Thesis Dissertation

ENCODING MULTI-TOKEN REVERSING PETRI NETS IN ANSWER SET PROGRAMMING FOR SIMULATION-BASED REASONING

Michail Angelos Pittakaras

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2021

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

ENCODING MULTI-TOKEN REVERSING PETRI NETS IN ANSWER SET PROGRAMMING FOR SIMULATION-BASED REASONING

Michail Angelos Pittakaras

Supervisor
Dr. Anna Philippou

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus
May 2021

Acknowledgments

I wish to express my deepest gratitude to my supervisor, Dr. Anna Philippou for her constant support and guidance that she provided during my study. She encouraged me to be more professional and showed her belief in me, and without her persistent help, this project would not have been completed.

I am extremely grateful to the PhD students, Eleftheria Kouppari and Kyriaki Psara, for always being available and willing to help me and transfer useful knowledge.

I would like to express my gratitude to Dr. Yannis Dimopoulos, for his assistance in better understanding the ASP language and optimizing my code.

I cannot dismiss to thank my family as well as my friends, for their support for the duration of this diploma thesis.

Abstract

A Petri Net is a mathematical modeling tool for the description and analysis of concurrent processes which arise in distributed systems. Multi-token Reversing Petri nets (MRPNs) are an extension to normal Petri nets that can be a very useful and powerful tool for describing applications which are reversible by nature.

Answer Set Programming language (ASP) is a declarative programming style aimed at solving difficult, NP-hard search problems and it is especially helpful in knowledge-intensive applications. Rather than using programs to decide how a problem should be solved as in conventional programming, the idea now is to create a supervised program that is general and can solve any instance of the problem. The solution in ASP is built using stable models and we can get multiple or all the stable models without adding too much overhead to the execution time. That works like parallel processing, giving us the option to choose the optimal answer.

In this thesis we describe MRPNs in ASP. MRPNs can be used as the base for describing other difficult problems which will then be solved automatically using ASP. The encoding of Multi-token Petri Nets in ASP along with some improvements for performance and memory usage are shown in detail in this thesis. We show the usefulness of our approach through a case study involving the multiple-input multiple-output (MIMO) problem of the Antenna Selection in wireless communication services. is described in Multi-token Petri nets and then solved using ASP.

Contents

Chapter **1** **Introduction**……………………………………………………………**1**

1.1 Motivation 1

1.2 Work Purpose 1

1.3 Work Methodology 2

1.4 Thesis Structure 3

Chapter **2** **Related Work**……………………………………………..…...………**4**

2.1 Reversible Computation 4

2.2 Reversible Modeling 5

2.3 Petri Nets 6

2.4 Reversible Petri Nets 9

2.5 Multi-token Petri Nets 12

2.5.1 Forward Execution 14

2.5.2 Reverse Execution 15

2.6 Answer Set Programming Language (ASP) 16

2.6.1 ASP syntax and semantics 17

2.7 Clingo 18

2.8 Python 19

2.8.1 Clingo API documentation 19

2.9 MIMO – Antenna Selection 19

Chapter **3** **Encoding Collective Petri Nets into ASP**………………………**23**

3.1 Forward Execution 23

3.2 Reversible Execution 43

3.3 Conditions 53

3.4 Extension to support Antenna Selection 63

3.5 Optimizations 74

Chapter **4** **Case Study**…………………….…………………….………….**88**

4.1 Antenna Selection Case Study 88

4.2 Reachability 94

4.3 Greedy Algorithm Comparison 96

Chapter **5** **Conclusions**…………………………...……………………………..**99**

5.1 Summary 99

5.2 Challenges 100

5.3 Future Work 101

References ………………………………………………………………… **103**

Chapter 1

Introduction

1.1 Motivation 1

1.2 Work Purpose 1

1.3 Work Methodology 2

1.4 Thesis Structure 3

1.1 Motivation

Multi-token Reversible Petri nets have been proposed in [4] and they are formed for describing applications which are reversible by nature. Encoding them in a language such as Answer Set Programming [23], which is oriented towards difficult search problems, can help us describe and solve difficult problems such as the MIMO problem of Antenna Selection proposed in [3].

1.2 Work Purpose

In this thesis, we consider a particular model of computation, known as Collective Petri nets and its implementation in Answer Set Programming language (ASP) using clingo as the grounder and solver. Collective Petri nets, a type of multi-token Petri nets (MRPNs), are an extended form of Petri nets that give more possibilities and can be applied in many problems and simulations. The purpose of analyzing this type of Petri nets is to show the properties of their behavior and use them as a base to describe difficult problems like the massive-MIMO antenna selection problem and solve it. MRPNs will be further extended with the addition of the *conditions* with the intent to make them more flexible and give more possibilities for describing applications. With some additional ASP and Python extensions, an algorithm for solving the MIMO problem of antenna selection is encoded and simulated on the Multi-token Collective Reversing Petri nets.

1.3 Work Methodology

The study of Petri nets and Reversing Petri nets, as well as their semantics and definitions, was the first step in putting this thesis into action. It was necessary to understand the Reversible computation separately. The next step was to learn how to use and code in Answer Set Programming and understand its abilities and potential use. Further research on the implementation of normal Petri nets and reversing Petri nets in Answer Set Programming was vital to explore Answer Set Programming and how Answer Set Programming works in practice.

After becoming acquainted with ASP, I began researching an extended form of Petri nets, MRPNs under the collective interpretation. I started encoding their forward computation on ASP and my methodology is described in this thesis. Then, to be able to encode and expand my ASP program with reversible execution, I continued my research on reverse semantics.

The validity and the correctness of the code I propose for the collective reversible Petri nets was then thoroughly checked using examples and test runs. To allow conditions to control both forward and reverse executions of the transitions in the Petri nets, the forward and reverse semantics for controlled execution were studied and subsequently added to the program. This was a vital step in order to solve the MIMO problem of the Antenna Selection, as conditions are required for the simulation, to control which transitions can fire according to the model's rules.

To extend and apply the research, the problem of Antenna Selection was then thoroughly investigated. I modified the algorithm proposed in [2][3] and added ASP and Python extensions on the Controlled Multi-Token Petri nets to solve the problem. Python is required for calculations and table multiplications, and it is dynamically called from ASP when required.

Finally, the ASP solving process was further studied and optimizations both on Python and ASP were made to reduce the complexity and make the model work more efficiently. Many attempts were made to reduce the time and the computation complexity of the model and of the solving process.

1.4 Thesis Structure

Chapter 2 starts with the description of Reversible Computation and Modeling. The following are the basics of Petri nets and Reversible Petri nets. Collective Petri nets are then explained, and their forward and reverse execution is shown. A basic explanation of Answer Set Programming is also provided in Chapter 2, along with its syntax and semantics. Clingo, the solver used in this thesis for ASP, is also included as well as the Python language, which is used for the problem of the Antenna Selection. Finally, the antenna selection problem is discussed in this chapter, together with an algorithm and its model on MRPNs.

In Chapter 3, emphasis is given on the encoding of a form of Multi-token Petri nets under the collective interpretation, in ASP language. The whole process of encoding is thoroughly explained in Chapter 3 with examples of execution. Subsequently, additions and slight modifications are made to the program, to support reversible execution and conditions. Then, the extensions needed for simulating and solving the MIMO problem of the Antenna Selection in a neighborhood are also added. Further optimizations are discussed for the program to achieve a better performance.

Chapter 4 contains a case study on the problem of the Antenna Selection where the program we encoded is used to solve an Antenna Selection example. The property of reachability is then explained and using it we compare a greedy approach solution to the non-greedy one proposed in [2].

Chapter 5 of this thesis summarizes the work done on this diploma thesis, noting the difficulties encountered during implementation, as well as the work's limitations. The chapter concludes with a brief mention of possible work that could be done based on this study.

Chapter 2

Related Work

2.1 Reversible Computation 4
2.2 Reversible Modeling 5
2.3 Petri Nets 6
2.4 Reversible Petri Nets 9
2.5 Collective Multi-token Petri Nets 12
2.5.1 Forward Execution 14
2.5.2 Reverse Execution 15
2.6 Answer Set Programming Language (ASP) 16
2.6.1 ASP syntax and semantics 17
2.7 Clingo 18
2.8 Python 19
2.8.1 Clingo API documentation 19
2.9 MIMO – Antenna Selection 19

2.1 Reversible Computation

Reversible computation [12] is an expanded type of standard-forward computation, that allows us to reverse the execution of an operation at any time. This makes it possible for the system to return to a previous state and possibly be used as a way to recover from errors.

Possibly the biggest reason for the study of reversible computation technologies is that they give the only potential way to increase computer computational energy efficiency. Due to the laws of physics, at some point, smaller transistors will no longer produce the improvements they used to yield, and at this stage the development of conventional semiconductor technology is going to stop as it is not going to be cost effective.

The concept of reversible computing is at the core of thermodynamics and information theory, and it is the only way, according to the laws of physics, to keep improving the cost and energy efficiency of general-purpose computing into the far future.

The history of reversible computing started with Landauer's finding, that only irreversible computing produces heat [16]. This finding led to the impetus for reversible computing, and a strong line of research into the development of reversible logical gates and circuits. Reversible computing never received much attention in the past because it is hard to implement. Now that the end is in sight, it is time for the finest physicists and engineers to begin a comprehensive effort.

Petri nets are a visual mathematical language that can be used to specify and test discrete event systems. They are linked to a rich mathematical theory and have widely been used to model and reason for a wide variety of applications.

The first analysis of reversible Petri nets was suggested in [9][10]. The authors of these works examined the effects after inserting reversed transitions in a Petri net. These transitions are obtained by reversing the direction of the arcs from a subset of selected Petri net transitions. In the resulting Petri nets, they then explore problems of decidability regarding reachability and coverability. This approach however, does not model reversible behavior since it allows to reverse non-executed transitions.

2.2 Reversible Modelling

Reversible computation has posed some unanswered questions. The key methods, effects, potential benefits, and applications of reversible computation must all be described. To answer these questions, it is helpful to apply various notions of reversibility to appropriate modeling languages and formulate theoretical foundations for what reversibility is, what it aims for, and what benefits it offers to systems.

Exploring reversibility across formal languages, would aid in understanding its definitions and semantics. It is easier to describe a unified theory for reversibility by applying reversibility to particular case studies and using various notions and strategies. This will also aid in comprehending how reversibility can improve specification, verification, and testing.

Once developed, reversible formalisms can also be useful to experts outside of the field of computer science. Since there are natural and artificial systems that embed or may use reversibility, it can be a useful setting for researching and analyzing systems. Such reversible formalism will also find application in biochemistry, mathematics, and material sciences.

2.3 Petri Nets

Petri Nets were developed by Carl Adam Petri [5] in 1962, as the subject of his dissertation. A Petri Net is a mathematical modeling tool for the description and analysis of concurrent processes which arise in distributed systems. Since 1962, they have been developed and extended, to support applications in many areas like protocols and networks, performance evaluation, hardware systems, telecommunications, e-commerce and many more.

A Petri Net consists of two types of elements, the places, drawn as circles and the transitions, drawn as rectangles. Places can contain a finite number of tokens drawn as black dots inside places. Tokens can represent the elements of a system that are subject to dynamic change. The number of tokens stored in different places in a specific state is called a marking. The starting token distribution in a Petri Net is called the initial marking. Transitions, however, have no capacity and they do not store tokens.

Directed Arcs, connect places to transitions or vice versa, forming a bipartite graph. Arcs connecting a place with a transition, form the input places. Arcs connect a transition to a place from which the output places are formed. These arcs are weighted and denote the number of tokens that will be moved through the arc. The weights are represented with a label above the arc.

A transition T when enabled may fire, moving tokens from input places to output places as declared on the arcs. For a transition T to be enabled, all its input places must have equal or more tokens than the weight of their corresponding arc that connects them with T.

Tokens will move from a place Pi to a place Pj, if there is an arc from Pi to a transition T, and an arc from the transition T to Pj. The transition T has to be enabled and it also has to fire. An enabled transition may or may not fire.

A Petri net is formally defined as a 4-tuple N = (P, T, F, W) and an initial marking M0, where:

1. P is a finite set of places.
2. T is a finite set of transitions, $P \cup T \neq \emptyset$, and $P \cap T = \emptyset$
3. F: is the set of directed arcs from places to transitions, and from transitions to places, $F \subseteq (P \times T) \cup (T \times P)$.
4. W: $F \rightarrow \mathbb{N}+$ is a function that assigns weights to the directed arcs.

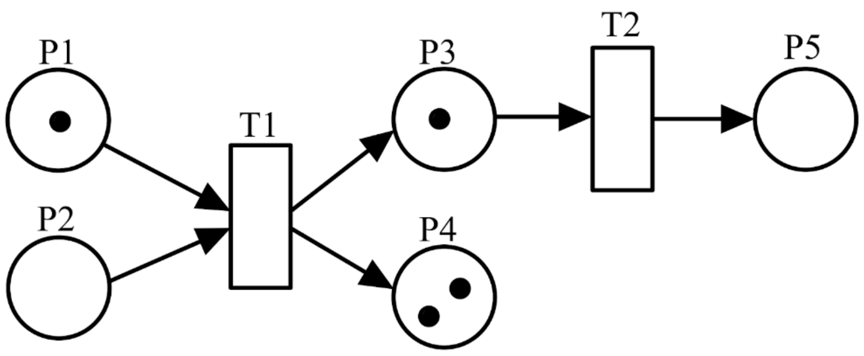


Figure 2.3.1: Petri Net

The Petri net in Figure 2.3.1 contains five places, P1, P2, P3, P4 and P5, and two transitions, T1 and T2. The arcs have no labels, so the weight by default is set to 1. In the initial marking M0, places P1 and P3 have 1 token each, and place P4 has two tokens. P1 and P2 are the input places for transition T1 and P3 and P4 its output places. For T2, the input place is P3 and the output place is P5.

A transition in a Petri Net is enabled if all its input places hold at least the number of tokens specified in the arc's label. Therefore, in figure 2.1, transition T2 is enabled as P3 holds a token and the arc connecting P3 and T2 has a weight of one. On the other hand, transition T1 is not enabled as place P2 has zero tokens.

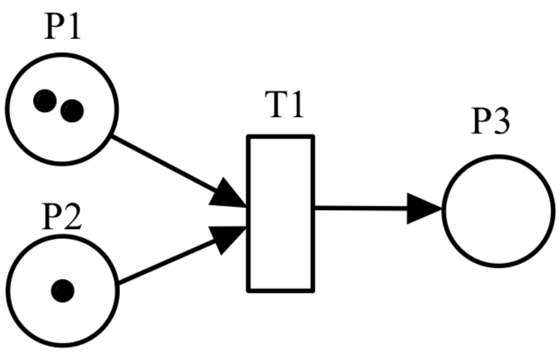


Figure 2.3.2: Petri Net before transition fire

The Petri Net in Figure 2.3.2 has three places, P1, P2 and P3 and one transition T1. T1 is enabled since all the input places of T1 have the required number of tokens. After transition T1 fires, tokens will move from input places P1 and P2 to the output place P3. The result of firing transition T1 is shown in figure 2.3.3. Input places and output places will lose and tokens respectively, equal to the weight of their arc and the fired transition.

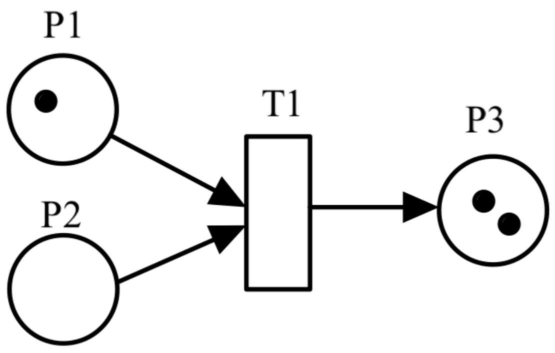


Figure 2.3.3: Petri Net after transition fire

P1 and P2 each lost one token in this example, while P3 gained one token because the arcs are all of weight one.

2.4 Reversible Petri Nets

On Petri nets, reversible computation aims to undo the consequences of a previously performed transition. This might be necessary for applications that naturally embed reversibility or to be used to go back to a previous state from an undesired one and recover from faults. The philosophy of Reversible Petri Nets (RPNs) will be explained in this subchapter.

Consider a catalyst c, which aids in the bonding of the otherwise inactive molecules a and b. Catalysis is achieved by element c bonding with a, which then allows a and b to bond. The catalyst is then no longer needed, and its bond with the other two molecules is broken.

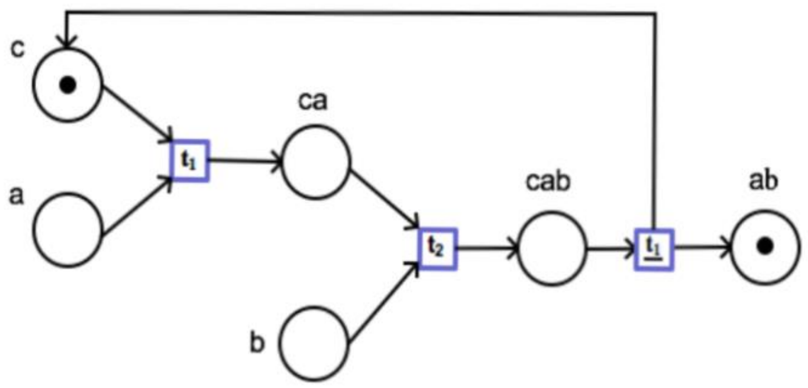


Figure 2.4.1: Biochemistry catalysis in normal Petri Nets

Figure 2.4.1 depicts a Petri net model of this operation. The Petri net performs transition t1 to create the bond c-a, then transition t2 to create c-a-b. Finally, action $\underline{t1}$ “reverses” the a-c bond, yielding a-b and releasing c. This example shows that Petri nets are not reversible by nature, in the way that no transition can be carried out in both directions. As a result, a supplementary forward transition (e.g., transition $\underline{t1}$ for undoing the effect of transition t1, bond c-a) is required to achieve the undoing of a previous action.

In systems that express several reversible patterns of execution, this explicit approach to modeling reversibility can be cumbersome, resulting in larger and more complex systems. In Figure 2.4.2, is a method for modeling reversible computation which does not involve the development of new, reversal transformations and instead allows transitions to be executed both forward and backward using reversible Petri nets. Note that this is the approach taken in RPNs and there could be other approaches as well.

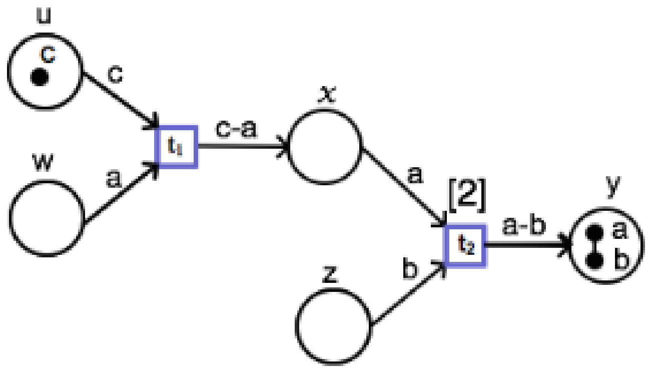


Figure 2.4.2: Biochemistry catalysis in reversible Petri Nets

This Petri net is only valid when tokens are individual, although this restriction will be lifted subsequently in this thesis. That means that for now, only one token of a certain type can be present in a reversing Petri net. While this restriction will be removed later in this thesis, this Petri net is only applicable when tokens are individual. That means that in a reversing Petri net, only one token of each kind can exist. Distinguishing the causal course of each transformation is another area where attention should be paid. Two new concepts, bases and bonds, have been implemented to accomplish this. A base is a non-consumable token that lasts forever. Via numerous transformations, a base safeguards the token's uniqueness. A bond is a token-to-token relation formed by transitions. A transition's reversal will break the bond, and thus the transition's effect.

Another decision that had to be made was whether to use a transition that only took a subset of the bonds and bases available as input. This ensures that only the ones required are chosen, while the others are available for any other transformation that may arise. As a result, negative bonds/bases are introduced. When a negative token ā is used to mark an arc, it means that token a should not be present in the incoming places of t for the transition to fire, and vice versa for bond $\overline{b}$.

Reversing Petri nets are formally defined [4] by the 5-tuple (A, P, B, T, F) where:

1. *A* is a finite set of bases or tokens ranged over by a, b, ... $\bar{A} = \{ \bar{a} \mid a \in A\}$where $\bar{A}$ contains negative instance for each token, and we write $\mathscr{A} = A \cup \bar{A}$.
2. *P* is a finite set of places.
3. *B* $\subseteq A \times A$ is a set of bonds ranged over by β, γ, ... A bond $(a, b) \in B$ is denoted by the notation a-b. $\overline{B} = \{ \overline{b} \mid b \in B\}$ contains negative instances of the bonds. We write $\mathscr{B} = B \cup \overline{B}$.
4. *T* is the set of the finite transitions.
5. *F:* $(P \times T \cup T \times P) \rightarrow 2^{\mathscr{A} \cup \mathscr{B}}$ is a set of directed arcs.

The places and transitions remain the same as the traditional Petri nets, as well as the arcs which remain labelled and directed. Their label now is a subset of $\mathscr{A} \cup \mathscr{B}$. Tokens are now being distinguished from each other as they have a unique name. Bases can exist as stand-alone elements or be combined to create bonds. The marking is the same as in traditional Petri nets with the addition of bonds. A bond between 2 tokens is graphically marked with a black line connecting the two black dots that represent the bonded tokens.

To be well formed, the following should hold true for the transitions of a Reversing Petri net:

1. $A \cap pre(t) = A \cap post(t)$.
2. If $a\text{-}b \in pre(t)$ then $a\text{-}b \in post(t)$.
3. $F(t, x) \cap F(t, y) = \emptyset$ for all $x, y \in P$ $x \neq y$.

According to the above, no transition should consume tokens (1), no bonds are destroyed during the execution of a transition (2) and no tokens or bonds are cloned into more than one outgoing place (3).

The history of the transition firings must be stored also to decide which transitions can be reversed. Three types of reversibility for reversing Petri nets, backtracking, causal reversibility, and out-of-causal-order reversibility are discussed in [4].

2.5 Multi-token Petri Nets

We define a new type of reversibility in this section based on the collective token interpretation philosophy [4], in which tokens of the same type are not differentiated. The limitation of standard reversible Petri nets is now lifted, and multiple tokens of the same type can exist in a marking, giving transitions the chance of multiple firings. Collective Reversible Petri nets is a type of Multi-token Reversible Petri net (MRPN). This philosophy is maintained by a variety of application domains, such as resource aware systems or biological systems, and is implemented as a firing rule for multi-token reversing Petri nets. This approach emphasizes the local nature of reversing Petri nets and introduces a new perspective on reversing systems that focuses on token location rather than transition relations.

Consider the example of Figure 2.5.1 that was originally presented in [13]. This example depicts two students attempting to purchase a present for their teacher. They both have a coin, denoted by C1 and C2. Their coins can be found in places S1 and S2. Transitions t1 and t2 represent the coin contributions, while transition t3 represents the act of purchasing the present which costs one coin.

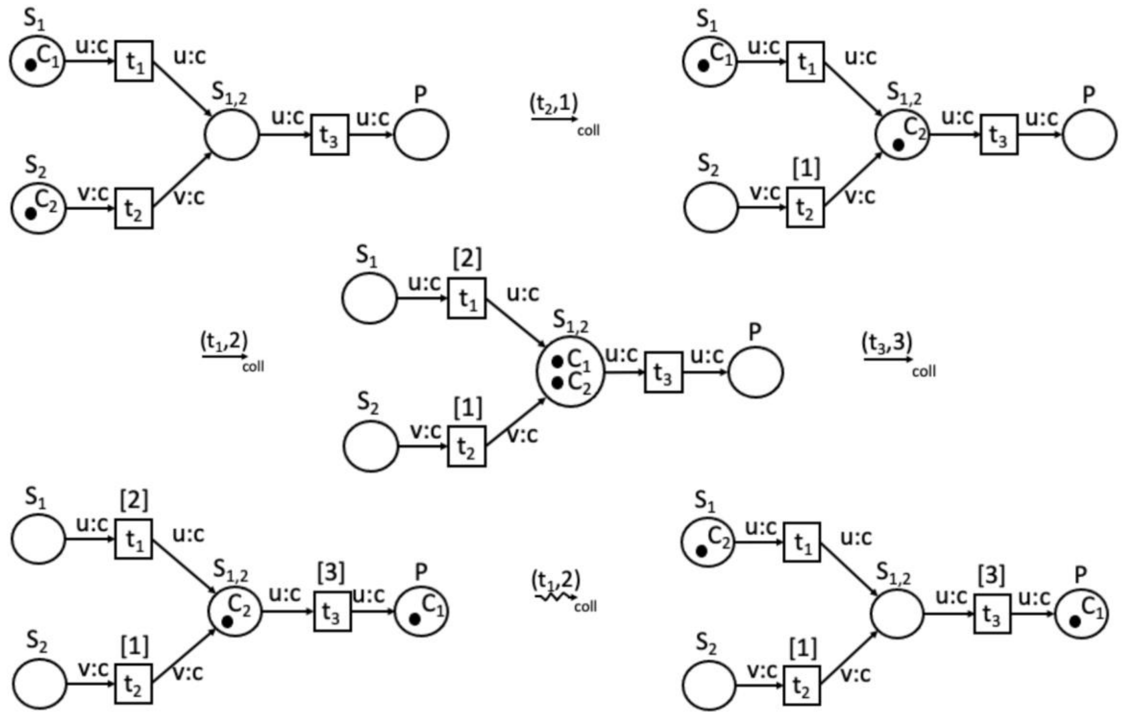


Figure 2.5.1: Students buying a present for their teacher.

After the contributions have been made and the gift was purchased, there remains a coin in S1.2, which must be returned. According to the collective token interpretation, this coin can be randomly returned to any of the students because the purchase was made possible by both of their contributions. This relation in which a transition may be causally dependent on one either of the transitions is called disjunctive causality.

A Collective Petri net is formally defined as a 6-tuple N = (P, T, A, A_V, B,F) and an initial marking M0.

1. P is a finite set of places.
2. T is a finite set of transitions, $P \cup T \neq \emptyset$, and $P \cap T = \emptyset$
3. A is a finite set of token types.

4. A_V is a finite set of variables.
5. $B \subseteq A \times A$ is a finite set of undirected bond types. We use the notation a-b for a bond $(a,b) \in B$.
6. F: $(P\times T \cup T\times P)$ is the set of directed labelled arcs from places to transitions, and from transitions to places.

