Solving wildlife conservation problems using Answer Set Programming

Tiago Almeida
Instituto Superior Técnico
INESC-ID
email: tiago.almeida@ist.utl.pt

Abstract

Preserving the biodiversity and avoiding the fragmentation of a land is an area that has been the subject of research since it is important to ensure that species are not put at risk when changing the available land for them. The wildlife conservation problem can be divided in two sub-problems. Given a land, that is split into smaller areas called sites that can have an utility and a cost value associated, the first sub-problem consists in, given certain key sites, finding the a continuous path with most utility between them under a certain cost constraint. The second consists in finding minimum amount of sites that are not scattered and that keep the maximum biodiversity (i.e. the sites must have at least one of each species represented). Those are important approaches and both are covered in this document. Recent work has developed either dedicated algorithms or integer linear programming approaches to solve this problem. Our goal is to solve this problem using answer set programming, which is based on a declarative language and has been successfully applied to different domain problems. We described the answer set programming language, including the semantics and syntax, and also referred one of the tools available to draw solutions, called ASPviz. The ASP programs to solve the two problem variants are in detail, including the impact of the choices. We compared the results obtained in answer set programming with one of the previous implementations and discuss the obtained results.

1 Introduction

The biodiversity is a fundamental characteristic for the survival of many species. When for some reason a land needs to be changed, it is essential to identify not only the fraction of the land with the maximum biodiversity, but is also essential to assure that the chosen territory will not be fragmented, since fragmentation promotes the species to be isolated, thus putting their survivability at risk. Most of the optimal choices with respect to the amount of different species result in very scattered lands. For that reason, some of the recent work is focused on ensuring that the chosen terrain is not fragmented[1, 2]. These approaches put emphasis on finding an area, restricted to some characteristics like land utility or land cost, that keeps the maximum biodiversity. The referred approaches are based on mathematical formulations of the problem, using integer linear programming to solve a set of equations that represent the problem restrictions. This work aims at introducing a new way of solving the connectivity problems, using the answer set programming language instead. Answer set programming is a declarative language in which we specify the constraints and let the program search for the solution. This allows to specify the problem in terms of some constraints (like the ones that prevent fragmentation, or to get below a certain cost) and let the program find a solution that does not break any restriction. Since we can restrict the problem to a few rules, this language is adequate for solving the wildlife conservation problems. With this we want to provide an efficient approach to solve the wildlife corridor and related problems. This work is organized as follows. In section 2 we will describe the problem variations and the existing approaches. In section 3 we will explain the answer set programming language using an example for better understanding.
Our approach will be detailed in section 4, where we explain the code that we created and describe our decisions for the implementation. The results will be displayed in section 5. In section 6 we will show our conclusions.

2 Wildlife Conservation

The wildlife conservation problem can be related to graphs, since it is an easy way to represent the sites and their adjacency. Therefore we will give a small notion of graph theory for better understanding of future concepts. We say that two sites are adjacent if they share a common border. The most basic type of graphs, undirected graphs, can be defined in the following way:

**Definition 1** A graph \( G = (V, E) \) consists in a set \( V \) of nodes, and a set \( E \) of edges, where each edge is pair \((v_1, v_2)\) of nodes from \( V \).

In the above definition, if every node is connected by an edge, then it is called a complete graph. When we want to refer to the number of nodes we use the notation \(|V|\). For the number of edges (also called size of the graph) we use \(|E|\). Another important definition is the definition of connected graph, since it is a main concern for the wildlife conservation problem. Before giving the definition we will introduce the notion of path. A path is an ordered set of nodes in which there is an edge from each node to the next node in the set. The definition for a connected graph is the following:

**Definition 2** A graph \( G = (V, E) \) is connected iff \( \forall v_i \in V \), there is a path to every other node in the graph.

This means that any node in a graph is reachable from any other node in the same graph. Another important definition is the definition of subgraph:

**Definition 3** A graph \( G' = (V', E') \), where \( E' \) are the edges from \( V' \), is a subgraph of \( G = (V, E) \) iff \( V' \subseteq V \) and \( E' \subseteq E \).

We can also have Weighted graphs. Weighted graphs are graphs where we can give a certain value to each edge. That value is called the weight of that edge. This weight is used to represent numeric attributes for using a certain edge, like a cost or a profit. With those notions we can now describe the problem itself in section 2.1.

