Solving Wildlife Conservation Problems using Answer Set Programming

Tiago Augusto Escudeiro Almeida

Dissertation submitted to obtain the Master Degree in Information Systems and Computer Engineering

Jury

President: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. Maria Inês Camarate de Campos Lynce de Faria
Member: Prof. Luís Jorge Brás Monteiro Guerra e Silva

October 2012
“The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.”

Ted Nelson,
Abstract

Preserving the biodiversity and avoiding the fragmentation of a land is an area that has been the subject of recent research. 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 the 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 are both covered in this document.

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 describe in detail the ASP language, including the semantics and syntax, and also refer one of the tools available to draw solutions, called ASPviz. The ASP programs to solve the two problem variants are explained in detail, including the impact of the choices. We compare the results obtained in Answer Set Programming with one of the existing implementations and discuss the obtained results.

Keywords

Wildlife Conservation
Answer Set Programming
Graph Theory
Integer Linear Programming
Preservar a biodiversidade e evitar a fragmentação de uma zona tem sido alvo de estudos recentes. É importante assegurar que a vida das espécies não é colocada em risco quando se muda o espaço em que estas habitam. O problema da conservação de espécies pode ser dividido em dois sub-problemas. Considerando uma zona, dividida em áreas mais pequenas às quais se associa um custo e um valor de utilidade, a primeira abordagem consiste em, considerando certas áreas chave, encontrar o caminho contínuo, passando pelas áreas chave, com mais utilidade que esteja abaixo de um determinado custo. A segunda abordagem consiste em encontrar o conjunto mínimo de áreas não dispersas que mantêm a máxima biodiversidade, isto é, que têm cada espécie representada em pelo menos uma das áreas escolhidas. Ambas as abordagens são importantes e são cobertas neste documento.

O nosso objetivo é resolver o problema usando Answer Set Programming, uma linguagem declarativa utilizada com sucesso em problemas de vários domínios. Descrevemos detalhadamente a linguagem ASP, incluindo a semântica e a sintaxe, referindo também uma das ferramentas disponíveis para representação visual da solução, o ASPviz. Os programas ASP criados para resolver ambas as abordagens do problema são explicados em detalhe, juntamente com o impacto das escolhas realizadas. Os resultados obtidos com Answer Set Programming são comparados com uma das implementações já existentes e fazemos a discussão desses resultados.

Palavras-Chave
Conservação de espécies
Answer Set Programming
Teoria de grafos
Programação linear
Acknowledgements

First I would like to thank my tutor Inês Lynce, who guided me through all the steps, always being available for help and showing me the right direction. This work would not be possible without her. I would also like to thank the opportunity given by project ASPEN (PTDC/EIA-CCO/110921/2009), financed by FCT. Also I would like to thank INESC-ID, a great Portuguese research institution.

Second I would like to thank my mother, father and sister, who have always been in my life and always believed in me. I could not ask for a better source of motivation, trust and availability. This is one more step of my life done with their full support. We cannot choose our family, but I feel lucky about mine.

At last but not least I would like to thank my friends. They represent a key role in my life by making me laugh and motivating me, taking the pressure from the hard work of every day. Also a special thanks to Filipe Gouveia and Rodrigo Santos, who helped me with the work done on this thesis, to Rui Claro who has always been there for me and to João Alves for all the technological discussions and laughs. I would also like to thank André Ramos, Diogo Capelo, Marcos Jerónimo, Bárbara Viotty, Luís Pedroso, Marília Barandas, Pedro Coelho, Bernardo Ivo and Rita Morbey for the social life that is necessary for an healthy mind.

Tiago Almeida

Algueirão, 15th October 2012
## Contents

1 Introduction 1  

2 Wildlife Conservation 3  
   2.1 Graph Theory 3  
   2.2 Problem Description 6  
   2.3 Related Work 8  
      2.3.1 Finding a sub-set with all species 8  
      2.3.2 Wildlife Corridor 13  

3 Answer Set Programming 21  
   3.1 A practical example: n-queens 26  

4 Solution Architecture 31  
   4.1 Wildlife Corridor 31  
      4.1.1 First Approach 32  
      4.1.2 Second Approach 39  
      4.1.3 Finding Minimum Cost 40  
   4.2 Minimum Connected Cover 41  
      4.2.1 First Approach 41  
      4.2.2 Second Approach 45  

5 Results 47  
   5.1 Generators 47  
      5.1.1 Minimum connected cover 47  
      5.1.2 Wildlife corridor 48  
      5.1.3 Implementation in CPLEX 48  
   5.2 Instance Selection 50  
   5.3 Results 51
6 Conclusion

A ASPviz 57
   A.1 ASPviz primitives .............................................. 57
   A.2 Representing with ASPViz ...................................... 58

B Input 61
   B.1 Wildlife corridor ............................................. 61
   B.2 Minimum connected cover .................................... 62

C ASP encoding 63
   C.1 Wildlife corridor - first approach .......................... 63
   C.2 Wildlife corridor - second approach ......................... 64
   C.3 Minimum connected cover - first approach .................. 65
   C.4 Minimum connected cover - second approach ................ 66
## List of Figures

2.1 Graph with 4 nodes and 3 edges .............................................. 3  
2.2 Directed graph example ..................................................... 4  
2.3 A Connected and a not connected graph ................................. 4  
2.4 A subgraph example .......................................................... 5  
2.5 Graph with edge weights ..................................................... 5  
2.6 Species distribution over a set of sites ................................. 7  
2.7 Graph representation of a set of sites ................................... 7  
2.8 Graph in which species distribution leads to $K = \{2, 4, 7\}$ ........ 11  
2.9 Result of $\{1, 2, 3, 4, 5, 6, 7\} \setminus \{2, 4, 7\} \cup \{7\}$ ............ 12  
2.10 Result of $\{1, 2, 3, 4, 5, 6, 7\} \setminus \{2, 4, 7\} \cup \{4\}$ ............... 12  
2.11 Flow demonstration on a Single Commodity Flow approach .......... 15  
2.12 Dantzig-Fulkerson-Johnson directed tree on a set of lands .......... 17  
2.13 Results on a randomly generated instance of wildlife corridor problem with 100 nodes .... 18  
2.14 Results on a real life instance of wildlife corridor problem with 242 nodes ........ 19  
3.1 Two solutions for the 4-queens problem instance ...................... 27  
3.2 ASPviz on two solutions for the 4-queens problem instance .......... 28  
4.1 4 by 4 area with costs and utilities represented ..................... 32  
4.2 4 by 4 area with six different species ................................ 42  
5.1 Representation of the adjacency choices ................................ 48  
5.2 Graph representing the choices and conflicts in a 10 by 10 instance ...... 53  
A.1 Empty grid representing the axis ........................................ 58  
A.2 ASPviz solution for one problem instance ............................. 59
List of Tables

3.1 Clasp options ......................................................... 25
5.1 Instance selection summary ........................................ 50
5.2 Ground size for two wildlife corridor approaches ............ 51
5.3 Time comparison between ASP and CPLEX .................... 52
5.4 Solving time for minimum connected cover instances .......... 54
A.1 ASPviz primitives ..................................................... 57
The biodiversity is a fundamental characteristic for the survival of many species. In fact, it is almost impossible for any species to live in an isolated environment. However, sometimes, some of the natural reserves need to be changed, since part of the land may be needed for other purposes (like construction). When that happens, it is essential to identify not only the fraction of the land with the maximum biodiversity, but is also essential to ensure that the chosen territory will not be fragmented, since fragmentation promotes the species to be isolated, thus putting their survivability at risk.

Some work has been done in the past that proposes a solution covering the maximum number of species\cite{22}. However, 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\cite{9, 3}. These approaches put emphasis on finding an area, restricted to some characteristics like land utility or land cost, that keeps the maximum biodiversity. Since the problem is given as a land, split into sites, a natural way of representing the problem is using a graph, where each node is one of the sites and the edges represent the sites adjacency, i.e, if two sites share a common border. 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 chapter 2 we will describe the problem variations and the existing approaches. In chapter 3 we will explain the answer set programming language using an example for better understanding. Our approach will be detailed in chapter 4, where we explain the code that we created and describe our decisions for the implementation. The results will be displayed in chapter 5. In chapter 6 we will show our conclusions.
In this chapter we will describe the problem in more detail and some of the work that was done in this
subject. Since we can relate the problem representation to graphs, we will give some basic notions of
graph theory in section 2.1. In section 2.2 we describe the problem itself and the relevant variants for
our work. In section 2.3 we will show the state of art approaches for this problem.

2.1 Graph Theory

The wildlife conservation problem can be related to graphs, since it is an easy way to represent the sites
and their adjacency. 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 a
pair $(v_1, v_2)$ of nodes from $V$.

In the above definition, if every pair of nodes 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 $|E|$. An example is given in figure 2.1, with $|V| = 4$ and $|E| = 3$,
where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, d)\}$.

![Figure 2.1: Graph with 4 nodes and 3 edges](image)

In this type of graphs, the edges are undirected, meaning that they do not have any specific direction.
In the example of figure 2.1, $(a, b)$ and $(b, a)$ refer to the same edge. However, sometimes, it is needed to
transform this graphs into directed graphs.

Definition 2 A directed graph $G=(V,D)$ consists in a set of $V$ nodes (equal to the undirected graph),
and a set of $D$ edges, where those edges are pairs with an order imposed on them.
This definition implies that having an edge \((v_1, v_2)\) is different from having an edge \((V_2, V_1)\). If we transform the undirected edges from figure 2.1 into directed edges, then the size of the graph becomes \(|D| = 6\), instead of 3. The directed version of the graph can be seen on figure 2.2.

![Directed graph example](image)

Another important definition is the definition of connected graph, since it is a main concern for the wildlife conservation problem, as detailed in section 2.2. 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 3** 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. In figure 2.3 we show on the left a connected graph, and on the right a not connected graph (since it is impossible from \(b\) to reach \(a\) or \(c\)).

