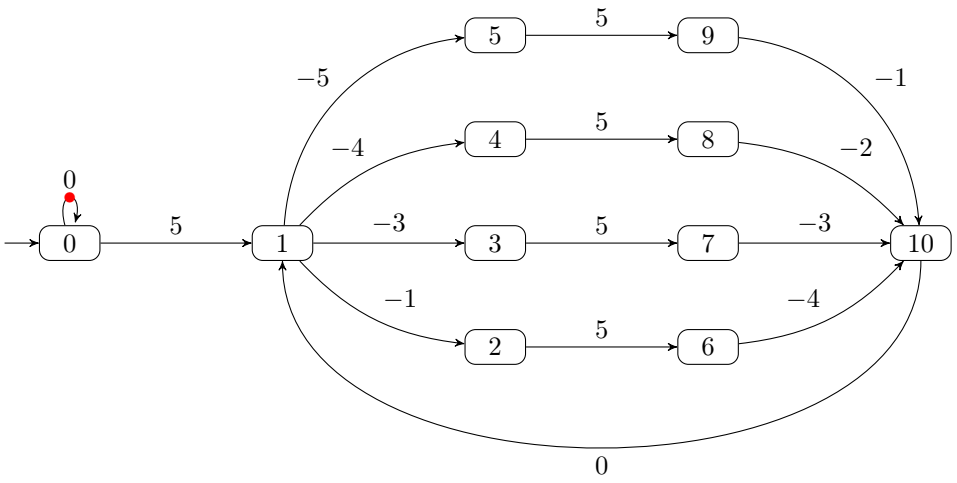


Figure 8: An automaton with one energy-feasible loop and several energy-negative loops. This example is inspired from Figure 6 in [8].

- the self-loop around 0 is removed since it accepts the only color of the acceptance condition;
- we start from state 0 with initial credit 0;
- as the path used to reach 0 is empty, we do not analyze the path;
- we continue the DFS: 0 is marked as discovered, we push its successor 1 to the stack, the energy attained at 1 is 5;
- state 1 is popped, we analyze the path used to reach it (the sole transition from 0 to 1 weighted with 5): as 1 does not appear twice in this path, we continue the DFS by pushing its successors 2, 3, 4 and 5 with associated reached energies;

- we pop state 5 with an inbound energy of 0. This still doesn't form a loop, so we push 9 with an attained energy of 5;
- we pop state 9 with an inbound energy of 5. This still doesn't form a loop, so we push 10 with an attained energy of 4;
- we pop state 10 with an inbound energy of 4. This still doesn't form a loop, so we push 1 with an attained energy of 4 (we already visited state 1 but with an inbound energy of 5 instead of 4);
- we pop state 1 with an inbound energy of 4. This forms a loop: the normalized form of this loop is $\{1, 5, 9, 10, 1\}$.

  Starting from state 1 with an optimal energy prefix of 5 (this was calculated at the start of the algorithm), we take every transition of this loop, but end with an energy of 4. We redo this process from the next state of the loop (5) but still end with a final energy that is lower than the starting energy at this state. The same applies for every remaining state of the loop: this is not an energy-feasible loop. As this is a loop, we do not continue the DFS from 1 with an inbound energy of 4;
- we continue the DFS instead from state 4 reached with an inbound energy of 1 (this couple was pushed when we first visited state 1), only to end with a loop that is not energy-feasible. The same applies for state 3 reached with an inbound energy of 2;
- we successively pop states 2 (inbound energy 4), 6 (inbound energy 5), 10 (inbound energy 1) and 1 (inbound energy 1). This forms the loop $\{1, 2, 6, 10, 1\}$.

  Starting from state 1 with an optimal energy prefix of 5, we end with an energy of 1. We shift the loop and start at state 2 with an optimal energy prefix of 4, but end with an energy of 0. We shift the loop again, starting at state 6 with an optimal energy prefix of 5, and end with an energy of 5: we have found an energy-feasible loop.

Instead of using a list of set of edges for storing already examined loops, we could have used a Python dictionary to reduce the time complexity of checking if a loop was already examined, with the first traversed edge being the key. However, edges in `wspot` are instances of the (C++) Spot built-in class `spot::twa_graph_edge_data`, and do not implement any hash function. As such, we cannot use such a dictionary unless we directly modify the Spot Python bindings, which is outside the scope of our work.

Using the same notations as in the naive algorithm complexity analysis ($k = |\mathcal{M}|$ is the number of colors, $n = |S|$ the number of states, $t = |T|$ the number of transitions), we can calculate the complexity of Algorithm 7.

For each color, we need to traverse all the n states of the automaton. Given a state s, checking if s appears twice in the path, i.e. checking if there is a loop in the path used to reach s, takes $O(t)$ operations.

If there is indeed a loop in this path, it is first normalized, which is an $O(t)$ operation. We then need to check if this loop is already in the list of examined loops (this means the current loop is not energy-feasible, since if this was the case, then the algorithm would return early). Unfortunately, we do not know a precise upper bound for the number of loops in an automaton that is not exponential in the number of states (in [2], which focuses on directed graphs with no short cycles, it is demonstrated that a directed graph, and by extension an automaton, with n states has $O(3^n)$ cycles if it has no short cycles, that is cycles of length greater than $n - k$ where $3k < n$). Therefore, the cycle storage based algorithm is also exponential in the number of states.

Algorithm 8, proposed by Philipp, is another variation of the DFS inspired by the original Büchi solving algorithm. It is similar to the solver using the cycle storage method (Algorithm 7), but differs in that the state pairs now contain the predecessor information instead of the incoming energy.

Algorithm 8: Co-Büchi solving using backtracking

Data: a co-Büchi (generalized) ω-automaton $A = (M, S, s_0, T)$, an initial state $s0$, a weak upper bound WU, an initial credit $c0$
Result: a `BuechiResult`

```
1  begin
2      for col ∈ M do
3          A' ⟵ A with color col removed
4          succ ⟵ [([], None, (s0,pred))]
5          discovered ⟵ []
6          while succ ≠ [] do
7              (path, currentEdge, (currentState, predecessor)) = succ.pop()
               /* Verify loop acceptance if they exist                    */
8              if path.length ≠ 0 then
9                  closingState ⟵ path.first
10                 if closingState occurs twice in the path then
11                     pump loop twice
12                     finalEnergy ⟵ energy at closingState
13                     backtrack loop from closingState
14                     if energy ≤ finalEnergy then
15                         return the loop
               /* Continue the DFS                                        */
16             if (currentState, predecessor) is not discovered then
17                 discover (currentState, predecessor)
18                 for e ∈ edges departing from currentState do
19                     if energy after traversing e ≥ 0 then
20                         add (path + [e], e, (e.destination, currentState))
               /* We reached the end of the DFS without finding an accepting loop */
21             return no loop
```

In a way similar to [8] and Algorithm 3, we can pump encountered loops (when we process a state that has already been processed) twice using predecessor information, and determine if this loop is energy-positive by starting from the end state, using the transitions that compose the loop backwards and comparing the resulting energy with the starting energy at the end state.

We use the same notations as in the previous complexity analysis ($k = |\mathcal{M}|$ is the number of colors, $n = |S|$ the number of states, $t = |T|$ the number of transitions) in the complexity analysis of the co-Büchi solving algorithm using backtracking.

For each color, we need to traverse all the n states of the automaton (just like the algorithm using cycle storage). Given a state s, a loop that terminates at s has a length of $O(n)$; we need to pump this loop twice (this is an $O(n)$ operation). Thus, the backtracking process inside a loop is also an $O(n)$ operation.

After the potential loop has been processed, the DFS continues by discovering the current (state, predecessor) couple and finding the next couples to be pushed on the stack. Since there are t transitions, there are $O(t)$ potential next elements to be pushed, giving us a total complexity of $O(k(n^2 + nt))$.

3.3 ▷ Energy functions

The motivation for using the Floyd-Warshall algorithm instead of the (modified version of the) Bellman-Ford algorithm used in Algorithm 1 is that we can run the former only once to find positive loops, since we can have a constant time access to the maximum energy difference obtained by going from a state to itself; whereas the latter must be run every time the starting point of a potential energy positive loop is found. Moreover, it also allows us to skip the optimal energy prefixes and predecessors calculation step, as we would have for each pair of states a direct access to possible values for the final energy reached, the states traversed to reach the destination state, and as such the optimal energy prefix.

We recall the general form of the Floyd-Warshall algorithm on weighted automata (Algorithm 9):

Algorithm 9: Standard Floyd-Warshall algorithm

Data: an integer-weighted automaton $A = (\mathcal{M}, S, s_0, T)$
Result: a matrix M of the shortest distances between each pair of states

```
1  begin
2      n ⟵ number of states of A
3      M ⟵ n × n matrix filled with ∞
4      for T ∋ e from u to v with weight k do
5          M[u][v] ⟵ k
6      for s ∈ S do
7          M[s][s] ⟵ 0
8      for k ∈ [1, n] do
9          for i ∈ [1, n] do
10             for j ∈ [1, n] do
11                 M[i][j] ⟵ min(M[i][j], M[i][k]+M[k][j])
12     return M
```

The Floyd-Warshall algorithm is usually used to compute minimal distances between every pair of states in directed graphs, stored in a square matrix of size the number of states of the graph. By inverting the sign of every weight, it is also possible to compute maximal distances to detect positive loops.

An application of this algorithm uses the *tropical semiring* (the semiring formed by the set of real numbers $\mathbb{R} \cup \{+\infty\}$ with min as the additive operator and the usual addition on real numbers as the multiplicative operator), also called *min-plus semiring*; but it can be extended to other semiring structures according to [11].

However, there are multiple reasons on why this algorithm cannot be used as is to solve energy problems:

- the algorithm loses the initial energy information. In the following two automata, depending on the initial energy, no energy-feasible runs may exist for $WU = 10$:

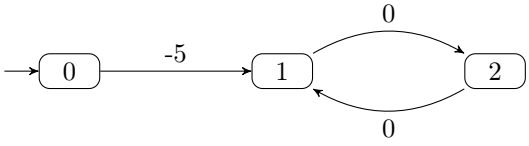


Figure 9: The loop $(1, 2)^\omega$, is only accessible if the initial energy is between 5 and 10 because of the transition going from 0 to 1.

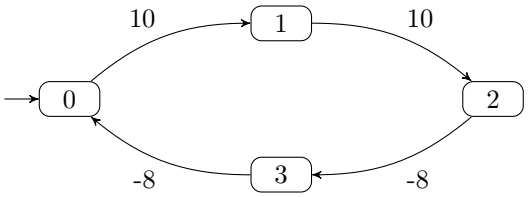


Figure 10: Due to the $WU = 10$ constraint, the transition going from 3 to 0 is unusable.

- in the case where multiple accepting loops exist, the Floyd-Warshall algorithm only returns the loop with the greatest (positive) energy difference. This can lead to scenarios where the algorithm fails to detect the existence of an infinite run, for example due to an insufficient initial energy. As an example, consider the following automaton for $WU = 10$:

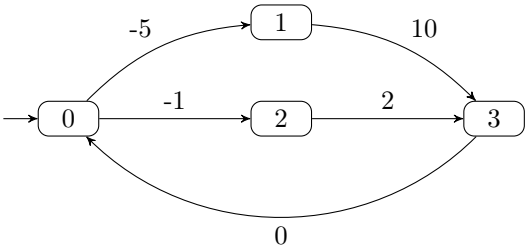


Figure 11: An automaton with two different positive loops.

  In this automaton, two positive loops exist: $(0, 1, 3, 0)$, with a net energy gain of 5, which is accessible if the initial energy is greater than 5, and $(0, 2, 3, 0)$, with a net energy gain of 1, which is accessible if the initial energy is greater than 1.

  The loop using state 1 is "more interesting" from an energy point of view, but it cannot be accessed if the initial energy is between 1 and 4. However, the classic Floyd-Warshall algorithm fails to detect the positive loop using state 2 since it is "less interesting" as the energy gain is lower: it will consider that no positive loop exists when searching for a loop with an initial energy of 1 to 4.

- most importantly, in energy problems, we are no longer working on a semiring, as transitions lose their associative property. In Figure 10, in a standard WWA context, it would have been possible to group together the transition going from 1 to 2 and the one going from 2 to 3 to form an unique transition going from 1 to 3 with weight 2. In an energy problem context, this is no longer possible as using the transition from 1 to 3, going above the WU, and then using the transition from 2 to 3 is different from simply adding 2 to the energy.

Instead, we elaborate on the notion of *energy functions*, already introduced in [4] but in the lower bound scenario (i.e. not in the weak upper bound scenario), and prove in Theorem 28 that they form a semiring structure, thus enabling us to use the Floyd-Warshall algorithm on automata weighted with them. Note that we will only define energy functions in this section; the link between energy functions and energy problems will be discussed in Section 4.

Our definitions follow the usual notations from the definition of $\mathcal{A}$ (such as WU). An energy function can be interpreted as a function of $[0, WU]$ into $[0, WU]$ that associates an input energy (for example, the energy before using a transition or a sequence of transitions) with an output energy (the energy after using a transition).

In the following, $\mathbb{N}$ is the set of natural numbers, $\mathbb{Z}$ the set of integers, and $\mathbb{N}^* = \mathbb{N} \setminus \{0\}$.

Definition 1 (Energy domain)**.** *An energy domain is an interval $I \subseteq [0, WU]$ of the form $[\alpha, \alpha]$, $[\alpha, \beta]$, $[\alpha, \beta[$, $]\alpha, \beta[$, or $]\alpha, \beta]$ for $(\alpha, \beta) \in [0, WU]^2$ and $\alpha < \beta$.*

An energy domain of the form $[\alpha, \alpha]$ is also called a point.

We will see in Section 4 that we need this specific structure to represent some edge cases that may appear in some energy problems.

Definition 2 (Energy segments)**.** *An energy segment is a function $f : I \longrightarrow [0, WU]$ defined on an energy domain I that is of the form $f : e_{in} \longmapsto ae_{in} + b$ where $a \in \{0, 1\}$ and $b \in \mathbb{Z}$. The image of f must remain in $[0, WU]$ for the energy segment to be defined. It is also equipped with a predecessor which is either $s \in S$ or the undefined predecessor, which is written as* `undef`*.*

As an exception to this definition, the null energy segment on I defined as null : $e_{in} \longmapsto \perp$, regardless of its predecessor, is also considered an energy segment even though $\perp$, a special value indicating inaccessibility, is not an element of $[0, WU]$. Unless explicitly specified, the predecessor for the null energy segment will be considered as `undef`*.*

In the following, when talking about an energy segment defined on I, we also include the null segment on I. I is called the energy domain of f, or simply domain of f, and is written as $\mathrm{dom}(f)$*.*

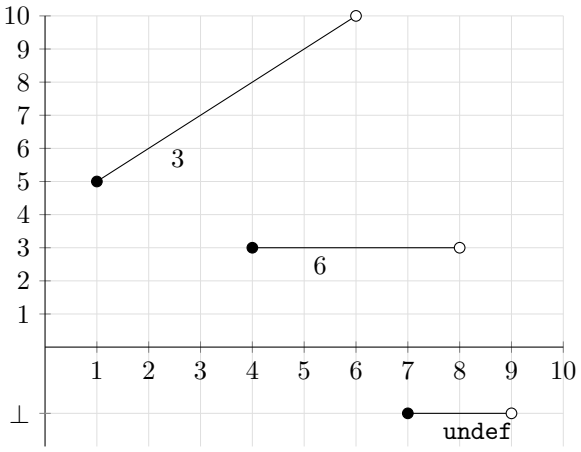


Figure 12: Examples of energy segments.