2.1 Problem description

In wildlife conservation, there is a need to identify the priority areas that ensure species viability. In the context of this problem, a land is defined as follows:

**Definition 4** A land is a representation of an area where there is a certain amount of biodiversity and some hostile land that is not populated by any species.

A land is divided into smaller areas, called sites.

**Definition 5** A site is a portion of a land that has the information about which species populate that site (or no species at all are there if it is an hostile area).

In the scope of this problem, the connectivity is defined as follows:

**Definition 6** Two sites are connected if they share a border with each other. A set of sites \( S \) is connected if for every site \( s \in S \) there is a way, using only connected sites, to reach any other site \( s' \in S \).

Also, a cover is defined as follows:

**Definition 7** Considering \( M \) as the set of species \( p \), a set of sites \( S \) is a cover if \( \forall p \in M \) there is at least 1 representative of \( p \) in \( S \).

This definition can be extended to an \( n \)-cover, where instead of having at least 1 representative, the set of sites must have at least \( n \) representatives.

Formally, the wildlife conservation problem is defined as follows:
Definition 8 Considering a set of species $M$, a land $L$, divided into a set of sites $S$, and a cover $p \in M$, the wildlife conservation problem is to find the minimum connected cover.

Although there are some approaches that try to find the smallest number of sites with maximum species, or the maximum species given a number of sites[3], connectivity is not a goal in those approaches, which might result in some scattered sites. Scattered sites are not desirable as they may halt the cycles needed for species viability, since some species need some adjacency in order to survive. The wildlife conservation problem we are trying to solve uses connectivity as a mandatory constraint, in order to find the best connected sub-sets of land that has either at least a specie of each kind, or that includes at least certain sites.

2.2 Related Work

There are a few approaches with respect to the two sub problems presented in this article. To the sub problem where we have to find the minimum connected cover there is an approach done by Cerdeira et al [2] that uses integer linear programming to find a solution. Here we will focus in an approach done by Dilkina et al [1] which is the one we will compare our work with. That approach is the Single Commodity Flow approach and it is related to the wildlife corridor problem.

2.2.1 Single commodity flow

In this approach, to each site is associated a cost and an utility value. There are a few sites denominated reserves, and those are the ones that need to be included in the final solution. The main goal is to connect those reserves, using the remaining sites and maximizing the utility. As an additional restriction, the sum of the costs of each site must be below a given cost (however, it is not a goal to minimize the cost). As referred earlier, these approaches represent the problem a graph $G=(V,E)$, which has some terminal nodes $T \subseteq V$, a function to determine the cost, $c : V \rightarrow R$ and a function for the profit, $u : V \rightarrow R$. Using such a graph, the goal is to find a sub-graph $G' = \{V', E'\}$, where $V' \subseteq V$ and $E' \subseteq E$, in which $V'$ contains all of the terminal nodes $T \subseteq V'$ and the sub-graph $G'$ is connected.

In these approaches each site is represented as $x_i$, where $i$ is a specific site, and the value for $x_i$ is 1 if the site is used in the final solution, 0 otherwise. The (mandatory) inclusion of the terminal nodes in the solution is represented on the following way:

$$x_i = 1, \quad \forall t \in T$$

$$x_i \in \{0, 1\}, \quad \forall i \in V$$

Since the goal is to maximize the utility of the subset of sites used in the solution, the following restriction is used:

$$\text{maximize} \sum_{i \in V} u_i x_i \quad (1)$$

The total cost of the sites included in the solution must also be restricted by $C$, so the following constraint is used to express that:

$$\sum_{i \in V} c_i x_i \leq C \quad (2)$$

For the connectivity between the terminal nodes, some authors[4, 5] consider the problem of as a single-commodity flow problem, where some properties like the flow conservation assure that the subgraph is connected. In order to do this, each previously undirected edge $(i, j) \in E$ is considered as two directed edges $(i, j)$ and $(j, i)$, and we will refer to the set of all the directed edges as $D$. It is considered that node act as a sink, consuming one unit of flow. The flow is introduced in the system in the following way:

- A random node $r$ from the set of terminal nodes $T$ is selected.
- A value for the maximum flow $f$ is determined by the total number of edges, $f = |V|$.
- A source node is introduced in the system and an edge $(0, r)$ is created to put the flow into the network.