In Collective Reversible Petri nets, we are not interested in identifying the exact tokens that participate in specific transitions. We assume that any token of the same type has identical capabilities on firing transitions and can only take part in transition involving variables of the same type. So, we let tokens that were not used in the forward execution of a transition, to reverse the transition if they are of the same type as the ones used in the forward execution.

2.5.1 Forward Execution

A transition that is enabled for forward execution, is characterized by the non-deterministic selection of tokens if they obey the transition's incoming arcs' variable types. We also want to allow tokens to reverse transitions in which they have not participated. Based on a selected forward enabling assignment, namely Uf, we can identify the tokens that are removed from places as defined by the •Uf and moved to places defined by Uf•. As a result, a transition firing in the forward direction, will pass a collection of token and bond instances, and potentially form or destroy bonds as required. These instances that will move, are specified by the transition's incoming arcs, and will be passed to the transition's outgoing places, as specified by the transition's outgoing arcs. Token and bond instances are transferred without being supplemented with memories of their previous transformations. For the transitions, their history of execution is updated accordingly, incrementing the number of times that they have been executed.

To be more specific, a transition occurrence is enabled:

(1) if token instances in the transition's incoming places are assigned to variables on the incoming edges in a type-respecting manner, with token instances originating from the appropriate input places and tokens connected with bonds as required by the transition's incoming edges.

(2) If the selected token instances are bonded together in the transition's incoming position, the bond should also exist on the variables labeling the incoming arcs (thus transitions do not recreate bonds).

(3) When removing the selected incoming tokens and adding the selected outgoing tokens, if two token instances are moved by a transition to separate outgoing places, these tokens do not remain bonded(we do not clone tokens).

2.5.2 Reverse Execution

Now we will focus on reversing transitions. If a transition has been previously executed and there are token instances in its output places that fit the requirements on its outgoing arcs, it can be reversed in that state. We neglect the causal paths allocated to the tokens during forward execution. As a result, tokens can reverse any transition given that the variable types are respected, regardless of whether the tokens were directly used to fire this particular transition event.

To be more specific, a transition occurrence is enabled to reverse:

(1) If there is a type-respecting assignment of token instances from the out-places of the transition to the variables on the transition's outgoing edges, and the instances are connected with bonds as required by the outgoing edges.

(2) If the selected token instances are bonded together in the transition's outgoing position, the bond should also exist on the variables labeling the outgoing arcs (thus we do not recreate existing bonds).

(3) If two tokens are passed to different incoming places by a transition, these tokens should not be linked when the selected outgoing tokens are removed and the selected incoming tokens are added (we do not clone tokens).

2.6 Answer Set Programming

Answer Set Programming (ASP) is a declarative programming style aimed at solving difficult, NP-hard search problems and it is especially helpful in knowledge-intensive applications. It is based on the stable model [7] semantics of logic programming which applies ideas from auto epistemic logic [11] and default logic [15] to the study of negation as failure. Years of research into information representation, logic programming, and constraint satisfaction have resulted in it. The planning method was proposed in 1997 and was an early example of answer set programming, based on the relationship between plans and stable models.

Unlike traditional programming, where the idea is to use programs to specify how a problem should be solved, the idea now is to view a program as a formal representation of the problem as such. In other words, instead of trying to solve each individual problem separately, we program the machine to be able to solve any problem of that kind. We encode the solving algorithm, and then we provide our encoded problem to it to solve it for us. The first step is to express the problem using first logic programming syntax. The grounder then produces a finite propositional representation of the input program. A ground program does not contain any variables but has the same answer sets as the original program. After producing the ground program, a solver then computes the stable models of the propositional program. The solution is then built using the stable models, computed in a bottom-up approach. The solving process is shown in Figure 2.6.1.

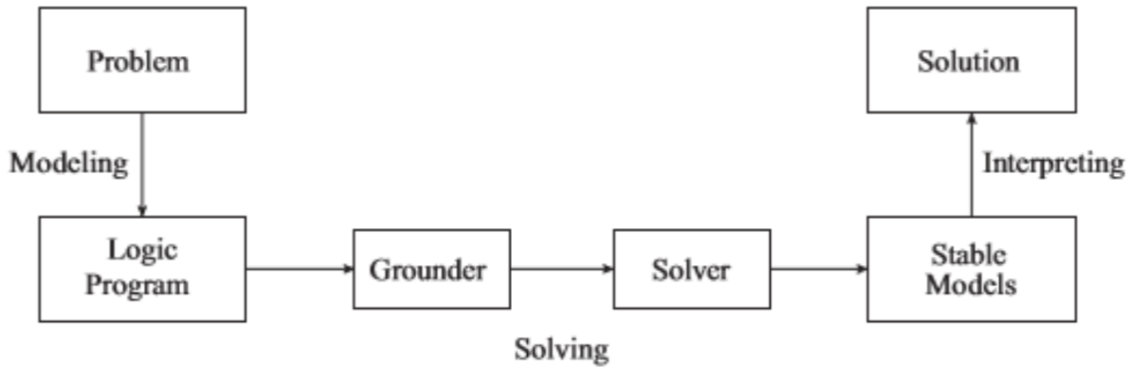


Figure 2.6.1: ASP solving process.

2.6.1 ASP syntax and Semantics

The building blocks of ASP are atoms, literals, and rules. Atoms are simple statements that may be true or false. Literals are atoms, or atoms preceded by not, declaring negation. An ASP rule is as follows:

$$h \text{ :- } p_1, \ldots, p_m, \textit{not } r_1, \ldots, \textit{not } r_k.$$

Where h, p_i's and r_i's are all atoms. In every rule, the part on the left of ':-' is called the head and the right part is called the body. In the above rule, h is the head and the p_i's and r_i's form the body of the rule.

A rule is an argument to derive that the head of a rule is correct if all literals in the body are true. A negated literal not r_j is true if the atom r_j does not have a derivation. For instance, from the rule

charging :- plugged_in, not broken.

we can conclude that something is charging if we establish that it is plugged in and nothing is broken.

Rules can have no body.

plugged_in :- .

Such rules are called **facts**. Facts are unconditionally true and usually " :- " is omitted.

Rules can also have no head.

:- broken.

Such rules are called **constraints** and they affect the collection of answer sets by eliminating all the answer sets where the body is true.

The atoms and facts can have several parameters. An atom with name weather with X parameters, $X \in Z^+$, is symbolized as *weather/X*. Next is a simple example of two ASP rules using the atoms *parent/2, father/1, mother/*1, *male/1*, *female/1*:

mother(A):- parent(A,_), female(A).

father(A):- parent(A,_), male(A).

Notice that underscores represent anonymous variables that can be used if their assignment is unimportant, i.e., if they are not used in another rule predicate or if they are irrelevant. For example, someone is a mother if is a parent to anyone and is a female.

Rules alone without facts will produce no more facts. Using the following facts,

female(alice). parent(alice,bob). male(bob).

our program will produce the fact *mother(alice)* from the first rule.

Semicolons in the head of a rule, will result in the creation of multiple facts:

human(A ; B) :- parent(A,B).
parent(alice,bob).

will produce the facts *human(alice)* and *human(bob).*

2.7 Clingo

Clingo is an ASP system to ground and solve logic programs of the Potsdam Answer Set Solving Collection (Potassco) [23]. Grounding is the procedure of computing the equivalent variable-free program. A grounder example is gringo [20], which is also used as a grounder for Clingo. The output of gringo can be further processed with clasp, claspfolio or clingcon.

For Clingo, the solver used is clasp. Clasp is a solver for (extended) normal and disjunctive logic programs. It integrates the high-level simulation capabilities of ASP with cutting-edge Boolean constraint solving techniques. The main clasp algorithm, which is based on conflict-driven nogood learning, is proven to be a very effective technique for satisfiability testing (SAT).

2.8 Python

Python is an object-oriented, high-level programming language with dynamic semantics that is interpreted. Python's plain, easy-to-learn syntax enhances readability, reducing software maintenance costs.

Its high-level built-in data structures, along with dynamic typing and dynamic binding, make it ideal for Rapid Application Development but also as a scripting or glue language for connecting existing components.

2.8.1 Clingo API documentation

The clingo module [21] can be used to provide functions and classes controllable grounding and solving. Clingo will be able to execute Python code embedded in logic programs if the clingo framework is built with Python support.

Functions which are defined inside a Python script block can be called during the instantiation process using @-syntax.

2.9 MIMO – Antenna Selection

Antenna selection in distributed Massive MIMO (Multiple Input Multiple Output) [18] antenna arrays is an optimization problem on a complex system with components that behave similarly to each other. It is possible to keep the benefits of a massive antenna array, such as interference reduction, spatial multiplexing, and diversity [1], while also reducing the amount of radio frequency (RF) chains and antennas to power at once. Utilizing all available antennas is not optimal, as certain antennas do not contribute to the operation [8].

Our goal is to find the optimal subset of antennas to power on, for us to have the optimal results. This problem however for large antenna arrays is computationally demanding [19]. As a result, for real-time use, suboptimal methods are sought.

A fast, environment-aware, asynchronous, distributed antenna selection algorithm that maximizes sum-capacity within a distributed array's RPN scheme is presented in [2].

The optimization problem we face here, is the maximization of the sum capacity in a distributed massive MIMO base station with N_T antennas. Our task is to select a subset of N_{TS} antennas in a cell with N_R users and attempt to maximize the sum capacity C.

$$C = \max_{\mathbf{P},\mathbf{H_c}} \log_2 \det \left(\mathbf{I} + \rho \frac{N_R}{N_{TS}} \mathbf{H_c} \mathbf{P} \mathbf{H_c}^H \right)$$

In the equation of C, ρ is the signal to noise ratio (SNR), matrix I is a $N_{TS}\, x\, N_{TS}$ identity matrix, P is a diagonal power distribution $N_R\, x\, N_R$ matrix initiated with $1/N_R$. H_c is the $N_{TS}\, x\, N_R$ channel submatrix for the subset of selected antennas from the $N_T\, x\, N_R$ channel matrix H, which is known in its entirety.

The RPN framework is independent of the array structure (or, in a broader sense, network topology) because it is not dependent on the number of antennas or how they are connected to one another. This hints at its generalizability as a resource allocation framework in wireless communications.

To model the antenna selection problem on Petri nets, we represent the antennas with places, and power tokens are held by the switched-on antennas. Transitions, depicted as bars between adjacent antenna places, allow the tokens to flow and move from one antenna to another. Transitions operate as follows:

1) If a token exists in exactly one of the two places it binds, the transformation is possible, otherwise it is not.
2) A transition can fire and move the power token from one antenna A to another antenna B, only if that leads to an increase in sum-capacity. In other words, if the sum-capacity with antenna B powered on and antenna A powered off is greater than the current sum-capacity, the transition is enabled.

3) In the case of multiple possible transitions from a place, the one with the maximum sum-capacity difference is preferred. Prioritisation is implemented in the transition condition function.
4) Transition execution has no set order, and transitions are carried out until a stable state is reached.

The state of the antennas is altered until the RPN converges, and the antennas with tokens are switched on for the period of the coherence interval. The algorithm resumes operation after the next update of the channel state information.

In the Multi-token Petri net interpretation, antennas are represented by places $A_1,...,A_n$, where n=NT, and the overlapping neighborhoods by places M1,...,Mh. When there is a connection link between antennas Ai and Aj, these places are connected via transitions ti,j, which connect Ai, Aj, and Mk. The transition captures the fact that antenna Ai may be preferred over Aj in place Mk based on local knowledge, or vice versa.

Three types of tokens are used to implement the intended mechanism. The power tokens $p_1,...,p_k$ are the first, with k denoting the number of enabled antennas. Antenna Ai is considered powered on if a token of type p is located on place Ai. The transfer of these tokens results in new antenna selections, which should converge to a locally optimal solution in the best-case scenario. Second, each neighborhood is represented by tokens m1,...,mh. The third type of tokens are the antenna tokens $a_1,...,a_n$ which represent the antennas. Here, is how the tokens are used:

Transition ti,j between antenna places Ai and Aj in the neighborhood Mk is enabled if token p is available on Ai, token aj on Aj, and bond (ai,mk) on Mk. The antennas Ai and Aj are on and off, respectively, in this configuration. The transition then causes the bond (ai,mk) to be broken, the token ai to be released to place Ai, the power token to be transferred to Aj, and the bond (aj, mk) to be created on Mk.

Finally, an antenna token ai is associated with data vector I(Ai)=**h_i**, $\mathbf{h_i} \in \mathbb{C}$, i.e. the corresponding row of H, to capture the transition's condition. The condition creates the matrix **Hc** by collecting the data vectors hi associated with the powered-on antennas. $\mathbf{Hc}=(\mathbf{h_1},...,\mathbf{h_n})^{T}$ where $\mathbf{h_i}$=I(A_i) if antenna Ai is powered on, otherwise $\mathbf{h_i}$=(0...0). If the sum capacity calculated for all currently active antennas (including ai), Cai, is less than the sum capacity calculated for the same neighbourhood with the antenna Ai replaced by Aj, Caj, i.e., Cai < Caj, the transition ti,j will occur.

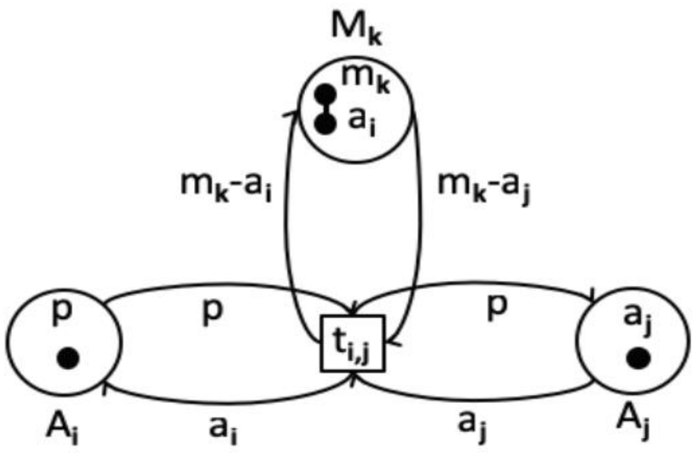


Figure 2.9.1: Part of MRPN model of Antenna Selection

Figure 2.9.1 shows the implementation of the transition for moving the power to antenna Ai from Aj through transition ti,j. Another transition exists, not shown above, that transfers the power from Aj to Ai, namely tj,i.

Chapter 3

Encoding Collective Petri Nets into ASP

3.1 Forward Execution 23

3.2 Reversible Execution 43

3.3 Conditions 53

3.4 Extension to support Antenna Selection 63

3.5 Optimizations 74

The aim of this Chapter is to describe the methodology followed, to create the rules and encode the Multi-token Petri nets into the ASP programming language. Each rule and fact are explained, and all together the program simulates the way a multi-token Petri net under the collective implementation behaves. The Extensions to support Antenna Selection are also discussed and then the code is being optimized for better performance.

3.1 Forward Execution

The time, places and transitions are represented with the following facts:

f1: time(ts), $0 \leq ts \leq k$.

f2: place(pi), $pi \in P$ is a place.

f3: trans(tj), $tj \in T$ is a transition.

The simulation of a Petri net is being split into different time instances from 0 to k, and at each time instance exactly one transition fires given that at least one is enabled.

We also need to represent the tokens of the Petri net.

f4: token(q), q ∈ A.

There can be multiple tokens of the same type in Collective Petri nets, as well as many different types of tokens. So, we need to clarify the type of each token with the following fact:

f5: type(ty,q), where ty ∈ Av is the type of token q.

Tokens are held by places.

mk1: holds(pi,q,ts), where pi ∈ P, q ∈ A, ts is a timestamp.

A place can also hold bonded tokens.

mk2: holdsbonds(pi,qa,qb,ts), where pi ∈ P, qa,qb ∈ A, ts is a timestamp

For simplifying things in future rules, we construct the following helping rules:

i. A bond exists both ways between the tokens.

```
holdsbonds(P,Q2,Q1,TS):-holdsbonds(P,Q1,Q2,TS).
```

ii. A token is only the same with itself.

```
same(Q,Q) :- token(Q).
```

iii. If a place either holds Q alone, or in a bond, we say place P has token Q.

```
has(P,Q,TS):- holds(P,Q,TS).
has(P,Q,TS):- holdsbonds(P,Q,_,TS).
```

In every Petri Net, there are arcs from places to transitions, and arcs from transitions to places. In Collective Petri Nets, arcs are labelled with variables or variable pairs. Variables ask for a specific type of token, and only tokens of that type can be assigned to it. We need to describe these arcs with facts and include in these facts the variables and the type of tokens they ask for. For this purpose, we use the following rules:

f6: ptarc(pi,tj,v,ty),

where pi ∈ P, tj ∈ T, v ∈ Av is a variable, ty ∈ A is a token type.

f7: tparc(tj,pi,v,ty),

where pi ∈ P, tj ∈ T, v ∈ Av is a variable, ty ∈ A is a token type.

f8: ptarcbond(pi,tj,va,ta,vb,tb),

where pi ∈ P, tj ∈ T, va,vb ∈ Av are variables, ta,tb ∈ A are token types.

f9: tparcbond(tj,pi,va,ta,vb,tb),

where pi ∈ P, tj ∈ T, va,vb ∈ Av are variables, ta,tb ∈ A are token types.

Fact f6, is a fact for a single token arc, from place pi to place tj. Similarly, f7 is a single token arc, from transition tj to place pi. Facts f8 and f9 encode the arcs moving bonds from place pi to transition tj and from transition tj to place pi respectively.

As we mentioned, our arcs now have variables, asking for a specific type of token. For a transition to be enabled, we need to find an enabling assignment that will respect all the restrictions on the types of tokens of the arc. In Figure 3.1.1 is an example with an

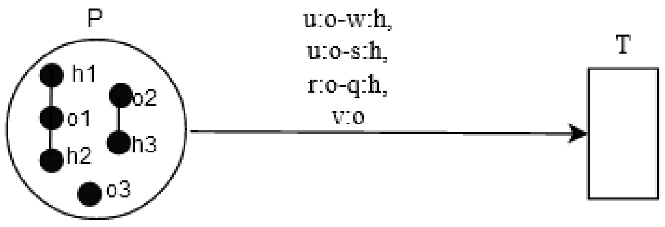


Figure 3.1.1: Enabling Transition

arc and 5 different variables, that we need to satisfy with the type of token they ask for. In this case, we need to have at least 3 different tokens of type “o” and at least 3 different type “h” tokens. One of the type “o” tokens, the one assigned to the variable u, has to be bonded with 2 different type “h” tokens. In this example, the only type “o” token that meets these criteria to be assigned to u, is token o1. Tokens h1 and h2 are assigned to w and s respectively, as they are the only two tokens bonded with o1. Similarly, we assign o2 to r, h3 to q and o3 to v, forming an enabling assignment. With this enabling assignment, transition T in in figure 3.1.1 is enabled with this current marking.

If we want to encode the arc from place P to transition T of the figure 3.1.1, we need to split all the different variables and bonded variables into *ptarc/4* facts or *ptarcbond/6* facts. This is needed because our facts are of static length and we cannot dynamically change them according to the number of parameters our arc needs.

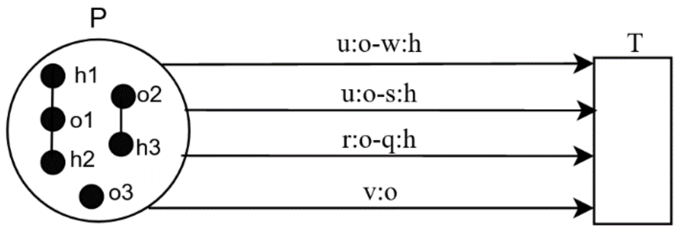


Figure 3.1.2: Enabling Transition for Encoding

Figure 3.1.2 demonstrates the way multiple arc requirements of Figure 3.1.1 are being split into *ptarc* and *ptrarcbond* facts. We need 3 different *ptarcbond/6* facts and 1 *ptarc/4* fact to encode this arc.

1. ptarcbond(p,t,u,o,w,h).
2. ptarcbond(p,t,u,o,s,h).
3. ptarcbond(p,t,r,o,q,h).
4. ptarc(p,t,v,o).

For a transition to be enabled, we need to check whether there is a type-respecting assignment of token instances in the incoming places of the transition, to the variables on the incoming edges. We need to satisfy every incoming arc to our transition, by assigning a token or a bond to the arc's variables. If the total number of satisfied incoming arcs to a transition T, is the same in quantity as all the arcs that point to T, it shows that all the arcs have been satisfied, there is an enabling assignment and transition T is now enabled.

Let us now see in detail the rules needed to encode the enabling assignment. We want our program to non-deterministically select an available token of the correct type, from the appropriate place, and assign it to a value. Then proceed with the remaining tokens that are available and try to assign them to the other values. The program should check all the available correct combinations and return the first one found.

For every place-transition arc that carries a single token (*ptarc*), if the place has an available token of the correct type, or a bond with at least one token of the correct type, then this token might be the one that satisfies the arc. This translates into the following rule:

```
r1a: {satisfied(P,T,V,Q,TS)}  :-    ptarc(P,T,V,X), has(P,Q,TS),
                                    not unavailable(Q,TS),
                                    type(X,Q), time(TS).
```

Rule r1a: If place P, has a token Q at time TS, that is not unavailable, and it is of the same type as the arc demands, then it might be the one satisfying the arc. Notice how the head of the rule is surrounded with curly brackets. The curly brackets indicate a choice rule. Even if all of the atoms in the body of the rule are true, the rule might not always execute and produce the fact *satisfied/5*. This is part of the random selection of the token that can satisfy the variable, but also because we only want one token to be assigned to V and travel through the arc. The fact *unavailable(Q,TS)* is also used here and it divulges that a token Q is unavailable for satisfying any variable at time TS and we will encode it subsequently. From r1a, a new fact is created:

f10: satisfied(pi,tj,v,q,ts), where pi ∈ P, tj ∈ T, v ∈ Av, q ∈ A, ts is a timestamp.

Until now, we have found a way to find the satisfied arcs of single tokens. From figure 3.1.2 that is only the last arc, the one defined with *ptarc(p,t,v,o)*. We need another rule to satisfy variables for bonds. For every place-transition arc that carries a bond (*ptarcb*), if the place has a bond of the correct type, that both tokens are available, then this bond might be the one that satisfies the arc. This translates into the following rule:

```
r1b: {satisfied(P,T,V1,Q1,V2,Q2,TS)} :-   ptarcb(P,T,V1,T1,V2,T2),
                                          holdsbonds(P,Q1,Q2,TS),
                                          type(T1,Q1), type(T2,Q2),
                                          not same(Q1,Q2),
                                          not unavailable(Q1,TS),
                                          not unavailable(Q2,TS),
                                          not splitcycle(P,Q1,Q2,TS),
                                          time(TS).
```

Rule r1b: If place P that relates to transition T with a *ptarcb/6*, holds a bond between two different type-respecting tokens Q1 and Q2 at time TS, that are not unavailable, then this bond might be the one satisfying the arc if it also does not cause un undecidable split.

Notice in rule r1b, we also include another atom, *not splitcycle/4*. This is for capturing circles in our tokens that combined with a transition that splits the bond, will lead to undecidable situations. This is captured by condition (3) in [4], saying that "if two token instances are transferred by a transition to different outgoing places, then these tokens should not remain connected when removing the selected incoming tokens and adding the selected outgoing tokens (we do not clone tokens)."

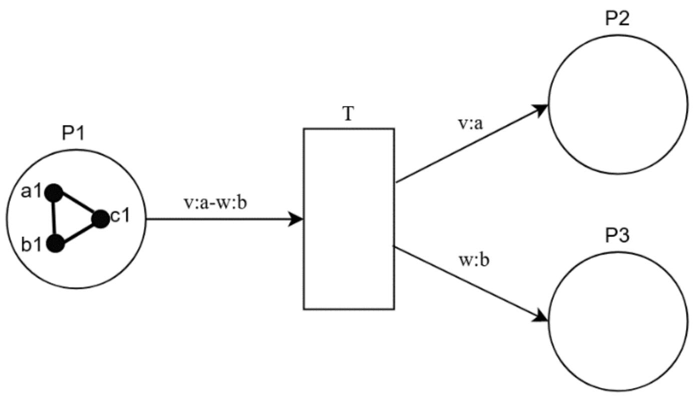


Figure 3.1.3: Undecidable Split

To encode *splitcycle/4* and prevent cloning tokens, we first need to identify when this problem might occur. In Figure 3.1.3 we see an example demonstrating this situation. Given an enabling assignment of v:a1 and w:b1, the execution of transition T will lead to a bond breakage between a1 and b1 as we can see from the outgoing arcs. Bonds between a1 and c1, and b1 and c1 remain unaffected from the transition's firing but c1 cannot follow both a1 and b1 to their new places as this will lead to token cloning. Those undecidable situations happen when tokens are circularly connected like in Figure 3.1.3, and we have a transition that destructs a bond of this circular chain.

Therefore, first we need to identify transitions that destroy bonds. This is done from the following rule:

```
r2:  splitbond(P,T,V1,T1,V2,T2)  :-      ptarcb(P,T,V1,T1,V2,T2),
                                         not tparcb(T ,_,V1,T1,V2,T2).
```

Fact splitbond/6 is created for every instance where there is an incoming bond-arc to a transition with bond between the tokens assigned to V1 and V2, but the same outgoing bond-arc is absent. That means the bond assigned to V1 and V2 is now destructed.

Now we need to create the *splitcycle/4* facts used in r1b for two tokens. The only thing left to do is to identify when two tokens remain connected through different chains of tokens, after their initial bond destruction. We do just this with the following rules:

```
r3a: chain(P,Q1,Q2,A,B,TS) :-    holdsbonds(P,Q1,Q2,TS),
                                 holdsbonds(P,A,B,TS),
                                 not same(Q1,A), not same(Q1,B),
                                 splitbond(P,T,V1,T1,V2,T2),
                                 type(T1,A), type(T2,B),
                                 time(TS).


r3b: chain(P,Q1,Q2,A,B,TS) :-    chain(P,Q1,Q3,A,B,TS),
                                 chain(P,Q2,Q3,A,B,TS),
                                 not same(Q1,Q2),
                                 time(TS).

r3c: chain(P,Q1,Q2,A,B,TS) :-    chain(P,Q2,Q1,A,B,TS).
```

With rule r3a, we construct a chain on place P, connecting all the tokens Q1, Q2 of the place, except from any bond A and B where A and B are the bonded tokens that might break from *splitbond(P,T,V1,T1,V2,T2)*, as A is type T1 and B is type T2. Rule r3b recursively captures that, if a token Q1 is chained with Q3 and Q2 is chained with Q3, then Q1 is chained with Q2, if Q1 and Q2 are not the same token. Finally, r3c defines the rule backwards as well for tokens Q1 and Q2.