Figure 12 illustrates some possible energy segments:

- an energy segment of equation $e_{in} \longmapsto e_{in} + 4$ defined on $[1, 6[$, with predecessor 3;
- an energy segment of equation $e_{in} \longmapsto 3$ defined on $[4, 8[$, with predecessor 6;
- the null energy segment defined on $[7, 9[$, with predecessor `undef`.

An energy segment with an `undef` predecessor usually means that it is better to stay at the current state rather than using a transition that departs from this state, while an energy segment with an equation of the form $e_{in} \longmapsto \perp$ means that the target state is unreachable, for example if there exists no path from the source towards the target or if there is not enough energy to use any available path reaching the target.

We use the notation $\underline{[0, WU]} = [0, WU] \cup \perp$ frequently when designating the set of possible values for an energy segment or function to increase readability.

Definition 3 (Energy functions)**.** *An energy function with a weak upper bound of WU is a function $F : [0, WU] \longrightarrow \underline{[0, WU]}$ composed of $p \in \mathbb{N}^*$ energy segments $\{f_k\}_{1 \leq k \leq p}$. Each segment f_k for $k \in [1, p]$ is defined on its domain $\mathrm{dom}(f_k)$, which respect the following properties:* $\forall k \in [1, p]$

- *if $k \neq p$ then $\sup(\mathrm{dom}(f_k)) = \inf(\mathrm{dom}(f_{k+1}))$ (energy segments are ordered by increasing domain and there are no gaps where no energy segment exists);*
- *if $k = 1$ then $\mathrm{dom}(f_k)$ is closed on the left and its minimum is 0;*
- *if $k = p$ then $\mathrm{dom}(f_k)$ is closed on the right and its maximum is WU;*
- *if $k \neq p$ and $\mathrm{dom}(f_k)$ is not a point then $\mathrm{dom}(f_k)$ is open on the right;*
- *if $k \neq p$ and $\mathrm{dom}(f_k)$ is a point then*
  - *$\mathrm{dom}(f_k)$ is closed by definition;*
  - *$\mathrm{dom}(f_{k+1})$ is open on the left, else $\mathrm{dom}(f_{k+1})$ is closed on the left.*

The null energy function is defined as the function that contains only the null energy segment defined on $[0, WU]$.

As a convention, we use lowercase letters to designate energy segments (such as f or g) and uppercase letters to designate energy functions (such as F or G).

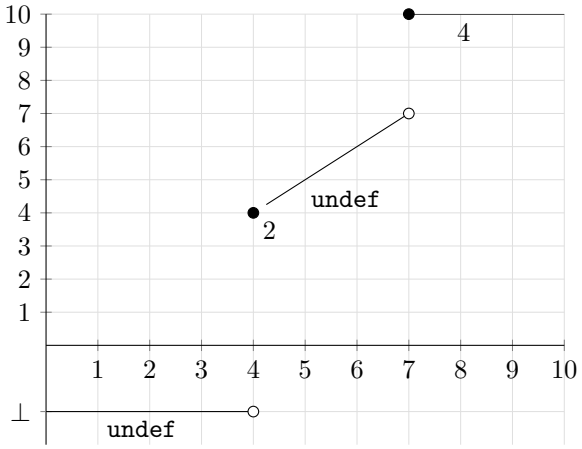


Figure 13: F_{test}, a fully defined energy function.

Figure 13 visually represents an example energy function called F_{test} for $WU = 10$. It is composed of four energy segments:

- the null energy segment defined on $[0, 4[$, with predecessor `undef`;
- the energy segment of equation $e_{in} \longmapsto e_{in}$ defined on $[4, 4]$, with predecessor 2;
- the energy segment of equation $e_{in} \longmapsto e_{in}$ defined on $]4, 7[$, with predecessor `undef`;
- the energy segment of equation $e_{in} \longmapsto 10$ defined on $[7, 10]$, with predecessor 4.

Proposition 4. *Let $F = (f_k)_{1 \leq k \leq p}$ an energy function. $\{\text{dom}(f_k)\}_{1 \leq k \leq p}$ forms a partition of $[0, WU]$.*

Proof.

We prove by induction for $u \in [1, p]$ that the $\{\text{dom}(f_k)\}_{1 \leq k \leq u}$ form a partition of P_u where $P_u = \bigcup_{1 \leq i \leq u} \text{dom}(f_i)$.

- For $u = 1$: $\text{dom}(f_1)$ is a partition of itself.
- Suppose that the $\{\text{dom}(f_k)\}_{1 \leq k \leq u}$ form a partition of P_u for $u \in [1, p-1]$ (the condition $u \neq p$ ensures that the next energy segment exists).

  By Definition 3, if P_u is closed on the right then the next energy domain is open on the left (i.e. they do not intersect), thus $\{\text{dom}(f_k)\}_{1 \leq k \leq u} \cup \text{dom}(f_{u+1})$ remains a partition of $P_u \cup \text{dom}(f_{u+1})$, which is also P_{u+1}. The reasoning if P_u is open on the right is similar (the next energy domain is closed on the left). Therefore, the property also holds for $u + 1$.

We remark that $P_p = [0, WU]$. Thus $\{\text{dom}(f_k)\}_{1 \leq k \leq p}$ forms a partition of $[0, WU]$. ■

The previous proposition allows us to consider that a set of $t \in \mathbb{N}^*$ energy segments $\{f_k\}_{1 \leq k \leq t}$ can be considered as an energy function if the domains of these segments do not overlap (i.e. $\forall (i, j) \in [1, t]^2$ s.t. $i \neq j \mid \text{dom}(f_i) \cap \text{dom}(f_j) = \emptyset$) and if the energy segments are ordered by ascending domain (i.e.: $\forall (i, j) \in [1, t]^2, \forall x \in \text{dom}(f_i), \forall y \in \text{dom}(f_j), x < y$).

Definition 5 (Set of energy functions on WU)**.** *The set of all energy functions with a weak upper bound of WU is written as $EF(WU)$.*

We suppose WU already defined, and we implicitly consider energy functions to have a weak upper bound of WU.

Definition 6 (Restriction of an energy segment)**.** *Let f an energy segment on its domain I. The restriction of f to the energy domain J is written as $f|_J$ and is defined when $I \cap J \neq \emptyset$. In that case, $f|_J$ is an energy segment defined on $I \cap J$ with the same equation and predecessor as f.*

Proposition 7. *Let J an energy domain and f an energy segment so that $f|_J$ is defined. $f|_J$ is an energy segment.*

Proof. f is already an energy segment and $J \subseteq [0, WU]$, thus the result. ■

Definition 8 (Restriction of an energy function)**.** *Let $F = \{f_k\}_{1 \leq k \leq p}$ an energy function and J an energy domain that will be considered w.l.o.g. closed to the left and open to the right.*

We define $F|_J$ the restriction of F to J as the set of every segment f_k for which $f_k|_J$ is defined, completed with null energy segments outside of J, that is:

$$F|_J := \{A, \{f_k|_J \mid \text{dom}(f_k) \cap J \neq \emptyset\}, B\}$$

where A is the null energy segment defined on $[0, \inf(J)[$ if $\inf(J) > 0$ and B is the null energy segment defined on $[\sup(J), WU]$ if $\sup(J) < WU$.

We now introduce the concept of *discontinuity*, which may also be seen as the boundaries of all the energy segments of an energy function.

Definition 9 (Discontinuities)**.** *The discontinuities of an energy function* $F = \{f_k\}_{1 \leq k \leq p}$ *are* $\text{Disc}(F) := \{0\} \cup \{\sup(\text{dom}(f_k))\}_{1 \leq k \leq p}$*. Note that* 0 *and* WU *are considered discontinuities.*

For example, the discontinuities of the F_{test} function defined in Figure 13 are $\{0, 4, 7, WU\}$.

Definition 10 (Pairs of discontinuities)**.** *Let* F *an energy function,* $\text{Disc}(F) = \{D_k\}_{0 \leq k \leq p}$ *its discontinuities and* $k \in [1, p]$*.*

By definition of F*, there exists an energy segment* f *which domain is an energy domain* $\text{dom}(f)$ *between* D_{k-1} *and* D_k*. Suppose w.l.o.g. that* δ_k *is closed to the left and open to the right. In this case, we call pair of discontinuities the energy domain* δ_k *between* D_{k-1} *and* D_k *that respects the following:*

- *if* $k = 1$*, then* δ_k *is closed on the left;*
- *if* $k = p$*, then* δ_k *is closed on the right;*
- *if* $D_{k-1} = D_k$*, then* δ_k *is closed on both sides;*
- *if* $k > 1$ *and* δ_{k-1} *is closed on the right, then* δ_k *is open on the left;*
- *else* δ_k *is closed on the left and open on the right.*

As such, the pairs of discontinuities of F *form the* $\{\delta_k\}_{1 \leq k \leq p}$*.*

This definition is similar to the definition of the energy domains of F, but unlike energy domains, it can be extended to multiple energy functions. This will be useful when examining the results of the $\oplus$ and $\otimes$ operations.

Definition 11. *In a definition, a proposition, or a proof, we say that we bind the* $\{\delta_k\}_{1 \leq k \leq n}$ *to an energy function* F *(or that the* $\{\delta_k\}_{1 \leq k \leq n}$ *are bound to* F*) when we consider inside the scope of that definition, proposition, or proof, that* $n = |\text{Disc}(F)| - 1$ *and that the pairs of discontinuities of* F*,* $\{\delta_k\}_{1 \leq k \leq n}$*, are generated from* $\text{Disc}(F)$*.*

Definition 12 (Combination of discontinuities)**.** *Let* F, G *two energy functions,* $\text{Disc}(F)$ *and* $\text{Disc}(G)$ *their respective discontinuities,* $p = |\text{Disc}(F)|$ *and* $q = |\text{Disc}(G)|$*.*

The list of discontinuities resulting from the combination of F*'s discontinuities and* G*'s, written as* $\text{Disc}(F, G)$*, is defined as*

$$\text{Disc}(F, G) := \text{Disc}(F) \cup \text{Disc}(G) \text{ so that } \text{Disc}(F, G) \text{ is ordered ascendingly}$$

This definition is extendable to more than two energy functions by considering the discontinuities of these other functions.

Proposition 13. *Let* F, G *two energy functions,* $p = |\text{Disc}(F)|$ *and* $q = |\text{Disc}(G)|$*.*

$$\max(p, q) \leq |\text{Disc}(F, G)| \leq p + q$$

Proof. Properties of the union on sets. ■

Definition 14. *Similarly to Definition 11, we say that we bind the* $\{\delta_k\}_{1\leq k\leq n}$ *to the* $t \in \mathbb{N}^*$ *energy functions* $(F_i)_{1\leq i\leq t}$ *(or that the* $\{\delta_k\}_{1\leq k\leq n}$ *are bound to the* $(F_i)_{1\leq i\leq t}$*) when we consider that* $n = |\mathrm{Disc}((F_i)_{1\leq i\leq t})| - 1$ *and that the pairs of discontinuities* $\{\delta_k\}_{1\leq k\leq n}$ *are generated from* $\mathrm{Disc}((F_i)_{1\leq i\leq t})$.

We now define the additive and multiplicative operations on energy functions.

Lemma 15. *Let* $F = \{f_k\}_{1\leq k\leq p}$ *and* $G = \{g_k\}_{1\leq k\leq q}$ *two energy functions,* $\{\delta_i\}_{1\leq i\leq n}$ *generated by the discontinuities of* F *and* G*, and* $i \in [1, n]$*. There is one and only one segment* f_u *of* F *and* g_v *of* G *so that* $\delta_i \subseteq \mathrm{dom}(f_u)$ *and* $\delta_i \subseteq \mathrm{dom}(g_v)$.

In other words, there are no situations where δ_i *overlaps with more than two segments of* F *or* G.

Proof. By contradiction:

suppose w.l.o.g. that δ_i overlaps with f_u and $f_{u'}$, two segments of F. Then, since f_u and $f_{u'}$ are two different segments, there exists an additional discontinuity $d \in \delta_i$. This contradicts with the definition of a pair of discontinuities. ■

Corollary 16. $F|_{\delta_i}$ *(*F *restricted to* δ_i*) contains a single energy segment.*

Proof. Using the previous lemma, we can affirm the existence of a segment f_u of F so that $\delta_i \subseteq \mathrm{dom}(f_u)$. This also means that $\delta_i \cap \mathrm{dom}(f_u) \neq \emptyset$. As $F|_{\mathrm{dom}(f_u)} = f_u$ is already an energy segment, according to Proposition 7, $f_u|_{\delta_i}$ is an energy segment. ■

We now define the additive ($\oplus$) and multiplicative ($\otimes$) operations on energy segments, then on energy functions. Translated to the domain of WBAs, if we have an existing energy function F associated with a certain sequence of transitions T between two states, the $\oplus$ operation represents the comparison of T with another sequence of transitions, while the $\otimes$ operation represents the composition of T with another transition that starts at the final state reached by using T.