![A Connected and a not connected graph](image)

Checking if a set of nodes \(V\) in a graph \(G = \{V, E\}\) is connected can be performed in linear time using the following algorithm:

1. Create an empty list \(L\) and add a random node from \(V\) to it.
2. Initiate a variable \(\text{count}\) starting at 0.
3. For each node \(V_i\) in \(L\):
   - Mark it and add 1 unit to the variable \(\text{count}\).
   - Add all of the adjacent nodes to \(L\).
   - Remove \(V_i\) from the list.
4. When the list becomes empty, compare count to |V| and if they are equal, then G is a connected graph.

Another important definition is the definition of subgraph:

**Definition 4** A graph \( G' = (V', E') \), where \( E' \) are the edges from \( V' \), is a subgraph of \( G = (V, E) \) iff \( V' \subseteq V \) and \( E' \subseteq E \).

An example of a subgraph of the graph in figure 2.1 is given in figure 2.4, where the node \( c \) and the edge \( (a, c) \) were removed from the original graph, but all of the remaining nodes and edges were kept.

![Figure 2.4: A subgraph example](image)

Weighted graphs are graphs where each edge can have a certain value. 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. As an example, consider the graph in figure 2.5. Suppose that it is desired to minimize the cost from of the path \( a \) to \( c \). There are two ways to go from \( a \) to \( c \), either using the path \{\((a,c)\)\} or the path \{\((a,b),(b,c)\)\}. Although the first only uses one edge (and if we were not considering the weights, it would have been the shortest path), the sum of the weights from the second is lower and, therefore, a better path for the example. This allows the definition of problems where there is a value associated with every edge, and that value represents some attribute we want to minimize (or it can even be only an upper limit).

![Figure 2.5: Graph with edge weights](image)

We can now introduce the minimum Steiner tree problem. A tree is a connected graph that has no cycles on it. Also, a terminal node is a node that has to be in the final solution. The minimum Steiner tree problem is defined as follows[7]:
Definition 5 For a given graph $G = \{V,E\}$, where each edge has an associated cost (weight), and a set of terminal nodes $T \in V$, the goal is to find the minimum cost tree that includes all the terminal nodes and eventually some non terminal nodes.

Computing the minimal Steiner tree is an NP-Hard problem[10]. The minimum Steiner tree is an important concept that is widely used when a minimum path between two nodes must be found.

2.2 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 6 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 7 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 8 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 9 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 10 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[17], 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 have either at least a specie of each kind, or that includes at
least certain sites.

As an example, consider the area in figure 2.6. Each of the numbered parcels (in blue) represents the
sites. The species are represented as $a$, $b$, $c$, $d$ and $e$. The black areas are parts of the land that are not
populated or are irrelevant for the problem (like an hostile land or volcano). Lands that share borders
with each other are connected lands (like sites 1 and 2). This also means that 2 is a neighbour of 1, since
the neighbours are the sites that are adjacent to a certain site (this can be changed for some purpose, as
shown in section 2.3.1). In this example, sites 4 and 8 include all the species and they represent a cover.
Sites 4, 6 and 8 make a connected cover, since they are all adjacent to each other and every specie is
included at least once. Most of the approaches to this problem represent the area as a graph $G = (V,E)$,
where the sites are represented as the vertices $V$ and the edges $E$ represent the adjacency of the sites. For
the given example, the graph would be like the one in figure 2.7. This allows the problem to be treated
as a graph problem, expanding the possibilities for solving it. In the following section, section 2.3, we
will describe some of the work done around this problem for which all of the approaches use graphs to
represent the sites.
In the wildlife conservation problem, usually a set of parcels (sites) is considered and the main goal is to obtain a connected area under certain restrictions. One of the variants includes finding a sub-set of sites that contain at least one species of each kind (to be detailed in section 2.3.1). There is also a related problem consisting in, given a certain set of lands and some mandatory sites (that can be seen as the most important sites), finding a connected area containing all of the mandatory sites (usually below a certain cost, maximizing the sites utility). This problem variant is usually called wildlife corridor problem and will be described in section 2.3.2. The main relation between those approaches is that they address the connectivity issue between sites, since it is not desired to get fragmentation.

2.3 Related Work

In this section two of the problem variants will be described, and we will describe related work in the area. Both approaches were programmed in Integer Linear Programming (ILP), which is based on mathematical constraints to model the problem. ILP allows modelling the problem as a mathematical problem, where the equations represent a certain constraint, and then a program (like cplex\(^1\)) finds the values that satisfy all the constraints.

2.3.1 Finding a sub-set with all species

Cerdeira et al. [3] focus into keeping at least one species of each kind in the minimum connected area, meaning that it is desired to minimize the number of connected sites while ensuring a cover. This work is divided into two steps to be described. First, in section 2.3.1, an heuristic approach is used. Then in section 2.3.1, an approach using ILP is used. Cerdeira also considers that the use of different neighbourhood levels can be helpful, and a real world application of that can be found in his paper.

**Heuristic approach**

Before trying to get an optimal result, an heuristic approach that will satisfy the connectivity and covering restrictions was considered. This heuristic is divided into three different steps, applied sequentially. Those steps are:

1. Covering, where the sites containing a cover are found.
2. Connecting, where the cover gets connected using other sites (if needed).
3. Pruning, in which some possibly unnecessary sites are removed.

More detail to each step will be given as follows. At the end of the third step the minimal cover is found.

\(^1\)http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/
Covering Since blindly choosing a set of sites that make a cover might result in very scattered sites, a distance based choice is used instead. Hence, a random node is first selected from the network and is put in a list $L$. After that, and if $L$ is not yet a cover, a site from the adjacent sites is chosen to be added to the list. To be chosen, that site must be the one with more new species amongst the adjacent sites (if two are tied, the choice is random). If none of the adjacent sites have new species, then a site that includes new species will be chosen from the whole set. This choice is done based on the distance between the new site and a site on $L$. The site with the minimum distance (or a random site chosen between two or more if they are tied) will be chosen. Here we consider distance as the number of sites between them. As an example, consider the map in figure 2.7. Starting by (randomly) choosing site 4 to be added to list $L$, we would have as candidates sites 1, 2 and 6. Since each of them adds only 1 new specie ($d$), let us assume that the random choice was site 2. Now $L = \{2, 4\}$. None of the adjacent sites of $L$ has any new specie, so the closest site with new specie will be added. As candidates we have site 7 and site 8 (both containing specie $e$). Since they are both at the same distance, we will randomly pick site 8. Now $L = \{2, 4, 8\}$, which is a cover, thus ending this step of the algorithm.

Connecting As referred on section 2.1, connecting a set of nodes using the minimum extra nodes is called the minimal Steiner tree problem, and it is an NP-Hard problem. A simpler approach is applied: considering a connected subset of $L$, that we will call $S$, the goal is to find the nearest subset of $L$ that is not yet in $S$, and the corresponding (shortest) path, and add it so $S$. This procedure is done until all of the components from $L$ are in $S$. Following the previews example, with $S = \{2, 4\}$, the shortest path to $\{8\}$ is through either site 5 or site 6 After picking randomly site 6, we get $S = \{2, 4, 6, 8\}$, thus obtaining a connected cover.

Pruning After the previous steps, a connected cover has been created. However, nothing guarantees that it is a minimal one, since minimality was not taken into account. In this step the goal is to take the resulting connected cover and turn it into a minimal one. Any site whose removal would break the covering restriction must be in the final solution. However, the same is not true for sites ensuring connectivity, since they may be no longer needed after the removal of some other sites. The pruning process is split into two steps. In step one, sites that have only one neighbour will be either marked as mandatory, if they are part of the sites needed for a cover, or removed since they will never break the connectivity constraint. When all of the sites from $S$ with only one neighbour are either removed or marked as mandatory, step one ends. For the given example, this step would remove site 2, since removing it preserves the covering, and mark sites 4 and 8 as mandatory. In step two, the remaining unmarked sites are checked for removal: for each one of the sites that may be removed, connectivity must be checked, using the algorithm shown in section 2.1). In the given example, site 6 would be checked and marked as mandatory, giving as final result $\{4, 6, 8\}$.
Integer linear programming approach

The heuristic approach gives no guarantee of the optimality of the solution. To obtain the optimal solution an integer linear programming formulation was created. We will refer to the complete set of sites from a land as \( L = \{1, 2, \ldots, N\} \), and to the set of different species \( M \). For each of the sites a variable \( x_i \in \{0, 1\} \), where \( i \in L \) is assigned, where the value 1 means the site is included in the solution and 0 means it is not. The constraints needed to encode the problem are the following:

\[
\min \sum_{i=1}^{N} x_i
\]  

\[
\sum_{i \in K} x_i \geq 1, \quad K \subseteq L
\]  

The constraint (2.1) is the objective function, since it minimizes the amount of sites to be used in the solution. In constraint (2.2) a new variable is used, \( K \), which is a subset of \( L \) for which \( L \setminus K \) does not contain any connected cover. One of the ways of ensuring that we do not have a connected cover when we remove the set \( K \) from \( L \) is by choosing such \( K \) in which all of the sites from a certain species are represented. In such situation, \( L \setminus K \) cannot have a cover. This way, the problem can be modelled using one constraint for each species, using a total of \( M \) constraints of the type (2.2). This model corresponds to the Set Covering Problem. As an example, using the figure 2.7 again as a reference, a constraint for the specie \( a \) would be \( x_1 + x_2 + x_6 + x_8 + x_9 \). To solve the problem with the covering inequalities, the resulting cover is checked. If it is connected, then it is an optimal cover. Else, a new equation needs to be created, where at least one of the sites that are not in current solution must be used. As an example, considering the figure 2.7, the result \( \{4, 8\} \) is a cover, but it is not connected. Then the constraint

\[
x_1 + x_2 + x_3 + x_5 + x_6 + x_7 + x_9 \geq 1
\]  

must be added. After that another solution is considered, and if that solution is still not connected then the procedure is repeated. The number of constraints similar to (2.3) can grow up to a number that will make the process too slow. In order to change that, polyhedral combinatorics are used (we refer to [21] for more detail on polyhedral combinatorics). The idea behind polyhedral combinatorics is to reduce the set of inequalities to certain inequalities (called facet defining) that are considered the essential ones, since they cover the solution space and cannot be reduced. This will reduce the number of constraints needed, since those inequalities still cover the solution space. More formally, facet defining inequalities can be defined in the following way[4]:

**Definition 11** Considering the set \( L \setminus K \) to be divided into connected components, being \( C_s \) the component including site \( s \) and \( K_s \) the sites that are adjacent to any site in \( C_s \), \( s \in L \setminus K \). An inequality is facet defining if the following conditions met:
1. \( \forall k \in K, L \setminus K \cup \{k\} \) has a connected cover.