Our goal is to identify and capture the bonds that will be susceptible to a splitcycle so that we can avoid assigning them to such variables. The following is how a splitcycle/4 is encoded:

```
r4:     splitcycle(P,Q1,Q2,TS) :-   splitbond(P,T,V1,T1,V2,T2),
                                    chain(P,Q1,Q2,Q1,Q2,TS),
                                    type(T1,Q1), type(T2,Q2).
```

The body of r4 contains four atoms. The chain atom indicates that, after creating a chain between all the bonds, except Q1 and Q2, Q1 and Q2 still form a chain at time TS. The other 3 atoms are included to create the fact *splitcycle/4* for forward execution only. Let us consider the example of figure 3.1.3 and follow the r3 rules.

Because there is no outgoing arc transferring the same bond as the incoming arc, we clearly have the fact *splitbond(p1,t,v,a,w,b)* from rule r2. A chain is created only when we have a *splitbond/6* fact and only for tokens of the same types as the ones that split. Other than that, there is no need for these facts, and we want to avoid generating them for performance reasons when they are not required. To avoid forming unnecessary rules for chains, we include in the rule's r3a body, which is the rule that will initiate the creation of the chain, the splitbond/6 requirement, and the types of the splitting bond.

With *splitbond(p1,t,v,a,w,b)* in mind, we can construct the *chain(P,Q1,Q2,A,B,TS)* and see that the only type a token is a1 and the only type b token is b1. Therefore, in *chain(P,Q1,Q2,A,B,TS)*, the ASP variable A is given the value of the token a1, and B of b1. We assume time TS is 0. So, we can replace the variables with constants and obtain *holds(p,Q1,Q2,a1,b1,0)*. We need two bonded tokens, Q1 and Q2, where Q1!=a1 and Q2!=b1 to replace the last variables. Q1 can be either b1 or c1 and Q2 can be a1 or c1. Since all of these tokens are bonded, we have 2 x 2 = 4 different combinations:

chain(p1,b1,a1,a1,b1,0), chain(p1,b1,c1,a1,b1,0),

chain(p1,c1,a1,a1,b1,0), chain(p1,c1,b1,a1,b1,0).

In our example, rule r3b will create more facts. But r3c will produce 2 more facts:

chain(p1,a1,c1,a1,b1,0) and chain(p1,a1,b1,a1,b1,0).

From the fact: *chain(p1,a1,b1,a1,b1,0)* we got what we were looking for. According to this fact, a1 and b1 will still be chained, even after destroying their bond. And this fact will be the body of r4 to give us the *splitcycle(p,a1,a2,0)* we need, in order to avoid assigning a1 to a and b1 to b.

The facts for the unavailable tokens are the final piece of the puzzle in fully encoding the rules r1a and r1b for satisfied incoming arcs and satisfied incoming bond arcs, respectively. It is a straightforward explanation, and the encoding is as follows:

```
r5a: unavailable(Q,TS)   :-        holdsbonds(P,Q,QQ,TS),
                                   not assigned(P,_,_,Q,TS),
                                   assigned(P,_,_,QQ,TS),
                                   time(TS).
```

```
r5b: unavailable(Q,TS) :-          holdsbonds(P,Q,QQ,TS),
                                   not assigned(P,_,_,Q,TS),
                                   unavailable(QQ,TS),
                                   time(TS).
```

A not assigned token Q, is unavailable if it is bonded with an assigned token QQ (r5a) or an unavailable token QQ (r5b). A token Q that is bonded with an assigned or an unavailable token QQ, will follow the assigned token when the transition fires. On that account, we cannot assign token Q to a different variable for this execution.

After making the *satisfied/5* and *satisfied/7* facts from r1a and r1b respectively, we save the assigned tokens to each variable on a new fact, *assigned/5*. We save single tokens to single variables this time because bonds might be formed or break after the execution. We capture for each time, which token, from which origin place P, is assigned to which variable V on an arc that travels to transition T.

```
r6a: assigned(P,T,V1,Q1,TS) :-   satisfied(P,T,V1,Q1,TS), time(TS).
r6b: assigned(P,T,V1,Q1,TS ; P,T,V2,Q2,TS) :-   satisfied(P,T,V1,Q1,V2,Q2,TS), time(TS).
```

Rule r6a captures the assignment of token Q1 to the variable V1. Rule r6b creates two new rules, one for V1 and Q1 and one for V2 and Q2 to capture the bond's assignment into a single token to variable assignment.

To complete the enabling assignment, a few things need to be sorted out first. As for now, because of the nature of the selective rules r1a and r1b, a couple of things could go wrong. The grounding process generates all the desirable satisfiable results, but it also creates some undesirable ones that we need to dispose of. First, a token can be assigned to more than one variable. Similarly, multiple tokens can be assigned to a single variable. Both scenarios contradict the implementation of the Collective Petri Nets, so we need to remove those instances from our results.

Count the number of variables assigned to a token for any transition. If more than one variable is assigned to a token at this timestamp for the same transition, then discard the set.

```
numVarAsgnToToken(P,T,Q,C,TS):-   C=#count{A:assigned(P,T,A,Q,TS)}, assigned(P,T,_,Q,TS).
:- numVarAsgnToToken(P,T,Q,C,TS), C>1 , time(TS).
```

Similarly, count the number of tokens assigned to a variable for any transition. If more than one token is assigned to a variable at this timestamp for the same transition, then discard the set.

```
numTokensAsgnToVar(P,T,V,C,TS):-   C=#count{A:assigned(P,T,V,A,TS)}, assigned(P,T,V,_,TS).
:- numTokensAsgnToVar(P,T,V,C,TS), C>1 , time(TS).
```

Rules with no head, constraint rules, show that a solution is deadlocked. Deadlocks are undesirable, and the current assignment to the variables is discarded if the body of the deadlock is true.

Note that if a place P is an incoming place to multiple transitions, then the token should be able to get assigned to more than one variable, but for different transitions. These rules do not prevent this from happening, since they count assignments of the token for the same transition.

Finally, to correctly finish the enabling assignment, another scenario must be considered. During the execution of the program, some transitions might not be shown as enabled at some time instance TS, despite having all the appropriate tokens to the incoming places at TS.

Let us consider the scenario of two transitions, namely A and B. We assume that all the incoming places to A and B have all the type-respecting tokens at time TS. That makes both A and B transitions enabled at TS. The problem is that the program, so far, can produce solutions where not all the arcs of A and B are getting satisfied. It can get along with just enough assignments, to make at least one transition enabled, because there is nothing to prevent this from happening. As a result, in some solutions we have neglected assignments that lead to having one of the two transitions shown as not enabled. This is happening again because of the optional satisfied rule and a non-complete solution can get along just fine, with fewer assignments.

To prevent this, we need to make sure that all the tokens that can get assigned do so. In other words, we run the satisfied rules r1a and r1b as many times as possible, assigning all tokens that can get assigned. Then all the transitions that should be enabled are indeed enabled in all the solutions, and one of them is chosen randomly at each distinct time to fire.

r7a:

```
:-        has(P,Q,TS), type(T1,Q), ptarc(P,T,V1,T1),
          not satisfied(P,T,V1,_,TS),
          not assigned(P,T,_,Q,TS),
          not unavailable(Q,TS), time(TS).
```

According to rule r7a, solutions are deadlocked if we have an arc from place P to a transition T that is not satisfied from any token, but place P has at least an appropriate, not unavailable, and not assigned token that does not get assigned to T and V1, at time TS.

With the same logic, rule r7b is used for deadlocking solutions for unassigned appropriate token bonds.

r7b:

```
:-        holdsbonds(P,Q1,Q2,TS), tparcb(T,P,V1,T1,V2,T2),
          type(T1,Q1), type(T2,Q2), time(TS),
          holdsbonds(P,Q1,Q2,TS), ptarcb(P,T,V1,T1,V2,T2),
          type(T1,Q1), type(T2,Q2),
          not satisfied(P,T,V1,_,V2,_,TS),
          not assigned(P,T,_,Q1,TS), not unavailable(Q1,TS),
          not assigned(P,T,_,Q2,TS), not unavailable(Q2,TS),
          not splitcycle(P,Q1,Q2,TS), time(TS).
```

Rule r7b is of the same logic as r7a, but this time for a bond instead of a token. The only extra rule here is the addition of *not splitcycle/4*, because we do not want to deadlock solutions where a bond is not assigned because it would lead to an undecidable situation.

We are now ready to declare the rule to capture the enabled transitions at each time.

```
r8:   enabled(T,TS) :-    C1=#count{P,F1,T1,F2,T2:ptarcb(P,T,F1,T1,F2,T2), place(P)},
                          C1{satisfied(_,T,_,_,_,_,TS)}C1,
                          C2=#count{P,F1,T1:ptarc(P,T,F1,T1), place(P)},
                          C2{satisfied(_,T,_,_,TS)}C2,
                          C1 + C2 > 0,
                          trans(T), time(TS).
```

A transition is set to "enabled" when the number of satisfied incoming arcs is equal to the number of the total incoming arcs of the same transition. This is captured from rule r8. C1 counts the number of the bond incoming arcs to T and then we surround *satisfied(_,T,_,_,_,_)* with curly brackets and C1, to say we want exactly C1 satisfied facts. We do the same thing, but now for single arcs and with C2 as our counter. If we have exactly C2 satisfied arcs at TS, then transition T is enabled at TS.

We now have a set of enabled transitions that may fire. We want a rule to encode the point that an enabled transition may fire.

```
r9: {fires(T,TS)} :- enabled(T,TS), time(TS).
```

In our simulation, we want exactly one transition to fire at each distinct time instance, assuming that at least one transition is enabled at that time instance. We force this by deadlocking situations in which multiple transitions fire, or when no transition fires, but we have an enabled transition.

```
firing(Q,TS) :- Q=#count{T:fires(T,TS)}, time(TS).
:-firing(Q,TS), Q>1, time(TS).
:-firing(Q,TS), Q=0, enabled(T,TS), time(TS).
```

There is only one thing left to do to finish the encoding of the forward firing Collective Petri Net. The assigned tokens must move through the arcs and go to their destined place. Also, tokens irrelevant to the firing must stay in the same place for the next time instance, after the fire.

We devised a useful rule, r10, to determine which tokens will split after a transition fire. It is based on the splitbond/6 rule that we discussed previously.

```
r10:  splittokens(T,Q1,Q2 ; T,Q2,Q1) :-  splitbond(P,T,V1,T1,V2,T2),
                                         assigned(PT,T,V1,Q1,TS), assigned(PT,T,V2,Q2,TS),
                                         time(TS).
```

This rule r10 captures all token bonds assigned to variables on a splitbond. In other words, the bonded tokens Q1 and Q2 will have their bond destroyed after the firing of transition T.

The following rules apply to tokens and bonds that are unrelated to transition T's firing:

```
r11:   holds(P,Q,TS+1) :-        fires(T,TS),
                                 holds(P,Q,TS),
                                 not assigned(P,T,_,Q,TS),
                                 not unavailable(Q,TS),
                                 time(TS).

r12:   holdsbonds(P,Q1,Q2,TS+1) :-    fires(T,TS),
                                      holdsbonds(P,Q1,Q2,TS),
                                      not assigned(P,T,_,Q1,TS),
                                      not unavailable(Q1,TS),
                                      time(TS).
```

Rule r11: Any token that was not assigned nor unavailable, will remain in the same place P for the next time instance.

Rule r12: Any bond that was not assigned nor unavailable, will remain in the same place P for the next time instance. We only need to include not assigned and not unavailable for one token, as if one is assigned then the other one is at least marked as unavailable.

We continue with tokens and bonds that were assigned to a fired transition.

```
r13:   holds(P,Q1,TS+1):-     fires(T,TS),
                              tparc(T,P,V1,T1),
                              assigned(PT,T,V1,Q1,TS), type(T1,Q1),
                              not holdsbonds(PT,Q1,_,TS),
                              time(TS).
```

```
r14:   holds(P,Q1,TS+1):-     fires(T,TS),
                              tparc(T,P,V1,T1),
                              assigned(PT,T,V1,Q1,TS), type(T1,Q1),
                              holdsbonds(PT,Q1,Q2,TS),
                              not holdsbonds(PT,Q1,Q3,TS),
                              token(Q3), not same(Q2,Q3),
                              splittokens(T,Q1,Q2),
                              time(TS).
```

```
r15:   holdsbonds(P,Q1,Q2,TS+1):-   fires(T,TS),
                                    tparcb(T,P,V1,T1,V2,T2),
                                    assigned(PT1,T,V1,Q1,TS), type(T1,Q1),
                                    assigned(PT2,T,V2,Q2,TS), type(T2,Q2),
                                    time(TS).
```

Rule r13: A place P will hold a token, if there is an incoming arc to P from the fired transition T, that carries a token assigned to V1. Token Q1 assigned to V1 was not bonded before.

Rule r14: A place P will hold a token, if the assigned token was bonded with exactly one other token and their bond brakes on T.

Rule r15: This rule captures two scenarios. If PT1 is equal with PT2 then a bond moved through the arc to the new place. If PT1 is not equal with PT2 then a new bond was formed from transition T and will end up to place P.

According to [4] definition (2), “if the selected token instances are bonded together in an outgoing place of the transition, then the bond should also exist on the variables labelling the outgoing arcs (thus we do not recreate existing bonds)”. Tokens bonded with assigned tokens must move with them to the place declared on the outgoing arc. These are the rules that capture this:

```
r16:   holdsbonds(P,Q1,Q2,TS+1):-     fires(T,TS),
                                      holdsbonds(PT,Q1,Q2,TS),
                                      assigned(PT,T,V1,Q1,TS), type(T1,Q1),
                                      not splittokens(T,Q1,Q2),
                                      tparc(T,P,V1,T1),
                                      unavailable(Q2,TS),
                                      time(TS).

r17:   holdsbonds(P,Q1,Q2,TS+1):-     fires(T,TS),
                                      holdsbonds(P,Q1,_,TS+1),
                                      holdsbonds(PT,Q1,Q2,TS),
                                      not splittokens(T,Q1,Q2),
                                      unavailable(Q2,TS),
                                      time(TS).
```

Rule r16: If a token Q2 was bonded at the origin place PT, with the assigned token Q1 and the two tokens did not split, then the destination place P will now hold Q1 and Q2 bonded.

Rule r17: This rule recursively captures that, if a bond has moved to place P at time instance TS+1, then all the bonds it previously had will also move to place P if they did not split.

Rules r11 to r17 keep the tokens in case of a firing. In case of no transition firing at a time instance, we want the tokens to stay at their places.

```
holds(A,B,TS+1) :- holds(A,B,TS),not fires(_,TS),time(TS).

holdsbonds(A,B,C,TS+1) :- holdsbonds(A,B,C,TS),not fires(_,TS),time(TS).
```

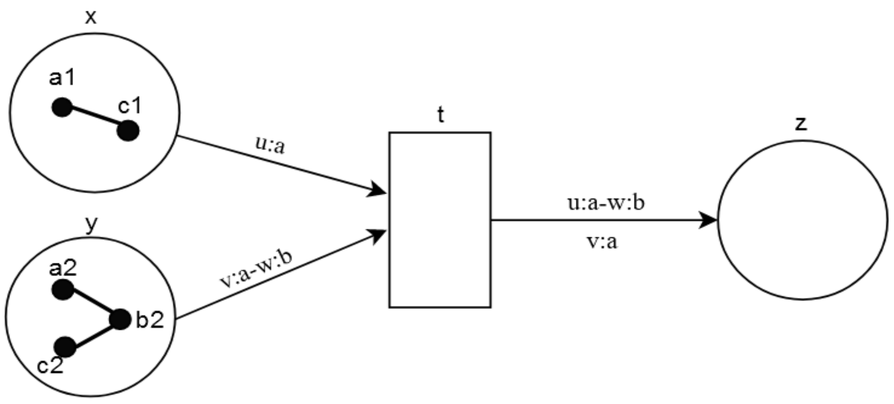


Figure 3.1.4: Encoding Petri Net Example

Consider the Collective Petri Net of Figure 3.1.4. If we want to describe our problem and simulate it on our program, we need to create facts that describe it. The only facts that someone needs to always declare, are the facts for all the arcs, the type of tokens, and the initial markings are *holds/3* and *holdsbonds/4*. Facts for tokens, places and transitions can be created using the following set of rules:

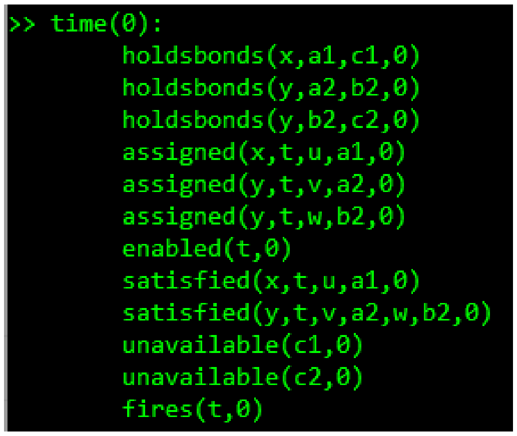
```
token(Q):- type(_,Q).

place(P):- ptarc(P,_,_,_).
place(P):- tparc(_,P,_,_).
place(P):- ptarcb(P,_,_,_,_,_).
place(P):- tparcb(_,P,_,_,_,_).


trans(T):- ptarc(_,T,_,_).
trans(T):- tparc(T,_,_,_).
trans(T):- ptarcb(_,T,_,_,_,_).
trans(T):- tparcb(T,_,_,_,_,_).
```

The Net of figure 3.1.4 can be described from the facts:

Arcs	***Type***	***Initial Marking***
ptarc(x,t,u,a).	type(a,a1). type(a,a2).	holdsbonds(x,a1,c1,0).
ptarcb(y,t,v,a,w,b).	type(b,b1). type(b,b2).	holdsbonds(y,a2,b2,0).
tparcb(t,z,u,a,w,b).	type(c,c1). type(c,c2).	holdsbonds(y,b2,c2,0).
tparc(t,z,v,a).		

We should also declare how many time instances we want our simulation to run. For example, for 5 instances we give the fact:

time(0..4).



```
>> time(0):
        holdsbonds(x,a1,c1,0)
        holdsbonds(y,a2,b2,0)
        holdsbonds(y,b2,c2,0)
        assigned(x,t,u,a1,0)
        assigned(y,t,v,a2,0)
        assigned(y,t,w,b2,0)
        enabled(t,0)
        satisfied(x,t,u,a1,0)
        satisfied(y,t,v,a2,w,b2,0)
        unavailable(c1,0)
        unavailable(c2,0)
        fires(t,0)
```

Figure 3.1.5: Encoding Petri Net Example transition fire clingo output time instance 0.

Figure 3.1.5 shows how tokens satisfy the arcs on time zero. Token a1 is assigned to u, and its bonded token c1 is marked as unavailable. Bond a2-b2 satisfies the *ptarcb(y,t,v,a,w,b)*, with a2 getting assigned to v and b2 to w. Token c2 is also marked unavailable as it is bonded to b2. Both arcs are satisfied, and now transition t is enabled and it fires. (Not all the facts generated are shown in Figure 3.1.5).

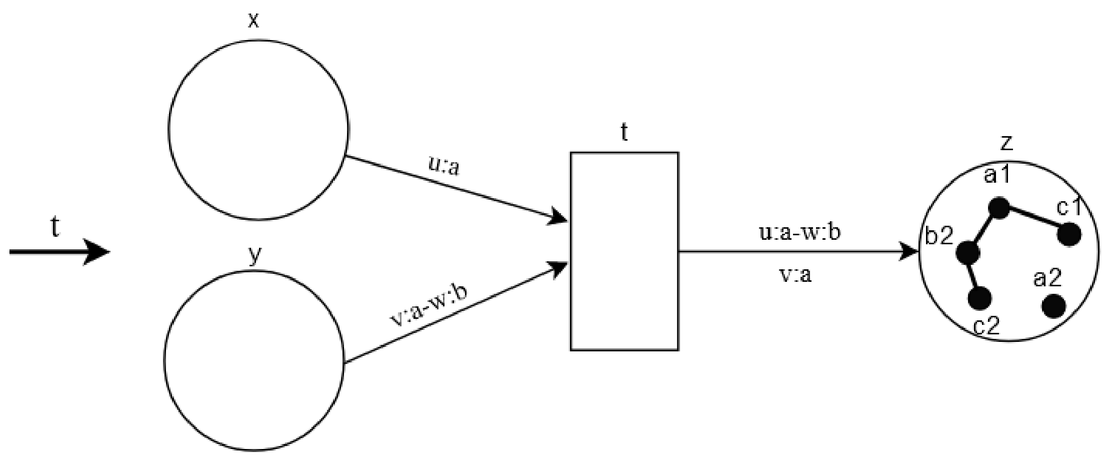
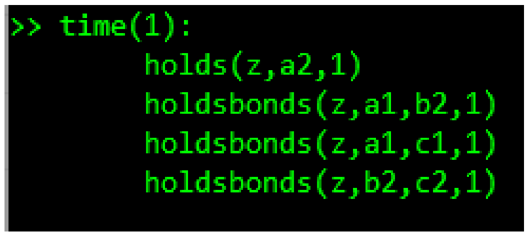


Figure 3.1.6 Encoding Petri Net Example transition fire

After the firing of t at time 0, the bond between a2 and b2 is destroyed and a new bond is being created between a1 and b2. Tokens c1 and c2 follow a1 and b2 respectively and stay bonded with them after the transition fire. Program output from clingo is shown in figure 3.1.7.



```
>> time(1):
       holds(z,a2,1)
       holdsbonds(z,a1,b2,1)
       holdsbonds(z,a1,c1,1)
       holdsbonds(z,b2,c2,1)
```

Figure 3.1.7: Encoding Petri Net Example transition fire clingo output time instance 1.

3.2 Reversible Execution

Reversible execution in Collective Petri nets, can take part in a certain state, if the transition has been previously executed and there are token instances in the outgoing places, that match the requirements of the outgoing arcs, requires a way of keeping track of when a transition has fired. We do not need to keep track of the paths of tokens, or which specific tokens were used in the forward executions, as this is not required. Any tokens that match the requirements of the outgoing arcs on the outgoing places, can be used to reverse a transition, given that the transition has been previously executed forward.

Because reverse execution cannot happen on its own without forward execution, the new code presented in this subchapter, must be used in conjunction with the code for the forward execution.

A way of keeping track of how many times a transition has fired, is required for this approach. Then, the transition can fire back, no more times than the times it has previously been fired forward. The first new rules that we need are the following:

```
r18a:   execcounter(T, 0, 0)  :- trans(T).
r18b:   execcounter(T,C,TS+1) :-  not fires(T,TS), not firesb(T,TS), execcounter(T,C,TS), time(TS+1).
r18c:   execcounter(T,C+1,TS+1) :-  fires(T,TS), execcounter(T,C,TS), time(TS+1).
r18d:   execcounter(T,C-1,TS+1)  :-  firesb(T,TS), execcounter(T,C,TS), time(TS+1).
```

The first parameter of execcounter/3 represents the transition, the second is its counter and the third parameter is the time instance. The rule r18a tells the program that every transition T has zero executions at first (at time instance zero). If the transition did not fire during time instance TS, rule r18b keeps the same count for the next time instance. If the transition fires forward at TS, we increment its count by one for TS+1 using rule r18c, and we decrement it using rule r18d for reverse fire.

Apart from the new rules above, everything else in reversing executions works with the same logic as in forward execution. The code looks similar, but we need to modify it to fit the reversing execution.

The first thing we change is the enabling assignment for the collective reversal. We are interested in the outgoing arcs of our transitions that can be reversed and we need to find a reversing enabling assignment for these arcs.

```
r19a:    {coll_satisfied(P,T,V,Q,TS)} :-    execcounter(T,C,TS), C>0,
                                             tparc(T,P,V,X),
                                             has(P,Q,TS), type(X,Q),
                                             not coll_unavailable(Q,TS),
                                             time(TS).
```

Notice in r19a, the addition of the *execcounter/3* atom in the body of the collective satisfied (*coll_satisfied/5*). In our *execcounter/3* atom, we want the counter C to be greater than zero. This addition was made to avoid reversing executions that were not fired forward more times than reverse. The next change is the replacement of the *ptarc/4* atom with *tparc/4* and this is done because we are now interested in satisfying the outgoing arcs of the transitions. Some tokens might be assigned to forward transitions as well as for reversing transitions, so we need to distinguish some facts from the forward facts via giving them new names. The fact *unavailable/2* is replaced with the *coll_unavailable/2* fact for reversing, as well as *coll_splitcycle/4* replaces splitcycle/4.

In a similar way, we encode *coll_satisfied* fact for outgoing arcs with bonds:

```
r19b:  {coll_satisfied(P,T,V1,Q1,V2,Q2,TS)} :-    execcounter(T,C,TS), C>0,
                                                   tparcb(T,P,V1,T1,V2,T2),
                                                   holdsbonds(P,Q1,Q2,TS,
                                                   type(T1,Q1), type(T2,Q2),
                                                   not same(Q1,Q2),
                                                   not coll_unavailable(Q1,TS),
                                                   not coll_unavailable(Q2,TS),
                                                   not coll_splitcycle(P,Q1,Q2,TS),
                                                   time(TS).
```

For reversing executions, splitting cycles may also apply here. To capture the transitions that break bonds, similarly with the forward execution, we used the following rule:

```
r20:    coll_splitbond(P,T,V1,T1,V2,T2) :-     tparcb(T,P,V1,T1,V2,T2),
                                               not ptarcb(_,T,V1,T1,V2,T2).
```

Rule r20 creates facts to help us create the coll_splitcycle. As with the forward execution, the same idea applies here as well. We want to see if 2 tokens remain chained after breaking their bond in a reverse execution. Therefore, we initiate a chain of two variables that will cause a *coll_splitbond/6*.

```
r21a: chain(P,Q1,Q2,A,B,TS) :-   holdsbonds(P,Q1,Q2,TS),
                                 holdsbonds(P,A,B,TS),
                                 not same(Q1,A), not same(Q1,B),
                                 execcounter(T,C,TS), C>0,
                                 coll_splitbond(P,T,V1,T1,V2,T2),
                                 type(T1,A), type(T2,B),
                                 time(TS).
```

Now, to identify that our chain refers to a splitting bond after reversal and avoid assigning it, we use the following rule:

```
r22:    coll_splitcycle(P,Q1,Q2,TS) :-    coll_splitbond(P,T,V1,T1,V2,T2),
                                          chain(P,Q1,Q2,Q1,Q2,TS),
                                          type(T1,Q1), type(T2,Q2).
```

This rule is the same as the r4 rule in forward execution with the exception of using *coll_splitbond/6* instead of *splitbond/6*. This is done to distinguish when a bond breaks after a forward or a reversing execution.