Definition 17 (Maximum of two energy segments)**.** *Let* $(\alpha, \beta) \in [0, WU]^2$*,* I *an energy domain between* α *and* β*, and* $f : x \mapsto a_1x + b_1$ *(predecessor* s_1*),* $g : x \mapsto a_2x + b_2$ *(predecessor* s_2*) two energy segments defined on* I*. The* $\oplus$ *operation between* f *and* g *is defined as*

$$f \oplus g := \begin{cases} g & \text{if } a_1 = a_2 \wedge b_1 = b_2 \wedge s_1 = \texttt{undef} \\ f & \text{if } a_1 = a_2 \wedge b_1 = b_2 \wedge s_1 \neq \texttt{undef} \\ f & \text{if } a_1 = a_2 \wedge b_1 > b_2 \\ g & \text{if } a_1 = a_2 \wedge b_1 < b_2 \\ f & \text{if } a_1 \neq a_2 \wedge f(\beta) > g(\beta) \wedge f \text{ and } g \text{ do not intersect} \\ g & \text{if } a_1 \neq a_2 \wedge f(\beta) < g(\beta) \wedge f \text{ and } g \text{ do not intersect} \\ \eta & \text{if } f \text{ and } g \text{ intersect} \end{cases}$$

where

$$\eta := \begin{cases} (f|_{[\alpha,\iota]}, g|_{[\iota,\beta]}) & \text{if } f(\alpha) > g(\alpha) \\ (g|_{[\alpha,\iota]}, f|_{[\iota,\beta]}) & \text{if } f(\alpha) < g(\alpha) \end{cases}$$

where $\iota = \frac{b_2 - b_1}{a_1 - a_2}$ *is the intersecting point of* f *and* g*, and* $f|_I$ *designates the restriction of the energy segment* f *to some interval* I *(which is itself an energy segment).*

Whether f *and* g *intersect can be determined, for example, using the intermediate value*

theorem:

$$f \text{ and } g \text{ intersect} \iff f(\alpha) > g(\alpha) \quad \texttt{XOR} \quad f(\beta) > g(\beta)$$

where `XOR` *denotes the usual exclusive-*`OR` *operation on booleans.*

An illustration of the $\oplus$ operation on energy segments is provided in Figures 14, 15 (non-intersecting segments) and 16 (intersecting segments).

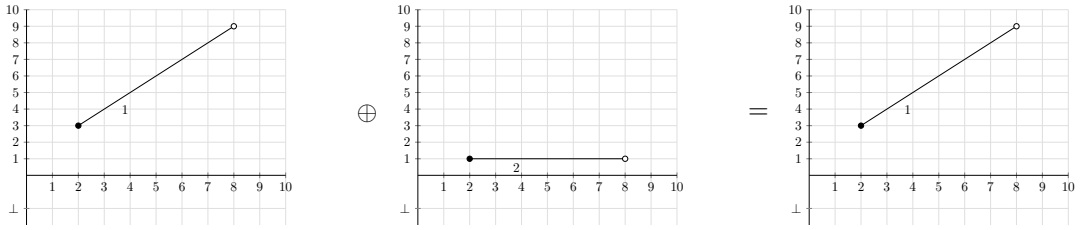


Figure 14: $\oplus$ operation on two non-intersecting energy segments.

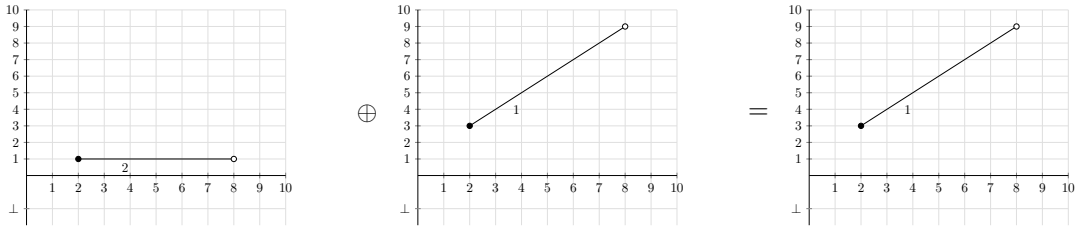


Figure 15: $\oplus$ on energy segments is commutative.

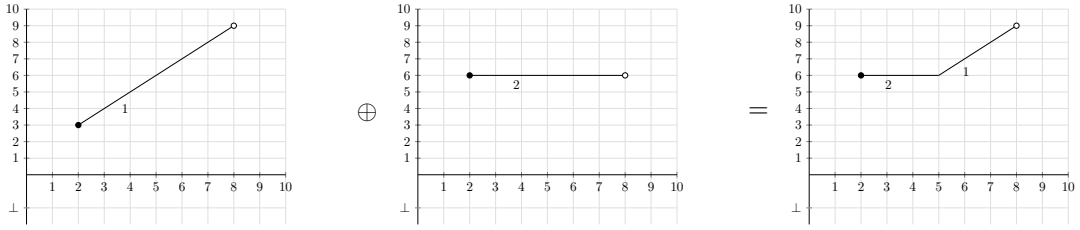


Figure 16: $\oplus$ operation on two intersecting energy segments.

We will define an intermediate operation called $\times$, or "*cross*", that will be used in the definition of $\otimes$.

Definition 18 (Cross operator)**.** *Let G an energy function, f an energy segment defined on an energy domain I between α and β that is supposed w.l.o.g. closed to the left and open to the right,* $\mathrm{Im}(f)$ *the image of f, i.e.* $\mathrm{Im}(f) = \{f(x) \mid x \in I\}$*, and f^{-1} the inverse function of f (obtained by inversing its equation) if f is not a constant energy segment.*

In the case where f is a constant energy segment, the energy segment has an equation of the form $e_{in} \longmapsto k$, where $k \in [0, WU]$. Since the domains of the energy segments of G form a partition of $[0, WU]$, there exists a segment of G, which we call g, such that $b \in \mathrm{dom}(g)$.

In the case where f is an ascending energy segment, the restriction of G to the interval

$\mathrm{Im}(f)$, *written as* $G|_{\mathrm{Im}(f)}$, *gives us a tuple of segments (potentially with only one segment in it).*

We consider $\{d_i\}_{1\leq i\leq n+1} = \{f(\alpha)\} \cup (\mathrm{Disc}(G) \cap \mathrm{Im}(f)) \cup \{f(\beta)\}$ *the ascendingly ordered set of discontinuities that remain in* $G|_{\mathrm{Im}(f)}$, *where* $n = |\mathrm{Disc}(G) \cap \mathrm{Im}(f)| + 1$.

We also consider the n *pairs of discontinuities formed by the* $\{d_i\}_{1\leq i\leq n+1}$, *written as* $(\theta_i)_{1\leq i\leq n}$ *to avoid confusion with pairs of discontinuities of classic energy functions defined on* $[0, WU]$ *(note that no* $\{\delta_i\}_{1\leq i\leq n}$ *are bound in this definition). According to Corollary 16, we can deduct energy segments from the* $(\theta_i)_{1\leq i\leq n}$*: for* $i \in [1, n]$, *we pose* $r_i = G|_{\theta_i}$ *the energy segment resulting from the restriction of* G *to* θ_i.

As such, we define the $\times$ *operation between an energy segment and an energy function as the following operation that returns a set of segments:*

$$f \times G := \begin{cases} \{f\} & \text{if } f \text{ is the null segment on } I \\ \{c\} & \text{if } f \text{ is a constant energy segment} \\ \{\xi_i \mid i \in [1, n]\} & \text{if } f \text{ is an increasing energy segment} \end{cases}$$

where

- c *is the energy segment of equation* $x \longmapsto g(k)$ *defined on* I *of predecessor the predecessor of the associated segment of* G *if it is not* `undef`, *the predecessor of* f *otherwise;*
- ξ_i *for* $i \in [1, n]$ *is the energy segment defined on* $[f^{-1}(d_i), f^{-1}(d_{i+1})[$ *of same equation and predecessor as* r_i.

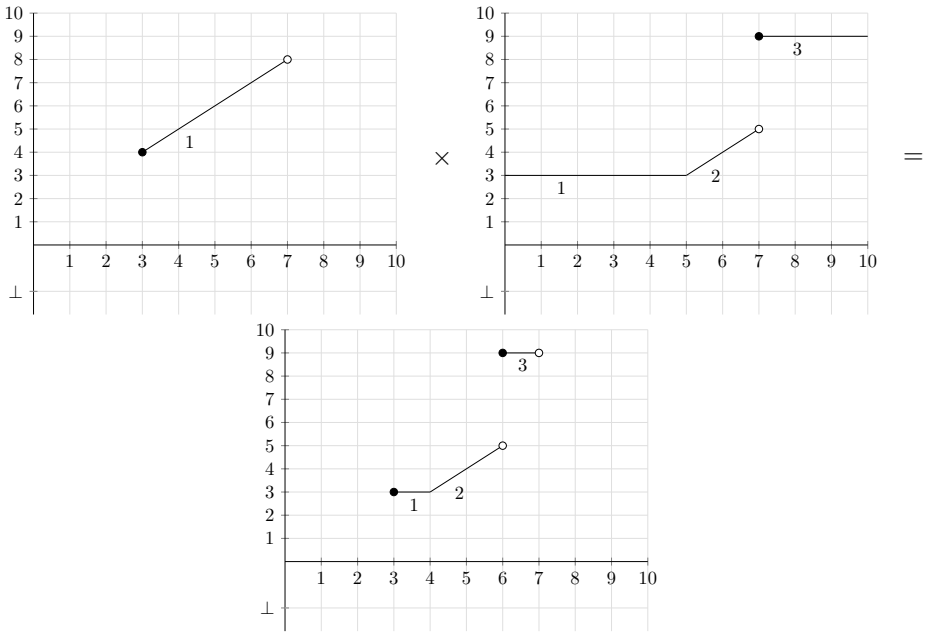


Figure 17: $\times$ operation on an energy segment and an energy function.

Definition 19 (Operations on energy functions)**.** *Let* F *and* G *two energy functions and* $\{\delta_i\}_{0\leq i\leq n}$ *bound to* (F, G), *where* $n = |\mathrm{Disc}(F, G)| - 1$.

The additive and multiplicative operations on two energy functions F and G are defined as

$$F \oplus G := \{F|_{\delta_i} \oplus G|_{\delta_i}\}_{1 \leq i \leq n} \qquad \text{and} \qquad F \otimes G := \{F|_{\delta_i} \times G\}_{1 \leq i \leq n}$$

Note that the definition of $\otimes$ holds thanks to Corollary 16, which ensures that no pair of energy segments of the resulting energy function overlaps.

Proposition 20.

$$\forall (F, G) \in EF(WU)^2, \forall x \in [0, WU] \mid (F \otimes G)(x) = G(F(x))$$

Proof.

Let F, G two energy functions, $x \in [0, WU]$, $n = |\text{Disc}(F, G)| - 1$ and $j \in [1, n]$ so that $x \in \delta_j$. We have $(F \otimes G)(x) = \{F|_{\delta_i} \times G\}_{1 \leq i \leq n}(x) = (F|_{\delta_j} \times G)(x)$.

We then reason on the type of the energy segment $F|_{\delta_j}$.

- If $F|_{\delta_j}$ is the null energy segment, then $F|_{\delta_j} \times G$ is the null energy segment on δ_j as well. This also means that $F(x) = \bot$, as such $(F|_{\delta_j} \times G)(x) = G(F(x)) = \bot$.
- If $F|_{\delta_j}$ is a constant energy segment of equation $e_{in} \longmapsto b$, then $F|_{\delta_j} \times G$ is a constant energy segment of equation $e_{in} \longmapsto G(b)$. This also means that $F(x) = b$, as such $(F|_{\delta_j} \times G)(x) = G(b) = G(F(x))$.
- If $F|_{\delta_j}$ is an ascending energy segment of equation $e_{in} \longmapsto e_{in} + b$, then $F|_{\delta_j} \times G$ is a set of energy segments $\{\xi_k\}_{1 \leq k \leq t}$, where the $\{\xi_k\}_{1 \leq k \leq t}$ are defined in Definition 18 and $t = |\text{Disc}(G) \cap \text{Im}(F|_{\delta_j})| + 1$. As $x \in \delta_j$, there is a k such that $x \in \text{dom}(\xi_k)$. We consider w.l.o.g. that $\text{dom}(\xi_k) = [\alpha, \beta[\subseteq \delta_j$.

  By definition, ξ_k is an energy segment which equation comes from G and respects the following property: $\forall u \in \text{dom}(\xi_k) \mid \xi_k(u) = G(F(u))$. In particular, $x \in \text{dom}(\xi_k)$. Thus $\xi_k(x) = (F|_{\delta_j} \times G)(x) = G(F(x))$.

■

Proposition 21. *$EF(WU)$ is closed under $\oplus$ and $\otimes$.*

Proof.

Let F and G two energy functions. We consider $H = \{h_i\}_{1 \leq i \leq n}$ the result of $F \oplus G$ (the proof is similar when considering $F \otimes G$).

For $i \in [1, p]$, h_i is either an energy segment defined on δ_i or a set of energy segments whose domains form a partition of δ_i. We deduce the result since the $(\delta_i)_{1 \leq i \leq n}$, by definition, form a partition of $[0, WU]$. ■

Definition 22. *Let f, g two energy segments defined on their respective energy domains.*

If $\text{dom}(f)$ *is closed to the right and* $\text{dom}(g)$ *is open to the left, or the other way around, and if f and g share the same equation and predecessor, we can form a new energy segment h with the same equation and predecessor as f and g that is defined on* $\text{dom}(f) \cup \text{dom}(g)$.

We say that h is the assembly of f and g, or that we assemble f and g to form the new energy segment h. This operation is written as $f \diamond g$.

We also say that f and g are assemblable if $f \diamond g$ exists.

In the following, if f is an energy segment f_1 and f_2 are assemblable energy segments so that $f_1 \diamond f_2 = f$, we directly write f instead of writing $\{f_1, f_2\}$ to designate the result of the assembly operation.

In particular, we also have:

Proposition 23. *Let f an energy segment defined w.l.o.g. on the energy domain $I = [\alpha, \beta[$, and $\tau \in I$ so that $\alpha < \tau < \beta$. We have*

$$f|_{[\alpha,\tau[} \diamond f|_{[\tau,\beta[} = f$$

Proof. We notice that $f|_{[\alpha,\tau[}$ and $f|_{[\tau,\beta[}$ have the same equation and predecessor, and $[\alpha, \tau[\cup [\tau, \beta[= [\alpha, \beta[= I$, thus the result. ■

Proposition 24. *Let $(\alpha, \beta) \in [0, WU]^2$ where $\alpha < \beta$, I an energy domain defined from α to β, which will be supposed w.l.o.g. closed to the left and open to the right, $\tau \in I$ so that $\alpha < \tau < \beta$, f and g two energy segments defined on I, and H an energy function.*

We have

$$\{f|_{[\alpha,\tau[} \oplus g|_{[\alpha,\tau[}, f|_{[\tau,\beta[} \oplus g|_{[\tau,\beta[}\} = f \oplus g$$

and

$$\{f|_{[\alpha,\tau[} \times H, f|_{[\tau,\beta[} \times H\} = f \times H$$

In other words, it is possible to cut f into two energy segments when calculating $f \oplus g$ or $f \times H$ without changing the result of the operation. A direct corollary of this proposition is that f can be further cut into more than two energy segments.

Proof. Let w.l.o.g. $I = [\alpha, \beta[$, $\tau \in I$ so that $\alpha < \tau < \beta$, f and g defined on I and H an energy function. To save space, let $A = [\alpha, \tau[$ and $B = [\tau, \beta[$.