2. \( \forall s \in L \setminus K \) exists \( k \in K_s \) such that \( (L \setminus K)\{s\} \cup \{k\} \) contains a connected cover.

As an example, considering equation (2.3), a connected cover cannot be found in \( L \setminus K \cup \{k\} \) if \( k = 1 \), as we get \( \{1, 4, 8\} \) and a connected cover cannot be found from this subset. This means that equation (2.3) is not facet defining, and therefore can be reduced in some way. The inequality

\[
x_2 + x_3 + x_5 + x_6 + x_7 + x_9 \geq 1
\]

implies (2.3), therefore being a stronger inequality. In the same way, other sites can be dropped until we get a minimal set that satisfies condition (i) in definition 11. In this case, the following inequality would meet the requirements for (i):

\[
x_5 + x_6 \geq 1
\]

This inequality also satisfies (ii). However, when it is not the case, there is a generic way to work around that. To exemplify, consider the graph representing a land in figure 2.8:

![Graph](image)

Figure 2.8: Graph in which species distribution leads to \( K = \{2, 4, 7\} \)

In this distribution, the chosen value for \( K \) is \( K = \{2, 4, 7\} \), since it meets the condition of not having a connected cover in \( V \setminus K \) and it satisfies condition (i) to be facet defining with the equation:

\[
x_2 + x_4 + x_7 \geq 1
\]

However, for condition (ii) this inequality fails if \( s = 6 \), since it is on the subset \( C_s = \{5, 6\} \) and no cover can be found with \( k \in K_s = \{4, 7\} \), as shown in figures 2.9 and 2.10, where the delimited area is the set of sites that are excluded, and as can be seen there is no connected cover amongst the others.

This means that if we want to choose site 6, then we must also choose one of the sites 4 or 7, as can be represented with the inequality

\[
x_4 + x_7 \geq x_6.
\]
Considering the inequalities (2.6) and (2.7), the following inequality is also valid:

\[ 2x_2 + x_4 + x_7 + x_6 \geq 2. \]  

(2.8)

Implying that either site 2 is used (which is a site from \( K \backslash K_s \)) or, if \( x_6 \) is used, then the inequality represents the same as (2.6); otherwise it forces the use of \( x_4 \) and \( x_7 \). In a generic way if condition (i) is not met, then it is possible to choose a proper subset of \( K \) to meet the requirements. For condition (ii), the following equations can be used in a generic way:

\[ \sum_{i \in K_s} x_i \geq x_s \]  

(2.9)

\[ \sum_{i \in K_s} x_i + 2 \sum_{i \in K \backslash K_s} x_i + x_s \geq 2 \]  

(2.10)

The algorithm used to solve this problem does multiple iterations, each one solving a relaxation, producing a cover at the end. It starts by checking if the current cover is connected. If it is not, then \( K \) is changed until it meets the condition (i) for being facet defining. Then condition (ii) is checked. If it holds, then the constraint is added to the constraints in the relaxation and then a new solution is produced. However in the case that the second condition does not hold, constraints (2.8) and (2.9) are considered, but before adding them the algorithm tries to create stronger versions of those inequalities. The resulting covers from each of the steps of the algorithms are a sequence of lower bonds for the optimal
value of the connected cover.

2.3.2 Wildlife Corridor

Dilkina et al. [9] deals with a different problem variant, denominated Wildlife Corridor. 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). Three different approaches were made to solve this problem and all of them represent the problem as a graph $G=(V,E)$ as shown in section 2.2. The first approach is based on the idea of a single commodity flow, where the flow conservation from the root to the selected nodes ensures that they are all connected[6]. Based on the flow conservation idea, there is an alternative way to solve the problem, where instead of having only one flow, the network is considered to have $K$ different flows, one for each selected node. A distinct approach is based on considering the problem as a Steiner Tree problem, where node costs and profit are considered, and use the Dantzig-Fulkerson-Johnson (DFJ) formulation[8], where the decision variables are the edges $E$ instead of the nodes $V$. All of the approaches were addressed using Integer Linear Programming.

There is some terminology common to all the problems. 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 subgraph $G' = \{V', E'\}$, where $V' \subseteq V$ and $E' \subseteq E$, in which $V'$ contains all of the terminal nodes $T \subseteq V'$ and the subgraph $G'$ 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:

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

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

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

$$\text{maximize} \sum_{i \in V} u_i x_i \quad (2.11)$$

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:

13
\[ \sum_{i \in V} c_i x_i \leq C \quad (2.12) \]

Those restrictions represent the basic restrictions of the problem except for the connectivity issue. For the connectivity there are three different approaches that will be explained in the next sub-sections, giving more detail to each of the approaches used.

**Single Commodity Flow**

One of the proposed approaches by some authors[6, 18] is to consider the problem of connectivity 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:

\[ x_0 + y_{0r} = f \quad (2.13) \]
\[ y_{ij} \leq f.x_j, \quad \forall (i, j) \in D \quad (2.14) \]
\[ \sum_{i:(i,j) \in A} y_{ij} = x_j + \sum_{i:(j,i) \in A} y_{ji}, \quad \forall j \in V \quad (2.15) \]
\[ \sum_{j \in V} x_j = y_{0r} \quad (2.16) \]
\[ y_{ij} \geq 0, \quad \forall (i, j) \in D \cup (0, r) \quad (2.17) \]
\[ x_0 \geq 0, \quad \forall (i, j) \in D \quad (2.18) \]

In the constraint (2.13), 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 (2.14) is used, since if the incoming flow is greater than zero, \(x_j\) must be assigned value one, otherwise the constraint would not be satisfied. Constraint
(2.15) 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 \( y_{ji} \) plus the flow in the node \( x_i \) itself. Constraint (2.16) enforces that the flow injected in the system is equal to the sum of the absorbed flows by each node. Finally constraints (2.17) and (2.18) are used to force the flow values to positive numbers. With these constraints it is ensured that the nodes are all connected. As an example, consider figure 2.11. Here the residual flow \( x_0 \) is 5, since there are a total of 10 sites. The sites marked with T are the mandatory sites (which are the terminal nodes in the graph). The remaining 5 units are injected into the network, and each of the selected sites consumes one unit of flow. The flow conservation is easy to observe, since for any given selected site, the incoming flow equals the sum of the outgoing flows, plus one.

![Figure 2.11: Flow demonstration on a Single Commodity Flow approach](image)

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.

**Multi-Commodity Flow**

This approach also uses the flow idea to ensure connectivity. However, instead of having a single flow, there is a different type of flow (a different commodity) for each of the nodes. Each node will potentially receive a unit of its type of flow from a root \( r \), which is chosen amongst the terminal nodes \( T \in V \). However, other types of flow may pass through a node, and those are routed to the corresponding node. For example if a node of type \( j \) receives a flow of type \( i \), it will not consume it and will route it to the next neighbour. If the same node of type \( j \) receives a flow of type \( j \), then it will be a sink for that flow. The neighbours of a node \( i \) will be designated as \( \delta(i) \), where \( \delta(i) = \{j|(i,j) \in A\} \). In the single commodity flow approach, edges had a variable to represent the flow on them. In this approach it is also needed to identify the type of the flow, and multiple types may be at the same time on an edge, so it is needed to create a variable \( y_{kij} > 0 \) that represents the amount of flow of type \( k \) that is being carried on the edge. Recall that \( x_i \) represents the inclusion of site \( i \) in the solution. The constraints used for this problem are the following:

\[
y_{kij} \leq x_i \quad \forall k, \forall i \in V - r, \forall j \in \delta(i) \quad (2.19)
\]
\[ y_{ki} \leq x_j \quad \forall k, \forall i \in V - r, \forall j \in \delta(i) \] (2.20)

\[ y_{kj} \geq 0 \quad \forall k \in V - r, \forall (i, j) \in A \] (2.21)

\[ \sum_{j \in \delta(j)} y_{kj} = 0, \quad \forall k \in V - r \] (2.22)

\[ \sum_{j \in \delta(k)} y_{kj} = x_k, \quad \forall k \in V - r \] (2.23)

\[ \sum_{j \in \delta(k)} y_{kj} = 0, \quad \forall k \in V - r \] (2.24)

\[ \sum_{j \in \delta(i)} y_{kj} = \sum_{i \in \delta(j)} y_{kji}, \quad \forall k, \forall i \in V - r, i \neq k \] (2.25)

Constraints (2.19) and (2.20) guarantee that the capacity of a node is either 0 (if not selected) or 1. Constraint (2.22) prevents the root from receiving any kind of flow, since it forces that every variable with flow into the root has value 0. Constraints (2.23) and (2.24) impose that if a node is selected, then it is a sink for its type of flow, and that this node cannot have outgoing flow of its own type (meaning it will take all the available flow of its type). Finally constraint (2.25) is used to ensure flow conservation, like in single commodity flow, but for each type of flow instead for only one type. This encoding will require more variables, \( 2|E|(|V| - 1) \), since it is now needed one variable for each type of commodity minus one, since the root will not need a specific type.

**Dantzig-Fulkerson-Johnson**

For this approach, a Steiner tree is used, and the goal is to find a set of edges that result into a directed Steiner tree covering all of the terminal nodes, making the solution a single path from the root to these nodes. Unlike the previous approaches, this one models the problem in a different way: instead of using variables \( x_i \) for the nodes, it uses variables \( y_{ij} \) for the edges, so there are as many variables as \( |D| \).