New unavailable facts named *coll_unavailable* for reversing execution were also used above.

```
r23a: coll_unavailable(Q,TS) :-   holdsbonds(P,Q,QQ,TS),
                                  not coll_assigned(P,_,_,Q,TS),
                                  coll_assigned(P,_,_,QQ,TS),
                                  time(TS).

r23b: coll_unavailable(Q,TS) :-   holdsbonds(P,Q,QQ,TS),
                                  not coll_assigned(P,_,_,Q,TS),
                                  coll_unavailable(QQ,TS),
                                  time(TS).
```

Moving on with the reverse assignments, they change accordingly to:

```
r24a:   coll_assigned(P,T,V1,Q1,TS) :-   coll_satisfied(P,T,V1,Q1,TS), time(TS).
r24b:   coll_assigned(P,T,V1,Q1,TS;P,T,V2,Q2,TS) :-   coll_satisfied(P,T,V1,Q1,V2,Q2,TS), time(TS).
```

To complete the reverse enabling assignment, we need to deadlock the unwanted situations using constraint rules similar to forward execution. The next two rules concern over-assignment.

```
numVarCollAsgnToToken(P,T,Q,C,TS):- C=#count{A:coll_assigned(P,T,A,Q,TS)}, coll_assigned(P,T,_,Q,TS).
:- numVarCollAsgnToToken(P,T,Q,C,TS), C>1 , time(TS).

numTokensCollAsgnToVar(P,T,V,C,TS):- C=#count{A:coll_assigned(P,T,V,A,TS)}, coll_assigned(P,T,V,_,TS).
:- numTokensCollAsgnToVar(P,T,V,C,TS), C>1 , time(TS).
```

Next, we also need to prevent under-assignment with the following two constraint rules for arcs and bond-arcs respectively:

```
:-  has(P,Q,TS), type(T1,Q), tparc(T,P,V1,T1),
    not coll_satisfied(P,T,V1,_,TS),
    not coll_assigned(P,T,_,Q,TS),
    not coll_unavailable(Q,TS),
    execcounter(T,C,TS), C>0,
    time(TS).
```

```
:-  holdsbonds(P,Q1,Q2,TS), tparcb(T,P,V1,T1,V2,T2),
    type(T1,Q1), type(T2,Q2), time(TS),
    not coll_satisfied(P,T,V1,_,V2,_,TS),
    not coll_assigned(P,T,_,Q1,TS), not coll_unavailable(Q1,TS),
    not coll_assigned(P,T,_,Q2,TS), not coll_unavailable(Q2,TS),
    execcounter(T,C,TS), C>0,
    not coll_splitcycle(P,Q1,Q2,TS).
```

A transition is set to "coll_enabled" when the number of satisfied outgoing arcs is equal to the number of the total outgoing arcs of the same transition. This is captured from the following rule r25:

```
r25:   enabled_coll(T,TS) :-   C1=#count{P,V1,T1,V2,T2:tparcb(T,P,V1,T1,V2,T2), place(P)},
                               C1{coll_satisfied(_,T,_,_,_,_,TS)}C1,
                               C2=#count{P,V1,T1:tparc(T,P,V1,T1), place(P)},
                               C2{coll_satisfied(_,T,_,_,TS)}C2,
                               C1 + C2 > 0,
                               trans(T), time(TS).
```

We now have a set of coll_enabled transitions that may fire back (in reverse). We want a rule to encode the point that an enabled transition may fire back.

```
r26:    {firesb(T,TS)} :- enabled_coll(T,TS), time(TS).
```

Given that at least one transition is enabled or coll_enabled, we want exactly one transition to fire, either forward or negative, at each time instance. The following rules are derived from the forward execution, along with two constraint rules. To apply this updated principal, we also need to update our rules.

```
firing(Q,TS) :- Q=#count{T:fires(T,TS)}, time(TS).
:-firing(Q,TS), Q>1, time(TS).
:-firing(Q,TS), Q=0, enabled(T,TS), time(TS).
```

Now we want to count both forward and backward firings and make sure they happen when possible and that only one takes place at every time instance. So, we replace the above three lines of code with the following:

```
firing(Q,TS) :- C1=#count{T:fires(T,TS)}, C2=#count{T1:firesb(T1,TS)}, Q = C1 + C2, time(TS).
:-firing(Q,TS), Q>1, time(TS).
:-firing(Q,TS), Q=0, enabled(T,TS), time(TS).
:-firing(Q,TS), Q=0, enabled_coll(T,TS),time(TS).
```

A fact *firing(Q, TS)* records the total number of firings Q for a time instance TS. If more than one firing happens at the same time, or if no firings happen when we could possibly have one, then with the constraint rules we deadlock the answers with these results.

To update the markings of the tokens after a transition reverses, we recreate the next helping rule for the reversing execution.

```
r27:  coll_splittokens(T,Q1,Q2 ; T,Q2,Q1) :-  coll_splitbond(P,T,V1,T1,V2,T2),
                                              coll_assigned(TP,T,V1,Q1,TS), coll_assigned(TP,T,V2,Q2,TS),
                                              time(TS).
```

Lastly, for moving the tokens after a reversing firing, we create new rules, similar to forward execution but this time we replace *fires/2* with *firesb/2*, *ptarc/4* and *ptarcb/6* with *tparc/4* and *tparcb/6* respectively, and all the facts we changed with their equal collective facts.

The new rules for tokens (r28) and bonds (r29) unrelated to transition T’s firing are:

```
r28:  holds(P,Q,TS+1):-   firesb(T,TS),
                          holds(P,Q,TS),
                          not coll_assigned(P,T,_,Q,TS),
                          not coll_unavailable(Q,TS),
                          time(TS).
```

```
r29:  holdsbonds(P,Q1,Q2,TS+1):-   firesb(T,TS),
                                   holdsbonds(P,Q1,Q2,TS),
                                   not coll_assigned(P,T,_,Q1,TS),
                                   not coll_unavailable(Q1,TS),
                                   time(TS).
```

For *coll_assigned* tokens and bonds:

```
r30:  holds(P,Q1,TS+1):-  firesb(T,TS),
                          ptarc(P,T,V1,T1),
                          coll_assigned(TP,T,V1,Q1,TS), type(T1,Q1),
                          not holdsbonds(TP,Q1,_,TS),
                          time(TS).
```

```
r31:  holds(P,Q1,TS+1):-  firesb(T,TS),
                          ptarc(P,T,V1,T1),
                          coll_assigned(TP,T,V1,Q1,TS), type(T1,Q1),
                          holdsbonds(TP,Q1,Q2,TS),
                          not holdsbonds(TP,Q1,Q3,TS),
                          token(Q3), not same(Q2,Q3),
                          coll_splittokens(T,Q1,Q2),
                          time(TS).
```

And finally for tokens that are bonded with assigned tokens and bonds:

```
r32:  holdsbonds(P,Q1,Q2,TS+1):-  firesb(T,TS),
                                  ptarcb(P,T,V1,T1,V2,T2),
                                  coll_assigned(TP1,T,V1,Q1,TS), type(T1,Q1),
                                  coll_assigned(TP2,T,V2,Q2,TS), type(T2,Q2),
                                  time(TS).
```

```
r33:  holdsbonds(P,Q1,Q2,TS+1):-  firesb(T,TS),
                                  holdsbonds(TP,Q1,Q2,TS),
                                  coll_assigned(TP,T,V1,Q1,TS), type(T1,Q1),
                                  not coll_splittokens(T,Q1,Q2),
                                  ptarc(P,T,V1,T1),
                                  coll_unavailable(Q2,TS),
                                  time(TS).


r34:  holdsbonds(P,Q1,Q2,TS+1):-  firesb(T,TS),
                                  holdsbonds(P,Q1,_,TS+1),
                                  holdsbonds(TP,Q1,Q2,TS),
                                  not coll_splittokens(T,Q1,Q2),
                                  coll_unavailable(Q2,TS),
                                  time(TS).
```

The not fire rules are updated as well:

```
holds(A,B,TS+1) :- holds(A,B,TS),not fires(_,TS),not firesb(_,TS),time(TS).

holdsbonds(A,B,C,TS+1) :- holdsbonds(A,B,C,TS),not fires(_,TS), not firesb(_,TS),time(TS).
```

Reverse Execution Example:

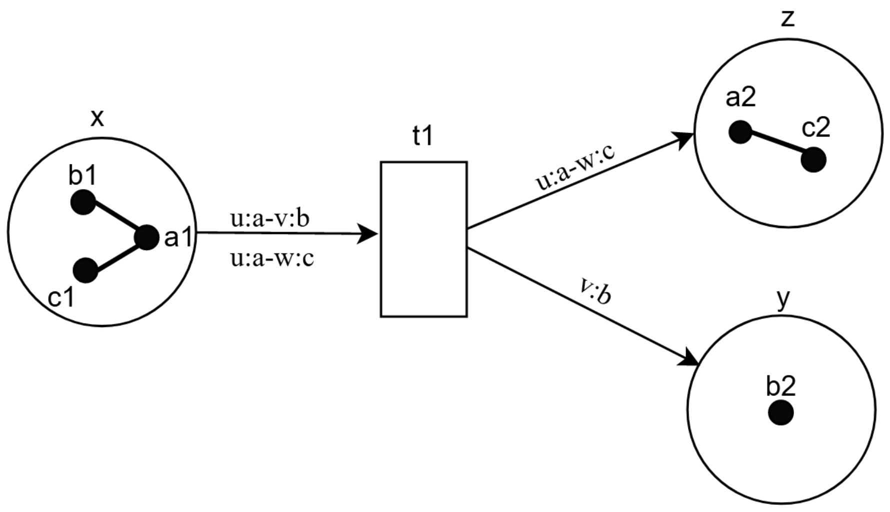


Figure 3.2.1: Reversing Petri Net example time 0.

Consider the Reversing Petri Net of Figure 3.2.1. To encode this Petri Net, we need at least the following facts:

Arcs	***Type***	***Initial Marking***
ptarcb(x,t1,u,a,v,b).	*type(a,a1). type(a,a2).*	*holdsbonds(x,a1,c1,0).*
ptarcb(x,t1,u,a,w,c).	*type(b,b1). type(b,b2).*	*holdsbonds(x,a1,b1,0).*
tparcb(t1,z,u,a,w,c).	*type(c,c1). type(c,c2).*	*holdsbonds(z,a2,c2,0).*
tparc(t1,y,v,b).		*holds(y,b2,0).*

There are many different scenarios and solutions to this problem. In every scenario, t1 is not coll_enabled at time zero, because it did not fire yet.

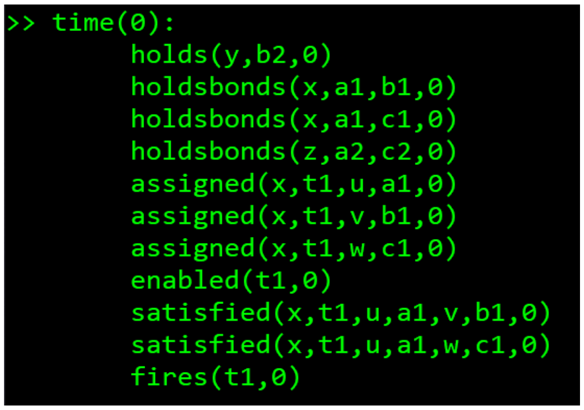

```
>> time(0):
        holds(y,b2,0)
        holdsbonds(x,a1,b1,0)
        holdsbonds(x,a1,c1,0)
        holdsbonds(z,a2,c2,0)
        assigned(x,t1,u,a1,0)
        assigned(x,t1,v,b1,0)
        assigned(x,t1,w,c1,0)
        enabled(t1,0)
        satisfied(x,t1,u,a1,v,b1,0)
        satisfied(x,t1,u,a1,w,c1,0)
        fires(t1,0)
```
Figure 3.2.2: Encoding Reversing Petri Net Example, transition fire, clingo output, time instance 0.

Figure 3.2.2 shows that at time zero, token a1 is assigned to u, token b1 is assigned to v and token c1 is assigned to w. This way the 2 bond-arcs are satisfied and the transition t1 fires. The visual result of the forward execution of t1 is shown in Figure 3.2.3.

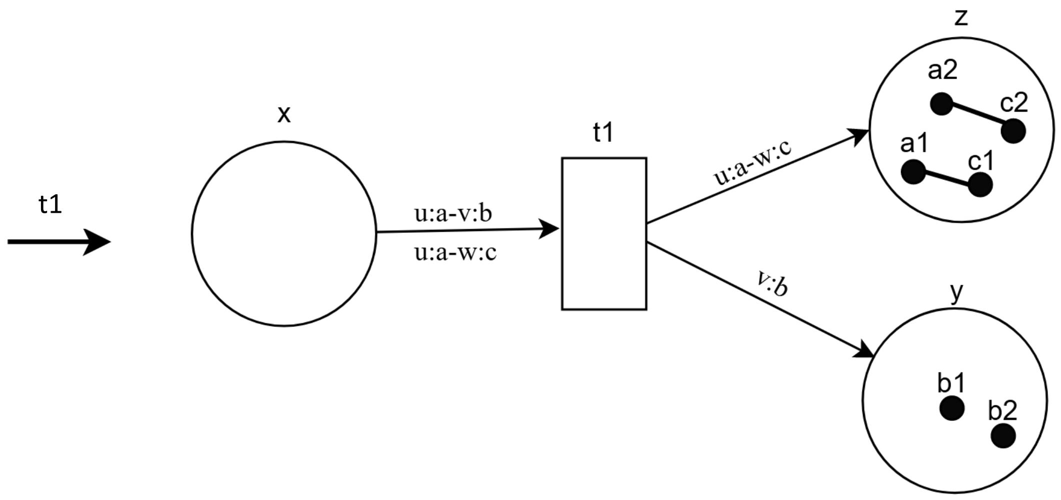
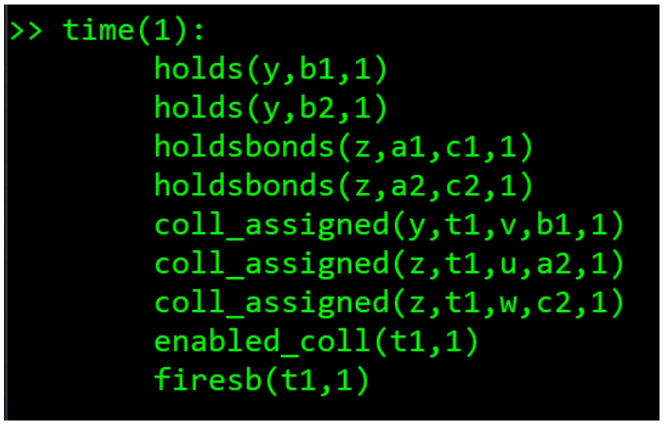


Figure 3.2.3: Reversing Petri Net example time 1.



```
>> time(1):
        holds(y,b1,1)
        holds(y,b2,1)
        holdsbonds(z,a1,c1,1)
        holdsbonds(z,a2,c2,1)
        coll_assigned(y,t1,v,b1,1)
        coll_assigned(z,t1,u,a2,1)
        coll_assigned(z,t1,w,c2,1)
        enabled_coll(t1,1)
        firesb(t1,1)
```

Figure 3.2.4: Encoding Reversing Petri Net Example, transition fire, clingo output, time instance 1.

Figure 3.2.4 shows that at time one, tokens are now assigned for a collective reverse execution of transition t. Also, at this solution, not all the same tokens that were used to fire the transition at time 0, are being used now at time 1 to reverse it.

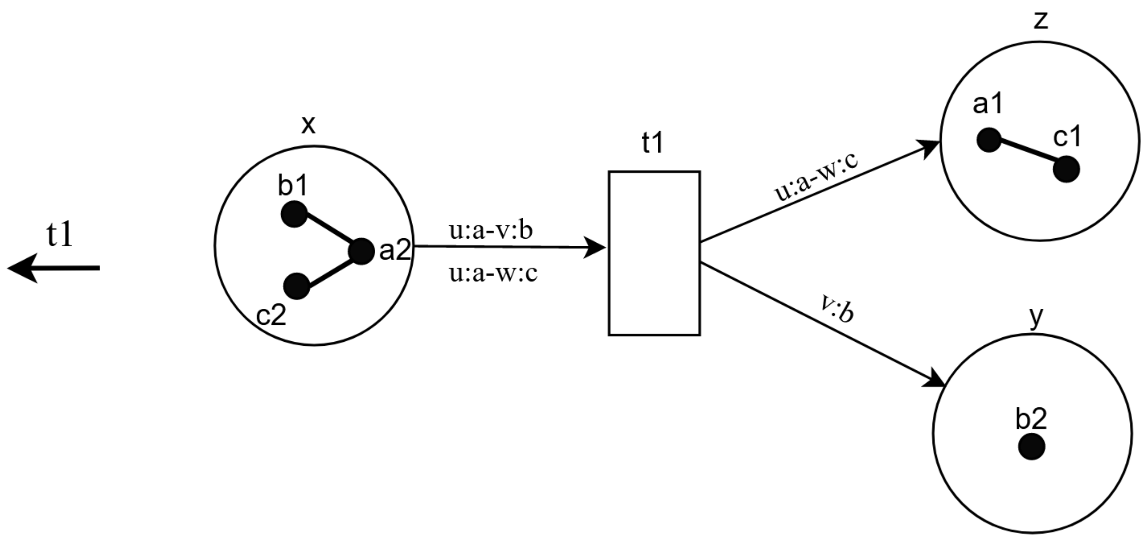
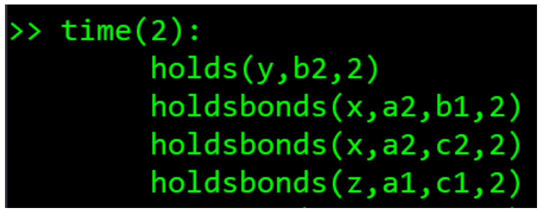


Figure 3.2.5: Reversing Petri Net example time 2.



```
>> time(2):
        holds(y,b2,2)
        holdsbonds(x,a2,b1,2)
        holdsbonds(x,a2,c2,2)
        holdsbonds(z,a1,c1,2)
```

Figure 3.2.6: Encoding Reversing Petri Net Example, transition fire, clingo output, time instance 2.

Figure 3.2.6 shows the marking of the tokens at time two, and Figure 3.2.5 shows the visual representation of the Net at time 2.

3.3 Conditions

An addition to Multi-token Petri nets, are the transition conditions. Apart from the rules that cause a transition to be forward or reverse enabled presented in subchapters 2.5.1 and 2.5.2 respectively, we add conditions on the transitions of a Petri net, that need to be respected as well. These conditions are not designed explicitly for the problem of Antenna selection. They are general and can be used for many problems needing controlled firing or conditions on some variables.

If a transition bears a forward condition on a variable, then the token assigned to this variable, not only has to respect the type of token that the variable is asking, but it also must respect the conditions of the transition.

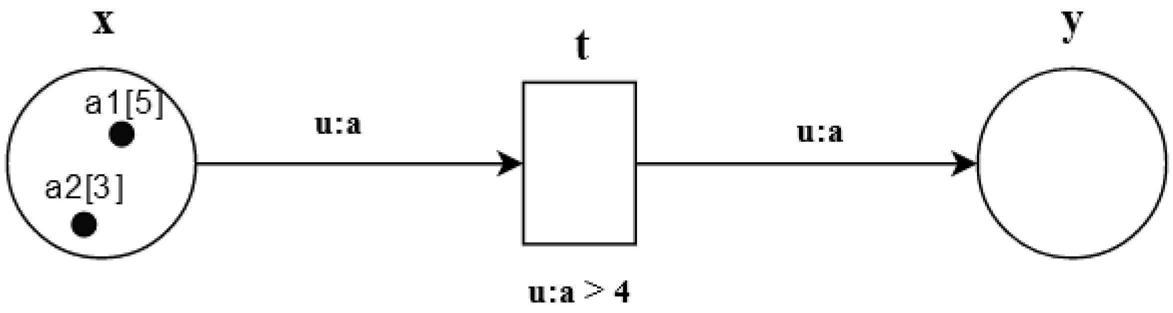


Figure 3.3.1: Forward Condition

Consider the example of Figure 3.3.1, with two type respecting tokens a1 and a2, held by the input place of transition t, x. In a normal Collective Petri Net without conditions, both a1 and a2 can be assigned to u on the incoming arc of t. With this addition, tokens now have values assigned to them and these values must meet the criteria of the transitions' conditions that may be present. In this example, transition t asks for a token of type a, that its value is greater than 4. So, the only token that can be assigned to variable u is token a1, which has a value of 5 and is of course greater than 4.

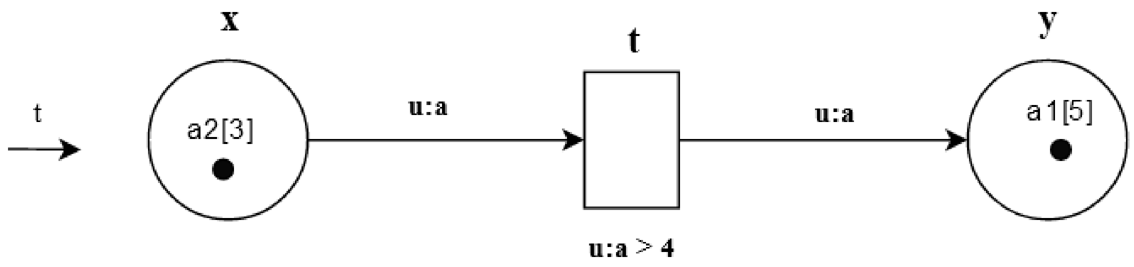


Figure 3.3.2: Forward Condition after transition fire

Token a2 is unable to satisfy the transition's t condition, and thus cannot be assigned to u and travel to place y, so in figure 3.3.2 transition t is not forward enabled.

The same idea applies to conditions for transition reversing. Figure 3.3.3 shows an example of a transition with conditions for both forward and reverse executions.

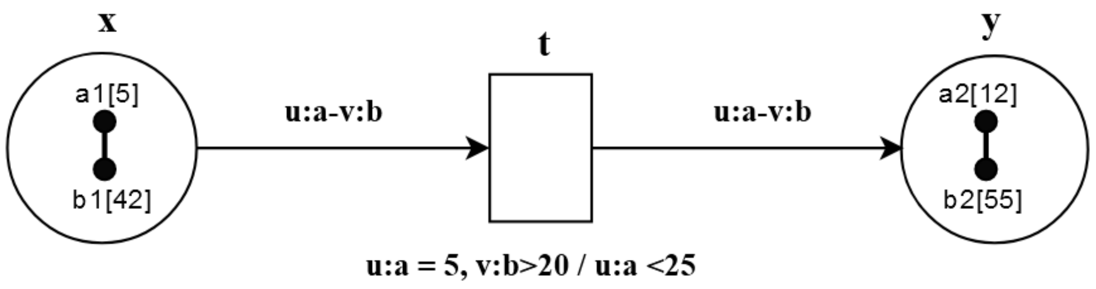


Figure 3.3.3: Forward and Reverse Conditions

The two first conditions (before the backslash) concern forward execution, whereas the last one (after the backslash) concerns reverse execution. From the initial marking shown in this example, we can see that bond a1-b1 is type-respecting and also condition respecting, as a1's value is indeed equal to five and b1's value is greater than 20. That makes transition t enabled to fire forward.

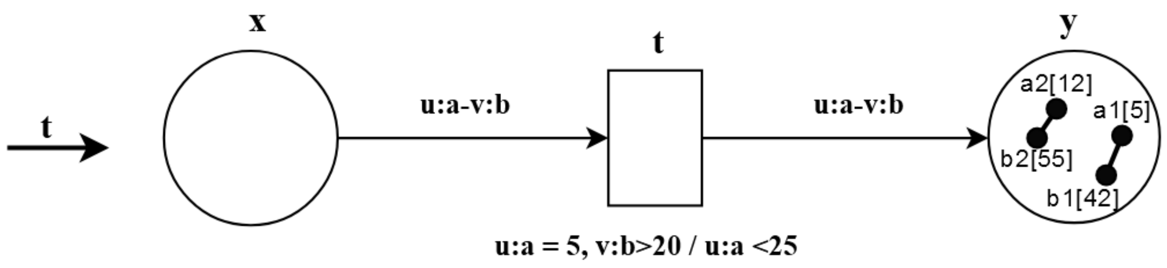


Figure 3.3.4: Forward and Reverse Conditions after forward fire

Figure 3.3.4 depicts the Petri Net after transition t fires forward. For reversing, we have a condition only on variable u. Because transition t fired once, it can now reverse once. Both bonds in y are type respecting the outgoing arc and both bonds are condition respecting. Remember that both bonds can be non-deterministically selected to reverse transition t, regardless of whether the tokens were directly used to fire transition t before.

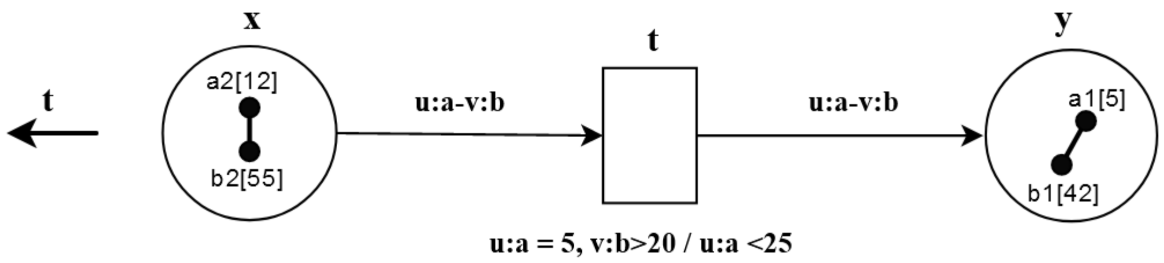


Figure 3.3.5: Forward and Reverse Conditions after backward fire

Assuming that bond a2-b2 was non-deterministically picked to reverse transition t, the outcome is shown in figure 3.3.5. Bond a2-b2 is now type-respecting in regards of the forward execution of t, but a2 does not comply with the condition. Transition t was already reversed the same amount of times as it was forwardly executed, so there are no more possible firings in this example.