- For $\oplus$:
  - if f and g do not intersect, then we suppose w.l.o.g. that $f \oplus g = f$. By definition of $\oplus$, we still have $f|_A \oplus g|_A = f|_A$ and $f|_B \oplus g|_B = f|_B$. We obtain the result by using Proposition 23.
  - else, f and g intersect at $\iota \in I$. We suppose w.l.o.g. that the maximal segment on $[\alpha, \iota[$ is f and the maximal segment on $[\iota, \beta[$ is g, i.e. $f \oplus g = \{f|_{[\alpha,\iota[}, g|_{[\iota,\beta[}\}$. We also suppose w.l.o.g. that $\tau \in [\alpha, \iota[$.

    By definition of $\oplus$, we have $f|_A \oplus g|_A = f|_A$. Also by definition of $\oplus$, we have $f|_B \oplus g|_B = \{f|_{[\tau,\iota[}, g|_{[\iota,\beta[}\}$. We obtain the result by noticing that $f|_A$ and $f|_{[\tau,\iota[}$ are assemblable and $f|_A \diamond f|_{[\tau,\iota[} = f|_{[\alpha,\iota[}$.
- For $\times$: the cases where f is the null energy segment on I or a constant energy segment are directly deductible from Definition 18. As such, we will consider f to be an increasing energy segment. We will also consider w.l.o.g. that the energy domain of f is of the form $[\alpha, \beta[$. According to Proposition 7, $f|_A$ and $f|_B$ are also (increasing) energy segments with the same equation and predecessor as f.

  In this case, $f \times H$ is a set of $p > 1$ energy segments, and $f \times H = \{\xi_i\}_{1 \leq i \leq p}$ where the $\{\xi_i\}_{1 \leq i \leq p}$ are defined in Definition 18, with their respective domains $\{[f^{-1}(d_i), f^{-1}(d_{i+1})[\}_{1 \leq i \leq p}$, and the $(d_i)_{1 \leq i \leq p+1}$ are the discontinuities that remain in $G|_{\mathrm{Im}(f)}$.

  By definition, there exists a ξ_k with $1 \leq k \leq p$ where $\tau \in \mathrm{dom}(\xi_k)$. τ partitions the domain of ξ_k into two intervals $[d_k, \tau[$ and $[\tau, d_{k+1}[$. The other $\{\xi_i\}_{1 \leq i \leq n | i \neq k}$ are identical by definition to the energy segments that appear in $\{f|_A \times H, f|_B \times H\}$. We deduce the result by noticing that the energy segment defined on $[d_k, \tau[$ and the energy segment defined on $[\tau, d_{k+1}[$ both with same equation and predecessor as ξ_k are assemblable and yield ξ_k once assembled.

■

Proposition 25. *Let F and G two energy functions, $n = |\mathrm{Disc}(F,G)| - 1$, $r \geq n$ and $D' = \{D'_k\}_{0 \leq k \leq r} \in [0, WU]^r$ an ascending set of discontinuities so that $\mathrm{Disc}(F,G) \subseteq D'$.*

The elements of D' form pairs of discontinuities written as $\{\theta_k\}_{1 \leq k \leq r}$ to avoid confusion with pairs of discontinuities generated by $\mathrm{Disc}(F,G)$. In this case, we have:

$$\{F|_{\theta_i} \oplus G|_{\theta_i}\}_{1 \leq i \leq r} = F \oplus G \qquad \text{and} \qquad \{F|_{\theta_i} \times G\}_{1 \leq i \leq r} = F \otimes G$$

Proof. This results from the application of Proposition 24 on every energy segment of F and G for the $\oplus$ case, and every energy segment of F for the $\otimes$ case. ■

This proposition allows us to introduce additional arbitrary discontinuities in the computation of $F \oplus G$ or $F \otimes G$.

It also allows us to introduce the cleaning algorithm, presented in Algorithm 10, that will be used as a handy utility tool for building energy functions in Section 5: it merges adjacent energy segments with the same equation and predecessor to reduce the total number of energy segments. Note that this algorithm clearly returns a set of energy segments where their domains form a partition of $[0, WU]$, i.e. the cleaning algorithm returns an energy function. This cleaned energy function is equivalent to its uncleaned counterpart thanks to Proposition 25.

Definition 26 (Cleaned function)**.** *An energy function $F = \{f_i\}_{1 \leq i \leq n}$ is cleaned iff no pairs of energy segments of F are assemblable, i.e.*

$$F \text{ is cleaned} \iff \forall (i,j) \in [1,n]^2 \mid i \neq j \implies f_i \text{ and } f_j \text{ are not assemblable}$$

Algorithm 10: Cleaning algorithm

Data: a set of energy segments Y that form an energy function
Result: a set of energy segments that is equivalent to Y

```
1  begin
2      new_segs ⟵ []
3      next_seg ⟵ None
4      for old_seg : e_in ⟼ a e_in + b ∈ Y do
           /* merge segments with the same equation                    */
5          if old_seg and next_seg are assemblable then
6              next_seg ⟵ next_seg ⋄ old_seg
7          else
8              add next_seg to new_segs
9              next_seg ⟵ old_seg
           /* add the remaining segment                                */
10     if next_seg ≠ None then
11         add next_seg to new_segs
12     return the cleaned energy function formed by new_segs
```

Theorem 27. *Let F an energy function. In the weak upper bound context, F is increasing.*

We will demonstrate Theorem 27 in Section 4. For now, we consider energy functions to be increasing.

Theorem 28. *Let $\bar{0}$ the energy function composed of the single segment $e_{in} \longmapsto \bot$ (predecessor* `undef`*) defined on $[0, WU]$, and $\bar{1}$ the energy function composed of the single segment $e_{in} \longmapsto e_{in}$ (predecessor* `undef`*) defined on $[0, WU]$.*
$(EF(WU), \oplus, \otimes, \bar{0}, \bar{1})$ is a semiring.

Proof.

- $EF(WU)$ with $\oplus$ and $\bar{0}$ is a commutative monoid:
  - $\oplus$ is stable in $EF(WU)$ by Proposition 21;
  - the neutral element for $\oplus$ is $\bar{0}$;
  - it is clear that $\oplus$ is associative as well as commutative (by properties of $\oplus$ on energy segments).
- $EF(WU)$ with $\otimes$ and $\bar{1}$ is a monoid:
  - $\otimes$ is stable in $EF(WU)$ by Proposition 21;
  - the neutral element for $\otimes$ is $\bar{1}$;
  - let F, G, H three energy functions and $x \in [0, WU]$. Using Proposition 20, we have

$$\begin{aligned}((F \otimes G) \otimes H)(x) &= H((F \otimes G)(x)) \\ &= H(G(F(x)))\end{aligned}$$

    and

$$\begin{aligned}(F \otimes (G \otimes H))(x) &= (G \otimes H)(F(x)) \\ &= H(G(F(x)))\end{aligned}$$

    Thus $\otimes$ is associative.
- $\otimes$ is distributive to the right over $\oplus$: let $(F, G, H) \in EF(WU)^3$, $D = (D_k)_{1 \leq k \leq r+1}$, $n = |\mathrm{Disc}(F, G, H)| - 1$ and $\{\delta_i\}_{1 \leq i \leq n}$ bound to F, G and H.

  We have

$$(F \oplus G) \otimes H = \big(((F \oplus G) \otimes H)|_{\delta_i}\big)_{1 \leq i \leq n}$$

  and

$$(F \otimes H) \oplus (G \otimes H) = \big(((F \otimes H) \oplus (G \otimes H))|_{\delta_i}\big)_{1 \leq i \leq n}$$

  By Proposition 21, these are energy functions.

  Let the following energy functions:

$$A = \big(((F \oplus G) \otimes H)|_{\delta_i}\big)_{1 \leq i \leq n} \text{ and } B = \big(((F \otimes H) \oplus (G \otimes H))|_{\delta_i}\big)_{1 \leq i \leq n}$$

  We can then prove that

$$\forall x \in [0, WU] \mid A(x) = B(x)$$

  Let $x \in [0, WU]$. By definition of D, there exists a $k \in [1, n]$ such that $x \in \delta_k$. Suppose w.l.o.g. that $F|_{\delta_k} \oplus G|_{\delta_k} = F|_{\delta_k}$, i.e. $\forall x \in \delta_k \mid F(x) \geq G(x)$ (if these energy segments intersect, we can restrict further δ_k by considering the intersection of the two relevant energy segments).

$$
\begin{aligned}
A(x) &= ((F \oplus G) \otimes H)|_{\delta_k}(x) \\
&= \{(F \oplus G)|_{\delta_j} \times H\}_{1 \leq j \leq n}|_{\delta_k}(x) && \text{[Definition 19]} \\
&= ((F \oplus G)|_{\delta_k} \times H)(x) \\
&= (\{F|_{\delta_j} \oplus G|_{\delta_j}\}_{1 \leq j \leq n}|_{\delta_k} \times H)(x) && \text{[Definition 19]} \\
&= ((F|_{\delta_k} \oplus G|_{\delta_k}) \times H)(x) \\
&= (F|_{\delta_k} \times H)(x) && \text{[initial supposition]} \\
&= (F \otimes H)(x) && [x \in \delta_k] \\
&= H(F(x)) && \text{[Proposition 20]}
\end{aligned}
$$

and

$$
\begin{aligned}
B(x) &= ((F \otimes H) \oplus (G \otimes H))|_{\delta_k}(x) \\
&= (\{F|_{\delta_j} \times H\}_{1 \leq j \leq n} \oplus \{G|_{\delta_j} \times H\}_{1 \leq j \leq n})|_{\delta_k}(x) && \text{[Definition 19]} \\
&= ((F|_{\delta_k} \times H) \oplus (G|_{\delta_k} \times H))(x) \\
&= ((F \otimes H)|_{\delta_k} \oplus (G \otimes H)|_{\delta_k})(x) && \text{[Definition 19]} \\
&= \max((F \otimes H)|_{\delta_k}(x), (G \otimes H)|_{\delta_k}(x)) && \text{[Definition 17]} \\
&= \max((F \otimes H)(x), (G \otimes H)(x)) && [x \in \delta_k] \\
&= \max(H(F(x)), H(G(x))) && \text{[Proposition 20]} \\
&= H(F(x)) && [H \text{ is increasing, initial supposition}] \\
&= A(x)
\end{aligned}
$$

Thus $\otimes$ is distributive to the right over $\oplus$. The demonstration of the distributivity of $\otimes$ to the left is similar.

- the null energy function is an annihilator for $\otimes$ to the left and to the right, by definition of $\otimes$ on energy segments.

■

This allows us to use the Floyd-Warshall algorithm on ω-automata weighted with energy functions. However, for the time being, $\mathcal{A}$ (the automaton we're working on) is still valued with integer weights. As such, we need a means of translating integer weights in $\mathcal{A}$ into elements of our newly defined semiring.

4 ▷ Application of energy functions to energy problems

In this section, we will use energy functions as a new means of solving energy problems on WWAs.

4.1 ▷ Mutators

To transform our WWA with integer weights $\mathcal{A}$ into an automaton compatible with our newly defined energy functions, we first need to define an intermediate function that returns an appropriate energy function from an integer-weighted transition in $\mathcal{A}$.

We recall that an integer-weighted WWA is a tuple of the form $(\mathcal{M}, S, s_0, T, \alpha)$ where:

- $\mathcal{M}$ is a finite set of integer-labeled colors;
- S is a set of integer-labeled states;

- $s_0 \in S$;
- $T \subseteq S \times 2^{\mathcal{M}} \times R \times S$ is a set of transitions valued in $\mathbb{Z}$. We call $U_{\mathbb{Z}} = \mathbb{N} \times 2^{\mathbb{N}} \times \mathbb{Z} \times \mathbb{N}$ the (infinite) set of all possible $\mathbb{Z}$-valued transitions of an automaton of $WWA(\mathbb{Z})$;
- α is an acceptance condition as defined in Section 1.

Let $WWA(\mathbb{Z})$ the set of integer-weighted ω-automata. We similarly call the set of energy function-weighted ω-automata $WWA(EF(WU))$.

As such, we provide means of converting the automaton $\mathcal{A} \in WWA(\mathbb{Z})$ to a new automaton $\mathcal{A}' \in WWA(EF(WU))$.

Definition 29 (Integer mutator)**.** *Let R a semiring. An integer mutator is a total function $\sigma_R : U_{\mathbb{Z}} \longrightarrow U_R$.*

An integer mutator is effectively a function that allows to convert the weight of an integer-weighted transition into an element of R. Since $EF(WU)$ is a semiring, we can define $\sigma_{EF(WU)}$.

Definition 30 (Integer $EF(WU)$-mutator)**.** *Let $t = (src, M, x, dst) \in U_{\mathbb{Z}}$ and F_t the energy function defined as*

- *if $x \geq WU$, then F_t is the energy function composed of the sole energy segment of equation $e_{in} \longmapsto WU$ defined on $[0, WU]$;*
- *if $0 > x > WU$, then F_t is composed of two energy segments:*
  - *an energy segment defined on $[0, WU - x[$ of equation $e_{in} \longmapsto e_{in} + x$,*
  - *an energy segment defined on $[WU - x, WU]$ of equation $e_{in} \longmapsto WU$;*
- *if $x = 0$, then F_t is composed of the sole energy segment of equation $e_{in} \longmapsto e_{in}$ defined on $[0, WU]$.*
- *if $-WU < x < 0$, then F_t is composed of two energy segments:*
  - *a null energy segment defined on $[0, -x[$,*
  - *an energy segment defined on $[-x, WU]$ of equation $e_{in} \longmapsto e_{in} + x$;*
- *if $x \leq -WU$, then F_t is composed of the sole null energy segment defined on $[0, WU]$;*

Every energy segment that appears in the definition of F_t that is not the null energy segment has a predecessor equal to the source state of t.

We then define the integer $EF(WU)$-mutator $\sigma_{EF(WU)}$ as the function that maps t to a new transition weighted with F_t of source state, acceptance sets and destination sets equal to the ones in t.

$\sigma_{EF(WU)}$ is well-defined, as it treats all possible cases for the weight of $t \in U_{\mathbb{Z}}$. Note that $\sigma_{EF(WU)}$ preserves the source / destination state and the colors information. The resulting transitions are elements of $U_{EF(WU)}$; some examples (where $WU = 10$) are provided in Figures 18 and 19.

By abuse of notation, if t is a transition weighted with the integer k, we also use $\sigma_{EF(WU)}(t)$ to refer to the weight of the transition that results from the application of the integer $EF(WU)$-mutator, i.e. the energy function that is associated with k. This allows us to write, for example, $\sigma_{EF(WU)}(t) \oplus \sigma_{EF(WU)}(t')$, where t and t' are arbitrary integer-weighted transitions, even though $\sigma_{EF(WU)}(t)$ and $\sigma_{EF(WU)}(t')$ are not energy functions but transitions.