Dantzig-Fulkerson-Johnson have used an approach to deal with the connectivity when giving one solution for the *Travel Salesman Problem* [8], which will be adapted in this solution. The constraints (2.12) and (2.11) must be redefined as follows, in order to keep the same functionality adapted to the use of edge variables instead of node variables, i.e maximize the utility and do not go over a cost, respectively:

\[
\text{maximize} \sum_{i \in V} \left( u \sum_{j \in \delta(i)} y_{ji} \right) \] (2.26)

\[
\sum_{i \in V} \left( c \sum_{j \in \delta(i)} y_{ji} \right) \leq C \] (2.27)

The constraints to ensure the connectivity between the nodes are the following:
\[
\sum_{j \in \delta(i)} y_{ji} = 1, \quad \forall i \in T \tag{2.28}
\]

\[
\sum_{j \in \delta(i)} y_{ji} \leq 1, \quad \forall i \in V - T \tag{2.29}
\]

\[
y_{ij} + y_{ji} \leq 1, \quad \forall i \in V - T, \forall j \in \delta(i) - r \tag{2.30}
\]

\[
\sum_{(i,j) \in A, j \in S, i \in V - S} y_{ij} \geq \sum_{j \in \delta(i)} y_{jk} \quad \forall k \in S \tag{2.31}
\]

The constraint (2.28) ensures that every terminal node is selected, since it forces that the sum of the incoming edges is equal to 1 (therefore forcing that one of them is actually selected). Since this approach is based on a tree concept, every node can have at most one incoming edge, as it is done in (2.29). Forcing this removes the possibility of cycles. This also means that we cannot have edges in more than one direction, so we must have a constraint that forbids more than one edge between two nodes, as it is done in constraint (2.30). Constraint (2.31) ensures the connectivity of the subgraph, where \( S \) is a sub-set of the set of nodes \( V \), that is created by subtour elimination (a concept used by Dantzig-Fulkerson-Johnson; we refer the reader to [8] for more detailed information). Subtour elimination is used to exclude some assignments that will not lead to a connected subgraph. An example of a resulting graph can be seen in figure 2.12. The arrows represent the directed edges, and the sites marked with T are the terminal nodes. The resulting graph is a tree, since there are no cycles.

![Dantzig-Fulkerson-Johnson directed tree on a set of lands](image)

Figure 2.12: Dantzig-Fulkerson-Johnson directed tree on a set of lands

Although this encoding creates yet more constraints, the used algorithm helps to deal with them. First, for the Dantzig-Folkerson-Johnson encoding algorithm, the variables \( y_{ij} \) that previously could only take \{0, 1\} as values, will now be continuous variables \( y_{ij} \in [0, 1] \). With this, a relaxed version of the problem can be solved first and then the result is used either as a solution (if it is an integral solution) or as a new constraint to help to narrow down the problem. The basic idea is there is no need to solve all the integral constraints from scratch, but instead some new constraints are "learned" and help to reduce the search space smaller, since they are always closer to the solution. The complete algorithm can be found at [9].
Results

To test the three different approaches the authors used a randomly generated instance (10 by 10), with 100 nodes in which 3 were terminal and a uniform cost and utility distribution, and two real world instances with different sizes (one with 242 nodes and the other with 3299 nodes). Before giving the results, the integrality gap and budget slack must be introduced. Considering the optimal objective of the relaxed solution as $R$ and the optimal objective of the solution of the problem as $S$, the integrality gap $I_g$ is given by

$$I_g = \frac{R - S}{S}$$

The budget slack represents how much the minimum cost value for a given sample was increased. For example, if the minimum cost to connect the terminal nodes is 10 and the budget slack is 10%, then every solution that has a cost below 11 is considered, since $10 + (0.1 \times 10) = 11$. The tests were made by changing the budget slack and observing the respective the time and integrality gap. For the randomly generated instance, the result are shown in figure 2.13, originally available in [9].

![Figure 2.13: Results on a randomly generated instance of wildlife corridor problem with 100 nodes](image)

For small budget slacks, the DFJ encoding gets much better integrality gaps, which come at the cost of running time, while the single commodity flow approach gets the best running time for a solution with a bigger integrality gap. The multi commodity flow approach has a considerable running time and it does not gives a solution as good as DFJ. When comparing the single commodity flow with the DFJ in the real world instances (with more nodes) in figure 2.14(also taken from [9]), it can be seen that the single commodity flow runs fast on very tight budget slacks, but it gets much slower when the slack is increased up to a certain point. On the other hand the DFJ encoding has a more constant increase on the runtime when increasing the budget slack.
Figure 2.14: Results on a real life instance of wildlife corridor problem with 242 nodes
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[16]. Informally, a stable model is a set of positive literals that make a program true. Before giving the formal definition for Stable Models, we must introduce first the concept of Answer Set Program. An Answer Set Program is built with atoms, literals and rules[2]. Atoms are in the form \( \text{atom}(V_1, ..., V_N) \) where \( \text{atom} \) is the atom name and \( N \) is its arity. Literals are either positive atoms or negated atoms (with strong or weak negation, as we will explain later in this section).

The rules are the basic way to generate new atoms. Formally, a rule is on the form

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

(3.1)

where \( \text{Atom} \) is an atom that is true if all of the literals (\( \text{Literal}_1 \) to \( \text{Literal}_N \)) are true. As an example

\[
p(A) \leftarrow q(A), r(A)
\]  

(3.2)

is a rule where \( p(A) \) is the head of the rule (which can only contain at most one literal, or an aggregate as we will explain later on this section) and \( q(A), r(A) \) is the body (which can contain any number of literals). Considering \( P \) as a program with rules on the form (3.1), where every variable was replaced by its ground instances (meaning that they have no variables), \( A \) as the set of atoms from \( P \), a stable model can be defined as:

**Definition 12** Let \( P_A \) be the program resulting from applying the following rules to \( P \):

1. Delete every rule that has in its body a negative literal \( \neg L \) where \( L \in A \).

2. Delete, for the remaining rules, all of their negative literals.

If \( P_A \) is equal to \( A \), then \( A \) is a stable model of \( P \).
A rule without a body is called a fact (and usually the ← is omitted). If we add the facts

\[ q(1).r(1). \]  \tag{3.3}

then the rule (3.2) would derive \( p(1) \). Thus, the stable model for that example would be

\[ \{ p(1), q(1), r(1) \} \]  \tag{3.4}

since \( q(1) \) and \( r(1) \) are facts and they make the body of the rule (3.2) true, its head is generated. If we want to give as facts \( q(1), q(2), \) and so on until \( q(n) \), we can use the shorter representation \( \{ q(1..n) \} \).

A rule without a head is called a constraint, and it is used to restrict the stable models returned by the program. If we add the constraint

\[ ← p(1). \]  \tag{3.5}

to the rules (3.2) and (3.3), the stable model (3.4) would not be valid, since it contains \( p(1) \), and the constraint (3.5) does not allow the appearance of \( p(1) \) in the answer sets. Another way to constrain the rule generation is to use the operator : . As an example, consider the following rule:

\[ p(A) : q(A) ← \text{Literal}_1, ..., \text{Literal}_N \]  \tag{3.6}

In this rule, assuming that the body is true, the program only generates \( p(A) \) if \( A \) also holds for \( q(A) \). This means that only the ground terms that hold \( q(A) \) are used to generate \( p(A) \).

In answer set programming is also possible to use choice rules. Choice rules (also called cardinality constraints) in the head are rules that allow the generation of multiple literals in one rule. Considering the following program

\[ p(1). \]  \tag{3.7}

\[ \{ q(A), r(A) \} ← p(A) \]  \tag{3.8}

This program would return four different stable models: \( \{ p(1) \}, \{ p(1), r(1) \}, \{ p(1), q(1) \} \) and \( \{ p(1), q(1), r(1) \} \). Since we had not limited the choices cardinality, the program can generate none of the literals, either one of them, or all of them. It is possible to restrict to minimum number of literals (adding a number to the left) and the maximum number of literals (adding a number to the right) of a choice rule. If we replace the rule (3.8) with

\[ 1 \{ q(A), r(A) \} 1 ← p(A) \]  \tag{3.9}
the program would only generate two stable models: \{p(1), r(1)\} and \{p(1), q(1)\}, since the answer set needs to have at least and at most one of the literals. Choice rules can also appear on the right side of a rule. In that case, it means that you need to have at least (or at most) a given number of the literals in order to satisfy the body of the rule. More formally, a choice rule is a constraint of the form

\[ n \#count\{a, b, c\} m \]  

(3.10)

where \#count is true if the number of distinct literals are between \(n\) and \(m\). In ASP is also possible to use weighted constraints. The weighted constraints appear in the form

\[ n \#sum\{a = v, b = w\} m \]  

(3.11)

such that \(v\) and \(w\) are real numbers. This is similar to \#count, but in \#count the weight of every literal is assumed to be 1. Here each literal has its own weight, namely \(a\) has weight \(v\) and \(b\) has weight \(w\), and the sum of all weights must not be smaller than \(n\) nor larger than \(m\). The operators \#count and \#sum are used in what is called an aggregate[11]. In a generic way, an aggregate is defined as

\[ n \#op\{Literal_1 = Weight_1, ..., Literal_N = Weight_N\} m \]  

(3.12)

where \#op is an operator. If \#op is omitted, then the default operator is used (\#sum) and we have a weighted constraint. If the weights (\(Weight_1\) to \(Weight_N\)) are omitted, then it is assumed that the literals have weight 1. There are also the aggregates \#min and \#max, that are used to check if the minimum (or maximum, respectively) weight in the aggregate is between \(n\) and \(m\).

Sometimes it is desirable to get only the best result according to some criterion. ASP supports the use of the \#maximize and \#minimize functions, in order to maximize (or minimize) the returned answer set according to some priority. An optimization statement comes in the form:

\[ \#opt[Literal_1 = Weight_1 @ Priority_1, ..., Literal_N = Weight_N @ Priority_N] \]

where the weights can be omitted. The priority defines the criteria for the optimization. As an example, consider the following program:

\[ 1\{\text{car("renault"), car("ferrari"), car("mercedes")}\} \]
\[ \text{cost("renault", 5).cost("ferrari", 50).cost("mercedes", 20).} \]
\[ \#\text{maximize}[\text{car}(X) : \text{speed}(X, Y) = Y @ 2]. \]
\[ \#\text{maximize}[\text{car}(X) : \text{speed}(X, Y) : \text{cost}(X, Z) = (Z/Y) @ 1]. \]
where we have three car brands (Renault, Ferrari and Mercedes), their maximum speeds and their costs. The priorities 1..n are made from the least priority (1) to the one with the most priority (n). In the given example, we prioritize the maximum speed of the car, meaning that this is the most relevant feature in our program. However, if two cars are tied for the most speed, then we have a second priority that will consider the car that has the best cost/speed relation. In this particular example, the first optimization would find two tied values, car("ferrari") and car("mercedes"). However, since the car("mercedes") has a better cost/speed ratio, the program would return {car("mercedes")} as the optimum answer set.