A variable is needed to represent the flow in an edge, so it will be defined that $y_{ij}$ represents the
amount of flow in the edge \((i, j) \in D\). The set of constraints needed to solve the connectivity issue in a single-commodity flow are the following:

\[
x_0 + y_0 = f \quad (3)
\]

\[
y_{ij} \leq f.x_j, \quad \forall (i, j) \in D \quad (4)
\]

\[
\sum_{i=1}^{A} y_{ij} = x_j + \sum_{i=1}^{A} y_{ji}, \quad \forall j \in V \quad (5)
\]

\[
\sum_{j \in V} x_j = y_0 \quad (6)
\]

\[
y_{ij} \geq 0, \quad \forall (i, j) \in D \cup (0, r) \quad (7)
\]

\[
x_0 \geq 0, \quad \forall (i, j) \in D \quad (8)
\]

In the constraint (3), a new variable \(x_o\) is introduced to allow residual flow. This is the flow that is not used in the network, since the solution might not include all the nodes. This constraint expresses exactly that: the sum of the unused flow with the used flow is equal to the maximum outgoing flow \(f\). To enforce the nodes to consume a unit of flow, rule (4) is used, since if the incoming flow is greater than zero, \(x_i\) must be assigned value one, otherwise the constraint would not be satisfied. Constraint (5) is used to ensure the flow conservation by checking if, for a certain node \(x_i\), the incoming flow \(y_{ij}\) is equal to the outgoing flow \(y_{ij}\) plus the flow in the node \(x_i\) itself. Constraint (6) ensures that the flow injected in the system is equal to the sum of the absorbed flows by each node. Finally constraints (7) and (8) are used to force the flow values to positive numbers. With this constraints it is ensured that the nodes are all connected. This approach creates a variable for each edge, \(y_{ij}\), to represent the amount of flow on them, and an additional variable for the edge between the source and the root. Therefore, this approach adds \(|D| + 1\) new variables.

### 3 Answer Set Programming

Answer Set Programming (ASP) is a declarative language created to solve problems where instead of creating an algorithm, we specify the constraints, the rules and the data of the problem and let the ASP solver find the answer. It was designed with the intention to be used mostly to solve search problems, although it can be used for other purposes. The goal of this language is to allow the user to simply specify the constraints of a problem and ask for a solution that does not violate any of those rules, instead of having to implement an algorithm to search for such solution. It is a declarative language based on stable models[6]. Informally, a stable model is a set of positive literals that make a program true. An answer set program is built with atoms, literals and rules[7]. The literals can be either positive or negative, and all the literals that are not stated in the program are considered do be negative (closed world assumption).

The rules are the basic way to generate new atoms. A rule is usually on the form

\[
\text{Atom} \leftarrow \text{Literal}_1, ..., \text{Literal}_N
\]

where \(\text{Atom}\) is an atom that is true if all of the literals (\(\text{Literal}_1\) to \(\text{Literal}_N\)) are true. If the left side is empty, then the rule means that we cannot have simultaneously all of the literals on the right side of the equation. If the right side is empty, then we have a fact, since the left side will always be true. ASP also supports functions like sum, count, maximize, minimize, etc. given in the form

\[
\#\text{opt}[\text{Literal}_1 = \text{Weight}_1@\text{Priority}_1, ..., \text{Literal}_N = \text{Weight}_N@\text{Priority}_N]
\]

where \(\#\text{opt}\) is the function. The ASP process can be split into two different tasks: the grounding process (made by grounders like lpars^1 and gringo[8]) and the solving process (made by solvers like smodels or clasp). For our program we used the gringo as a grounder and clasp as a solver, since they are the most promising and updated programs. The grounder is responsible for computing all the ground terms, i.e., terms with no variable on them. This means that the grounder makes all the possible attributions for the variables on the rules. Then the solver will find a set of such terms that does not violate any of the constraints, which represents the answer set for the

---

1http://www.tcs.hut.fi/Software/smodels/
given instance. One of the relevant advantages of the ASP for our work is that it allows the search for the optimal solution. For example, it includes a maximize statement that is useful when we are trying to find the solution with the maximum utility with respect to the wildlife corridor problem.

4 Solution architecture

In this section we will describe the proposed solutions. Since we have two different problem variants we will deal with each of them individually. First in section 4.1 we will describe the wildlife corridor approach. In section 4.2 we will explain the approach for the minimum connected cover problem.