It is now time to encode conditions and add them to the Collective Petri Nets program. The first thing we need is to add values to our tokens. This is done initially using the fact *value/3*.

f6: value(q,v,ts), q ∈ A, v ∈ Z, ts is a time instance.

Fact f6 declares that token Q has a value V at time instance TS. Values can be different in different time instances. But if a value is not given for some time instances, the same value is being kept for the token. This is captured by:

```
r35: updated(Q,TS+1):- value(Q,V,TS), value(Q,VN,TS+1), VN!=V, time(TS+1).
r36: value(Q,V,TS+1):- value(Q,V,TS), not updated(Q,TS+1), time(TS+1).
```

Rule r35 is responsible for creating a fact *updated/2* if a token's value has been updated to a new value for the next time instance TS+1. If it has not been updated with a new value, rule r36 renews the current value for TS+1. In other words, the value of a token is kept the same for all time instances, unless explicitly updated.

Forward conditions:

We want our conditions to be flexible through different time instances and allow different time instances to have different conditions. We use the following fact for forward conditions:

f7: fwcond(tj,v,s,x,ts) , tj ∈ T is a transition, v ∈ Av is a variable,
s ∈ { > , < , !=, = }, x ∈ Z^+ is the condition value and ts is a time instance.

For convenience, for transitions that do not change during the simulation, we can use f7 with only the four first parameters (ts is excluded) and the following rule generates the conditions for every time instance of our simulation:

```
fwcond(T,Var,Eq,Val,TS) :- fwcond(T,Var,Eq,Val), time(TS).
```

Our next goal is to find tokens that satisfy all the conditions of a transition for a variable that they can be assigned to. If this is the case, the token can be assigned to the variable.

Using the following rules, we capture the value type pairs to minimize later on the code needed to encode the Conditions.

```
ptValTypePair(P,T,V,TY):-ptarc(P,T,V,TY).
ptValTypePair(P,T,V1,T1 ; P,T,V2,T2):-ptarcb(P,T,V1,T1,V2,T2).
```

The symbols =, ≠, <, ≤, >, and ≥ are used to compare integers. We must write one rule for each one of them in ASP to encode them. Next is an example for encoding greater than (>) but the rest are very similar.

```
r37:  fwsatcond(T,V,Q,">",X,TS) :-  fwcond(T,V,">",X,TS),
                                    ptValTypePair(P,T,V,TY),
                                    has(P,Q,TS), type(TY,Q),
                                    value(Q,A,TS), A > X, time(TS).
```

Rule r37 states that a token Q, satisfies the forward condition on value V of transition T, for having a value greater than X in time TS. In the body of the rule, we have a forward condition of transition T, a place P connected with T that holds a token Q, the same type asked from the condition, with value A that is greater than X at time instance TS. We encode other symbols similar to this one, changing the body's comparison symbol as well as the fourth and third parameters of *fwsatcond/6* and *fwcond/5* respectively, according to the symbol we try to encode.

In many problems, more than one rule might apply to a variable. We want to find the tokens of the variable's asking type that also fulfill all the condition requirements.

In general, the idea here is to find the tokens that satisfy all the rules of a transition for a variable and save them in facts. The first thing is to count the conditions for a variable on a transition, and then find the tokens, whose type matches the variable's type, and count how many of these conditions they satisfy. If the number of conditions they satisfy is the same as the total number of the conditions for the variable, then we create a fact *satAllFwCond/4*, saying that the token satisfies all the conditions of a variable of a transition.

```
r38: satAllFwCond(T,V,Q,TS) :-   C1=#count{S,X:fwcond(T,V,S,X,TS)},
                                 C1{fwsatcond(T,V,Q,_,_,TS)}C1,
                                 ptValTypePair(P,T,V,TY),has(P,Q,TS),type(TY,Q),
                                 place(P), time(TS).
```

One might notice there are some unwanted scenarios. In the event of multiple conditions for multiple variables that concern tokens of the same type, even if there is a perfect matching, because tokens are non-deterministically assigned to variables, there will be “correct” answer sets, where the tokens are not perfectly matched with the variables. Although there is always going to be the desired solution with the perfect matching, if it exists, optimally we would like to get the maximum matching every time and ignore the other solutions.

Figure 3.3.6 shows an example of this problem:

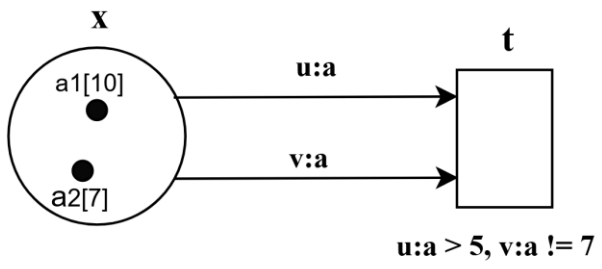


Figure 3.3.6: Variable matching problem

The problem here is that we will have 2 answer sets, one where a1 and a2 are assigned to variables v and u respectively and one answer set where only a1 gets assigned to u. The second answer set does not have a maximum/optimal matching and thus the transition t will be not enabled.

To solve this problem for 2 variables, we can deadlock the scenario with a constraint rule:

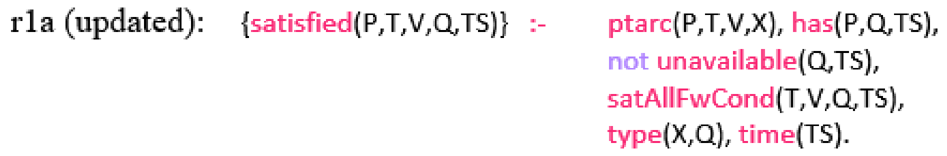
```
:-  not assigned(P,T,Vns,_,TS), satAllFwCond(T,Vns,Q,TS), assigned(P,T,V,Q,TS), Vns!=V,
    satAllFwCond(T,V,Q2,TS), not same(Q,Q2), not assigned(P,T,_,Q2,TS), not unavailable(Q2,TS),
    time(TS).
```

The constraint rule above suggests deadlocking an answer set, if there is a token Q that can be assigned to a not assigned variable Vns, but instead is assigned to a variable that can be satisfied by a different available token Q2. This solution will deadlock the second answer set for Figure 3.3.6.

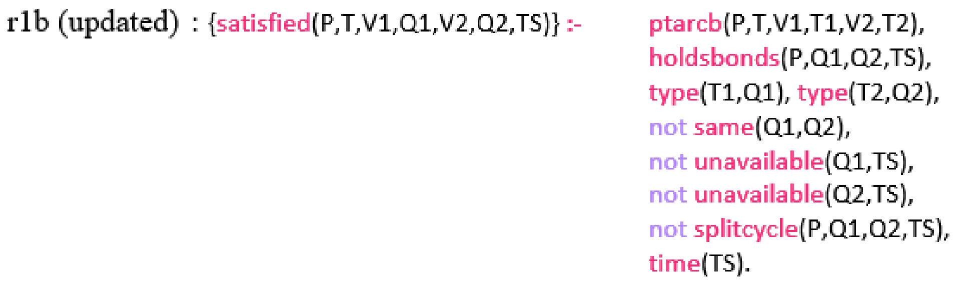
For more than two variables asking for tokens of the same type, the deadlock above will not deadlock all the non-perfect matchings. However, extra constraint rules, much more complex but similar to this one, can be added to achieve a perfect matching in every answer set. To avoid adding extra rules, another approach is to use reachability constraints for problems of this kind if they might exist.

Generally, the problem of perfect matching can be solved in polynomial time using the algorithm of Hopcroft–Karp [22] that is based on augmenting paths.

Conditions are successfully encoded in ASP, but we need to take them into account before our assignments. We avoid satisfying arcs with tokens that do not respect the conditions and to do that, we add the atom *satAllFwCond/4* in the body of the rules r1a and r1b as follows:



```
r1a (updated):   {satisfied(P,T,V,Q,TS)}  :-   ptarc(P,T,V,X), has(P,Q,TS),
                                                not unavailable(Q,TS),
                                                satAllFwCond(T,V,Q,TS),
                                                type(X,Q), time(TS).
```



```
r1b (updated) : {satisfied(P,T,V1,Q1,V2,Q2,TS)} :-   ptarcb(P,T,V1,T1,V2,T2),
                                                      holdsbonds(P,Q1,Q2,TS),
                                                      type(T1,Q1), type(T2,Q2),
                                                      not same(Q1,Q2),
                                                      not unavailable(Q1,TS),
                                                      not unavailable(Q2,TS),
                                                      not splitcycle(P,Q1,Q2,TS),
                                                      time(TS).
```

We also need to relax the constraint rules r7a and r7b, and stop deadlocking solutions where tokens are not assigned because of not satisfying conditions.

r7a (updated) :

```
:-      has(P,Q,TS), type(T1,Q), ptarc(P,T,V1,T1),
        not satisfied(P,T,V1,_,TS),
        not assigned(P,T,_,Q,TS),
        not unavailable(Q,TS),
        satAllFwCond(T,V1,Q,TS),
        time(TS).
```

r7b (updated) :

```
:-      holdsbonds(P,Q1,Q2,TS), tparcb(T,P,V1,T1,V2,T2),
        type(T1,Q1), type(T2,Q2), time(TS),
        holdsbonds(P,Q1,Q2,TS), ptarcb(P,T,V1,T1,V2,T2),
        type(T1,Q1), type(T2,Q2),
        not satisfied(P,T,V1,_,V2,_,TS),
        not assigned(P,T,_,Q1,TS), not unavailable(Q1,TS),
        not assigned(P,T,_,Q2,TS), not unavailable(Q2,TS),
        satAllFwCond(T,V1,Q1,TS), satAllFwCond(T,V2,Q2,TS),
        not splitcycle(P,Q1,Q2,TS), time(TS).
```

Reverse conditions:

Similarly with the forward conditions, reverse conditions are described with the following fact:

f8: rvcond(tj,v,s,x,ts) , tj ∈ T is a transition, v ∈ Av is a variable, s ∈ { > , < , !=, = }, x ∈ Z^+ is the condition value and ts is a time instance.

For convenience, for transitions that do not change during the simulation, we can use f8 with only the four first parameters (ts is excluded) and the following rule generates the conditions for every time instance of our simulation, exactly like in forward conditions.

```
rvcond(T,Var,Eq,Val,TS) :- rvcond(T,Var,Eq,Val), time(TS).
```

Our next goal is to find tokens that satisfy all the conditions of a transition for a variable that they can be assigned to. If this is the case, the token can be assigned to the variable.

This time we want to capture the value pairs of tparc and tparcb instead, because our conditions apply to the variables of the outgoing arcs of a transition.

```
tpValTypePair(P,T,V,TY):-tparc(T,P,V,TY).
tpValTypePair(P,T,V1,T1 ; P,T,V2,T2):-tparcb(T,P,V1,T1,V2,T2).
```

The symbols =, ≠, <, ≤, >, and ≥ are used to compare integers. We must write one rule for each one of them in ASP to encode them. Next is an example of encoding less than (<) for a reversing condition, but the rest are very similar.

```
r39:  rvsatcond(T,V,Q,"<",X,TS) :-  rvcond(T,V,"<",X,TS),
                                    tpValTypePair(P,T,V,TY),
                                    has(P,Q,TS), type(TY,Q),
                                    value(Q,A,TS), A < X, time(TS).
```

Rule r39 states that a token Q, satisfies the reverse condition on value V of transition T, for having a value less than X in time TS. In the body of the rule, we have a reverse condition of transition T, a place P connected with T that holds a token Q, the same type asked from the condition, with value A that is less than X at time instance TS. We encode other symbols similar to this one, changing the body's comparison symbol as well as the fourth and third parameters of *rvsatcond/6* and *rvcond/5* respectively, according to the symbol we try to encode.

The matching deadlock rule for the reverse conditions is:

```
:-    not coll_assigned(P,T,Vns,_,TS), satAllRvCond(T,Vns,Q,TS),
      coll_assigned(P,T,V,Q,TS), Vns!=V, satAllRvCond(T,V,Q2,TS),
      not same(Q,Q2), not coll_assigned(P,T,_,Q2,TS),
      not coll_unavailable(Q2,TS), time(TS).
```

Similarly to forward conditions, more than one rule might apply to a variable. We want to find the tokens of the variable's asking type that also fulfill all the condition requirements.

In general, the idea here is to find the tokens that satisfy all the rules of a transition for a variable and save them in facts. The first thing is to count the reverse conditions for a variable on a transition, and then find the tokens, whose type matches the variable's type, and count how many of these reverse conditions they satisfy. If the number of reverse conditions they satisfy equals the total number of conditions for the variable, then we create a fact *satAllRvCond/4*, saying that the token satisfies all the conditions of a variable of a transition.

```
r40: satAllRvCond(T,V,Q,TS) :-   C1=#count{S,X:rvcond(T,V,S,X,TS)},
                                  C1{rvsatcond(T,V,Q,_,_,TS)}C1,
                                  tpValTypePair(P,T,V,TY),has(P,Q,TS),type(TY,Q),
                                  place(P), time(TS).
```

Just like in forward execution, we update the relative satisfied rules and constraint rules, adding *satAllRvCond/4* this time as an atom to their body.

3.4 Extension to support Antenna Selection.

With Controlled Multi-token Petri nets (CMPN) coded in ASP, we are now almost ready to describe a problem of antenna selection as a CMPN and solve it. Reversible execution will not be used for this problem as it is not required by the algorithm. What is left to do, is to figure out a way of calculating and storing the sum capacity and create the conditions for transitions. The algorithm proposed in [7], does not have a low-level way of updating a neighborhood when its place is being changed from a different neighborhood. That can happen in the situation of overlapping neighborhoods or when an antenna on the edge of a neighborhood passes the token to the adjacent neighborhood. So in this implementation, we use only one neighborhood, without dividing it into smaller ones.

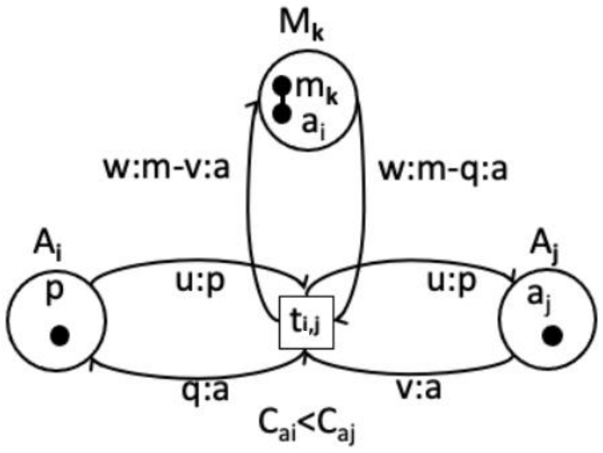


Figure 3.4.1: Part of the RPN model for Antenna Selection

Problem Encoding:

Figure 3.4.1 shows a part of the CMPN model for moving the power from place Ai to place Aj via the execution of transition $t_{i,j}$. Notice that this is the half image, and there exists another transition $t_{j,i}$, with similar arcs but in reverse, moving the power to place Ai from place Aj. Generally, for every two adjacent antennas we need two places representing each antenna, one place for the neighborhood, two transitions for moving tokens between the antennas, and twelve arcs, eight of them are single token arcs and the rest four are bonded arcs.

Having to declare all these facts to describe a problem, makes it very susceptible to mistakes. A problem with just 10 connections between antennas requires 120 arcs to be declared. To avoid deadly type mistakes which are not being identified by ASP, and make the life of the user easier, we use a custom constructor for the problem which is a 5-tuple, defined with *constructor(Ai, Aj, Tij, Tji, Mk)* where Ai and Aj are the two antennas, Tij is the transition for moving the power from Ai to Aj, Tji is the transition for moving the power from Aj to Ai and Mk is the neighborhood.

For each constructor, using the following rules we produce all the arcs needed:

```
ptarc(Ai,Tij,u,p ; Aj,Tji,u,p ; Aj,Tij,v,a ; Ai,Tji,q,a) :- construct(Ai,Aj,Tij,Tji,Mk).
tparc(Tji,Ai,u,p ; Tij,Aj,u,p ; Tji,Aj,v,a ; Tij,Ai,q,a) :- construct(Ai,Aj,Tij,Tji,Mk).

tparcb(Tij,Mk,w,m,v,a ; Tji,Mk,w,m,q,a) :- construct(Ai,Aj,Tij,Tji,Mk).
ptarcb(Mk,Tji,w,m,v,a ; Mk,Tij,w,m,q,a) :- construct(Ai,Aj,Tij,Tji,Mk).
```

As previously mentioned in subchapter 3.2, to encode a problem we need to define the ***arcs***, the facts for token ***types*** and the ***initial marking*** using the facts *holds/3* and *holdsbonds/4*. We have shown how the rest of the facts are being generated in subchapter 3.1.

There are three types of tokens in this algorithm, as mentioned previously. These are:

1. the type ‘p’ tokens that represent the power,
2. the tokens of type ‘m’ that represent the neighborhood and
3. the tokens of type ‘a’ that represent the antennas.

Places that represent powered-on antennas, hold a token of type **p**, and their antenna token is bonded with a neighborhood token, type **m**, at the neighborhood place. Places that represent powered off antennas, hold a token of type **a**. The number of the tokens that are of type **m** and type **p**, are equal to the number of the powered-on antennas, while the number of type **a** tokens is equal to the number of the total antennas of our net = NT. The antenna users are not being represented graphically but their contribution can be seen on the matrix H. For our simulation, we assume that users do not change during the execution.

Since we have an antenna token for every antenna on our net, it is easy to declare the types of the antenna tokens automatically, creating tokens with the same names as the antennas, for every antenna used.

```
type(a,Ai ; a,Aj ) :- construct(Ai,Aj,Tij,Tji,Mk).
```

It is only necessary to give facts for the tokens of type p and m manually. We declare the same number of type p tokens and type m tokens, equal to the number of powered on antennas we want to have.

Finally, the initial marking must be given, expressing which antennas are powered on initially. With *holds/3* facts we write facts for the antennas that hold each power token and with *holdsbonds/3* facts we write facts for the bonds created on the neighborhood place, between the antenna tokens of powered on antenna places and the type m tokens of the neighborhood place.

The next rule can also help placing the antenna tokens to the antennas without any power token, making the definition of the problem as easy as possible.

```
holds(A,A,0):- #count{P:holds(A,P,0),type(p,P)}=0, type(a,A).
```

Some mistakes in ASP are hard to identify and problems like these will lead to wrong results and results. To help identify some errors, the following rules were created:

```
error("Type_Not_Specified",A):-holdsbonds(_,A,_,_),not type(_,A).
error("Type_Not_Specified",A):-holdsbonds(_,_,A,_),not type(_,A).
error("Type_Not_Specified",A):-holds(_,A,_),not type(_,A).
error("Token_Not_Used",Q):-type(T,Q), not holds(_,Q,0),not holdsbonds(_,Q,_,0),T!=a.
```

ASP Extension:

To solve the problem of Antenna selection, we must create a few extra rules that will create the conditions and give values to our tokens. In short, we first identify the antennas that might get powered on, also known as the unpowered antennas adjacent to powered on antennas. Then we calculate the new sum capacity that will be produced from moving the power to the candidate antenna from their neighbour’s powered-on antenna. The sum capacity is calculated using Python and is being stored in the value of the antenna tokens found in antenna places. The current sum capacity is also stored in power tokens' value. Then we create conditions and allow transitions to fire only if this exchange of power results in an increase of the sum capacity.

We use the fact *candidateOn/3* to capture the candidates for getting the power. The first parameter is of *candidateOn/3* is the candidate antenna, the second one is the adjacent powered-on antenna, symbolizing the antenna that might give its power to the candidate, and the last parameter is the current time instance. The rule for this fact is the following:

```
m1:     candidateOn(A,P,TS) :-  connected(A,T), connected(T,P),
                                holds(A,O,TS), holds(P,Q,TS),
                                type(a,O), type(p,Q), time(TS).
```

The Atom connected is created using the following rules:

```
connected(P,T):-ptarc(P,T,_,_).
connected(P,T):-ptarcb(P,T,_,_,_,_).
connected(T,P):-tparc(T,P,_,_).
connected(T,P):-tparcb(T,P,_,_,_,_).
```

Rule m1 construes that an antenna place A is candidate on, if it is connected with a place P through transition T, place P is currently on while place A is not. We can see that place P is indeed powered on, as it holds a token of type 'p' and place A is not powered on as it holds a token of type 'a'.

For the calculation of the sum capacity, we first need to identify the antennas that are powered on and then construct the matrix Hc, by collecting their data vectors from the corresponding rows in matrix H. Sum capacity will be calculated using python, and a way of communicating to python which antennas are currently on is needed, otherwise matrix Hc cannot be created, and sum capacity cannot be calculated.

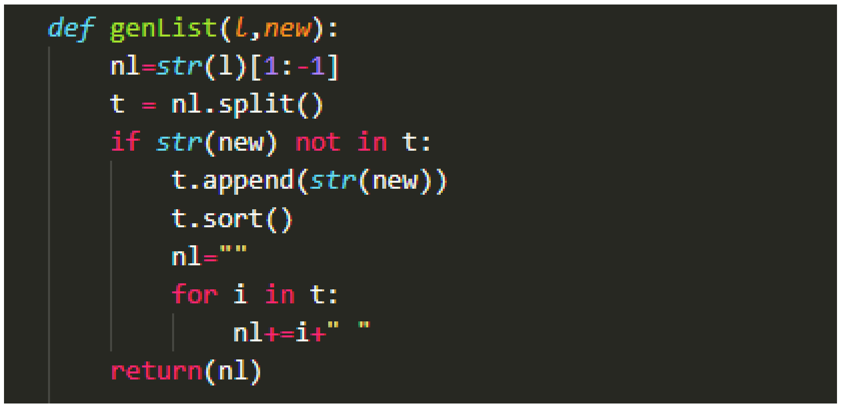
This is the trickiest part of calculating the sum capacity because ASP does not support lists and thus, we cannot send a list with all the currently powered on antennas. For every problem, the number of power tokens/powered-on antennas may vary and the number of parameters of each fact is static. So, it is not practical to have different rules and facts with different number of parameters, one for every possible number of powered on antennas.

The first approach was to send each powered-on antenna one by one to python and create a list with the powered-on antennas stored in a global variable in python. However, that does not produce the desired results every time, because of the way grounding works. Because every combination of possible constants is used to replace the variables, the calculation of sum-capacity was not reliable this way. Using non static global variables in Python in conjunction with Clingo turned out to be not such a good idea.

The solution used for this problem, is to use recursive rules in ASP and store the list as a string in a parameter of a fact, and then this list is translated back to a list in Python.

```
m2:     list("",0,TS):-time(TS).
m3:     list(@genList(L, A),TS):- holds(A,Q,TS), type(p,Q), list(L,S,TS).
m4:     list(L,@getSize(L),TS) :- list(L,TS),time(TS).
```

We want the parts of this list to be the names of the places that hold a power token at time instance TS. So, we initiate an empty list of length zero for every TS using rule m2. Then for every place A that holds a type ‘p’ token Q, we add it to the list L, and using the function *genList* from python, shown in Figure 3.4.1, we get the new list with A appended to L alphabetically.

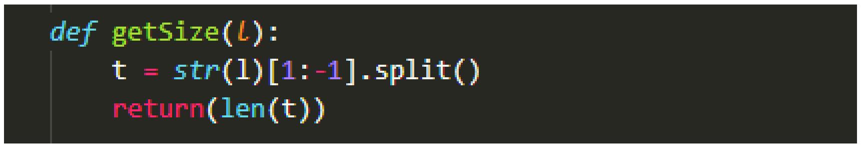


```python
def genList(l,new):
    nl=str(l)[1:-1]
    t = nl.split()
    if str(new) not in t:
        t.append(str(new))
        t.sort()
        nl=""
        for i in t:
            nl+=i+" "
    return(nl)
```

Figure 3.4.1: Python function genList for ASP

Alphabetical order is used to reduce the number of facts created in ASP, as if it was not alphabetical, we would create a number of lists, equal to all the different combinations of powered-on places starting with length one.

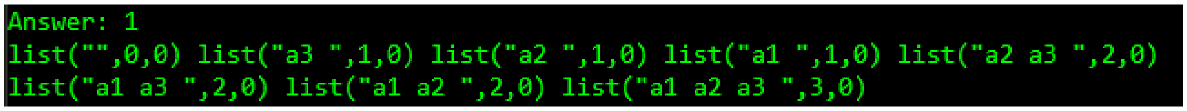
Finally, rule m4 creates a list with 3 parameters, including the length of the list L. The length is vital to know which is the complete list. The complete list is a list with a length equal to the number of power tokens.



```python
def getSize(l):
    t = str(l)[1:-1].split()
    return(len(t))
```

Figure 3.4.3: Python function getSize for ASP

The results of the facts created for three power tokens, held in places a1, a2, and a3, are shown in Figure 3.4.3. This is not optimal in terms of the number of facts created, but list creation in ASP will be further optimised in the next chapter. The fact that we are interested in, is the list of length 3, which is equal to the number of all powered on antennas, and is the last fact in Figure 3.4.3.



```
Answer: 1
list("",0,0) list("a3 ",1,0) list("a2 ",1,0) list("a1 ",1,0) list("a2 a3 ",2,0)
list("a1 a3 ",2,0) list("a1 a2 ",2,0) list("a1 a2 a3 ",3,0)
```

Figure 3.4.4: List results

Now that the lists have been prepared, everything is in place for the sum capacity to be calculated. For every candidate on place, we want to store the new sum capacity produced, if we exchange the power token with its powered-on neighbor. This value of sum capacity can be stored at the value of the antenna token held by the place. It is important to note that a place can be a candidate from multiple adjacent powered places. We need an additional *value/4* fact, that will also save the adjacent antenna's name, from which we acquire the sum capacity calculated if the power token is exchanged.

```
m5:  value(Q,@c(A,P,L),TS; Q,@c(A,P,L),P,TS):-   candidateOn(A,P,TS),
                                                  holds(A,Q,TS), type(a,Q),
                                                  list(L,C,TS), C=#count{T:type(p,T)},
                                                  time(TS).
```

In rule m5, we use python function c with 3 parameters. The first parameter is the candidate place, next is the token donor adjacent place, and last is the list of all the powered-on places. We use C to get the list of the correct length, containing all these places. Python, will turn on place A and turn off place P, and calculate the sum capacity based on the exchange of power between A and P. The process of calculating the sum capacity using Python function c will be explained subsequently.