So far we have worked only with positive literals, since the negation notion is not trivial. In ASP there are two kinds of negation: strong negation and weak negation[20]. For a literal p(1) the weak negation representation is not p(1) and the strong negation is represented as ¬p(1). The weak negation is the most used negation in ASP. Since in ASP only the atoms used in the program are known to be true, everything else is not known. However the weak negation assumes that everything that is not in the program is false. For example in the following program

\[ p(1). \] (3.13)

\[ q(A) ← p(A), not \ s(A). \] (3.14)

we would get the answer set \{p(1), q(1)\}, since the literal s(1) never appears in the program, it is assumed to be false, making the body of the rule (3.14) true. However, it is not really known if s(1) is true or false. The weak negation assumes it is false because there is no evidence that is true. On the other hand, the strong negation is used when it is known that something is false. As an example

\[ p(1). \] (3.15)

\[ q(A) ← p(A), ¬s(A). \] (3.16)

would have as answer set \{p(1)\}, since q(1) could not be derived, as we specifically said that for q(A) to exist, it must be known that s(A) does not exist. Not knowing if it exists or not is not enough. In order to derive q(1), ¬s(1) would have to be a given fact, which would generate the answer set \{p(1), q(1), ¬s(1)\}.

The ASP process can be split into two different tasks: the grounding process (made by grounders like lparsel and gringo[14]) and the solving process (made by solvers like smodels or clasp). In order to understand the grounding process, we must first introduce the Herbrand Universe notion.

1http://www.tcs.hut.fi/Software/smodels/
Definition 13 The Herbrand universe $H$ of a program $P$, denoted as $H_P$, is the set of all constants in the program $P$. If no constants exists in $P$, then $H_P$ has an arbitrary constant $c$.

The set of all ground atoms of a program, generated by the predicates in $P$ and the terms in $H$ is called the Herbrand Base of $P$. Ground terms are terms that have no variables, meaning that the grounder has the job of assigning every possible variable from the domain to all the terms. The grounder also needs to ensure that every rule in the program is safe. Let $H_r$ be the head of a rule $r$, $B_r^{\text{pos}}$ be the set of positive literals in the body and $B_r^{\text{neg}}$ be the set of negative literals on the body. The definition for a safe rule is as defined below:[19]:

Definition 14 A rule $r$ is safe if for every variable $X \in H_r \cup B_r^{\text{neg}}$, $X$ is also in $B_r^{\text{pos}}$.

The grounders usually produce the output in a format that the solver can read. The solving process is done by a solver (like smodels for lparse and clasp for gringo), and consists in finding the stable models for the program. Solvers like Clasp have many options that can be used to change the way it handles the search space, which is useful for some problems. Exploring the Clasp options can change the solving time by a huge amount, so it is always recommended to do it. Some of the options include changing the used heuristic for the search, changing the number of conflicts before the solver restarts the search and even some SAT preprocessing can be done (we refer the reader to [15] for more information on preprocessing in ASP). Some of the options and a short description can be found in table 3.1 (the complete set of options can be found in [11]).

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- -- trans -- ext = all</td>
<td>Transforms choice or weight rules (according to the option used) into simple rules.</td>
</tr>
<tr>
<td>-- sat prepro = yes</td>
<td>Runs Sat preprocessing, with $n_1$ being the maximum iterations, $n_2$ the cut-off for variable elimination and $n_3$ the maximum time (in seconds).</td>
</tr>
<tr>
<td>-- heuristic = Berkmin</td>
<td>Determines the decision heuristic to be used, with Berkmin being the default.</td>
</tr>
<tr>
<td>-- rand freq = p</td>
<td>Used to give the probability $p$ for the program make a random decision instead of an heuristic based one.</td>
</tr>
<tr>
<td>-- restarts, -rn1, n2, n3</td>
<td>Used to chance the restart policy. $n_1$, $n_2$ and $n_3$ are used to parametrize after how many conflicts the search should restart.</td>
</tr>
<tr>
<td>-- brave</td>
<td>Used to compute the union of all answer sets.</td>
</tr>
<tr>
<td>-- cautious</td>
<td>Used to compute the interception of all answer sets.</td>
</tr>
<tr>
<td>-- search limit = n, m</td>
<td>Stop searching after $n$ conflicts or $m$ restart, whichever comes first.</td>
</tr>
</tbody>
</table>

Table 3.1: Clasp options

After the solving process, the solver gives either all or a specific number of the answer sets for the program (or unsatisfied if there is none), and some options can be used to get more information on the
solving process, like the number of conflicts, the number of restarts or the solving time.

Since most programs return only some literals, it might be useful to represent those literals graphically in order to quickly understand the solution, specially for problems that are graphically represented themselves, for example games like magic squares\(^2\), where there are some positions and some numbers, or the Maximum Density Still Life problem\(^3\). For this purpose there is a tool called ASPviz\(^5\). This tool uses the ASP semantic and allows the users to draw some forms as output. It provides as basic functions the ability to draw ellipsis, squares and characters, to define some colours, fill positions with some colour and even use some images. With those primitives we can draw a very simple scheme that might be enough to represent the solution of the program, allowing a quick view on that solution. Some of the primitives and a respective short description are given in appendix A. We refer the reader to [5] and [1] for more detailed usage and functions.

In section 3.1 we will give an example of ASPviz, in the context of a specific problem. The ASPviz program is given to the grounder together with the program itself, so that clasp generates answers sets that include the ASPviz primitives. After that, ASPviz uses the answer sets as input to draw the respective image, according to the primitives it finds on that answer sets.

### 3.1 A practical example: n-queens

Writing and reading a program in answer set programming is fairly easy, since it feels a lot like writing the rules for a game. As a classic example in ASP problems\(^{12}\) we will show a n-Queens encoding. The n-Queens problem consists into putting \(n\) queens in a board with the size \(n \times n\) in such way that no queen attacks each other (meaning two queens cannot be in the same row, column or diagonal). In this example we are using the notation used for the implementation itself, where \(\rightarrow\) represents the implication symbol, \(\leftarrow\). A simple straightforward way to encode this problem would be:

\[
\text{position}(1..n, 1..n).
\] (3.17)

\[
n\{\text{queen}(R, C) : \text{position}(R, C)\}n.
\] (3.18)

\[
\rightarrow \text{queen}(R, C) \leftarrow \text{queen}(R, C2), C \neq C2.
\] (3.19)

\[
\rightarrow \text{queen}(R, C) \leftarrow \text{queen}(R2, C), R \neq R2.
\] (3.20)

\(^2\)http://www.cs.st-andrews.ac.uk/ ianm/CSPLib/prob/prob019/spec.html

\(^3\)http://www.cs.st-andrews.ac.uk/ ianm/CSPLib/prob/prob032/spec.html
−queen(R,C), queen(R2,C2), R ! = R2, C ! = C2, \#abs(R − R2) == \#abs(C − C2).

The rule (3.17) generates the board with \( n \) positions. Rule (3.18) ensures that we have exactly \( n \) queens in the board, since the choice rule is lower bounded and upper bounded by \( n \), and each queen is identified by its position on the board, row and column, respectively. Rule (3.19) ensures that no pair of queens is on the same row, by saying that we cannot have two queens with different columns in the same row. In an analogous way, rule (3.20) prevents two queens from being on the same column. The last rule (3.21) prevents two queens from being in the same diagonal, by ensuring that it is impossible to have two distinct queens such that the row and column difference between them is the same. This is an easy to understand approach since we give as constraints exactly the constraints of the problem itself, as we have an \( n \times n \) board (3.17) with \( n \) queens (3.18) that cannot be on the same row (3.19), column (3.20) or diagonal (3.21) of each other.

To run the previous example in a grounder and solver like gringo and clasp, respectively, we might want to add the following rules to the program:

\[
\text{#hide.}\text{#showqueen(\_\_, \_\_).}
\]

Those rules allow us to see as a resulting set only the literals \((queen(R,C))\), which are the ones that we are interested to know. If we run the program with those rules and giving \( n=4 \), we get the answer sets \{queen(4,3), queen(3,1), queen(2,4), queen(1,2)\} and \{queen(4,2), queen(3,4), queen(2,1), queen(1,3)\}, which correspond to the boards in 3.1. The figure on the left is the first answer set and the figure on the right is the second answer set.

Figure 3.1: Two solutions for the 4-queens problem instance

Figure 3.1 was drawn by hand, and since it is a solution for the 4-queens problem it is really easy to read the output and get the respective board. However, for bigger instances it might not be desirable to read the whole output, and that is where the ASPviz becomes handy. We can easily draw the output with ASPviz, since we can draw the positions as white squares and the queens as blue circles. The program
to draw the boards is the following:

\[
\text{brush}(\text{standard}).\text{brush}_\text{color}(\text{standard}, \text{black}).\text{brush}_\text{width}(\text{standard}, 1). \quad (3.23)
\]

\[
\text{fill}_\text{ellipse}(\text{standard}, \text{blue}, p((X - 1) * 40 + 20, (Y - 1) * 40 + 20), 40, 40) : -\text{queen}(X, Y). \quad (3.24)
\]

\[
\text{fill}_\text{rect}(\text{standard}, \text{white}, p((X - 1) * 40, (Y - 1) * 40), 40, 40) : -\text{position}(X, Y). \quad (3.25)
\]

The rules in (3.23) define the brush colour and width. Rule (3.24) draws the queens as circles, since in the head of the rule are the ellipsis and in the body the queens. The last rule (3.25) draws the board itself, since it draws a square for each literal \text{position}(X, Y). The result of this program combined with the \textit{n-queens} program can be seen in figure 3.2.