4.1 Wildlife Corridor

The first thing we know is that there will be a certain number of sites, so we can represent them as

\[ \text{node}(i). \]

where \( i \) is the node number. The cost of the node must be referred, and that can be done using the following representation

\[ \text{cost}(i, c). \]

where \( i \) is the node number and \( c \) is its cost. In a similar way, the utility can be represented by

\[ \text{util}(i, u). \]

being \( u \) the utility of site \( i \). The adjacency between sites must be referred as well, so the program knows that two sites are connected. We can represent that with

\[ \text{edge}(i, j). \]

where \( i \) and \( j \) are two distinct sites that are adjacent. Two sites are considered adjacent if they share any borders between them. At last we must refer the mandatory sites and that is done by including the predicate

\[ \text{terminal}(i). \]

where \( i \) is a mandatory site.

In order to make a program in ASP to solve this problem, we should take a closer look at the restrictions in the problem. Considering \( C \) as the maximum cost, \( c_i \) the cost of the site \( i \) and \( u_i \) the utility of site \( i \), the restrictions to which the problem is subject are the following:

\( R1 \): The mandatory sites must be included.

\( R2 \): The mandatory sites must be connected by some path of sites.

\( R3 \): The sum of the costs of each chosen site must not be greater than \( C \).

\( R4 \): The utility of the chosen sites must be the maximum value possible under the cost limitation \( C \).

Having the rules discriminated as above is very helpful when trying to implement a program in ASP. In order to allow the ASP program to choose certain sites we must have a rule that, from any site, possibly generates a chosen site. Such rule can be written as follows:

\[ \text{chosen}(N) : \text{node}(N). \]  \hspace{1cm} \text{(9)}

The meaning of this line is: for each site \( N \), we can have either 0 or 1 \( \text{chosen}(N) \). Another factor to consider is that the input might only have the adjacency in one direction. Intuitively we know that if \( A \) is adjacent to \( B \), then \( B \) is adjacent to \( A \), but we need to say that explicitly to the program, so we add the following line to the program:

\[ \text{edge}(N, N_2) : -\text{edge}(N_2, N). \]  \hspace{1cm} \text{(10)}

With this, the program will look at every edge given at the input, and generate the edge in the opposite direction. The restriction \( R1 \) is a really direct rule, since all we have to do is to look at the sites that are considered mandatory and force the program to keep them in any of the solutions. To do such thing, we can simply write the following line:

\[ : -\text{terminal}(N), \text{notchosen}(N). \]  \hspace{1cm} \text{(11)}

Line 11 explicitly says that we cannot have a final answer set that contains simultaneously a terminal node \( N \) (which is a mandatory one as referred previously) and does not have \( N \) as one of the chosen nodes. In order to make sure that the chosen sites are always connected in some way to each other, we must define what are the possibilities to chose given certain choices. To make it easier to define connectivity in ASP, we give as input the predicate

\[
\text{start}(N).
\]

where \( N \) is one of the terminal sites, chosen at random. To represent the connectivity in ASP we use a recursive solution, shown in rules 12 - 14.

\[
\begin{align*}
\text{poss}(N) & : \neg \text{start}(N). \\
\text{poss}(N) & : \neg \text{poss}(M), \text{edge}(N, M), \text{chosen}(M). \\
& : \neg \text{chosen}(N), \neg \text{poss}(N).
\end{align*}
\]

Rule 12 says that the start node is a possibility, meaning that it is one of the allowed sites, since it is connected to a terminal node (in this case, it is a terminal node itself). Rule 13 is the recursive rule, where we restrict the possibilities to nodes that are adjacent to possible nodes that are chosen nodes. Rule 14 ensures that there is no site that is not a possibility from the adjacency, making it impossible to have isolated sites. With this set of rules we addressed the connectivity issue, covering rule \( R2 \). Rule \( R3 \) is a cost restriction, to ensure that we are searching in a certain scope of possibilities. Considering the first site of the sample, we check if it is a chosen site or not. If it is a chosen site, we say that the sum of the costs until site 1 is its cost. If it is not a chosen site, then the sum is 0. The rule is represented in ASP by 15 and 16:

\[
\begin{align*}
\text{sum}(1, C) & : \neg \text{cost}(1, C), \text{chosen}(1). \\
\text{sum}(1, 0) & : \neg \text{notchosen}(1).
\end{align*}
\]

From now, we can just say that the sum for a site \( N \) is equal to the sum of site \( N-1 \) plus either the cost of site \( N \) if it is chosen or 0 otherwise. To represent this in ASP we use the following lines, 17 and 18:

\[
\begin{align*}
\text{sum}(N, C + C2) & : \neg \text{sum}(N - 1, C), \\
\text{cost}(N, C2), \text{chosen}(N). \\
\text{sum}(N, C) & : \neg \text{sum}(N - 1, C), \\
\text{notchosen}(N), \text{node}(N).
\end{align*}
\]