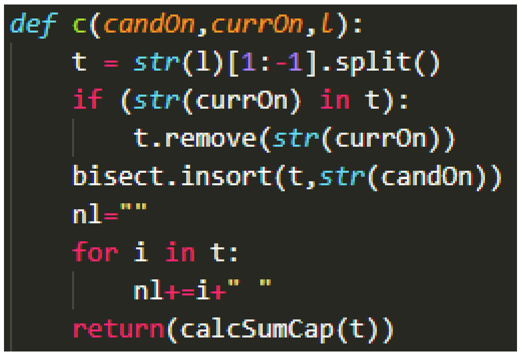
Similarly, rule m6 calculates the sum capacity for the currently powered on places. To limit the functions used, we use the same python function using P as the first and second parameter, where P is the place holding a power token.

```
m6: value(Q,@c(P,P,L),TS):-   holds(P,Q,TS), type(p,Q),
                              list(L,C,TS), C=#count{T:type(p,T)},
                              time(TS).
```

Finally, we give value 0 to antenna tokens that are not candidates to be turned on, to avoid extra Python calls that decrease the performance.

```
m7:  value(Q,0,TS) :-   not candidateOn(A,_,TS), holds(A,Q,TS), type(a,Q), time(TS).
```

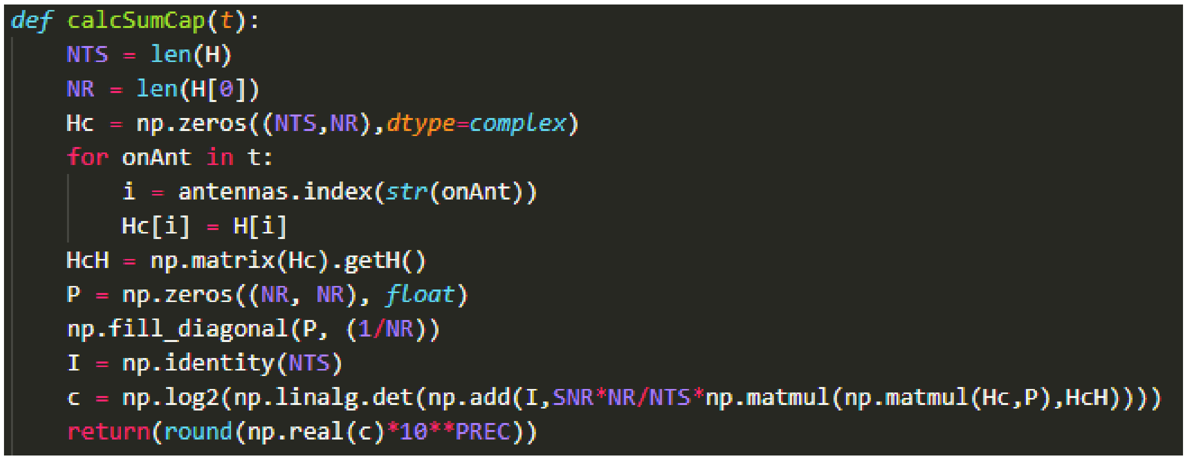
For the calculation of sum capacity, as mentioned previously, we used the function ***c*** of Python. Python code in ASP file is surrounded with #script (python) … #end. There we have as a global variable the table H and a parallel matrix named antennas, for the matching of each antenna name (names must be the same ones used in ASP code), with its corresponding row in H. We also have ρ defined in a global variable named SNR.



```
def c(candOn,currOn,l):
    t = str(l)[1:-1].split()
    if (str(currOn) in t):
        t.remove(str(currOn))
    bisect.insort(t,str(candOn))
    nl=""
    for i in t:
        nl+=i+" "
    return(calcSumCap(t))
```

Figure 3.4.5: Python function c for ASP

The function c, depicted in Figure 3.4.5, begins by translating the list *l* obtained by the fact from ASP code. The currently on antenna, neighbor of candidate on antenna, is shut down, and power is transferred to the candidate. Using the function *calcSumCap* shown in figure 3.4.6, sum capacity for sorted list ***t*** that contains the names of the new powered on antennas is calculated and returned.



```
def calcSumCap(t):
    NTS = len(H)
    NR = len(H[0])
    Hc = np.zeros((NTS,NR),dtype=complex)
    for onAnt in t:
        i = antennas.index(str(onAnt))
        Hc[i] = H[i]
    HcH = np.matrix(Hc).getH()
    P = np.zeros((NR, NR), float)
    np.fill_diagonal(P, (1/NR))
    I = np.identity(NTS)
    c = np.log2(np.linalg.det(np.add(I,SNR*NR/NTS*np.matmul(np.matmul(Hc,P),HcH))))
    return(round(np.real(c)*10**PREC))
```

Figure 3.4.6: Python function calcSumCap for ASP

The problem faced here with calculating and storing the sum capacity is that clingo cannot have complex nor real parameters, and thus we must turn the result into an integer value.

The imaginary part of the complex number is zero, so on the last line of the code, we multiply the real part of the result stored in variable c, with a power of 10. The power of 10 in variable PREC (10^{PREC}) symbolizes the precision we want to have. Notice here that the result after multiplying the real part with 10 to PREC power, our result must remain within the Integer range. For maximum integer values in matrix H (unrealistic scenario), the max PREC number we can have is 7, since the logarithmic operation at the end brings the max c value at around 62, and 62×10^7 does not escape the Integer range (620,000,000 < 2,147,483,647).

For more realistic values in table H, we can safely set PREC to bigger values. Figure 3.4.6 shows an example for PREC=15 and realistic complex values in H.

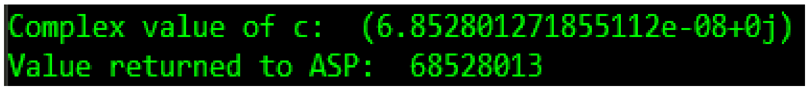
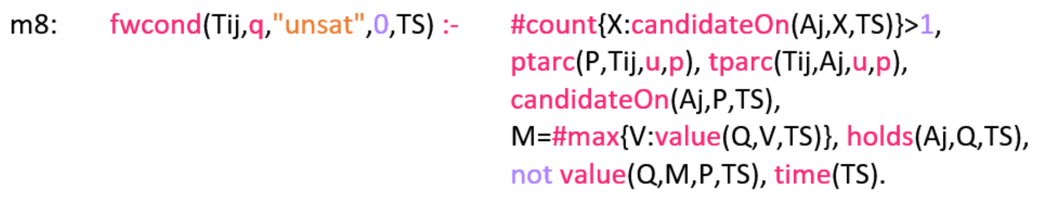

```
Complex value of c:  (6.852801271855112e-08+0j)
Value returned to ASP:  68528013
```

Figure 3.4.7: Sum Capacity values returned to ASP

The final step is to create the conditions and control how transitions fire in order to achieve an increase in total capacity every time. The general conditions coded in subchapter 3.3 are used to create conditions for our problem.

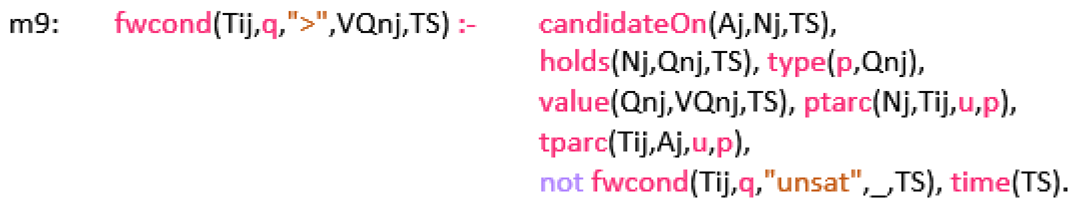
One antenna place can be a candidate on from multiple adjacent powered antennas. The algorithm in this scenario, can choose only the transition that will increase the sum capacity the most. With the following rule, we exclude all the transitions that do not offer the maximum local increase.



```
m8:     fwcond(Tij,q,"unsat",0,TS) :-     #count{X:candidateOn(Aj,X,TS)}>1,
                                          ptarc(P,Tij,u,p), tparc(Tij,Aj,u,p),
                                          candidateOn(Aj,P,TS),
                                          M=#max{V:value(Q,V,TS)}, holds(Aj,Q,TS),
                                          not value(Q,M,P,TS), time(TS).
```

If an antenna Aj is candidate-on from multiple places X, then set the transitions from all the places to Aj to “unsat”, except for the transition that brings the power token from the place that gives the maximum increase of sum capacity when its power is transferred to place Aj. So, this basically creates a condition that can never be satisfied by any token and shuts down the transitions that do not offer the maximum local increase, only for this time instance.

The most important condition for a transition to fire is when it increases the sum capacity.



```
m9:     fwcond(Tij,q,">",VQnj,TS) :-      candidateOn(Aj,Nj,TS),
                                          holds(Nj,Qnj,TS), type(p,Qnj),
                                          value(Qnj,VQnj,TS), ptarc(Nj,Tij,u,p),
                                          tparc(Tij,Aj,u,p),
                                          not fwcond(Tij,q,"unsat",_,TS), time(TS).
```

Rule m9 creates a condition for a transition Tij (Tij moves token from to Aj) to be enabled, only when a token can be assigned to variable q (q takes values ‘a’, antenna tokens) that its value is greater than VQnj. Value VQnj is the value of the token held at the adjacent power antenna. If the token in the candidate antenna has greater value than this one, it means that this condition is satisfied, and the antenna token held by the candidate place will be assigned to q.

There is also the option to make this algorithm greedy and choose the transition only if it offers the overall max increase in sum capacity on the net. In other words, for multiple power tokens and candidate on antennas, always choose the one with the max impact on sum capacity.

```
m10:    maxdiff(M,TS):- M=#max{V:value(Q,V,TS)},time(TS).
```

The fact *maxdiff/2* created from rule m10, holds the maximum increase M stored in value of a token for time TS.

```
m11:  fwcond(Tij,q,"=",M,TS) :-   candidateOn(Aj,Nj,TS), maxdiff(M,TS),
                                  ptarc(Nj,Tij,u,p), tparc(Tij,Aj,u,p),
                                  not fwcond(Tij,q,"unsat",_,TS).
```

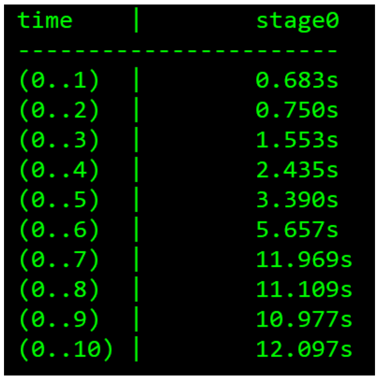
Then, with rule m11, we create conditions for every transition moving a token to a candidate place, to only fire when the value of the token in the candidate place is the one that offers the maximum overall increase.

3.5 Optimizations

The current solution is correct, but it does not provide the best possible performance. ASP and Python communication can be slow if Python is called a lot of times and that will cost in performance. Some optimizations are made in this subchapter, and the performance benefits are significant, while the program remains correct.

To inspect the performance gains of each optimization phase, a Python program on my personal computer is used to run a problem 10 times, incrementing by 1 the number of time instances in each run, starting with 2 time instances, time(0..1), and finishing with 11 time instances, time(0..10).

While the performance on small nets is fast, we will see subsequently that it grows exponentially for bigger nets with more variables. Worth noting is that most of the time is spend for the grounding phase and a much smaller fraction of the time is spent on the solving phase. That gives a hint that reducing the number of constants in the problem and the number of facts created are the bottleneck of the performance, and thus we should focus on them.



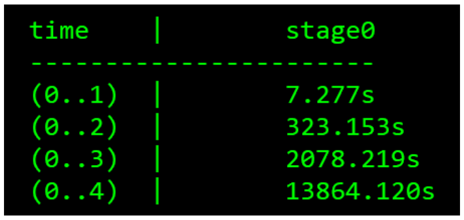
```
time     |          stage0
-----------------------------
(0..1)   |          0.683s
(0..2)   |          0.750s
(0..3)   |          1.553s
(0..4)   |          2.435s
(0..5)   |          3.390s
(0..6)   |          5.657s
(0..7)   |          11.969s
(0..8)   |          11.109s
(0..9)   |          10.977s
(0..10)  |          12.097s
```

Figure 3.5.1: Solving Time for small net

Figure 3.5.1 shows the solving time, which is the time needed for grounding the variables and producing an answer set, for a net with 5 antennas, with 6 connections between them and 2 power tokens in total.

From now on, a problem with 10 antennas and 4 power tokens will be used to compare the results after each optimization. There are 12 connections between the antennas, and we run the non-greedy approach. The power tokens are placed statically in the same places every time.

Below are the initial results for the current stage without optimizations:



```
time     |          stage0
-----------------------------
(0..1)   |          7.277s
(0..2)   |          323.153s
(0..3)   |          2078.219s
(0..4)   |          13864.120s
```

Figure 3.5.2: Results optimization stage 0

Results in Figure 3.5.2 show that the initial performance of the algorithm is very slow for the more complex net and it is unable to run past 5 time instances.

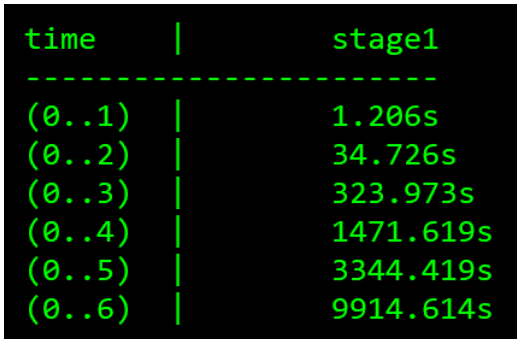
Stage 1:

As mentioned previously, due to some neighborhood communication problems, the algorithm proposed in [7] can run only for one neighborhood. So, the first optimization is to remove the neighborhood places and bond arcs, because distinguishing between the different neighborhoods is unnecessary because we can only have one. Instead of bonding the tokens of the powered-on antennas, places simply now exchange their power and antenna tokens.

The arcs remaining are created by the following rules.

```
ptarc(Ai,Tij,u,p ; Aj,Tji,u,p ; Aj,Tij,q,a ; Ai,Tji,q,a) :- construct(Ai,Aj,Tij,Tji,Mk).
tparc(Tji,Ai,u,p ; Tij,Aj,u,p ; Tji,Aj,q,a ; Tij,Ai,q,a) :- construct(Ai,Aj,Tij,Tji,Mk).
```

With this optimization not only we decreased the number of arcs and places but also the number of tokens and variables. When we run the problem with the new optimizations, we can see that the performance has greatly improved, and we can now run for more time instances, without running out of memory and crashing.



```
time     |         stage1
------------------------
(0..1)   |         1.206s
(0..2)   |         34.726s
(0..3)   |         323.973s
(0..4)   |         1471.619s
(0..5)   |         3344.419s
(0..6)   |         9914.614s
```

Figure 3.5.1: Results after stage 1 optimizations

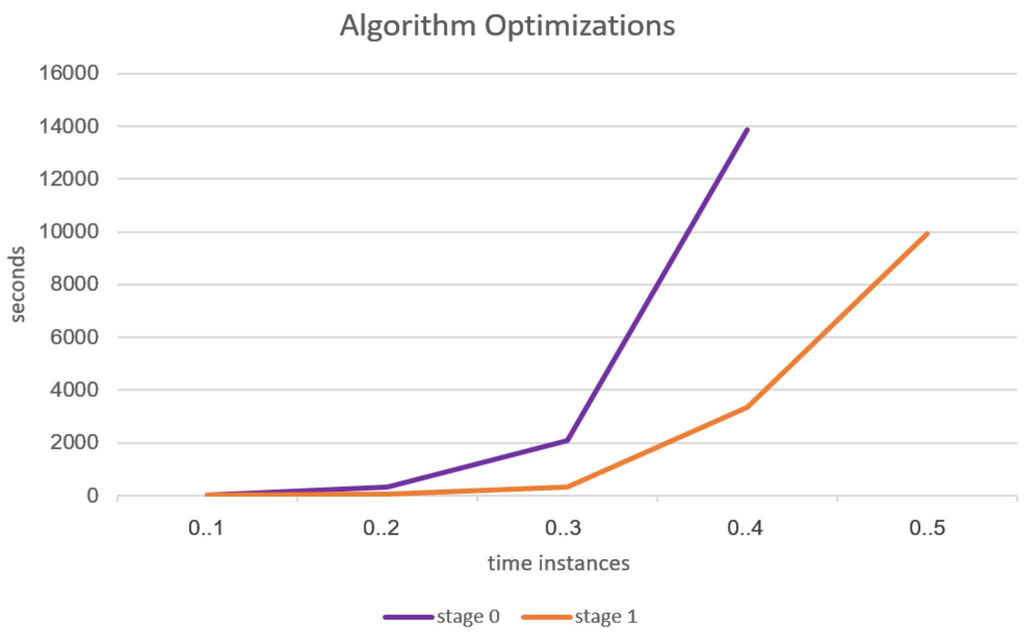
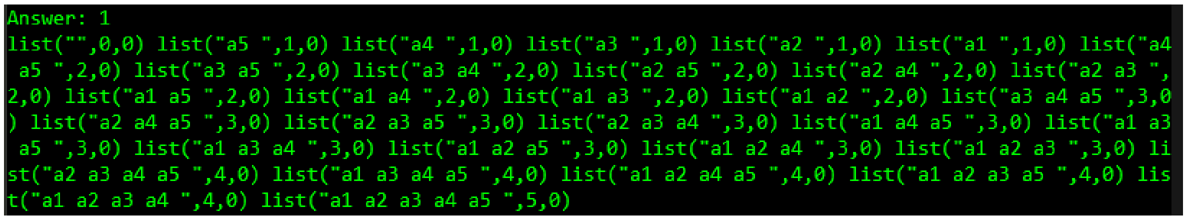


Figure 3.5.2: Graph comparing optimization stage 0 vs stage 1.

<u>*Stage 2:*</u>

Another approach for optimization is to reduce the number of facts we create. Optimizing the custom lists created can reduce the number of facts created. Because lists are created with the help of Python, we also significantly reduce the number of calls made to Python, so this change has a double benefit in terms of performance. Without the alphabetical condition in Python, creating a list of five components would require $1 + 5 + 5^2 + 5^3 + 5^4 + 5^5 = 3906$ facts because the rule would generate all possible combinations of the powered-on places names.

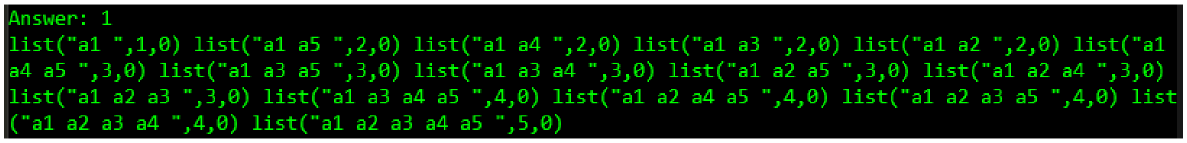
Currently, for five powered-on places, our program with Python using alphabetical order, will create 32 different lists/facts.



```
Answer: 1
list("",0,0) list("a5 ",1,0) list("a4 ",1,0) list("a3 ",1,0) list("a2 ",1,0) list("a1 ",1,0) list("a4
 a5 ",2,0) list("a3 a5 ",2,0) list("a3 a4 ",2,0) list("a2 a5 ",2,0) list("a2 a4 ",2,0) list("a2 a3 ",
2,0) list("a1 a5 ",2,0) list("a1 a4 ",2,0) list("a1 a3 ",2,0) list("a1 a2 ",2,0) list("a3 a4 a5 ",3,0
) list("a2 a4 a5 ",3,0) list("a2 a3 a5 ",3,0) list("a2 a3 a4 ",3,0) list("a1 a4 a5 ",3,0) list("a1 a3
 a5 ",3,0) list("a1 a3 a4 ",3,0) list("a1 a2 a5 ",3,0) list("a1 a2 a4 ",3,0) list("a1 a2 a3 ",3,0) li
st("a2 a3 a4 a5 ",4,0) list("a1 a3 a4 a5 ",4,0) list("a1 a2 a4 a5 ",4,0) list("a1 a2 a3 a5 ",4,0) lis
t("a1 a2 a3 a4 ",4,0) list("a1 a2 a3 a4 a5 ",5,0)
```

Figure 3.5.3: List optimisation starting stage

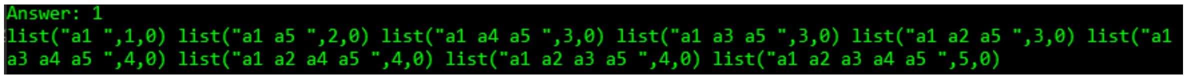
Instead of starting with an empty list, we can start with a list of one antenna, containing the antenna that holds the alphabetically first power token. This way, we eliminate half the facts in total for this scenario, left with 16 facts.



```
Answer: 1
list("a1 ",1,0) list("a1 a5 ",2,0) list("a1 a4 ",2,0) list("a1 a3 ",2,0) list("a1 a2 ",2,0) list("a1
a4 a5 ",3,0) list("a1 a3 a5 ",3,0) list("a1 a3 a4 ",3,0) list("a1 a2 a5 ",3,0) list("a1 a2 a4 ",3,0)
list("a1 a2 a3 ",3,0) list("a1 a3 a4 a5 ",4,0) list("a1 a2 a4 a5 ",4,0) list("a1 a2 a3 a5 ",4,0) list
("a1 a2 a3 a4 ",4,0) list("a1 a2 a3 a4 a5 ",5,0)
```

Figure 3.5.4: List optimisation optimisations 1

There is also the ability to select only the last power token, alphabetically sorted, and create just one list of length two, list with a1 and a5 in this case.

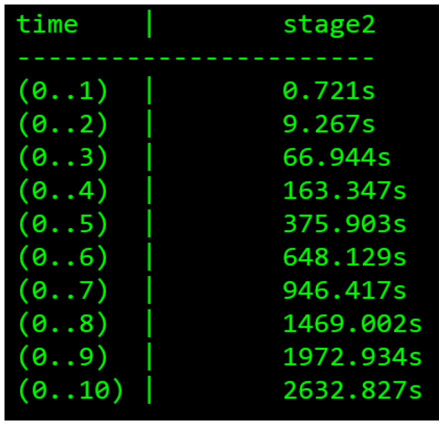


```
Answer: 1
list("a1 ",1,0) list("a1 a5 ",2,0) list("a1 a4 a5 ",3,0) list("a1 a3 a5 ",3,0) list("a1 a2 a5 ",3,0) list("a1
a3 a4 a5 ",4,0) list("a1 a2 a4 a5 ",4,0) list("a1 a2 a3 a5 ",4,0) list("a1 a2 a3 a4 a5 ",5,0)
```

Figure 3.5.5: List optimisation optimisations 2

That leaves us with only 9 facts. The optimal would be 5 facts, starting at length one, and appending one antenna every time until length five. However, adding more rules to bring down the number of lists created is not beneficial at this point. The following, is the final code for creating the custom lists in ASP using Python, replacing rules m2, m3 and m4 shown in subchapter 3.4.

```
list(@genList("",A),TS):- holds(A,Q,TS), Q=#min{O:type(p,O)},time(TS).
list(@genList(L ,A),TS):- holds(A,Q,TS), Q=#max{O:type(p,O)},list(L,1,TS),time(TS).
list(@genList(L, A),TS):- holds(A,Q,TS), type(p,Q), list(L,S,TS), S>1.
list(L,@getSize(L),TS) :- list(L,TS),time(TS).
```



```
time      |        stage2
-----------------------
(0..1)  |        0.721s
(0..2)  |        9.267s
(0..3)  |        66.944s
(0..4)  |        163.347s
(0..5)  |        375.903s
(0..6)  |        648.129s
(0..7)  |        946.417s
(0..8)  |        1469.002s
(0..9)  |        1972.934s
(0..10) |        2632.827s
```

Figure 3.5.6: Results after stage 2 optimizations

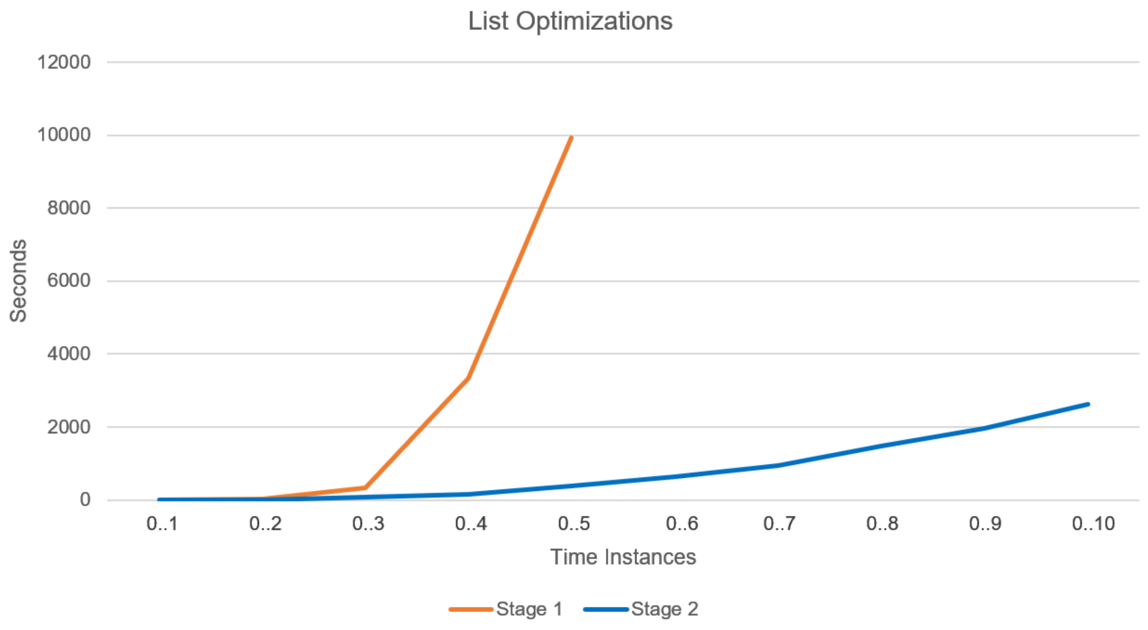


Figure 3.5.7: Graph comparing optimization stage1 vs stage 2.