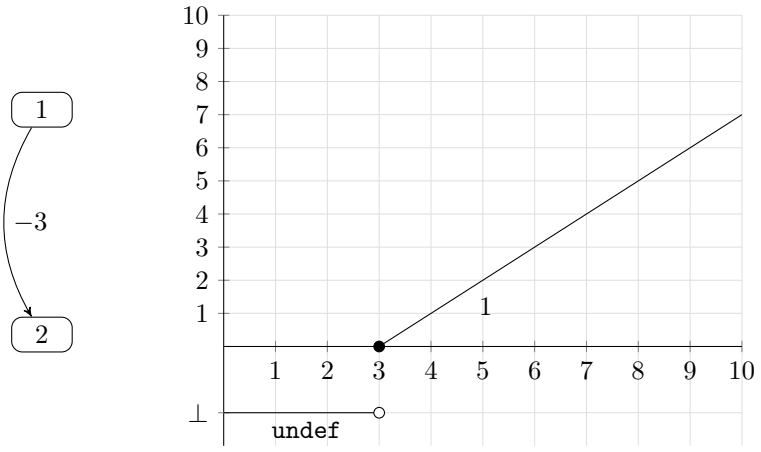


Figure 18: Equivalent energy function for a transition going from 1 to 2 with weight -3.

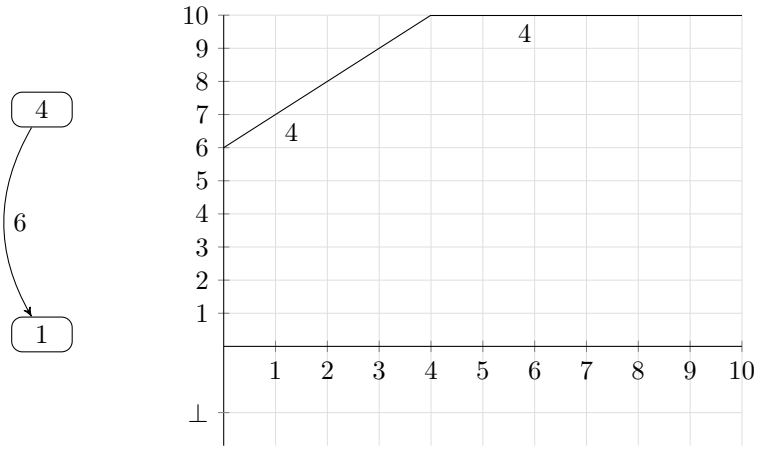


Figure 19: Equivalent energy function for a transition going from 4 to 1 with weight 6.

As seen in the previous section, the $\oplus$ and $\otimes$ operators on an energy function generated by a sequence of transitions T can be interpreted in WWAs as respectively the comparison of T with another sequence of transitions, and the addition of a new transition that starts from the final state reached by using T.

In Figure 20, to go from 1 to 3, we need to use the transition t from 1 to 2, then the transition t' from 2 to 3. Thus, to calculate the energy function associated with transitions from 1 to 3, we need to calculate the energy function $\sigma_{EF(WU)}(t) \otimes \sigma_{EF(WU)}(t')$. Note that since $\otimes$ is not commutative, the result energy function is different from $\sigma_{EF(WU)}(t') \otimes \sigma_{EF(WU)}(t)$, which would be the use of a transition weighted with 8 followed by the use of a transition weighted with -5.

In Figure 21, to go from 1 to 4, we have two choices:

- use the transition t_{12} from 1 to 2 weighted with -2 then the transition t_{24} from 2 to 4 weighted with 4 (path T_2);
- or use the transition t_{13} from 1 to 3 weighted with -5 then the transition t_{34} from 3 to 4 weighted with 10 (path T_3).

The energy functions associated with paths T_2 and T_3 can be calculated using $\otimes$:

$$\sigma_{EF(WU)}(T_2) = \sigma_{EF(WU)}(t_{12}) \otimes \sigma_{EF(WU)}(t_{24})$$
$$\sigma_{EF(WU)}(T_3) = \sigma_{EF(WU)}(t_{13}) \otimes \sigma_{EF(WU)}(t_{34})$$

We can then use $\oplus$ to calculate the energy function F associated with the optimal paths to use when going from 1 to 4:

$$F = \sigma_{EF(WU)}(T_2) \oplus \sigma_{EF(WU)}(T_3)$$

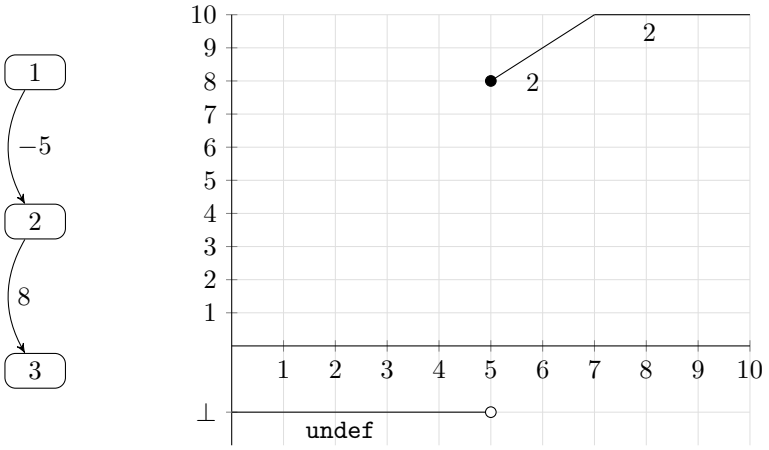


Figure 20: Energy function associated with transitions from 1 to 3.

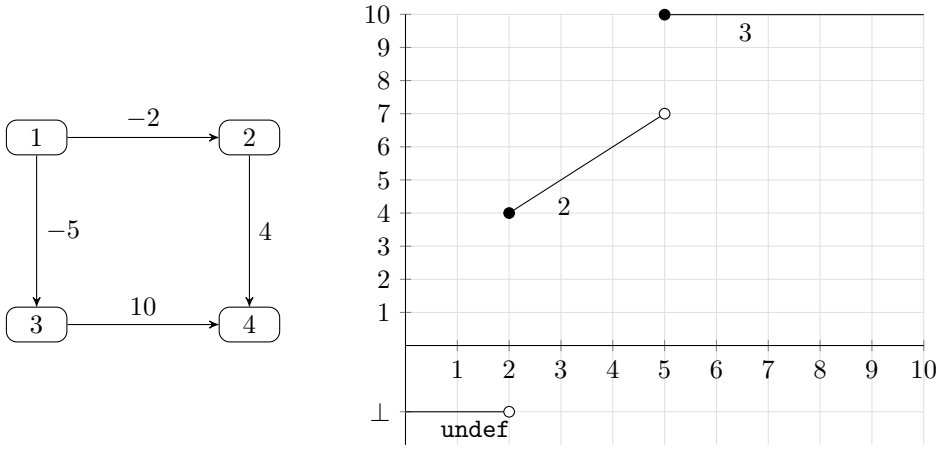


Figure 21: Energy function associated with transitions from 1 to 4.

We can also initially associate paths from a state to itself as the identity energy function, that is the energy function with the sole energy segment of equation $e_{in} \longmapsto e_{in}$ defined on $[0, WU]$ with predecessor `undef`. In this case, we can understand this function as "no transition was taken", but in a more global scope, it is also understandable as "it is (energetically) better to stay at the current state rather than use some transitions to loop back at this state".

This finally allows us to explain why we introduced the notion of energy domain in Definition 1: when searching for energy feasible loops in a WWA, we may stumble upon a particular energy-neutral loop, such as in Figure 22. This kind of loop, even though it does not change the attained energy at a state, is better than staying at the state without using a loop.

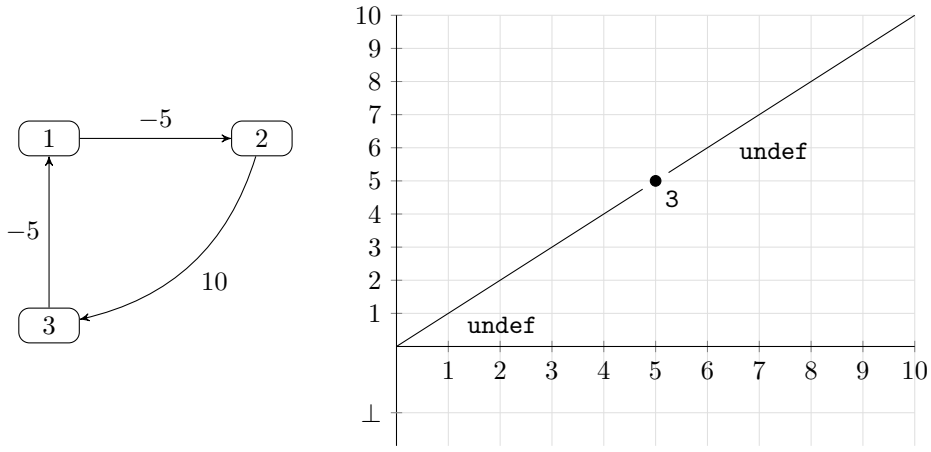


Figure 22: An automaton with an energy-neutral loop symbolized by an energy point in the energy function associated with the optimal energy path from 1 to 1. Here, $WU = 10$.

In this portion of automaton, there is an energy-feasible loop that always allows us to attain a final energy of 5 if the initial energy is greater than 5. However, this also means we would lose energy if the initial energy is strictly greater than 5. As such, the loop is only interesting if the initial energy is equal to 5. This case cannot be represented in an energy function if we only allow energy domains of the form $[\alpha, \beta[$ (eventually closed to the left if $\beta = WU$).

This finally allows us to demonstrate Theorem 27 presented in Section 3.3.

Proof. (*Theorem 27*)
Let F an energy function. If F is not an increasing energy function, this means that there exists two consecutive energy segments f and f' of F, respectively associated with paths T_1 and T_2 in the initial WWA, where the value of f' at the boundary $x \in [0, WU]$ is strictly inferior to the value of f if it was extended to x (as energy segments form a partition of $[0, WU]$, f is not defined at x, so we use the limit), i.e. $f'(x) < \lim_{t \to x} f(t)$.

This would mean that it is more efficient to use path T_2 than path T_1 when the initial energy is x, which is a contradiction: the condition $f'(x) < \lim_{t \to x} f(t)$ means that it is possible to use path T_1 with an initial energy of x to attain a final energy of $\lim_{t \to x} f(t)$. ■

Definition 31. *Let $\mathcal{A} = (\mathcal{M}, S, s_0, T, \alpha) \in WWA(\mathbb{Z})$.*

We can build a set of transitions T' equivalent to T by applying $\sigma_{EF(WU)}$ to every element of T:

$$T' = \{\sigma_{EF(WU)}(t) \mid t \in T\}$$

This set of transitions is, by definition, weighted in $EF(WU)$. By abuse of notation, to save space, we write $T' = \sigma_{EF(WU)}(T)$ to designate the resulting transitions instead of the set notation.

We define the mutation of automata weighted in $\mathbb{Z}$ to automata weighted in $EF(WU)$, written as $\mathrm{mut}_{EF(WU)}$, as the function from $WWA(\mathbb{Z})$ to $WWA(EF(WU))$ defined as

$$\mathrm{mut}_{EF(WU)} : \mathcal{A} = (\mathcal{M}, S, s_0, T, \alpha) \longmapsto (\mathcal{M}, S, s_0, \sigma_{EF(WU)}(T), \alpha)$$

The previous two operations can be visualized as the following:

$$U_{\mathbb{Z}} \xrightarrow{\sigma_{EF(WU)}} U_{EF(WU)}$$

$$WWA(\mathbb{Z}) \xrightarrow{\text{mut}_{EF(WU)}} WWA(EF(WU))$$

Let's not forget the original motivation for using WWAs weighted with energy functions, that is to have an efficient algorithm for solving energy problems in co-Büchi automata: we can apply the Floyd-Warshall algorithm to our newly defined automaton $\mathcal{A}' = \text{mut}_{EF(WU)}(\mathcal{A})$, as $\mathcal{A}'$ is still weighted with elements of a semiring.

4.2 ▷ Floyd-Warshall algorithm with energy functions

The Floyd-Warshall algorithm has already been explained in Algorithm 9 on standard co-Büchi automata weighted with integers, but not yet on automata weighted with energy functions. Moreover, as we only need to know if there exists one energy-positive loop in the automaton, we may return a result early if we manage to find one.

Algorithm 11 presents the Floyd-Warshall algorithm adapted to an arbitrary semiring with the additive operation $\oplus$ and the multiplicative operation $\otimes$. $\bar{0}$ designates the neutral element for $\oplus$ (in $EF(WU)$, this is the null energy function composed of the sole energy segment of equation $e_{in} \longmapsto \bot$ defined on $[0, WU]$ with predecessor `undef`), while $\bar{1}$ designates the neutral element for $\otimes$ (in $EF(WU)$, this is the identity energy function composed of the sole energy segment of equation $e_{in} \longmapsto e_{in}$ defined on $[0, WU]$ with predecessor `undef`). Like the traditional Floyd-Warshall algorithm, this algorithm returns a matrix of the "shortest distances" in the chosen semiring, which is a square matrix of size the number of states of the automaton.

Algorithm 11: Floyd-Warshall algorithm for arbitrary semirings

Data: an energy function-weighted automaton $A = (\mathcal{M}, S, s_0, T)$, a semiring $(R, \oplus, \otimes)$ with neutral elements $\bar{0}$ and $\bar{1}$ and the associated integer mutator σ_R, a short-circuit function *check*

```
1  begin
2      n ⟵ number of states of A
3      M ⟵ n × n matrix filled with 0̄
4      for T ∋ e from u to v with weight k do
5          M[u][v] ⟵ σ_R(T)
6      for s ∈ S do
7          M[s][s] ⟵ 1̄
8      for k ∈ [1, n] do
9          for i ∈ [1, n] do
10             for j ∈ [1, n] do
11                 M[i][j] ⟵ M[i][j]⊕M[i][k]⊗M[k][j]
12                 if check(M, i, j) then
13                     return positively
14     return negatively
```

We add to the innermost loop of the Floyd-Warshall algorithm (Line 11 in Algorithm 9) a *short-circuiting function*, which is a function that takes as arguments the matrix, a source state *src* and a destination state *dst*, and that returns positively if the energy function in the matrix from *src* to *dst* represents a positive loop. This function is useful when we don't want to calculate the full matrix and only want to determine if there is an energy-feasible loop in the automaton.