Rule 15 to 18 make it possible to know the sum of the costs until a given site is reached. As a final constraint to make rule \( R3 \) true, we must now say that there is no site in which the sum is greater than the input value. To do that we can use a simple constraint with a value \( \text{max} \) that is given as input, being it the maximum cost allowed for that instance. Line 19 represents the ASP constraint needed to ensure that no solutions have a cost greater than the given cost.

\[
\neg \text{sum}(N, C), \text{node}(N), C > \text{max}.
\]

Finally, to cover the rule \( R4 \), which is the optimization criterion, we can use the optimization function available in ASP. Line 20 will associate the utility with the respective chosen site.

\[
\text{chosenUtil}(X, Y) : \neg \text{chosen}(X), \text{util}(X, Y).
\]

With this information, we only need to do the trivial rule to get optimality, since ASP guarantees optimality of the solutions when using the maximize predicate and the program ends [9].

\[
\text{#maximize[chosenUtil}(N, U) = U].
\]

With line 21 we ensure such optimality, since we want to maximize the sum of \( U \), which in this case is the utility for each site. This implementation generates a large number of ground terms, so we created an alternative version that replaced lines 15 to 19 with more efficient lines that serve the same purpose. Those lines are the lines 22 and 23. They will generate less ground terms and with that we want to achieve better solving times for our problem.
\[\text{costC}(N, C) : -\text{cost}(N, C), \text{chosen}(N). \]  
\[\text{sum}(C) : -C = \#\text{sum}\{\text{costC}(C2) = C2}. \]  

4.2 Minimum Connected Cover

In this section we will give a short description on the program for the minimum connected cover problem. The input is very similar to the input for the previous approach, but instead of costs and utilities, we have

\[\text{species}(i, s).\]

which represents that the site \( i \) is inhabited by the species \( s \). In this approach we need to ensure that the solution includes a cover. In order to ensure that we use the rules 24 and 25.

\[\text{covered\_species}(X) : -\text{chosen}(N), \text{species}(N, X). \]  
\[\text{:} -\text{species}(X), \text{notcovered\_species}(X). \]  

With those rules we ensure that there is no solution without at least 1 of each species, since the predicate \(\text{covered\_species}(X)\) is only generated for chosen sites and we cannot have a species that is not on such predicate, due to rule 25. The connectivity rules are very similar but we do have to intrude two extra rules, because we cannot choose any random site as a starting point, it must be one of the sites in the solution. Rules 26 and 27 handle that issue.

\[\text{1start}(N) : \text{node}(N)1. \]  
\[\text{:} -\text{start}(N), \text{notchosen}(N). \]  

5 Results

In this chapter we will show and discuss the result of our implementations and compare the new wildlife corridor implementations to the CPLEX implementation. We created a generator and generated some instances for both approaches with the characteristics summarized on table 1. In the wildlife corridor problem there are two instances for each size because we have an instance with a short range of costs and utilities, and an instance with a long range of costs and utilities. For the minimum connected cover we also have two instances of each size, one with 20 species appearing 10 times each and another with 10 species appearing 20 times each.

<table>
<thead>
<tr>
<th>Problem variant</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild. corridor</td>
<td>Two 9x9 and two 10x10 instances with cost range from 100% to 190%</td>
</tr>
<tr>
<td>Min. con. cover</td>
<td>Two 7x7, two 8x8, two 9x9 and two 10x10 instances</td>
</tr>
</tbody>
</table>

Table 1: Instance selection summary

We will compare our approach with the Single Commodity Flow approach used to solve the wildlife corridor problem. In figures 1 and 2 we show the graph representing the time results for the second approach S.A. and CPLEX. The cost limit increment is represented by the % value in the second column, where 0% is the minimum cost that is needed for a solution. The time is given in seconds, and the instances that took longer than a certain time to solve (50 seconds for 9 by 9 and 250 seconds for 10 by 10 instances) or didn’t solve in 720 seconds are not in the graphs. The instances were run in an Intel Xeon 5160 dual-core with 3.00 GHz of clock speed, 4 GB of RAM with the 64-bit version of Linux 2.6.33 as operating system. The clasp version used was the version 2.0.5 and the gringo version was 3.0.4. The time is given in seconds and is the average of the times for the 10 instances of each type.