Stage 3:

Stage three is an attempt to reduce the number of times Python is called by ASP. Because of the way our value rule has been structured thus far, variable L can be ground to any possible list. As a result, Python is called unnecessarily to calculate the sum capacity of an incorrect list that will never be used. For this reason, a new fact *corrlist/2* is created, that holds the single correct list for each time instance.

```
corrlist(L,TS):- list(L,C,TS), C=#count{T:type(p,T)}.
```

We replace *list(L,C,TS), C=#count{T:type(p,T)}* with *corrlist(L,TS)* in the rules that calculate the value and now the new L can only be grounded to the correct lists of each time instance only. Furthermore, rather than calling c twice to create facts *value/3* and *value/4* like is done in rule m5, we call python once and then use an extra rule *m5b* that uses *value/4* to create *value/3*.

old rule:

```
m5:
    value(Q,@c(A,P,L),TS; Q,@c(A,P,L),P,TS):-   candidateOn(A,P,TS),
                                                holds(A,Q,TS), type(a,Q),
                                                list(L,C,TS), C=#count{T:type(p,T)},
                                                time(TS).
```

new rules:

```
m5(updated):
    value(Q,@c(A,P,L),P,TS):-  candidateOn(A,P,TS),
                               holds(A,Q,TS), type(a,Q),
                               corrlist(L,TS), time(TS).

m5b:
    value(Q,V,TS):-value(Q,V,P,TS).
```

For power tokens, the value they have depends on the transition that fires, and is the same as the value of the antenna token used to execute a transition. Thus, if a transition fires, we update the values of the power tokens to the old value of the antenna token, and we only call Python for power tokens once at the earliest time instance.

old rule:

```
m6:   value(Q,@c(P,P,L),TS):-   holds(P,Q,TS), type(p,Q),
                                list(L,C,TS), C=#count{T:type(p,T)},
                                time(TS).
```

new rules:

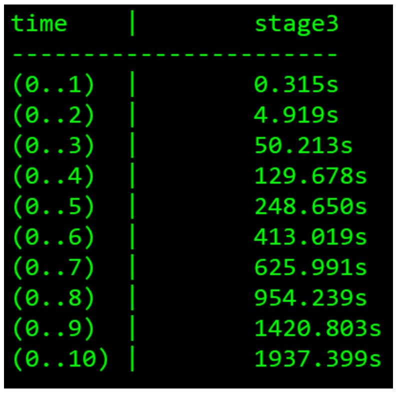
m6(updated):

```
value(Q,@c(P,P,L),TS) :-   holds(P,Q,TS), type(p,Q),
                           corrlist(L,TS),
                           time(TS),TS=#min{X:time(X)}.
```

m6b:

```
value(Qp,V,TS):-   type(p,Qp), TS>#min{X:time(X)}, time(TS), fires(T,TS-1),
                   assigned(P,T,q,A,TS-1), V=#max{Z:value(A,Z,TS-1)}.f
```

The value of power tokens after the minimum time instance, get the maximum value of the token of type ‘a’ that was used to fire a transition. That value is equal to the next sum capacity of the net. We can be sure the maximum value will be used, because locally we only enable the transition with the maximum positive impact on the sum-capacity. All the other transitions that offer less or negative gains, will have an unsatisfiable condition that will not let them fire, as we seen with rule *m8* in the previous subchapter.



```
time     |        stage3
-----------------------
(0..1)   |        0.315s
(0..2)   |        4.919s
(0..3)   |        50.213s
(0..4)   |        129.678s
(0..5)   |        248.650s
(0..6)   |        413.019s
(0..7)   |        625.991s
(0..8)   |        954.239s
(0..9)   |        1420.803s
(0..10)  |        1937.399s
```

Figure 3.5.8: Results after stage 3 optimizations

Those changes contribute to the decrease of Python calls from ASP, and that positively impacts the performance.

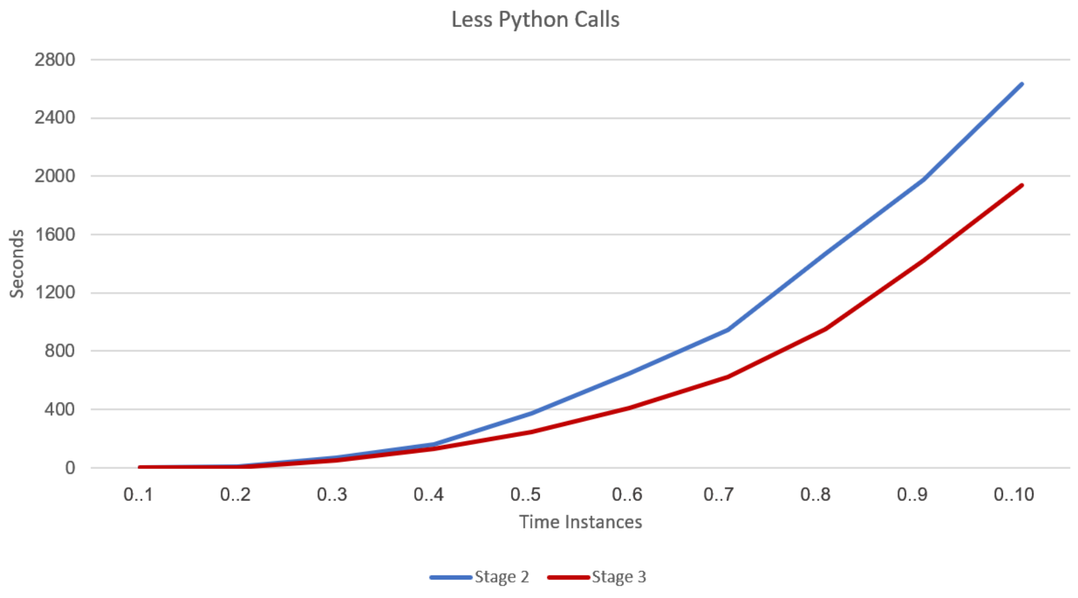


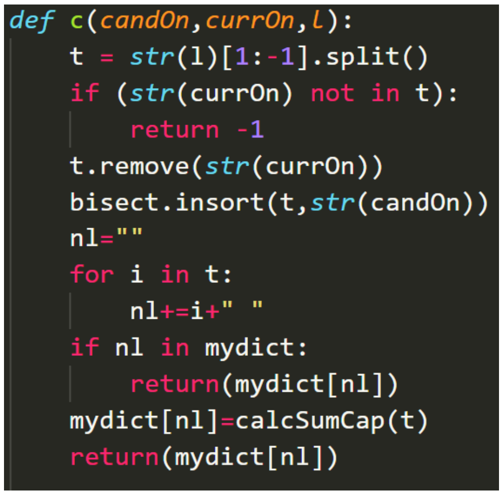
Figure 3.5.9: Graph comparing optimization stage 2 vs stage 3.

Stage 4:

In this stage we optimize the python script, limiting the amount of times sum capacity is calculated. We can save the results for a combination of turned-on Antennas in a HashMap using the list as the key, and before we calculate it check if it was previously calculated.

The second improvement in this stage is to stop calculating the sum capacity if the *currOn* antenna which is specified as parameter to function c, is not found in the list of turned-on antennas. Normally the currently on (*currOn*) antenna should be found in the list of turned-on antennas, but since all the different combinations are grounded, antenna place P, which is the parameter sent as the *currOn* antenna, can get any possible place constant value. This can be fought in a similar manner with the lists in stage 3 and the *corrlist/2*, but is proven to be worse that way, due to the large number of new rules and facts needed to implement a similar idea.

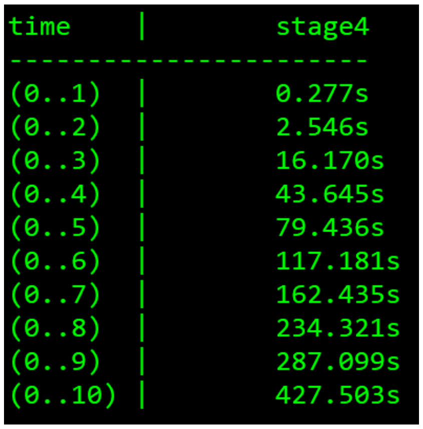
The solution that gave better results in this case, was to just return any value (-1 is returned below) and not calculate the sum capacity, because we are sure the value created will not be included in any answer set as the rest of the atoms in the body will not match to true.



```
def c(candOn,currOn,l):
    t = str(l)[1:-1].split()
    if (str(currOn) not in t):
        return -1
    t.remove(str(currOn))
    bisect.insort(t,str(candOn))
    nl=""
    for i in t:
        nl+=i+" "
    if nl in mydict:
        return(mydict[nl])
    mydict[nl]=calcSumCap(t)
    return(mydict[nl])
```

Figure 3.5.10: Updated function c for optimisations

A global variable ***mydict={ }*** must also be declared. The hit ratio for this problem is around 98%, so the number of times sum capacity is calculated is greatly reduced.



```
time     |        stage4
-----------------------
(0..1)   |        0.277s
(0..2)   |        2.546s
(0..3)   |        16.170s
(0..4)   |        43.645s
(0..5)   |        79.436s
(0..6)   |        117.181s
(0..7)   |        162.435s
(0..8)   |        234.321s
(0..9)   |        287.099s
(0..10)  |        427.503s
```

Figure 3.5.11: Results after stage 4 optimizations

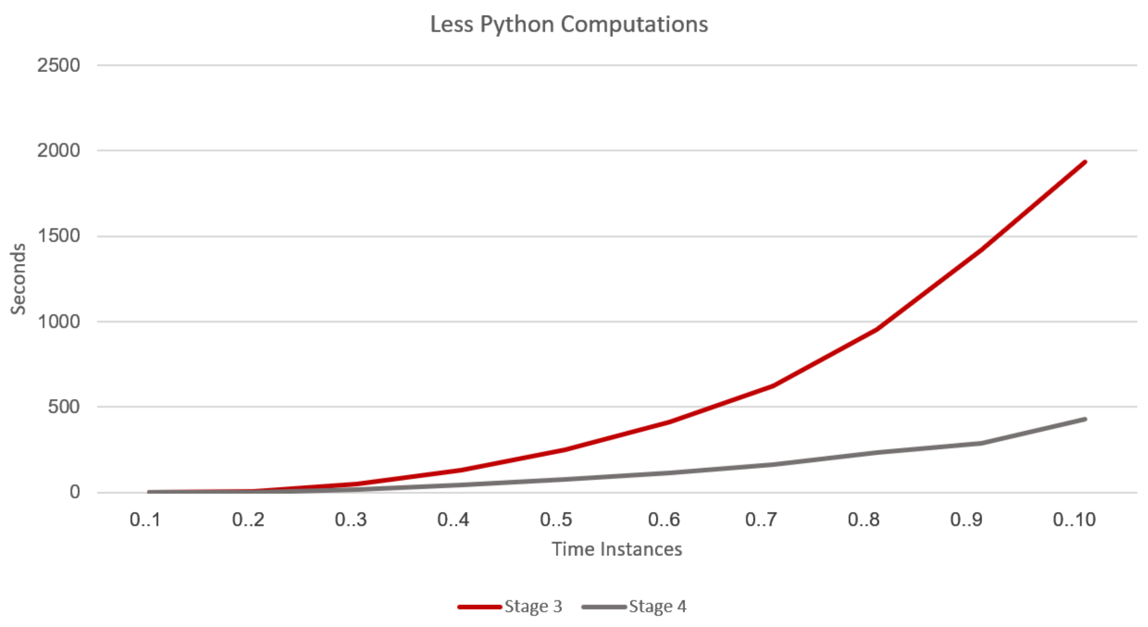


Figure 3.5.12: Graph comparing optimization stage 3 vs stage 4.

Stage 5:

Another idea for achieving better performance on ASP language is to limit the number of parameters used in facts. One extra parameter can exponentially increase the number of grounded facts.

Take as an example the code below:

```
temp(A,B,C,D,E,F) :- a(A), b(B), c(C), d(D), e(E), f(F).

a(0..9).  b(0..9).  c(0..9).  d(0..9).  e(0..9).  f(0..9).

goal:- temp(0,0,0,0,0,0).

:- not goal.
```

Even though there is only one correct answer set, with the fact *temp(0,0,0,0,0,0)*, the grounder will ground every variable in the rule above with all the possible constant values it can take, and create all the possible combinations of facts *temp/6*. It will only then go to the solving process and start checking for the answer sets. For this example, there will be 10^6 possible facts created during grounding. Removing one parameter from the above rule, will result in 10^5 grounded facts, **90%** less than with six parameters.

Quite a big amount of time is trimmed from the results, if we remove the conditions for the problem of the Antenna Selection. That leads to the conclusion that conditions should be optimized to get better results. Currently we count how many conditions of a variable in a transition a token satisfies marking them as satisfied using the fact *fwsatcond/6* (rule r37). If it satisfies the same number of conditions as the total number of conditions, we say it satisfies all the conditions with the fact *satAllFwCond/4*.

Instead of creating a fact for every condition satisfied, we create facts for every variable condition that is not satisfied from a token in a transition. If there are no unsatisfied conditions, then we say that the token satisfies all the conditions with the fact *satAllFwCond/4*. That brings the number of parameters down to 4 from 6.

Rule r37 changes to:

```
r37(updated):     fwUnsatcond(T,V,Q,TS) :-     fwcond(T,V,">",X,TS),
                                               ptValTypePair(P,T,V,TY),
                                               has(P,Q,TS), type(TY,Q),
                                               A=#max{M:value(Q,M,TS)}, not A > X,
                                               time(TS).
```

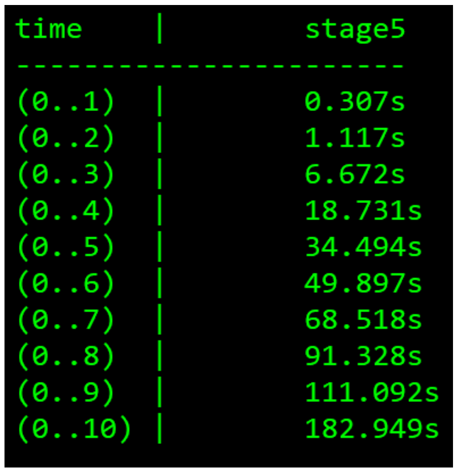
There are two changes in the body of the rule r37. The first one is the addition of the negation (*not*) when comparing A and X, to now check for non-satisfiability of the condition. The second one is that we only use the maximum value of a token, that can have multiple values if it has multiple powered-on neighbors. We are only interested in the maximum values because only the greatest increase in sum-capacity may fire locally. Rule r37u cancels all other transitions from the same location that offer less total capacity gains. Similarly, we apply these changes of r37 to the rest of the comparing symbols (>,=.!=…).

```
r37u:   fwUnsatcond(T,V,Q,TS) :-   fwcond(T,V,"unsat",X,TS),
                                   ptValTypePair(P,T,V,TY),
                                   has(P,Q,TS), type(TY,Q),
                                   value(Q,M,TS), time(TS).
```

Now a token Q satisfies a variable V of a transition T when there are no unsatisfied conditions.

```
satAllFwCond(T,V,Q,TS):-   not fwUnsatcond(T,V,Q,TS), place(P),
                           ptValTypePair(P,T,V,TY), has(P,Q,TS), type(TY,Q), time(TS).
```

Next are the performance results for the changes mentioned above:



```
time     |        stage5
------------------------
(0..1)   |        0.307s
(0..2)   |        1.117s
(0..3)   |        6.672s
(0..4)   |        18.731s
(0..5)   |        34.494s
(0..6)   |        49.897s
(0..7)   |        68.518s
(0..8)   |        91.328s
(0..9)   |        111.092s
(0..10)  |        182.949s
```

Figure 3.5.13: Results after stage 5 optimizations

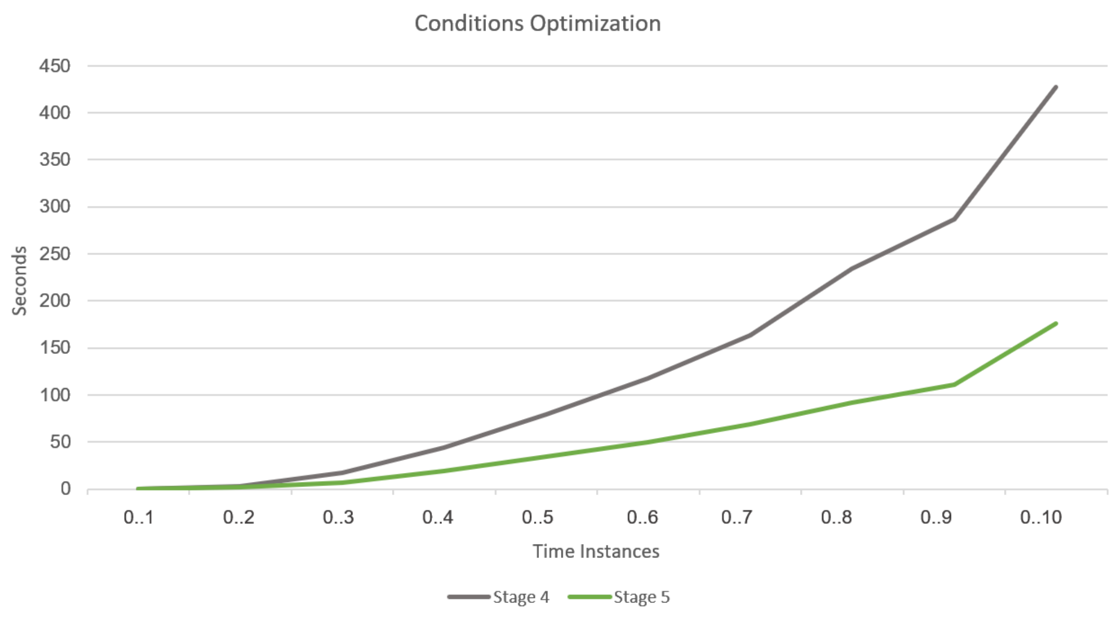


Figure 3.5.14: Graph comparing optimization stage 4 vs stage 5.

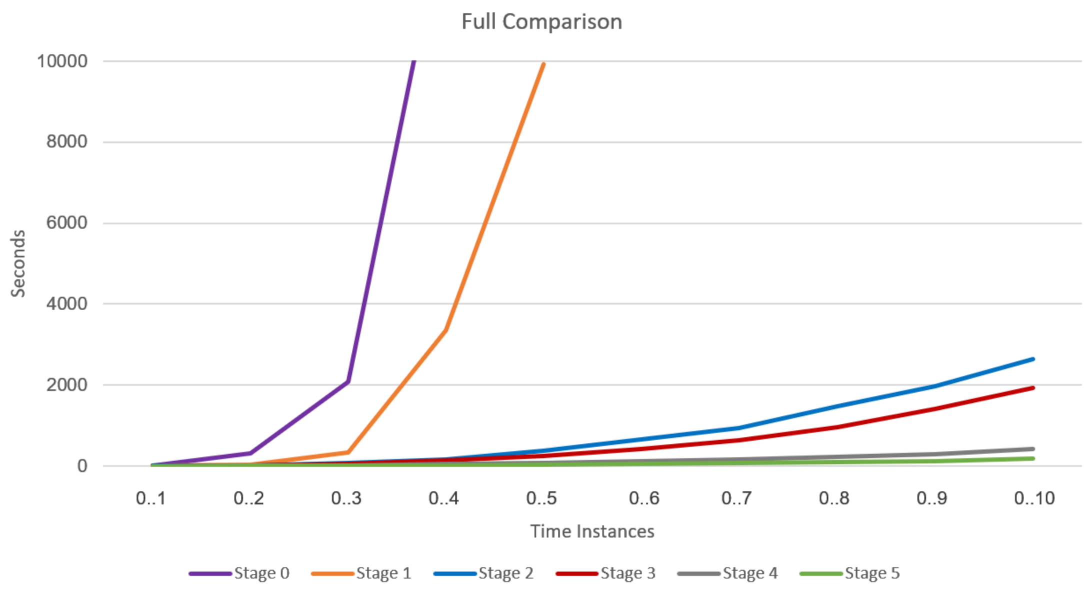


Figure 3.5.15: Graph comparing all optimization stages.

Concluding with this chapter, we see that after the optimizations, the performance of the algorithm has been dramatically improved while still producing the same correct results as before. The machine used for running these experiments is an outdated laptop and much better results are expected on more capable machines.

Chapter 4

Case Study

4.1 Antenna Selection Case Study 88
4.2 Reachability 94
4.3 Greedy Algorithm Comparison 96

4.1 Antenna Selection Case Study

We will specify a problem to the ASP program and then execute the algorithm to obtain the results in this subchapter. The following example, in Figure 4.1.1, is a case of an Antenna Selection problem for a neighborhood. The lines between the antenna places indicate that the two antennas connected are adjucent and thus we can exchange power between them if the criteria are met. Every line is an abstract representation of Figure 4.1.2, which shows the arcs and transitions between two adjucent antennas. If we replace each line with the corresponding schema like in Figure 4.1.2, we will get the real image of how the problem is described to ASP and which transitions are created. There are ten antennas, the number of users is four and the SNR is 0.5.

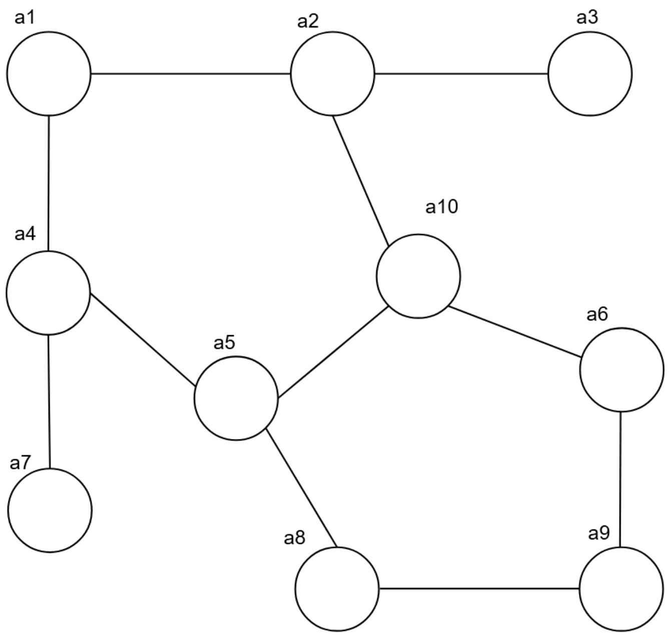


Figure 4.1.1: Antenna Selection Case

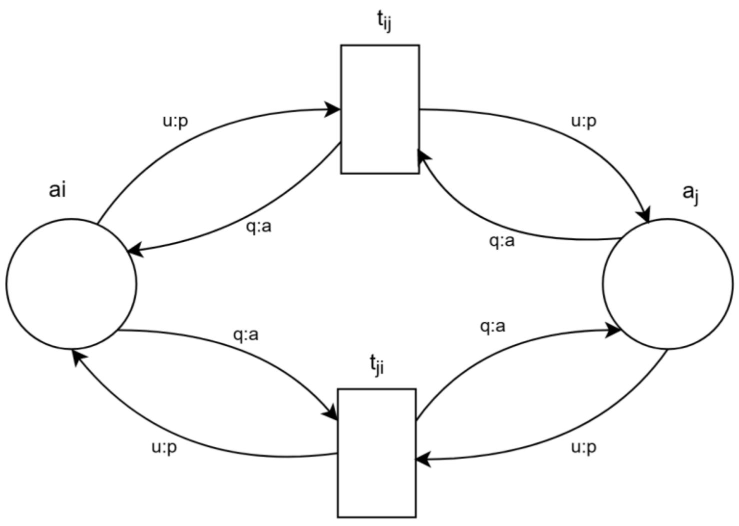


Figure 4.1.2: Adjacent Antennas real schema

To describe the problem in ASP, we use the *construct/5* facts.

construct(a1,a2,t12,t21,m).
construct(a1,a4,t14,t41,m).
construct(a1,a5,t15,t51,m).
construct(a2,a3,t23,t32,m).
construct(a3,a6,t36,t63,m).
construct(a4,a7,t47,t74,m).
construct(a5,a7,t57,t75,m).
construct(a5,a9,t59,t95,m).
construct(a6,a9,t69,t96,m).
construct(a7,a8,t78,t87,m).
construct(a8,a9,t89,t98,m).

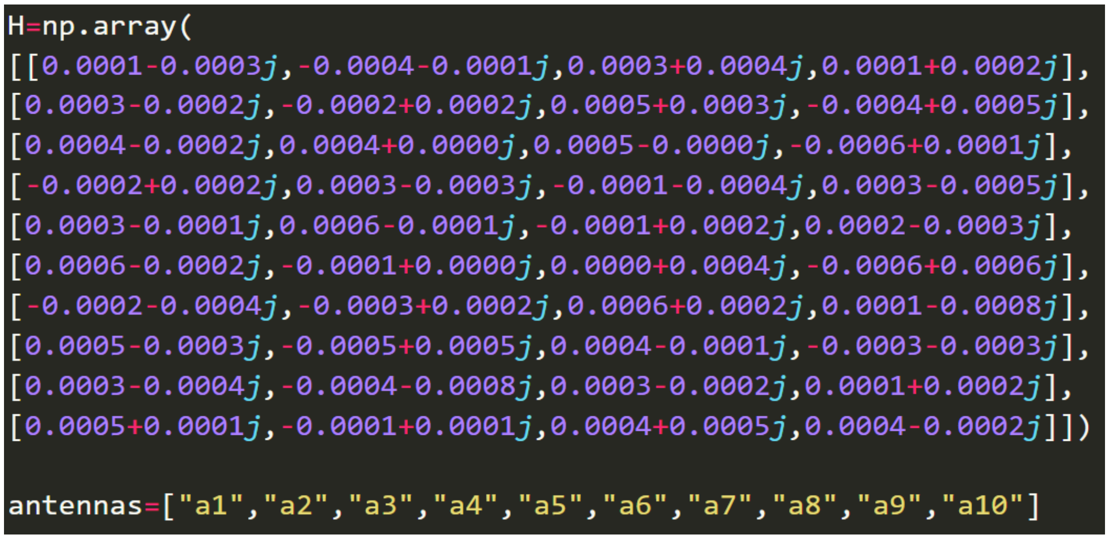
We need eleven *construct/5* facts to show which antennas are adjacent in our neighborhood m. The rest of the facts for the arcs and transitions shown in Figure 4.1.2 between every two adjacent antennas will be generated by the rules having construct/5 in their body.