By definition of energy functions, we can check if there exists an energy segment which is greater than $\bar{1}$: this is the role of the `is_above_one` class method, which is used if $src = dst$, i.e. if the current energy function corresponds to a path from a state to itself.

5 ▷ Implementation of energy functions

5.1 ▷ Structure

We recall the address of the GitHub repository containing our implementations, which is `https://github.com/PhilippSchlehuberCaissier/wspot/tree/thay`. This repository contains the `tests` directory containing various test automata in HOA format (presented in [3]), whereas the `code` directory contains the bulk of our implementations:

- `*_benchmark*.py` designate one-off scripts that are made for benchmarking various aspects of our implementations: execution time, allocated memory...
- there are `ipynb` files, which are Jupyter notebooks that provide a visual interface for our algorithms. For example, this allows us via the Spot library (that is, its Python bindings) to visualize automata that are in the HOA format.
- `energy.py` is the implementation of energy-related classes;
- `integer.py` is an implementation of the tropical semiring used in classic applications of the Floyd-Warshall algorithm on weighted directed graphs;
- `ipython_utils.py` are a set of functions targeted towards Jupyter and IPython sessions that print debug information. These functions are controlled by a `IPythonUtils` object that determines the quantity of information to be printed (textual information in IPython sessions, graphical output in Jupyter notebooks); this class implements the Singleton pattern;
- `*_builder.py` are scripts containing classes designed to automatically build automata;
- `semiring.py` contains an abstract class describing a semiring with its associated integer mutator;
- `tests_energy.py` uses the `unittest` Python library to test our implementations of various semirings and their operations;
- `WBA_FW.py` is an implementation of the generalized Floyd-Warshall algorithm described in Algorithm 11;
- `WBA_solvers.py` contains implementations of various energy problem solvers;
- `WBA_utils.py` functions as a switch that chooses the correct solver for a WWA depending on the type of its acceptance condition (Büchi, co-Büchi, parity, etc.);
- other files are either the result of previous works, such as `tchecker` which is used in [8], or utility files such as the Doxygen documentation generation scripts.

The solver for co-Büchi using the Floyd-Warshall algorithm and energy functions has also been implemented in the `WBA_solvers.py` script. Energy functions and segments are implemented in a separate file, `energy.py`.

To be able to use the Floyd-Warshall algorithm on weighted automata using other weight types than integers, we use an abstract class representing a semiring equipped with its associated integer mutator (defined in Definition 29) called `transition_to_sr`:

Semiring
__add__(other: Semiring): Semiring *__mul__(other: Semiring): Semiring* *transition_to_sr(e: edge, weight: int): Semiring* *zero(): Semiring* *one(): Semiring*

To test our Floyd-Warshall algorithm implementation, we also use the `Semiring` abstract class to implement the tropical semiring (the semiring of integers equipped with min as additive operator and + as multiplicative operator) in the `integer.py` script.

Our implementation of energy functions in `wspot` revolves mainly around the `EnergySegment` and `EnergyFunction` classes that are described below.

EnergySegment
lowerBound: int upperBound: str pred: int a: int b: int is_zero: bool is_above_one: bool domain: (int, int) image: (int, int)
__add__(other: EnergySegment): EnergyFunction is_in_domain(x: int): bool evaluate(x: int): int restriction(low: int, upp: int): EnergySegment zero(low: int, upp: int, pred: int?): EnergySegment one(low: int, upp: int, pred: int?): EnergySegment const(low: int, upp: int, pred: int?, k: int): EnergySegment incr(low: int, upp: int, pred: int?, b: int): EnergySegment

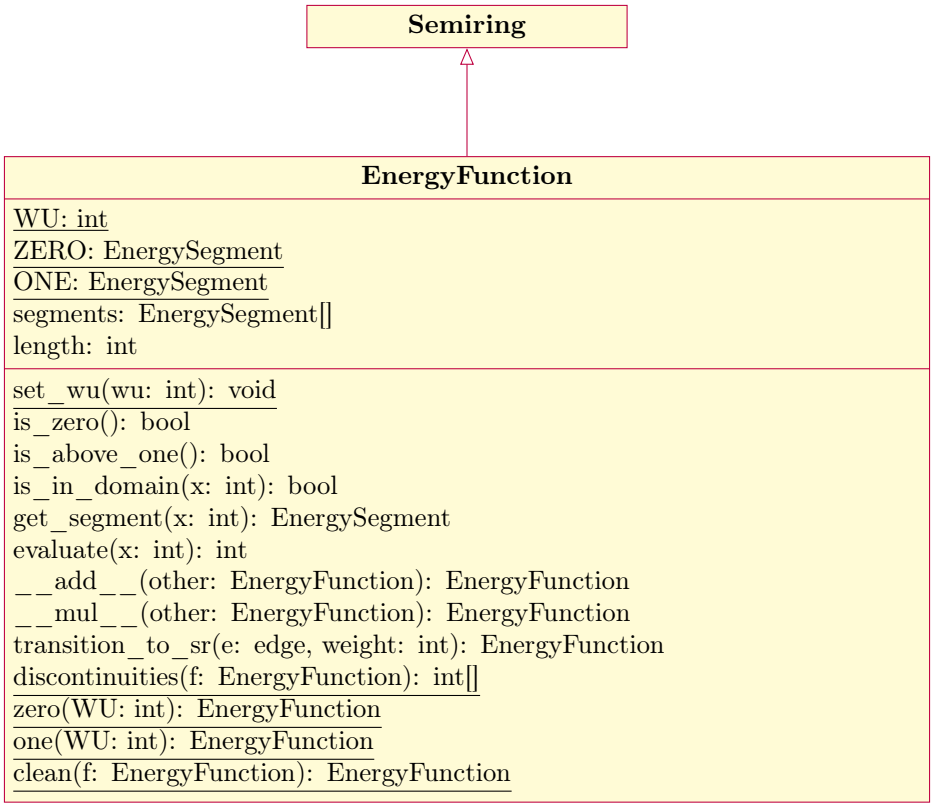
The `EnergyFunction` class is a full implementation of the `Semiring` abstract class. It also contains other utility functions such as the cleaning algorithm already presented in Algorithm 10.

However, we do not implement energy domains in this class and rather consider all energy domains to be of the form $[\alpha, \beta[\in [0, WU]$. This may not make sense in the case an energy segment's domain is a point (an energy domain of the form $[\alpha, \alpha]$), but we can show that this does not matter when searching energy feasible loops.

Indeed, when checking if an energy function corresponds to an energy-feasible loop with the `is_above_one` method, we only check if there exists a segment that is above $\bar{1}$. To do so, we only check for each segment f if its predecessor is not `undef` and if there exists an $x \in \mathrm{dom}(f)$ so that $f(x) \leq x$. Since energy segments that have a point as their energy domain respect this condition, as seen in Figure 22, the `is_above_one` method will detect that this corresponds to an energy-feasible loop.

In the following, we detail the implementation of the `__add__` ($\oplus$, Algorithm 12) and `__mul__` ($\otimes$, Algorithm 13) operators.

The additive operation between two energy functions uses `segment_at_disc`, a Python dictionary which stores for both energy functions the (unique) segment of the function that is relevant at each discontinuity, eliminating the need to use the `get_segment` method for each use of this operator as the latter is in $O(n)$. We suspect that this dictionary might grow exponentially in



size if the combined two energy functions have a high number of discontinuities, but we chose this solution over a more traditional list structure storing which segments are valid between two discontinuities (for example, segments f_1 and g_1 are used on δ_1, segments f_1 and g_2 on δ_2, etc.) for the constant access time the hash set structure provides (calculating the hash of a discontinuity, i.e. an integer, is in $O(1)$).

To reduce the total number of energy segments, we also reuse the cleaning algorithm already seen in Algorithm 10.

The multiplicative operation is more straightforward, as we only need to map the cross function over the list of segments of F.

We can calculate the complexity of these to operations to deduce the complexity of the modified Floyd-Warshall algorithm. Let F, G two energy functions and $D = |\text{Disc}(F, G)|$; we can assume that $D = O(WU)$ since there cannot be more discontinuities than WU (one per integer in $[0, WU]$) in the integer case:

- for the $\oplus$ operation, we first need an intermediate function that returns an energy segment given an energy function and an $x \in [0, WU]$. As we use a dichotomic approach for this, this intermediate function is logarithmic in the number of segments. There may be as much as WU segments in an energy function, giving us a complexity of $O(\log(WU))$. We call this intermediate function for every discontinuity in $\text{Disc}(F, G)$, so the complexity of building the `segment_at_disc` dictionary is $O(WU \log(WU))$.

  After building `segment_at_disc`, we iterate again over each discontinuity in $\text{Disc}(F, G)$. Getting the relevant energy segment given a discontinuity is in constant time, since access in a Python dictionary is in constant time. The restriction of an energy segment is also in constant time, but it creates a new energy segment. Finally, we use the $\oplus$ operator on energy segments, which is in $O(1)$ in our implementation.

  Therefore, the complexity of the $\oplus$ operation on energy segments is equal to the complexity

Algorithm 12: Additive operation ($\oplus$) on energy functions

Data: two energy functions F and G
Result: the energy function $F \oplus G$

```
begin
    new_segs ⟵ []
    discs ⟵ Disc(F, G)
    segment_at_disc ⟵ {F : {}, G : {}}
    for d ∈ discs do
        segment_at_disc[F][d] ⟵ segment of F at d
        segment_at_disc[G][d] ⟵ segment of G at d
    for i ∈ [0, discs.length − 2] do
        lower ⟵ discs[i]
        upper ⟵ discs[i + 1]
        seg1 ⟵ segment_at_disc[F][lower]
        seg2 ⟵ segment_at_disc[G][lower]
        restrict seg1 and seg2 to [lower, upper]
        add the segments from seg1 ⊕ seg2 to new_segs
    return the cleaned energy function generated by new_segs
```

Algorithm 13: Multiplicative operation ($\otimes$) on two energy functions

Data: two energy functions F and G
Result: the energy function $F \otimes G$

```
begin
    new_segs ⟵ []
    for seg ∈ F do
        add the segments from seg × G to new_segs
    return the cleaned energy function generated by new_segs
```

of creating the `segment_at_disc` dictionary, that is $O(WU \log(WU))$.

- for the $\otimes$ operation, we only need to know the complexity of the cross operator. In our implementation of this operator, which appears in the `energy.py` script as a top-level function, the worst-case complexity occurs if the energy segment f passed in argument is an increasing energy segment. In either case, the final complexity of the $\otimes$ operator only depends on the number of discontinuities of F, which is nevertheless $O(WU)$.

  In the cross operator, with an increasing energy segment f, we need to build the appropriate $\{\xi_i\}_{1 \leq i \leq n}$ defined in Definition 18. The number of new energy segments n depends on the number of discontinuities of G that intersect $\mathrm{Im}(f)$, but can be expressed as $O(WU)$. For each one of these new energy segments, we need to get the equation and predecessor of another energy segment in G: this operation, as seen in the $\oplus$ analysis, is in $O(\log(WU))$. Afterwards, we create a new energy segment, which is constant in time (but not in memory), yielding us a total complexity of $O(WU \log(WU))$.

  Since F contains $O(WU)$ energy segments, we deduct the final complexity of the $\otimes$ operator, which is $O(WU^2 \log(WU))$.

- The cleaning algorithm (Algorithm 10) is called after every use of the $\oplus$ or $\otimes$ operator. However, we can see that it only needs to iterate over the energy segments of the energy function, yielding us a complexity of $O(WU)$; we can ignore this complexity.

- Finally, in the modified Floyd-Warshall algorithm, we call both the $\oplus$ and the $\otimes$ operators inside the three nested loops. As such, the final complexity of the algorithm is $O(n^3 \cdot WU^2 \log(WU))$ where $n = |S|$ is the number of states of $\mathcal{A}$.

5.2 ▷ Performance and optimization

We test the execution times of four co-Büchi energy problem solving algorithms: the naive algorithm (Algorithm 6), the cycle storage algorithm (Algorithm 7), the algorithm using backtracking (Algorithm 8), and the modified Floyd-Warshall algorithm using energy functions (Algorithm 11 with the `EnergyFunction` class).

We first present interesting automata that can be used to test these algorithms.

Figures 23 and 24 present the *nested loops* automaton, composed of $n \geq 1$ loops: a standard DFS would have to traverse $n-1$ energy negative loops with an increasing number of states (if $k \in [1, n]$, the k-th loop will contain k states) before finding an energy positive loop. Thus, an algorithm based on a DFS, such as the cycle storage one, would need to traverse every state. Regardless of the number of loops in a nested loops automaton, we always consider $WU = 10$ in such automata. We also use a variant of this automaton, called the *non-feasible nested loops* automaton, which replaces the last positive loop by a negative loop (the last transition has a weight of -1 instead of 1): in such an automaton, there is no longer an energy-feasible loop.

Figures 25 and 26 present the "*stairs*" automaton, composed of $n \geq 1$ loops. For $k \in [1, n]$, the k-th loop is composed of a "barrier" that is only passable if the energy is greater than k. In that case, the accumulated energy is maxed to WU and then reduced to k. Thus, it is interesting to use the loop k only if the accumulated energy is equal to k (otherwise, either the energy is lesser than k and the loop is unusable, or it is greater than k but there exist other loops that will result in a higher final energy).

When using energy functions and the Floyd-Warshall algorithm *without the short-circuit checking function* to solve an energy problem on a "stairs" automaton, the energy function from 1 to itself will contain n constant energy segments (thus the "stairs" denomination). However, for an algorithm based on a DFS, measured times should be comparatively low, since every loop in such an automaton are energy-feasible.

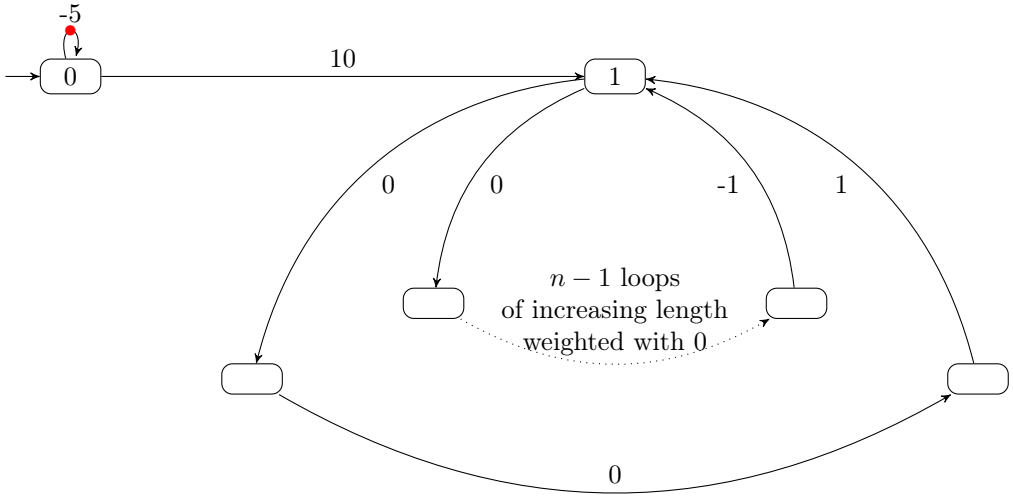


Figure 23: General form of the nested loops automaton with n loops.