When comparing the ASP solution with the CPLEX solution, we can see that ASP is faster for the cases where the budget is close to the minimum, but it gets much slower as we increase that constraint. This means that ASP performs worse when we have a large pool of solutions and we try to find the best one, as opposed to the cases
where the majority of site combinations are over that budget. One of the possible explanations is that it is harder for ASP to deal with large search space, since it will not be able to cut as much information as it potentially could if the budget was lower. It seems that the CPLEX can find better ways than ASP to automatically cut the search space for this problem. The ideal would be to exclude part of the search space by adding some constraints, but since the instances are random and not symmetric, we cannot know beforehand what kind of information could be excluded of the decision process. As an example, when we deal with symmetric data, we can use only half of the information to find a solution, since it is always possible to retrieve other solutions from the one we get, so we cut part of the search space. In this case, the costs and utilities are not symmetric at all, so we have to consider the whole data. Given that we end up having a solution that is worse than the CPLEX implementation for large budgets, but it can still solve the smaller cases relatively fast. It also has the advantage of being easy to write and understand, which is one of the main features of ASP.

Table 2 shows the results for running some instances for the minimum connected cover problem. On the left we have the instance identification, where the ones saying 10 species have each species present in 20 distinct sites, while the ones saying 20 species have each species in 10 distinct sites. The second column shows the time until the optimal solution was found, in seconds. We gave the program 720 seconds for each instance before time-out.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7x7 - 10 species</td>
<td>0,01</td>
</tr>
<tr>
<td>7x7 - 20 species</td>
<td>0,99</td>
</tr>
<tr>
<td>8x8 - 10 species</td>
<td>0,03</td>
</tr>
<tr>
<td>8x8 - 20 species</td>
<td>28,95</td>
</tr>
<tr>
<td>9x9 - 10 species</td>
<td>0,2</td>
</tr>
<tr>
<td>9x9 - 20 species</td>
<td>time-out</td>
</tr>
<tr>
<td>10x10 - 10 species</td>
<td>9,26</td>
</tr>
<tr>
<td>10x10 - 20 species</td>
<td>time-out</td>
</tr>
</tbody>
</table>

Table 2: Solving time for minimum connected cover instances

As we can see, in this case the ASP solver also quickly starts to take much longer to solve the instances. There is a huge difference between having to cover 10 or 20 species, since finding a cover for more species is naturally harder. It is important to refer that in both wildlife corridor and minimum connected cover variants we have the connectivity issue, which generates a big amount of ground terms. Having such a big number of ground terms for connectivity is one of the causes for the exponential solving time, since the solver has to consider a large number of rules.

6 Conclusion

When we started by choosing ASP as the programming language for the problem we were motivated by the purpose of the language. It is a simple language were we can just specify our constraints in a very intuitive way and let the solver find the solutions. Since it is also a language used mostly for solving search problems or problem with spe-
specific constraints, we decided to apply ASP to the wildlife conservation problems.

As a result of this work we found out that it really is intuitive to write a program in ASP even to find optimal solutions. The program has proven to be really fast for small instances, even faster than the CPLEX implementation that we used for comparison. Unfortunately, despite our attempts to improve the encoding, when we increase the budget constraint on wildlife corridor approach, the ASP encoding takes longer than the CPLEX implementation. However, we do consider that the work we developed is important, since it allows to quickly create a program to solve those problems and return the optimal solution using a language that despite not very well known has a lot of potential. We gave importance to the understanding of ASP, which can be helpful for future work since it lets us understand better which rules generate more constraints. Overall, working with ASP is an interesting challenge, because ASP is a language that is easy to understand as a concept, but yet is far from being trivial to understand how it really works.

**Future Work** Considering the advantages of ASP, we consider that it is important to find an alternative way to represent the issues that prevented the efficiency of the program, like the connectivity representation. There are always other possible representation of the data (domain) that might be more advantageous for the solving phase, since choosing the domain can influence the problem resolution time. The change of the domain can even allow different approaches that can be more efficient. Also, the Clasp tool is currently in development, so the solver can be improved in the future in a way that benefits solving this type of problem. For example, really close to the writing of this document, a more recent version of clasp (2.1.0) was released. Although this particular version did not improved the time for our implementation, it is important that the tool is in constant update. There is no recipe for programming in ASP, and even the good practices do not work in the same way for every type of program, so exploring alternative ways can result in very different results, for better or worse. We do encourage the use of Gringo with Clasp since they are the most promising grounder and solver, respectively.

**References**