![Figure 3.2: ASPviz on two solutions for the 4-queens problem instance](image)

The program we just created is intuitive and easy to read. However, it is far from being the best approach, since it generates unnecessary literals. A good modeling of an ASP program is essential, since it can reduce the total time to run it (grounding and solving). There are no standards in terms of program optimization, since it depends on the problem itself. However, there are some guidelines that can be followed and usually (but not always) improve the program quality. Martin Gebser et al. [12] give some optimizations for the \textit{n-queens} problem. For example, the literal position is not really needed, since it is being used to define a place where the queens are placed into, but the queens also have the position on them, making the position a redundant literal. They define the queen placement as follow:

\[
n\{\text{queen}(1..n, 1..n)\}n \quad (3.26)
\]

meaning that we have exactly \( n \) queens and each queen has a position between 1 and \( n \) in row and column. By doing this, we can completely remove the \text{position}(X, Y) literals, thus generating less ground terms, speeding up de grounding process. As another typical example also referred in Gebser’s work, we will also explain symmetry breaking. Rules (3.19) and (3.20) generate some symmetric ground terms, for
example: $-queen(2,1), queen(2,4)$. and $-queen(2,4), queen(2,1)$. Those constraints are redundant, since they do not allow the placement of the same queens. In bigger instances, this can be a problem for the grounding process. In order to prevent the generation of such constraints, the $C \neq C2$ can be replaced by $C < C2$ for rule (3.19) and $R \neq R2$ can be replaced by $R < R2$ for rule (3.20), since this will still ensure the difference and eliminates the symmetric constraints.

There are more ways to optimize each program, but generally based on the problem itself, on which some particular rules have benefits (that can even be problematic to some other problems) and others just make the solving time longer.
Since we have two different problem variants we will deal with each of them individually, giving details about every step of the encoding. We will provide an illustrative example for each approach so we can show the impact of those steps in the problem resolution. Since the first step of solving problems with ASP is grounding, where all the rules are replaced with the ground terms (rules with no variables) we will give special attention to how the chosen lines of code impact the grounding for the program. In section 4.1 we will cover the wildlife corridor variant and in section 4.2 we will approach the minimum connected cover approach. In appendix A we have the ASPviz implementation for the approaches described on this chapter.

4.1 Wildlife Corridor

In this section we will approach the variant where we must connect a certain number of mandatory sites, under a certain cost, with maximum utility. To address this problem in answer set programming, we must at first take a look at what do we have as input: we do know the shape of the area, so we know how many sites exist and how they are connected, the cost and utility of each site and the sites that are mandatory. Now that we know what is given, we must find a reasonable way to represent the data in answer set programming. 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
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 the next sections we will give two approaches to solve this problem. In both approaches we will use the area in figure 4.1 as an example, in order to illustrate every step of the program. Each square represents one site, where the site cost is represented in red (upper right corner), the site utility is represented in green (upper left corner) and the site name is represented by a black number in the middle. The terminal sites are the sites 1, 3 and 11. The complete input for this area can be found in the appendix B.

![Figure 4.1: 4 by 4 area with costs and utilities represented](image)

### 4.1.1 First Approach

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, because the constraints used in the program are closely related at those rules. We will now take a closer look to each rule and represent it in ASP, so in the end we have a program (that might not be optimized yet) capable of solving correctly the problem instances. But before we look at the rules, we must have a way to say that certain sites are the sites that we want to use in our solution. Such sites will be referred as chosen sites. 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:

$$\text{chosen}(N) : \text{node}(N).$$  \hfill (4.1)

The meaning of this line is: for each site $N$, we can have either 0 or 1 $\text{chosen}(N)$. This means that every site can potentially be a chosen site, and the constraints will take care of ensuring that will only be chosen sites that do not violate any constraint. 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:

$$\text{edge}(N, N^2) : -\text{edge}(N^2, N).$$  \hfill (4.2)

With this, the program will look at every edge given at the input, and generate the edge in the opposite direction. Although this is not strictly necessary (we can always take in consideration that the edge might only be represented in a single direction in every rule), it makes the other rules simpler and easier to read. This decision also goes in the direction of the previous studied approaches that considered the problem as a bidirectional graph.

After those introductory lines, we can now focus on the program restrictions. In order to make this easier to understand, we will use the example in the figure 4.1 to illustrate the ground terms generated with every rule. Although the grounder by default generates data that is not easy to understand by humans, an option can be given when grounding in order to get a human readable version of the rules generated, in order to understand the data that is given as input to the solver. The restriction $RI$ 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:

$$: -\text{terminal}(N), \text{notchosen}(N).$$  \hfill (4.3)

Line 4.3 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. With this rule, we ensure that there is no mandatory site that is not on the final solution. In the given example, it will generate the three following restrictions:

\[ \neg \text{notchosen}(3). \quad \neg \text{notchosen}(11). \quad \neg \text{notchosen}(1). \]

With those restrictions the program will never find an answer set that does not contain the sites 1, 3 and 11 as chosen sites. Since these are the terminal sites, we ensure that they will be included. Rule R2 restricts the sites that can be chosen, since we cannot simply pick an isolated site that has a low cost and high utility just to make our solution better. We recall that this optimization is related to the wildlife corridor, where having isolated species is not an option. 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. This predicate is used to say that site \( N \) is the starting point to consider connectivity, in such a way that every other site must be connected either to the start site or to a site that is somehow connected to it. To represent the connectivity in ASP we use a recursive solution, shown in rules 4.4 - 4.6.

\[
\text{poss}(N) : \neg \text{start}(N). \quad (4.4)
\]

\[
\text{poss}(N) : \neg \text{poss}(M), \text{edge}(N, M), \text{chosen}(M). \quad (4.5)
\]

\[
\neg \text{chosen}(N), \neg \text{poss}(N). \quad (4.6)
\]

Rule 4.4 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 4.5 is the recursive rule, where we restrict the possibilities to nodes that are adjacent to possible nodes that are chosen nodes. Since every terminal node is a chosen node and the start node is both a possible node and a terminal node, a site \( A \) adjacent to the start node can always be chosen. From that, the next site \( B \) to be chosen can be either adjacent to the start node, or to the \( A \) node, because both are possible and chosen, and if \( B \) is adjacent to any of them, then there is an edge from \( B \) to either \( A \) or start (or even both). Rule 4.6 ensures that there is no site that is not a possibility from the adjacency, making it impossible to have isolated sites. Using the example, consider the start node

\[ \text{start}(1). \]
It will generate the following rules from 4.4:

\[
\text{poss}(5) : \neg \text{chosen}(1).
\]

\[
\text{poss}(2) : \neg \text{chosen}(1).
\]

\[
\text{poss}(6) : \neg \text{chosen}(1).
\]

since the node 1 is the start node and the nodes 2, 5 and 6 are adjacent to it. Rule 4.5 will generate several similar rules related to the connectivity, which can be read as “if site A is a chosen site, then you can choose B”. For the example, the grounder would generate the following line:

\[
\text{poss}(10) : \neg \text{poss}(5), \text{chosen}(5).
\]

This line means that the site 10 could be chosen if it was decided that site 5 was part of the solution. We also know that site 5 could have been chosen from 4.4, as shown above, so all that matters is whether the site 5 is actually part of the solution. For every adjacency, there will be a similar rule. With these rules all the chosen sites are connected, since there is no way to choose a site that is not somehow already connected to the start site. Rule 4.6 will generate constraints such as

\[
\neg \text{chosen}(2), \neg \text{poss}(2).
\]

\[
\neg \text{chosen}(3), \neg \text{poss}(3).
\]

for every site, excluding the start site (because it is already a possibility). 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. To represent this rule in answer set programming we will need to somehow represent the sum of the costs in the sites that are being considered. Since that information is not easily available, in our first approach we decided to introduce yet another recursive rule. Since we cannot give any extra input information for this case to get a start value for our recursion we do the following. 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 4.7 and 4.8:

\[
\text{sum}(1, C) : \neg \text{cost}(1, C), \text{chosen}(1).
\] (4.7)

\[
\text{sum}(1, 0) : \neg \text{notchosen}(1).
\] (4.8)

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, 4.9 and 4.10:

\[
\text{sum}(N, C + C2) : \neg \text{sum}(N - 1, C), \text{cost}(N, C2), \text{chosen}(N). \tag{4.9}
\]

\[
\text{sum}(N, C) : \neg \text{sum}(N - 1, C), \text{notchosen}(N), \text{node}(N). \tag{4.10}
\]

Rule 4.7 to 4.10 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 4.11 represents the ASP constraint needed to ensure that no solutions have a cost greater than the given cost.

\[
: \neg \text{sum}(N, C), \text{node}(N), C > \text{max}. \tag{4.11}
\]

Going back to our example, lines 4.7 and 4.8 will generate the following ground terms

\[
\text{sum}(1, 3) : \neg \text{chosen}(1).
\]

\[
\text{sum}(1, 0) : \neg \text{notchosen}(1).
\]

since the cost of the site 1 is 3. If site 1 is chosen, the sum until site 1 will be 3. Although the grounder cannot make that assumption, we intuitively know that it will be chosen, because it is a terminal one. If site 1 is not chosen, the sum will be 0. Lines 4.9 and 4.10 will generate a large number of rules in the grounding. For example, the following lines will be generated:

\[
\text{sum}(2, 3) : \neg \text{sum}(1, 0), \text{chosen}(2).
\]

\[
\text{sum}(2, 6) : \neg \text{sum}(1, 3), \text{chosen}(2).
\]

\[
\text{sum}(2, 0) : \neg \text{sum}(1, 0), \text{notchosen}(2).
\]

\[
\text{sum}(2, 3) : \neg \text{sum}(1, 3), \text{notchosen}(2).
\]

Those rules include every possible combination for the sum until site 2 is reached. For this case, only 4 rules are needed, since they cover the value of the sum in case site 2 was chosen and in case it was not, plus the cases where the site 3 itself is chosen and where it is not. Following this reasoning, when we reach site 3 we will have to consider all the possible values for the sum at site 2, and the inclusion or not of the site 3 itself, shown as follows:
\[ \text{sum}(3, 0) : -\text{sum}(2, 0), \text{notchosen}(3). \]
\[ \text{sum}(3, 3) : -\text{sum}(2, 0), \text{chosen}(3). \]
\[ \text{sum}(3, 3) : -\text{sum}(2, 3), \text{notchosen}(3). \]
\[ \text{sum}(3, 6) : -\text{sum}(2, 3), \text{chosen}(3). \]
\[ \text{sum}(3, 6) : -\text{sum}(2, 6), \text{notchosen}(3). \]
\[ \text{sum}(3, 9) : -\text{sum}(2, 6), \text{chosen}(3). \]