For the calculation of sum capacity, we need to declare two parallel arrays in the Python script part inside the ASP program. The first array ***antennas*** will include the names of the antennas as given in ASP, and the array ***H***, parallel to the other array, will contain the values of each antenna in the array ***antennas.***

The values for each antenna are shown below:

a1:	0.0001-0.0003i	-0.0004-0.0001i	0.0003+0.0004i	0.0001+0.0002i
a2:	0.0003-0.0002i	-0.0002+0.0002i	0.0005+0.0003i	-0.0004+0.0005i
a3:	0.0004-0.0002i	0.0004+0.0000i	0.0005-0.0000i	-0.0006+0.0001i
a4:	-0.0002+0.0002i	0.0003-0.0003i	-0.0001-0.0004i	0.0003-0.0005i
a5:	0.0003-0.0001i	0.0006-0.0001i	-0.0001+0.0002i	0.0002-0.0003i
a6:	0.0006-0.0002i	-0.0001+0.0000i	0.0000+0.0004i	-0.0006+0.0006i
a7:	-0.0002-0.0004i	-0.0003+0.0002i	0.0006+0.0002i	0.0001-0.0008i
a8:	0.0005-0.0003i	-0.0005+0.0005i	0.0004-0.0001i	-0.0003-0.0003i
a9:	0.0003-0.0004i	-0.0004-0.0008i	0.0003-0.0002i	0.0001+0.0002i
a10:	0.0005+0.0001i	-0.0001+0.0001i	0.0004+0.0005i	0.0004-0.0002i

The array ***H*** has dimensions $N_T \, x \, N_R$ where $\mathrm{N_T}$ is the number of antennas and $\mathrm{N_R}$ is the number of users.



```
H=np.array(
[[0.0001-0.0003j,-0.0004-0.0001j,0.0003+0.0004j,0.0001+0.0002j],
[0.0003-0.0002j,-0.0002+0.0002j,0.0005+0.0003j,-0.0004+0.0005j],
[0.0004-0.0002j,0.0004+0.0000j,0.0005-0.0000j,-0.0006+0.0001j],
[-0.0002+0.0002j,0.0003-0.0003j,-0.0001-0.0004j,0.0003-0.0005j],
[0.0003-0.0001j,0.0006-0.0001j,-0.0001+0.0002j,0.0002-0.0003j],
[0.0006-0.0002j,-0.0001+0.0000j,0.0000+0.0004j,-0.0006+0.0006j],
[-0.0002-0.0004j,-0.0003+0.0002j,0.0006+0.0002j,0.0001-0.0008j],
[0.0005-0.0003j,-0.0005+0.0005j,0.0004-0.0001j,-0.0003-0.0003j],
[0.0003-0.0004j,-0.0004-0.0008j,0.0003-0.0002j,0.0001+0.0002j],
[0.0005+0.0001j,-0.0001+0.0001j,0.0004+0.0005j,0.0004-0.0002j]])

antennas=["a1","a2","a3","a4","a5","a6","a7","a8","a9","a10"]
```

Figure 4.1.3: Python Arrays for case study

Figure 4.1.3 shows the arrays for the values of our problem. Besides the two arrays, global variable SNR is set to 0.5 and we initialize the dictionary as seen in the next lines. Precision is set to 15.

```
SNR = 0.5
PREC=15
mydict = {}
```

We are using three power tokens p1, p2 and p3 and we assume they are initially positioned at places a1, a5 and a10 respectively.

type(p,p1). *holds(a1,p1,0).*
type(p,p2). *holds(a5,p2,0).*
type(p,p3). *holds(a10,p3,0).*

After running the program, we obtained the first solution with the following results: ***fires(t14,0). fires(t58,1). fires(t47,2). fires(t102,3). fires(t89,4). fires(t92,5). fires(t23,6).***

Figure 4.1.4 Demonstrates the visual result after running the program.

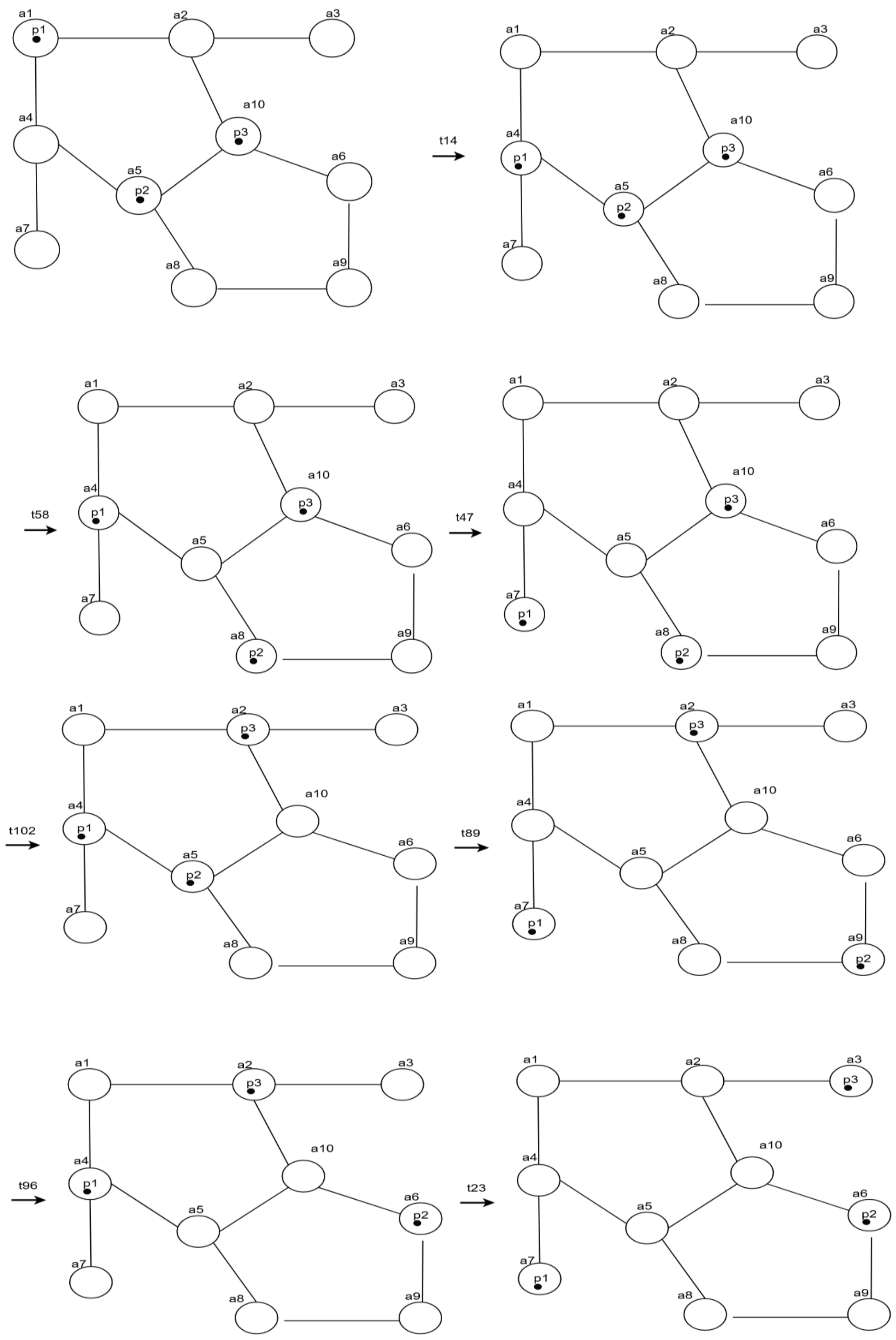
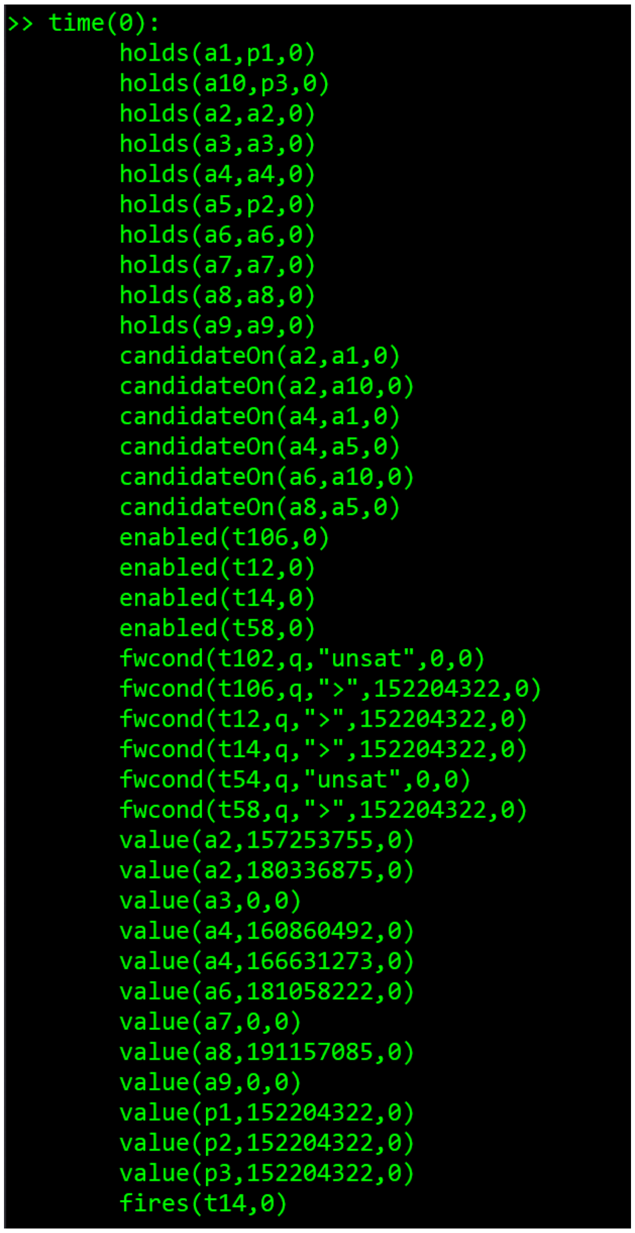


Figure 4.1.4: Case Study visual run results.

Following are the clingo results for *time(0).*



```
>> time(0):
        holds(a1,p1,0)
        holds(a10,p3,0)
        holds(a2,a2,0)
        holds(a3,a3,0)
        holds(a4,a4,0)
        holds(a5,p2,0)
        holds(a6,a6,0)
        holds(a7,a7,0)
        holds(a8,a8,0)
        holds(a9,a9,0)
        candidateOn(a2,a1,0)
        candidateOn(a2,a10,0)
        candidateOn(a4,a1,0)
        candidateOn(a4,a5,0)
        candidateOn(a6,a10,0)
        candidateOn(a8,a5,0)
        enabled(t106,0)
        enabled(t12,0)
        enabled(t14,0)
        enabled(t58,0)
        fwcond(t102,q,"unsat",0,0)
        fwcond(t106,q,">",152204322,0)
        fwcond(t12,q,">",152204322,0)
        fwcond(t14,q,">",152204322,0)
        fwcond(t54,q,"unsat",0,0)
        fwcond(t58,q,">",152204322,0)
        value(a2,157253755,0)
        value(a2,180336875,0)
        value(a3,0,0)
        value(a4,160860492,0)
        value(a4,166631273,0)
        value(a6,181058222,0)
        value(a7,0,0)
        value(a8,191157085,0)
        value(a9,0,0)
        value(p1,152204322,0)
        value(p2,152204322,0)
        value(p3,152204322,0)
        fires(t14,0)
```

Figure 4.1.5: case study clingo results time(0).

The enabled transitions at time 0 are t106, t12, t14 and t58. Transitions t54 and t102 have an ***unsat*** forward condition *fwcond/5*, as places a4 and a2 are candidate on from 2 places each, and only the one with the maximum increase can be chosen. In the end of time(0), we can see that the transition t14 is chosen to fire, non-deterministically, in this answer set.

The net converges within seven time instances in this solution, with final sum capacity of ***2,63291834 x 10⁻⁷*** and antennas a3, a6 and a7 being powered-on. To allow the program to run until it converges, we use the following rule:

```
time(0).
time(TS+1):- trans(T), enabled(T,TS), time(TS), TS<20.
```

If the current time instance has a transition enabled, the above rule will create the next time instance.

The results of this answer set are sub-optimal. The optimal sum capacity is given when enabling antennas a6, a7 and a9 and it is equal to $\boldsymbol{2{,}81325521 \times 10^{-7}}$.

The firings that result in this sum capacity are given from a different answer set and this will be further discussed in subchapter 4.3 where the non-greedy algorithm is being compared to the greedy one.

4.2 Reachability

In a Petri net NP with initial marking M0, $<NP, M_0>$, reachability is the question of determining whether a marking Mn is reachable. A marking M_n is said to be reachable in $<NP,M_0>$ if M0 can be transformed to Mn after a sequence of firings, $\sigma=M_0\ T_1\ M_1\ T_2\ M_2\ T_3 \ldots T_N\ M_n$. $L(N_P,M_0)$ denotes the set of all markings that are reachable by $<N_P,M_0>$. Although it takes exponential space and time to verify [14], the issue of reachability has been shown to be decidable [6][17].

We can test if a specific state of the program is reachable, by restricting the solutions that do not contain it. We can restrict these solutions using constrain rules and deadlock solutions that do not contain our desired state. If all the solutions are deadlocked and no answer is given, then we get to the conclusion that our desired state was not reachable.

On the other hand, if the state is reachable, the solutions containing the given state will be printed.

Any fact and combination of facts can be used to see if there is a solution containing them, or in other words to see if they are reachable. For example, if we want to test whether a transition t1 will fire forward after time instance 3, we can use the following set of rules:

goal :- fires(t1,TS), TS>3.

:- not goal.

In this example, we deadlock all the solutions where transition t1 does not fire after time instance three. If the state is reachable, then the output of clingo will be SATISFIABLE and all the answer sets will be printed, and will contain the fact fires(t1,TS), with TS being greater than three.

We can also get more creative with more atoms in the rule's body and multiple goals:

s1: *goal1 :- holds(P1,Q1,TS), type(cat,Q1), not holds(P1,Q2,TS), type(dog,Q2), place(P1), time(TS).*

s2: *goal2 :- holds(zoo,Q3,_), type(lion,Q3), place(zoo).*

State s1 sets a goal to find a solution where any place holds a token of type cat and a token of type dog at the same time. Similarly, state s2 checks the reachability of a state where place zoo holds a token of type lion. We could combine the two rules into one rule, but this rule separation gives us the opportunity to check if any of the two states is reachable, using one constraint rule for both.

:- not goal1, not goal2.

This constraint rule will block solutions where neither of the 2 desired states, s1 and s2, is reachable. For our example with the above constraint, we check if any place can simultaneously have a cat and a dog OR if the zoo has a lion at some point. If any of the two or both are reachable, then solutions will be given from the solver.

We can still constrain the solutions via using 2 separate constraint rules, one for each goal, and check if they can appear together:

:- not goal1.

:- not goal2.

It is enough for either of the states to be unreachable for our solver to produce no solutions and give UNSATISFIABLE as the answer. With the above constraint, we check the reachability of a state where a place has a cat and a dog at the same time AND if the zoo has a lion at any time instance.

Combining facts and different rules, we can check the reachability of any state. We are not limited to which atoms we can use in the constraint rule's body. Any atom or combination of atoms can be used to check for their reachability.

For the problem of Antenna Selection, we can use a rule like the following, and check whether a higher value of sum capacity is reachable in our net.

goal:- value(Q,X,_), X>1000000, type(p,Q).

:- not goal.

4.3 Greedy Algorithm Comparison

The results in subchapter 4.1 were given by the non-greedy algorithm where a transition is selected to fire non-deterministically among all the enabled transitions. The

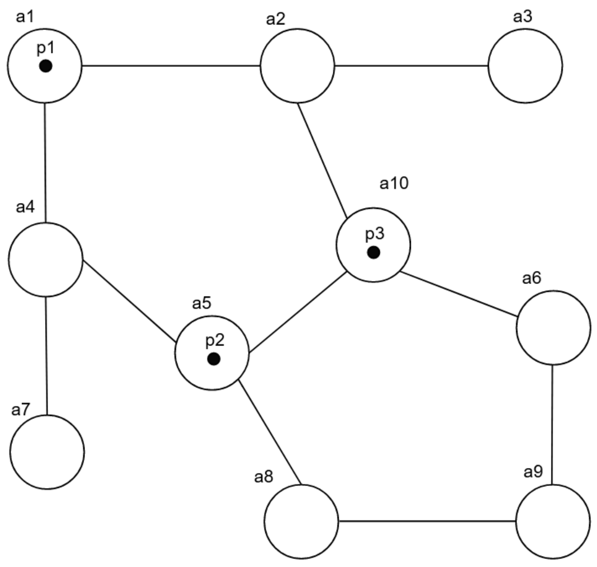


Figure 4.3.1: Antenna net for greedy vs non-greedy

greedy algorithm on the other hand, will always select the transition that gives the maximum increase in sum capacity for the next step. As we will see in this subchapter, the greedy algorithm produces worse results compared to the non-greedy algorithm while also being slower in performance.

Adding back to the program the extra rules for greedy-solving, and running it again for the net shown in Figure 4.3.1, we obtain the following results:

fires(t58,0). fires(t106,1). fires(t12,2). fires(t89,3). fires(t23,4).

The powered-on antennas when converging, are the antennas a3, a6 and a9. The final sum-capacity is ***2,52471621 x 10***$^{-7}$, and it is inferior to the first answer set of the non-greedy algorithm which was ***2,63291834 x 10***$^{-7}$.

The problem of the greedy algorithm is that it does not explore sufficiently, and it can miss turning on a beneficial antenna. On the other hand, the non-greedy algorithm allows for partial exploration in the net, and the power can end up in multiple locations in different answer sets.

The way that Clingo works, it can produce multiple answer sets without adding a lot of overhead to the solving process. This works similarly with running the algorithm in parallel. However, the greedy algorithm has only one solution, as it always selects the one transition that increases the sum capacity by the biggest amount.

The non-greedy algorithm can produce different answer sets and if we get all the answer sets, the results from the best answer set will be **equal or better** than the greedy algorithm, while also being faster in performance (due to fewer rules). This is guaranteed because the greedy solution can also always be found in the non-greedy algorithm results.

We can use some extra constraint rules to check the reachability of an answer set that has better results than what we currently have, or we can print all the answer sets and choose the best one manually.

```
goal:- value(Q,X,_), X>263291834, type(p,Q).
:- not goal.
```

The best solution returned from the non-greedy algorithm, is a solution where eventually antennas a6, a7 and a9 are turned on. The sequence of firings is as follows:

fires(t58,0). fires(t89,1). fires(t14,2). fires(t47,3). fires(t106,4).

and the resulting sum capacity is ***2,81325521 x*** $\boldsymbol{10^{-7}}$ and it is equal to what an **optimal** algorithm would return.

Chapter 5

Conclusions

5.1 Summary 99

5.2 Challenges 100

5.3 Future Work 101

5.1 Summary

The subject of this diploma thesis is focused on encoding the Multi-token Petri Nets (MPNs) into Answer Set Programming language. Algorithms were developed for MPN and reversible MPN, also called MRPN, as well as Controlled MPN and Controlled MRPN. Additional extensions were also developed for an algorithm proposed towards solving the optimization problem of Antenna Selection, which belongs to the category of Multiple Input Multiple Output (MIMO) problems and is an NP-hard problem. Python was necessary to work in conjunction with ASP and provide a way to compute the complex equation that measures the sum capacity of an Antenna network. Following the creation of the program, some optimizations were made to achieve better performance results. The program was then demonstrated in the case study for an antenna selection algorithm. Finally, a comparison was made between the greedy and the non-greedy algorithm for finding the optimal token distribution on a net.

5.2 Challenges

Several challenges were faced during the implementation of this diploma thesis. One of the bigger challenges was to start getting into the correct mindset, for writing code in a completely different manner than I was used to. Trying to understand how Petri nets and Reversible Petri nets were implemented in ASP, and how the code works seemed impossible at first, especially since there is little to no online support. After spending more time experimenting with ASP, and better understanding Petri Nets, how they work and how they can be reversed, I started feeling more confident.

When the time came for me to start implementing a form of Multi-token Petri Nets, the Collective Petri Nets I had to rethink the process from almost the very beginning, as Collective Petri Nets need many different facts to be created from the very early stages. Many hours of thinking about the algorithm and the rules that I needed to implement for each step of the coding process were needed and I had to have a clear and fresh mind to be productive. However, I really enjoyed the challenge involved and solving a problem was very satisfying and motivational to move forward.

Another challenge with ASP is the difficulty of finding simple problems like typographical errors, especially when searching in hundreds of lines of code. Aside from that, when implementing the Multi-token Petri Nets, there are many different scenarios that must be thoroughly tested and are difficult to find and solve in some cases, such as the problem of identifying cycles in token bonds.

When something is not working as expected, there is no formal debugger. The programmer must search and follow the code and while this is simple for small programs, it gets much harder for bigger programs with many facts and rules. After becoming more familiar with ASP, I found out ways to "debug" my problems and find where the mistake is and why the desired results are not produced, by having extra debugging and error rules, or adding extra parameters in some facts and atoms. The time spent developing this custom debugging process was well worth it in the long run. I also automated the process of declaring a problem as much as possible, leaving the minimal number of facts needed to be declared and minimizing the room for errors.

ASP does not support lists of any kind, but to encode the problem of Antenna Selection in ASP, the use of lists at some point is inevitable. Several other ways without the use of lists were tried but none had the desired outcome. This was one of the most difficult problems I faced, and I solved it using custom lists that were created with the help of Python calls.

Changing a basic fact in ASP, will result in many changes in the program, as most likely, all the rules that contain the fact must now be changed as well, and that continues recursively. This makes it hard to change fundamental facts after creating rules based on them. To avoid this mistake, extensively checking each stage is required.

While solving the problem of the Antenna Selection, difficult mathematic equations and matrix multiplications arise which are not possible to be calculated using just ASP. At that point, a higher-level language was needed. Getting Python to work with ASP for calculating difficult equations was also a difficult task due to limited online help.

Finding the source of “low” performance and fixing it was also a challenge. Better understanding of ASP was vital at this point, in order to understand what is stalling performance and what could be a better alternative.

Regardless, ASP is now one of my favorite programming languages because it involves a challenge, it forces you to think of a way to solve your own problems, and not just use the internet to find the solution.

5.3 Future Work

Future work on this topic could involve further optimizations on the MPN and MRPN algorithms, making them more efficient. Other problems like the Antenna Selection problem can be simulated using the CMPNs or CMRPNs and solved using ASP and clingo. ASP is a very powerful tool that can help solve NP-hard problems and give all the possible answer sets/solutions to a problem without increasing the time needed by much compared to a single answer.

Specifically for the problem of the Antenna Selection, future work can be done to further optimize the algorithm suggested for MPNs. The algorithm can be extended to support multiple neighborhoods. A dynamically changing matrix H can also make the problem more realistic, with users changing over time. In this case using reversible execution would be reasonable for error recovering. Further optimisations to the problem are also possible.

More properties can also be implemented for the multi-token reversing Petri nets. Some examples are the property of Persistence, a transitions' liveness, finding home states, finding the shortest path etc.

The code is available at: https://github.com/PittMichaelAngelo/ThesisRepository

REFERENCES

[1] A. Ozgur, O. Lévêque, and D. Tse, “Spatial degrees of freedom of large distributed MIMO systems and wireless ad hoc networks” IEEE Journalon Selected Areas in Communications, vol.31, no.2, pages 202–214, 2013.

[2] A. Philippou, K. Psara, and H. Siljak. Distributed antenna selection for massive MIMO using reversing Petri nets. IEEE Wireless Communications Letters, 2019.

[3] A. Philippou, K. Psara, and H. Siljak, “Controlling Reversibility in Reversing Petri Nets with Application to Wireless Communications,” *in 11th International Conference on Reversible Computation,* pages 1-7, Springer, 2019.

[4] A. Philippou and K. Psara, Reversible computation in Petri nets, in International Conference on Reversible Computation, pages 136-148, Springer, 2018.

[5] C. A Petri, Kommunikation mit Automaten. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962, Second Edition: New York: Griffiss Air Force Base, Technical Report RADC-TR-65--377, Suppl. 1, English translation, Vol.1, 1966.

[6] E. W. Mayr. An algorithm for the general Petri net reachability problem. SIAM, vol. 13 no.3, pages 441-460, 1984.

[7] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., Proceedings of International Logic Programming Conference and Symposium, pages 1070–1080. MIT Press, 1988

[8] J. Hoydis, S. t. Brink, and M. Debbah, “Massive MIMO in the UL/DL of Cellular Networks: How Many Antennas Do We Need?” IEEE Journal on Selected Areas in Communications, vol.31, no.2, pages 160–171, 2013.

[9] K. Barylska, M. Koutny, L. Mikulski, and M. Piatkowski. Reversible computation vs. reversibility in Petri nets. In Proceedings of RC 2016, LNCS 9720, pages 105–118. Springer, 2016.

[10] K. Barylska, L. Mikulski, M. Piatkowski, M. Koutny, and E. Erofeev. Reversing transitions in bounded Petri nets. In Proceedings of CS&P 2016, volume 1698 of CEUR Workshop Proceedings, pages 74–85. CEUR-WS.org, 2016.

[11] Moore, R. Semantical considerations on non-monotonic logic. Artificial Intelligence 25(1):75–94, 1985.

[12] M. Frank, "The Future of Computing Depends on Making It Reversible", *IEEE Spectrum: Technology, Engineering, and Science News*, 2017.

[13] R. J. van Glabbeek. The individual and collective token interpretations of Petri nets. In Proceedings of CONCUR 2005, LNCS 3653, pages 323–337. Springer, 2005.

[14] R. J. Lipton. The reachability problem requires exponential space. New Haven, CT, Yale University, Department of Computer Science, Res. Rep. 62, 1976.

[15] R., R. A logic for default reasoning. Artificial Intelligence 13: pages 81–132, 1980.

[16] R. Landauer. Irreversibility and heat generation in the computing process. IBM Journal of Research and Development, 5(3):183–191, 1961.

[17] S. R. Kosaraju. Decidability of reachability in vector addition systems. In Proceedings of 74th Annual ACM Symp. Theory Computing, San Francisco, pages 267-281,1982.

[18] X. Gao, O. Edfors, F. Tufvesson, and E. G. Larsson. Massive MIMO in real propagation environments: Do all antennas contribute equally? IEEE Transactions on Communications, 63(11):3917–3928, 2015.

[19] Y. Gao, H. Vinck, and T. Kaiser, “Massive MIMO antenna selection: Switching architectures, capacity bounds, and optimal antenna selection algorithms,” IEEE Transactions on Signal Processing, vol.66, no.5, pages 1346–1360, 2018.

[20] clingo and gringo. Available: https://potassco.org/clingo/

[21] "Clingo API documentation" Available: https://potassco.org/clingo/python-api/5.4

[22]"Hopcroft–Karp algorithm - Wikipedia", *En.wikipedia.org*. Available: https://en.wikipedia.org/wiki/Hopcroft%E2%80%93Karp_algorithm.

[23] Potassco , *Potassco.org*. Available: https://potassco.org/