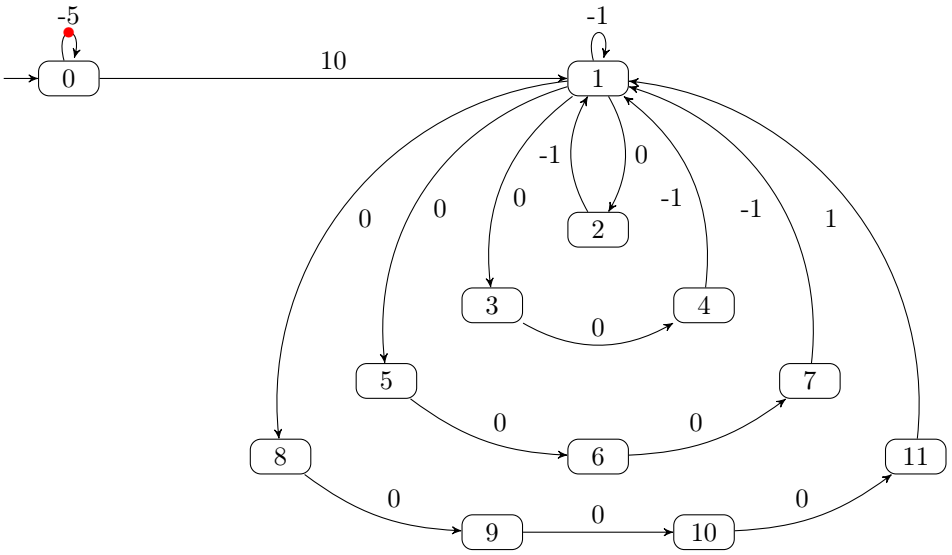


Figure 24: Automaton with five nested loops.

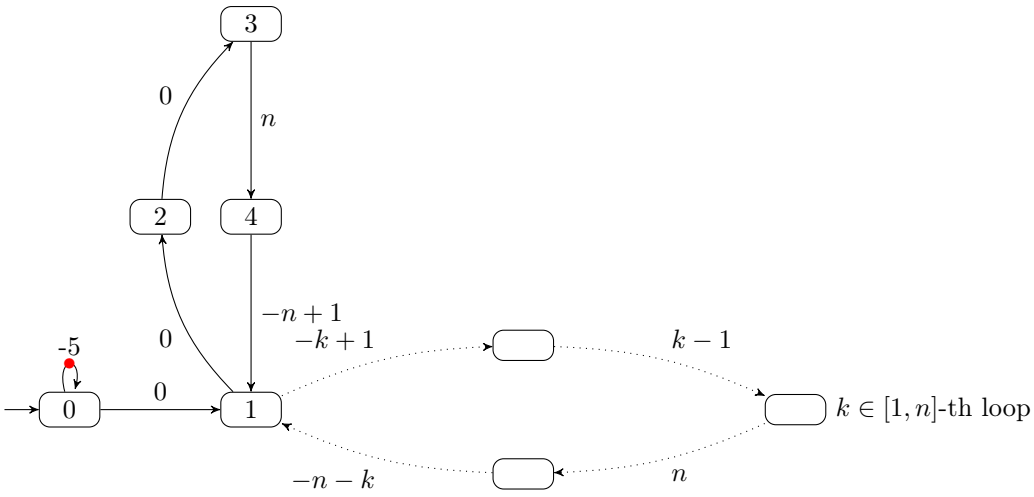


Figure 25: General form of the "stairs" automaton with n loops.

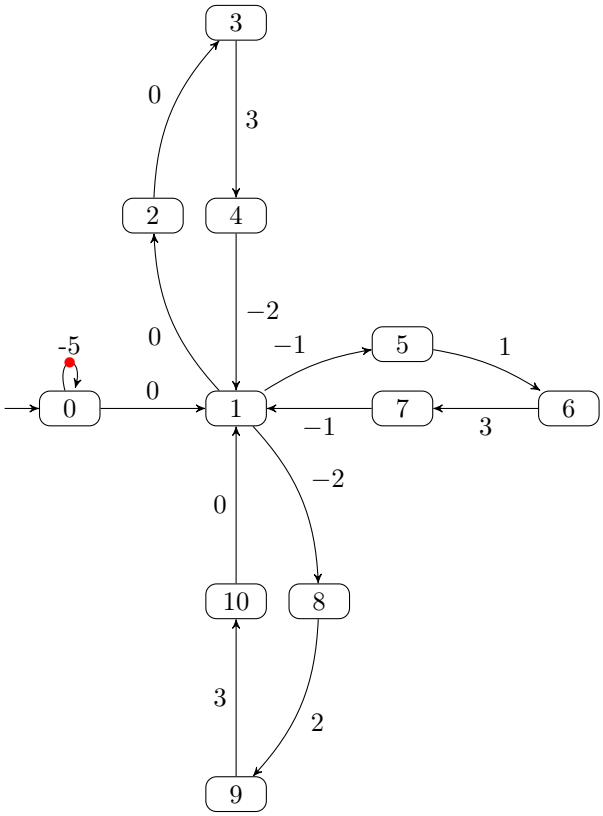


Figure 26: "Stairs" automaton with three loops.

The circling automaton (an idea of Philipp), not depicted here due to its complexity, is also another type of WWA which contains several nested loops with varying depth levels, but no energy-feasible loop: the goal of this type of automaton is to maximize the number of loops that need to be examined. We will not describe in detail these automata, but the idea for circling automata is that they have a nesting level of $N > 3$, an incremental energy of $NM \geq N$, and a recursive construction based on nested modules of N states.

These types of co-Büchi automata, which can be arbitrarily big, allow us to test our algorithms by varying their number of loops.

To build nested loops automata, we use the `NestedLoopsBuilder` class, which is implemented in the `nested_loops_builder.py` script, that builds the associated nested loops automaton given a number of loops. This script also contains the `AltNestedLoopsBuilder` class which produces nested loops automata without an energy-feasible path. Similar builders exist for stairs automata (in the `stairs_builder.py` script) and for the circling automaton (in the `devilCircles.py` script).

Execution times presented in Figure 27 were measured using the `time` Python library. We use the `algo_benchmark.py` script as a global platform for execution time measurements, as well as the usual `matplotlib` Python library for displaying our results.

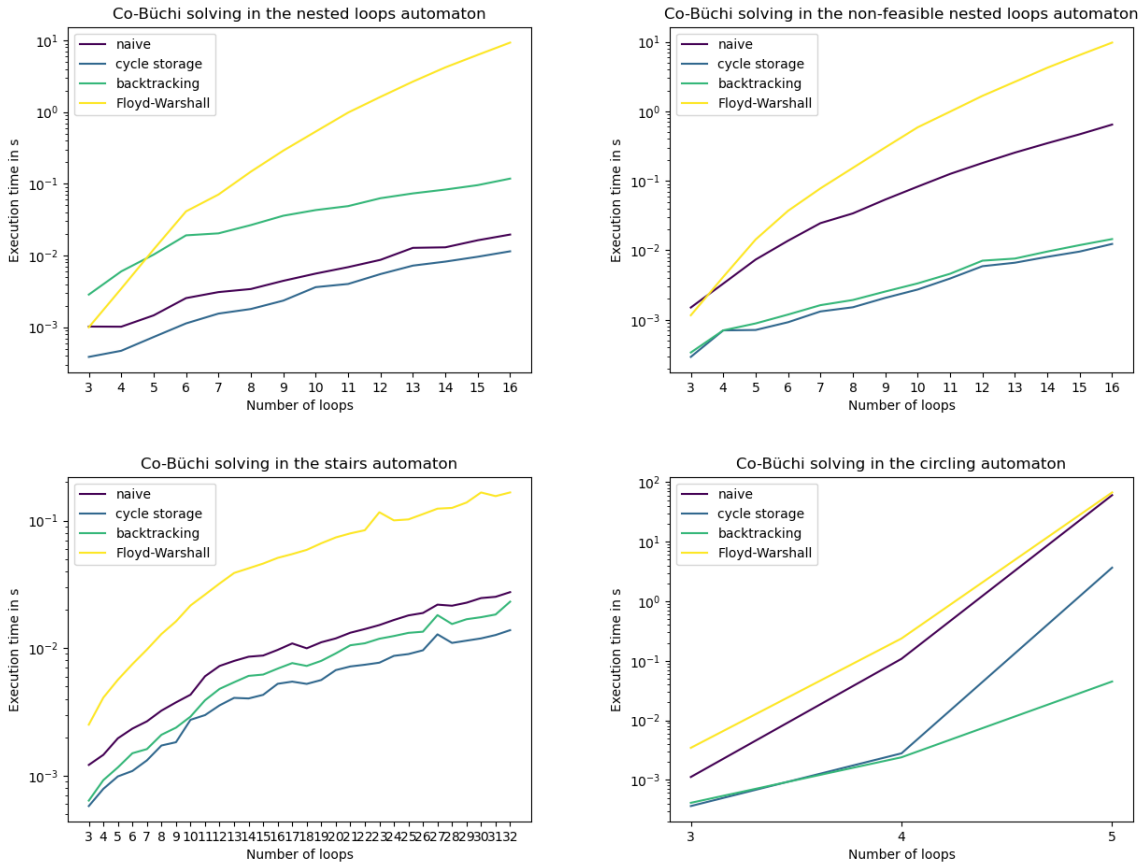


Figure 27: Execution times for various co-Büchi solving techniques in (top to bottom, left to right): the nested loops automaton, the non-feasible nested loops automaton, the "stairs" automaton and the circling automaton.

We notice that for every time of automaton except the circling one, our Floyd-Warshall based approach is consistently slower than the other three algorithms.

We also notice that our cycle storage based approach is the fastest on both types of nested loops automata as well as stairs automata, being only outperformed by the backtracking-based algorithm when solving co-Büchi problems on circling automata. The naive algorithm, as unoptimized as it may have looked when being presented in Algorithm 6, actually has similar performances to the cycle storage and backtracking algorithms on energy-feasible nested loops automata and stairs automata, but underperforms on energy-unfeasible nested loops automata and circling automata, being almost as slow as the modified Floyd-Warshall algorithm on the latter.

To find potential bottlenecks in our implementation of energy functions and their use with the Floyd-Warshall algorithm, we use the `cProfile` Python profiler [22]: implemented in C, thus minimizing its overhead; this profiler allows us to know what are the slowest functions or the most called ones. We also use the `gprof2dot` utility tool [9] that converts the output of this profiler (raw text) into a `dot` graph. The results of the profiler when applied to the use of the Floyd-Warshall algorithm to find energy-feasible paths on a nested loops automaton is illustrated in Figure 28.

In this figure, we can see that the $\oplus$ operation on energy functions is the operation that takes the most time, representing 80.05% of the total execution time of the solver, while the $\otimes$ operation

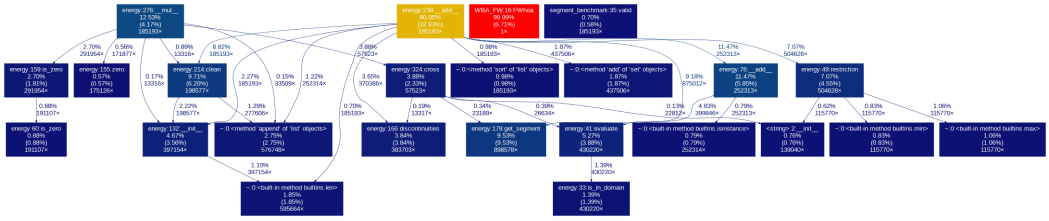


Figure 28: Graphical representation of the `cProfile` profiler run on the FLOYD-WARSHALL based co-Büchi solver on the nested loops automaton with 11 loops.

only represents 12.53% of the total execution time. We can also see that the three sub-functions called by these two operators that take the most time are the $\oplus$ operation on energy segments (in constant time but with $O(1)$ new energy segments being created), the cleaning function (linear in the number of segments of the original energy function, but it also needs to create a new energy function), and the `get_segment` function that returns an energy segment from an energy function F given an $x \in [0, WU]$ which is logarithmic in the number of segments of F, but can be considered linear in this example as the number of energy segments is low (there are many transitions weighted with 0, these transitions do not change the number of energy segments of an energy function when the $\otimes$ operator is used).

To improve the performance of the FLOYD-WARSHALL based co-Büchi solver, we have used multiple approaches:

- as mentioned in the modified FLOYD-WARSHALL algorithm complexity analysis, we used a dichotomic approach to get an energy segment of a function given an $x \in [0, WU]$;
- we have tried replacing the `segment_at_disc` dictionary generated at each call of the $\oplus$ operator on energy functions by a more general equivalent dictionary directly stored in instances of `EnergyFunctions` updated each time this energy function is used in the $\oplus$ operation. However, we figured out that since a new energy function was created for each iteration of the inner loop of the algorithm, this would represent a waste of memory space and would only be useful if energy functions were not recreated or replaced as often in the FLOYD-WARSHALL matrix;
- to check if the number of segments played a significant role in the complexity of the algorithm, we made a benchmark script (`segment_benchmark_legacy.py`) that counts for each energy function of the result matrix its number of segments. However, this didn't really matter in the end since even in the unfeasible nested loops automaton where most energy functions are identity energy functions (it is not worth using any loop in the automaton since they are all negative), execution times are still an order of magnitude higher than the other co-Büchi solving algorithms;
- we also tried to determine if the allocated memory during the problem solving was abnormally higher than other algorithms with the `memory_benchmark_new.py` script (mainly using nested loop automata), but we dropped this idea after noticing that differences were not significant between algorithms (around 120 MB allocated for all four algorithms including Python overhead). To begin to notice differences in allocated memory that are superior to the Python overhead, we may need large automata with at least thousands of states (more than 10^6 energy functions in the FLOYD-WARSHALL result matrix); with our current implementation, co-Büchi problems cannot be solved in a reasonable amount of time in such large automata;
- initially, the WU was stored in a file in `/tmp` and updated only when the `OmegaEnergy` function, in `WBA_utils.py`, was called. This would have generated a high number of I/Os,

since access to the WU is needed at least every time the $\otimes$ operator is called (i.e. at every iteration of the inner loop of the Floyd-Warshall algorithm). This was changed to use class variables and a `set_wup` static method, which updates the WU as well as the neutral energy functions for $\oplus$ and $\otimes$, instead.