With this approach, the number of rules will grow quickly, which results in a really quick growth of the ground file. For our example when we reach the site 16, we will have 72 ground rules only for itself, and a total of 598 rules generated by 4.9 and 4.10 alone, which represents more than 50% of the size of the complete grounded file. With this in mind, another version of the program was created, to be explained in section 4.1.2. Rule 4.11 will generate different rules based on the limit chosen. For this example, the limit 12 was given, and the site with the lowest number that can have such sum is site 6, because the sum of all sites from 1 to 5 is 12. As an example we can find in the ground file the lines

\[ : -\text{sum}(6, 16). \]
\[ : -\text{sum}(6, 13). \]
\[ : -\text{sum}(6, 15). \]
\[ : -\text{sum}(6, 17). \]
\[ : -\text{sum}(6, 14). \]

which represent sums for combinations that cannot happen. For example

\[ : -\text{sum}(6, 16). \]

can be the result of choosing the sites 1, 2, 3, 5 and 6, which sum is 16, a value above the given limit of 12. For every site after site 6 we will see similar rules with values that invalidate the solution, in order to ensure the correction of the solution. Finally, to cover the rule R4, which is the optimization criterion, we can use the optimization function available in ASP. All we need to do is to represent that we want the maximum utility of the chosen sites. In order to make that, we will first generate a predicate that says the utility of a site if it is a chosen one, since it makes it easier to represent the information we want to optimize. Line 4.12 will do the trick, associating the utility with the respective chosen site.
chosenUtil(X, Y) : -chosen(X), util(X, Y).

(4.12)

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 [13].

#maximize[chosenUtil(N, U) = U].

(4.13)

With line 4.13 we ensure such optimality, since we want to maximize the sum of $U$, which in this case is the utility for each site. With those 13 lines we can take any input and the solution will be correct and optimal. The grounding generated by 4.12 is easy to understand since it represents that if a certain site $N$ is chosen, then the predicate chosenUtil(N, U) has the respective value of utility $U$. As an example, for sites 1 and 2 the following lines are generated:

chosenUtil(2, 4) : -chosen(2).

chosenUtil(1, 5) : -chosen(1).

Rule 4.13 generates a single line, since it is an optimization function, with the following aspect, where some of the terms where omitted due to its extension:

#maximize[chosenUtil(1, 5) = 5@1, chosenUtil(2, 4) = 4@1, (...), chosenUtil(16, 2) = 2@1].

This line contains all the chosenUtil(N, U) predicates and gives different weights to each predicate that correspond to the utility value, which is the one that we want to optimize. In order to see the desired output, which corresponds to the sites that are chosen as solution, two additional lines are added to the program, lines 4.14 and 4.15.

#hide.

(4.14)

#showchosen(_).

(4.15)

Line 4.14 will just hide every output line, and then with line 4.15 we say that we want to see the predicates chosen(N) as output. With this we get as a result the sites that represent the answer set, which is the desired solution. The complete code can be seen in appendix C.
4.1.2 Second Approach

In the previous topic we showed a correct solution for the problem. However, during the design of the program, we noticed that the predicate sum was generating way too many lines on the grounding. Since most of times a huge grounding represents a slower program, because of the large amount of data that needs to be checked, we wanted to try a different approach on the sum predicates. Considering that the main goal of this approach is to change the sum predicate, we will keep lines 4.1 to 4.6 and lines 4.12 to 4.15. The remaining lines (4.7 to 4.11) will be replaced by the lines described in this section. The main problem with the previous approach resides in the recursion used for the sum, which causes the grounder to have to cover every possibility until a given site is reached. The ASP language provides a large amount of functions, one of them being the sum function, which might be a good alternative to the used approach. However, in order to use the sum function, we need to do something similar to what we have done in the maximize function, because the cost in which we are interested in is related to the chosen sites only. Line 4.12 addresses a very similar issue but for the utility, so all we need to do is to adjust to use the cost instead

\[ costC(N, C) : \neg cost(N, C), chosen(N). \] (4.16)

and with this we have a predicate that contains the cost for each individual site that was chosen. Now all we have to do is to restrict the sum of those sites to the given limit. That can be done using the code in line 4.17, where we say that a sum\( (C) \) is equal to the sum of every value \( C2 \) in the predicates \( costC(N, C2) \), which are the predicates that only contain chosen sites. This way we prevent the generation of the huge amount of lines that we had in the previous approach.

\[ sum(C) : \neg C = \#sum[costC(.C2) = C2]. \] (4.17)

Using the same example 4.1, the grounding generated by the line 4.16 is one line for each site that creates the predicate with the respective cost in case the site is chosen, as shown below for sites 1 and 2.

\[ costC(2, 3) : \neg chosen(2). \]

\[ costC(1, 3) : \neg chosen(1). \]

Line 4.17 will generate for each possible value of a sum a very similar line to the following one, with some terms omitted due to the extension of the line:

\[ sum(0) : \neg \#sum[costC(1, 3) = 3, costC(2, 3) = 3, ..., costC(16, 4) = 4]0. \]

The only differences between each of the lines created is on the lower bounds, upper bounds and the
value on the predicate sum that refers to the value of the sum to be considered. For our example, the sum of the costs of every site is 39, so the higher value that we can find on the grounder for the sum is \( \text{sum}(39) \), which generates the line

\[
\text{sum}(39) : -39 \# \text{sum}[	ext{costC}(1,3) = 3, \text{costC}(2,3) = 3, \ldots, \text{costC}(16,4) = 4]39.
\]

that is very similar to the previous line. The meaning of this line is: considering that the predicate \( \text{costC}(1,3) \) has the value 3, the predicate \( \text{costC}(2,3) \) has the value 3, etc. the sum of the value for all predicates must be higher then or equal to 39 and lower then or equal to 39 (i.e. it must be 39) to generate the predicate \( \text{sum}(39) \). The complete encoding for this approach can be found on appendix C.

The goal in this approach was to reduce the amount of predicates that were related to the sum predicate, so we will do a small comparison for our example. As we referred previously, the lines 4.9 and 4.10 have generated 598 lines by themselves. The sum component of the new approach, including the new predicates \( \text{costC}(N,C) \), use only 59 lines, which is a significant difference. The grounded size for this small example is different for the two approaches where the file with the first approach had 17.591 bytes, and the file using this second approach has 14.075 bytes, which is almost half of the size. If we consider a random 10 by 10 instance, the first approach generates a file with 889.874 bytes, where the second approach generates a file with 4.462.959 bytes, where the same input in the second approach generates a file with 2.510.182 bytes, which is considerably smaller. The performance of this solution in comparison to the previous one will be discussed in chapter 5. In any case, the goal of making a program that generates a smaller grounding has been accomplished.

### 4.1.3 Finding Minimum Cost

In both sections 4.1.1 and 4.1.2 one of the input values is the cost limit to find a solution. Although this boundary is supposed to be changed, it is convenient that we know the minimum cost necessary to find at least a solution. To find such value we also used an ASP implementation since it is quite simple to adapt from the given implementations to a program that finds the minimum cost. The connectivity of the solution and the inclusion of the terminal nodes are just like in the previous implementations:

\[
\text{poss}(N) : -\text{start}(N).
\]

\[
\text{poss}(N) : -\text{poss}(M), \text{edge}(N,M), \text{chosen}(M).
\]

\[
: -\text{chosen}(N), \text{notposs}(N).
\]

\[
: -\text{terminal}(N), \text{notchosen}(N).
\]
To find the minimum cost, the following two lines are added:

\[
\text{chosenCost}(X,Y) : \neg\text{chosen}(X), \text{cost}(X,Y).
\]

\[
\#\text{minimize}[\text{chosenCost}(N,C) = C].
\]

Those lines work in a very similar way to lines 4.12 and 4.13 introduced in section 4.1.1. However, instead of maximizing the utility value, it minimizes the sum of the costs. When we run this program we get at least one solution with an optimization value that is equal to the minimum cost needed to connect all the terminal nodes. That value will be the reference for our tests since it is not useful to search below that minimum as we would not get any answer set.

4.2 Minimum Connected Cover

In this chapter we will address the problem variant where we must find a minimum connected cover, i.e., the minimum number of sites that is connected and has at least 1 individual of each species. In a similar way to the approach in section 4.1, the input will contain the information about the sites and their adjacency. However, instead of costs and utilities, here we have the information about the species. That information can be represented in a similar way to the costs or the utilities, having a predicate with the site and the respective cost. Having this in consideration, the input for this problem can be given by the following predicates:

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

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

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

where \text{node}(i) and \text{edge}(i,j) represent, like in the problem described in section 4.1, the site \(i\) and that \(i\) is adjacent to \(j\). The predicate \text{species}(i,s) represents that the site \(i\) is inhabited by the species \(s\). This problem variant was addressed to do some further tests to ASP. However this was not the main focus of our work. In the next sections we will use figure 4.2 as an example of a zone that will be used to explain the program. In the figure, each square represents a site where the species are represented by letters in blue (on top of each site), each letter corresponding to a different species. In black, on the bottom of each site, we have the site name, represented by a number. The complete input code can be found on the appendix B.

4.2.1 First Approach

In a similar way to what we have done before, we will take a closer look at every rule that we need to take in consideration for this problem. Those rules can be discriminated as follows:
R1: At least one of each distinct species must be included in the final solution.

R2: The chosen sites must be connected to each other.

R3: The amount of chosen sites must be the minimum possible.