6 ▷ Perspectives

For the time being, we are still in the process of finding efficient algorithms to solve WWAs using other types of acceptance conditions such as Streett ($\alpha = \bigwedge_{i=0}^{k\in\mathbb{N}^*} \texttt{Fin}(2i) \vee \texttt{Inf}(2i+1)$). The objective would be to be able to solve energy problems in Emerson-Lei automata, i.e. to solve energy problems with arbitrary acceptance conditions.

We saw that the Floyd-Warshall based co-Büchi solver underperforms compared to our other approaches. However, this algorithm has other applications: for example, in the case of the Büchi case, it would remove the time complexity dependency on the number of back-edges, the Floyd-Warshall approach having a fixed complexity of $O(n^3 \cdot WU^2 \log(WU))$, where $n = |S|$ is the number of states of the automaton. This can be useful if the number of back-edges in a WBA is high, or more generally in WBAs where there are a high number of colored transitions.

There are still cases where the modified Floyd-Warshall algorithm needs further improvements. In Figure 29, we can see that there are two energy-feasible loops (and by extension, lassos with no prefixes), one using the transition from 0 to 1 then from 1 to 2 (path T_1) and one using directly the transition from 0 to 2 (path T_2). However, when running the current version of the modified Floyd-Warshall algorithm, we would find an optimal energy path from 2 to itself that uses path T_2, which does not respect the Büchi condition, and path T_1 would be discarded even though it satisfies the acceptance condition.

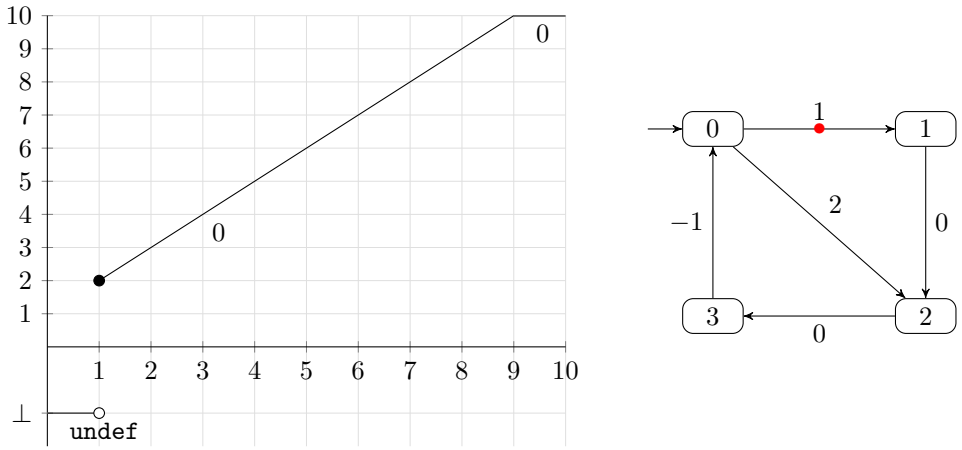


Figure 29: Our modified Floyd-Warshall algorithm applied to a Büchi automaton. The energy function from 2 to itself is depicted.

Furthermore, in the case we found an energy-feasible loop in an automaton, we would still need to determine if this loop is accessible, such as in Figure 30 where there exists an energy-feasible accepting loop ($(1 \to 2 \to 1)^\omega$) that is not accessible for $WU = 10$. We can however solve the accessibility problem by pruning parts of the automaton that are not accessible from the initial state with the initial credit, for example by using a DFS.

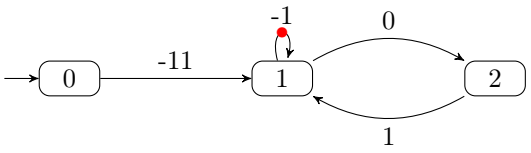


Figure 30: The Büchi accepting loop of this automaton is not accessible due to the transition between 0 and 1.

6.1 ▷ Towards a generalization of $\mathrm{mut}_{EF(WU)}$?

A new question arises with the definition of $\mathrm{mut}_{EF(WU)}$: is it possible to extend the definition of an integer mutator to any semiring? Put otherwise, is it possible to convert an automaton weighted with an arbitrary semiring R into another equivalent automaton weighted with another arbitrary semiring R', i.e. have an automaton mutator from $WWA(R)$ to $WWA(R')$? This would allow solving co-Büchi energy problems in automata with weights that are not integers (such as automata with weights in $\mathbb{R}$) but also find other exotic semirings (to be defined...) that could have interesting applications in WWAs.

We can already see that $\sigma_{EF(WU)}$ has no inverse: there exist energy functions that cannot be associated to a single transition with an integer weight, such as the one illustrated in Figure 31.

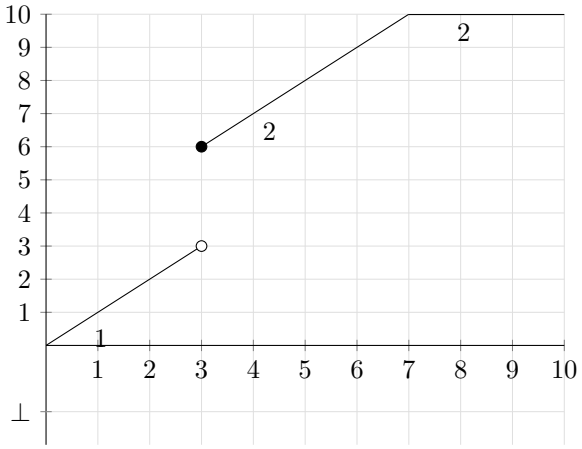


Figure 31: An energy function that cannot be interpreted as a single integer-weighted transition.

▷ References

[1] E. Allen Emerson and C.-L. Lei, "Modalities for model checking: Branching time logic strikes back," *Science of Computer Programming*, vol. 8, no. 3, pp. 275–306, 1987, ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90036-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167642387900360.

[2] E. W. Allender, "On the number of cycles possible in digraphs with large girth," *Discrete Applied Mathematics*, vol. 10, no. 3, pp. 211–225, 1985, ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(85)90044-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0166218X85900447.

[3] T. Babiak, F. Blahoudek, A. Duret-Lutz, *et al.*, "The hanoi omega-automata format," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds., Cham: Springer International Publishing, 2015, pp. 479–486, ISBN: 978-3-319-21690-4.

[4] P. Bouyer, U. Fahrenberg, K. G. Larsen, and N. Markey, "Timed automata with observers under energy constraints," in *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '10, Stockholm, Sweden: Association for Computing Machinery, 2010, pp. 61–70, ISBN: 9781605589558. DOI: 10.1145/1755952.1755963. [Online]. Available: https://doi.org/10.1145/1755952.1755963.

[5] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, and J. Srba, "Infinite Runs in Weighted Timed Automata with Energy Constraints," in *Lecture Notes in Computer Science*, Cassez, Franck, Jard, and Claude, Eds., ser. Lecture Notes in Computer Science, vol. 5215, Saint-Malo, France: Springer, 2008, pp. 33–47. DOI: 10.1007/978-3-540-85778-5_4. [Online]. Available: https://hal.science/hal-01194594.

[6] V. Capraro, A. Lentsch, D. Acemoglu, *et al.*, "The impact of generative artificial intelligence on socioeconomic inequalities and policy making," *PNAS Nexus*, vol. 3, no. 6, pgae191, Jun. 2024. DOI: 10.1093/pnasnexus/pgae191.

[7] A. Casares, "Transition-based vs stated-based acceptance for automata over infinite words," *arXiv preprint arXiv:2508.15402*, 2025.

[8] S. Dziadek, U. Fahrenberg, and P. Schlehuber, "ω-regular energy problems," *Formal Aspects of Computing*, 2022.

[9] *GitHub - jrfonseca/gprof2dot at main*, [Online; accessed 29. Sep. 2025], Sep. 2025. [Online]. Available: https://github.com/jrfonseca/gprof2dot.

[10] M. Hosseini, P. Gao, and C. Vivas-Valencia, "A social-environmental impact perspective of generative artificial intelligence," *Environmental Science and Ecotechnology*, vol. 23, p. 100 520, 2025, ISSN: 2666-4984. DOI: https://doi.org/10.1016/j.ese.2024.100520. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666498424001340.

[11] M. Mohri, "Semiring frameworks and algorithms for shortest-distance problems," *J. Autom. Lang. Comb.*, vol. 7, no. 3, pp. 321–350, Jan. 2002, ISSN: 1430-189X.

[12] M. Mohri, F. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech & Language*, vol. 16, no. 1, pp. 69–88, 2002.

[13] A. W. Mostowski, "Regular expressions for infinite trees and a standard form of automata," in *Computation Theory*, A. Skowron, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 157–168, ISBN: 978-3-540-39748-9.

[14] D. E. Muller, "Infinite sequences and finite machines," in *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design (swct 1963)*, 1963, pp. 3–16. DOI: 10.1109/SWCT.1963.8.

[15] *Pédagothèque numérique: Les jeudis de l'IA Gen*, [Online; accessed 30. Sep. 2025], Sep. 2025. [Online]. Available: https://pedagotheque.imt.fr/mod/page/view.php?id=26571.

[16] PhilippSchlehuberCaissier, *GitHub - PhilippSchlehuberCaissier/wspot at sven/loop_ bugfix*, en. [Online]. Available: https://github.com/PhilippSchlehuberCaissier/wspot (visited on 06/16/2025).

[17] *Plan Stratégique de la Transition Écologique et Sociétale*, [Online; accessed 1. Oct. 2025], Jun. 2024. [Online]. Available: https://www.telecom-sudparis.eu/plan-strategique-transition-ecologique-et-societale.

[18] *Rapport de mission d'audit - institut mines-télécom, télécom sudparis*, [Online; accessed 29. Sep. 2025], 2025. [Online]. Available: https://www.cti-commission.fr/wp-content/uploads/2025/04/telecomsudParis_rmad_202502.pdf.

[19] F. Renkin, A. Duret-Lutz, and A. Pommellet, "Practical "Paritizing" of Emerson-Lei Automata," in *Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA'20)*, ser. Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA'20), vol. 12302, Hanoi, Vietnam: Springer, Oct. 2020, pp. 127–143. DOI: 10.1007/978-3-030-59152-6_7. [Online]. Available: https://hal.science/hal-02891970.

[20] J. Richard Büchi, "Symposium on decision problems: On a decision method in restricted second order arithmetic," in *Logic, Methodology and Philosophy of Science*, ser. Studies in Logic and the Foundations of Mathematics, E. Nagel, P. Suppes, and A. Tarski, Eds., vol. 44, Elsevier, 1966, pp. 1–11. DOI: https://doi.org/10.1016/S0049-237X(09)70564-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0049237X09705646.

[21] *Spot: A platform for LTL and ω-automata manipulation*. [Online]. Available: https://spot.lre.epita.fr/ (visited on 06/16/2025).

[22] *The Python Profilers*, [Online; accessed 29. Sep. 2025], Sep. 2025. [Online]. Available: https://docs.python.org/3/library/profile.html.

▷ Appendices

A ▷ About Télécom SudParis

Télécom SudParis (TSP, former Télécom INT) [18], is an engineering school that is also part of the Institut Polytechnique de Paris (IP Paris) alongside École Polytechnique, ENSTA Paris, École des Ponts et Chaussées, ENSAE and Télécom Paris.

This public engineering school is part of the Institut Mines-Télécom (IMT), a public institution under the supervision of the Ministère de l'Économie, des Finances et de la Souveraineté industrielle et numérique.

Télécom SudParis' activities are split between two main sites:

- Évry-Courcouronnes, the historical site which also houses the Institut Mines-Télécom Business School;
- Palaiseau (Saclay), on the campus of the Institut polytechnique de Paris, along with the general management of the Institut Mines-Télécom and Télécom Paris, and nearby the other school members of IP Paris.

Télécom SudParis is also equipped with a research lab specialized in Information and Communication Technologies (ICTs).

B ▷ Sustainable and Socially Responsible Development (DD&RS)

B.1 ▷ Green Transition and Social Transition Master Plan

According to Télécom SudParis' website [17], the school aims to excel in digital engineering while integrating sustainable development and social responsibility principles in the institution's identity. To attain this level of excellence, five main strategic objectives are presented:

- integrating an environmental and social dimension to curriculums;
- actively promoting efforts in research and innovation oriented towards digital sustainability;
- promoting social and gender diversity by setting up inclusion and support programs;
- building partnerships with actors committed to sustainability;
- reducing the campus' carbon footprint through transportation, energy consumption and digital use policies.

B.2 ▷ A heavy reliance on AI technologies

Télécom SudParis and the Institut Mines-Télécom organize a monthly online seminar about the use of Generative AI (in the following, GenAI) for education purposes, called "Les jeudis de l'IAGen" (*GenAI Thursdays*) [15]. These talks cover a variety of topics such as the use of ChatGPT to generate course outlines or discussions on how students tend to use GenAI in their studies. They also present a curated list of GenAI tools for various usages: converting a textual prompt to an image or a video, summarizing the contents of a resource, or generating presentation slides.

However, this seminar fails to tackle GenAI's environmental and social impacts. According to [10], the high environmental impact of GenAI technologies mainly comes from two contributors:

- the hardware needed to power GenAI models: GPU manufacturing, which needs high electricity, water and non-recyclable rare metals inputs (as an example, as much as 25 % of the electrical consumption of Taichung City (3 million inhabitants) in Taiwan); as well as data center building and upkeep. The social repercussions of GenAI are also presented, both at the data center level (terrain and ressource usage that do not cause any benefit for local communities, a cited example is xAI's latest data center in Memphis, Tennessee).
- the training and development of GenAI models: quantitatively, it is estimated that the training of OpenAI's GPT-3 model consumes nearly 1300 MWh of electricity, the equivalent of 300 French households[1], and emits about 550 tons of CO_2, the equivalent of 312 round trips from Paris to New York[2]. The paper also notes that there exist inequalities in GenAI accessibility depending on the users' income, and that these models tend to favor English-speaking users at the expense of other languages.

Capraro et al., in [6], address the question of the social impacts of GenAI in four domains (information, work, education, and healthcare): they note that while these tools can provide a better access to information or be used to assist humans in critical contexts such as in medical image analysis, they tend to be used as a replacement for human workforce instead of a complementary tool to assist humans.

The authors are concerned that these tools could form the basis for a "surveillance capitalist" system, by exploiting GenAI's widespread usage to collect information about its users (possibly in violation of privacy regulations such as the GDPR in the European Union, the Loi 25 in Québec, or the California Consumer Privacy Act (CCPA) in the United States), but also that they could worsen the digital divide that already appeared at the time the personal computer was introduced.

[1] ENGIE website
[2] Impact CO_2, ADEME