We recall that rule R1 is relative to the covering portion of the problem whereas rule R2 is related to the connectivity. Since the input model used for both this problem and the problem shown in section 4.1 is very similar (and uses the same predicates for the sites), the following lines are also needed in this program:

\[
\{\text{chosen}(N) : \text{node}(N)\}. \quad (4.18)
\]

\[
\text{edge}(N, N2) : -\text{edge}(N2, N). \quad (4.19)
\]

having the exact same function they had in section 4.1. After those rules, we can focus on rules R1, R2 and R3. Rule R1 is a rule that requires checking whether all the species are being considered in a solution. One way to make sure that every species were included in a solution is to create a new predicate, \(\text{covered}\_\text{species}(X)\), in which \(X\) is a species and that is generated only when we consider a site for the solution that contains the species mentioned. After that, all we need to do is to ensure that no species were forgotten by checking that every species has one \(\text{covered}\_\text{species}(X)\) predicate. These rules are written in ASP in the following way:
covered_species(X) : -chosen(N), species(N,X).
(4.20)

: -species(X), notcovered_species(X).
(4.21)

In our example, grounding with rule 4.20 would generate the following lines:

covered_species(d) : -chosen(1).

covered_species(c) : -chosen(1).

representing that if site 1 is chosen, then we have species c and d covered. There are similar rules for every site. Line 4.21 would generate a restriction for every species saying that is must be included (since the rule says that we cannot *not have* the species covered). The grounder would simply generate constraints like

: -notcovered_species(d).

: -notcovered_species(c).

for every species on every site. This will, however, create redundant constraints because for every time that some species appear in a site a constraint is generated, even if that same constraint was already generated. For our example, there will be eleven constraints *notcovered_species(a)*, because there are eleven sites where the species a appears. Since this information is strictly redundant, a slight modification to this program related to this phenomenon will be explained later in section 4.2.2. By only adding lines 4.20 and 4.21 we have a program that is capable of finding covers on a given input but they are not minimal and not necessarily connected yet. To ensure connectivity we will now address rule R2. Unlike the problem described in section 4.1, we do not have any information about any site that must be kept in the final solution.

In order to work around that issue we will say that exactly one chosen site will be the start site, meaning that the connectivity will be checked from such site, in the same way that we checked connectivity from a certain mandatory site in section 4.1.1. To generate such starting point we use lines 4.22 and 4.23. At a first glance it looks like we should have just generated a *start(N)* predicate from a chosen site in the first place, so the constraint 4.23 would not be needed. However, Clasp can only generate predicates in this way using domain predicates, i.e., predicates that are given as input such as *node(N)*. Since *chosen(N)* is a predicate generated by the solver itself, it cannot be used to generate predicates in this kind of rule

1*{start(N) : node(N)}*.  
(4.22)
The grounding of the rule 4.22 is a count function that is limited by 1 in both upper and lower bounds, having the following aspect for our example:

\[ 1 \# \text{count}\{ \text{start}(16), \text{start}(15), \ldots, \text{start}(1) \} \]

Since the value of the count is given by the amount of existing predicates from the ones that are inside the function, and that value must be exactly 1, we ensure that only one of the chosen sites is considered to be the starting one. This is important because if we had two distinct starting points, we could have a partially connected solution where a site could only be connected to one of the starting sites but not to the other and still be considered a connected solution, which violates rule \( R2 \). Line 4.23 will generate a constraint for every site that prohibits such site from being a starting site if it is not a chosen one. Those constraints will look like the following ones:

\[ : - \text{start}(1), \text{notchosen}(1). \]
\[ : - \text{start}(2), \text{notchosen}(2). \]

After having these rules we can use the same rules 4.7, 4.8 and 4.9 used in section 4.1.1 to check connectivity. Now that rule \( R2 \) has been dealt with, all we need to do is to ensure the optimality which is given by rule \( R3 \). Once again we will use one of the built-in functions of ASP to find our optimal solution. In this case we want to state that the best solution is the one with the least amount of sites, so we will apply the minimize function to the number of sites that are chosen. To do that we use line 4.24, where we say that we want to minimize the number of nodes that are chosen.

\[ \#\text{minimize}[\text{chosen}(N): \text{node}(N)]. \quad (4.24) \]

This will generate a ground rule with the following aspect:

\[ \#\text{minimize}[\text{chosen}(1) = 1 \oplus 1, \text{chosen}(2) = 1 \oplus 1, \ldots, \text{chosen}(16) = 1 \oplus 1]. \]

This rule says that the weight of each site is equal to one, so all it needs to check is how many sites are being used in the solution, because all of them have the same weight. The complete encoding for this approach can be found on appendix C.
### 4.2.2 Second Approach

As we mentioned in the previous section, the rule 4.21 generates some redundant grounding, which is something that is not desired, specially because the redundancy comes from duplicated lines, so there is absolutely no new information on those lines. To work around that issue, we will assume that the input can include the list of different species that exist in an area, which is fairly reasonable considering that we known which species inhabit in each of the sites. Consider that we can have as input to our program the following representation:

\[
\text{distinct species}(X)
\]

where \(X\) represent one of the species, existing a total of one of those representations for each existing species. Having this information we can replace rule 4.21 by rule 4.25, which has the exact same purpose of ensuring that every species are included.

\[
: \neg \text{distinct species}(X), \text{notcovered species}(X).
\]  

(4.25)

This line will only generate one constraint for each species, instead of generating a rule for each time some species appeared on a site, like it did before. With this we do introduce some extra predicates as input, but we do know that we will have as many lines as twice the number of species, since we have one line for \(\text{distinct species}(X)\) and one line for \(\neg \text{distinct species}(X), \text{notcovered species}(X)\). Comparing this approach with the previous one, it is reasonable to use it as long as each species appear on average in more than two sites. Comparing the values for our small example, the first approach generated 29 lines in the grounding process with respect to the rule 4.21 alone, while the second approach generated only 10 lines to represent the same rules, which includes the predicate \(\text{distinct species}(X)\) and the rule \(\neg \text{distinct species}(X), \text{notcovered species}(X)\). Although this might not be a huge change, we consider it a positive one since we are getting rid of duplicated information that will only make the grounding file bigger. However, as mentioned before, there are some special cases where this approach ends up using more space than the first one, like the cases where there are very few to none species repetition across the sites. The complete encoding for this approach can be found on appendix C.
In this chapter we will show and discuss the result of our implementations and compare the new wildlife corridor implementations to the CPLEX implementation. Since we need some instances to compare results, we have created two generators, one for each type of problem. In section 5.1 we will describe the generators that were created and the choice of instances while in section 5.2 we will explain the instances we have selected to run. In section 5.3 we will show the results for the chosen instances.

5.1 Generators

The generation of instances for the problem is something that needs some attention. We want to have a way to test our program and compare with the existing one, so we can have some idea about the efficiency of the program. To do that we created two instance generators, one for each type of problem. Both generators were developed in Java and both produce input for the answer set programming programs. The generator that produces instances requiring the connection of mandatory sites also produces the CPLEX code as referred in 5.1.3. In section 5.1.1 we will describe the instance generation with respect to the problem were we need to find the minimum cover and in section 5.1.2 we will describe the generation of the wildlife corridor instances. Since we needed a CPLEX implementation to evaluate our results, we found it more convenient to incorporate such implementation in the instance generation. With that we generate simultaneously the ASP domain and the CPLEX program.

5.1.1 Minimum connected cover

The first thing to consider is the area representation, since we need to have multiple sites and connections. We opted for rectangular areas, where we give as input the number of columns and the number of lines. With that we can generate all the \textit{node}(N) and the \textit{edge}(N,M) predicates. To do that, for each site, we add the edges in the way that is represented in figure 5.1, where the blue circle represents a site that we are considering and the arrows point to the sites that are considered adjacent. Since this procedure is done for every site, all the sites will have their adjacency represented, either being the source or the destination. We always check if the site is not in a corner or at the bottom, to avoid representing edges that do not exist.

Now that we have the area represented, we need to generate some species. In order to make the generator a little versatile we have some parameters that can be given as arguments. First we have the number of species as argument, since it is one of the major changes in any area. Second, each species
might be present in more than one site, so we can give as an argument how many times each of the species appears. This is a global modifier that affects all species, so if we have 3 different species and we say each appears 2 times, we will end up having 6 different \( \text{species}(N, S) \) predicates. At last we can also define how many different species are at most in each site in order to ensure that we will not end up having more than the given number of species in any site. With these parameters we can generate either areas with lots of different species but few species for each site or areas with few distinct species but with great species abundance in each site.

5.1.2 Wildlife corridor

Like in the previous case, we will need to generate an area. For that we used the exact same criteria as before, giving the lines and columns desired and generating the \( \text{node}(N) \) and \( \text{edge}(N, M) \) predicates. We also give as input how many sites we want to be considered the mandatory sites, so the program marks them as \( \text{terminal}(N) \). Those terminals are chosen randomly.

In order to give some additional flexibility, the generator also receives as argument the range of costs and the range of utilities that we want to our sites, so the random chooses a value that is always in the range. With that we can generate instances with very close utility values where might be there many solutions, or just use big ranges to increase the probability of an instance to have very few or even only one solution. Since we can choose the range at will, we can even generate instances with either uniform costs or uniform utilities (or even both) if we want to test some particular cases.

Finally, since sometimes the fact that we use random values might generate particular instances that can have peculiar results, we have a last argument that indicates how many instances should be generated with the same size, number of terminal nodes and cost and utility range and with that information we can average the time to solve all the instances and use that as a more consistent result.

5.1.3 Implementation in CPLEX

In order to compare our work with the state of art work, we had to run a CPLEX implementation for our main problem, the wildlife corridor problem. We decided to integrate the CPLEX implementation with the generators, since it added some simplicity to the process. However, it can be adapted to a converter as long as it is provided some way to identify the existing sites and the respective edges. We used the Single Commodity Flow implementation referred in 2.3.2 using a Java program that has the information of the

48
existing sites, the edges of those sites, and the terminal nodes. That information is available because our
generators keep the information in the form of classes, having a class Site and a class Edge corresponding
to the existing sites and to the existing edges, respectively. Each site has a list of its edges so we know
at any time to which other sites it is connected to. To write the CPLEX implementation we followed
closely the implementation in section 2.3.2.

First we write the optimization line, writing a line with maximize and the sum of the product of each
site with its utility, in the form

\[ u_1x_1 + u_2x_2 + \ldots + u_Nx_N \]

where \( u_n \) represents the utility of site \( N \) and \( x_N \) the inclusion or not of the site \( N \). Then we write subject
to to start with the constraint of the problem, starting with the expression

\[ c_1x_1 + c_2x_2 + \ldots + c_nx_n < \text{Cost} \]

to ensure that the cost \( c \) of the sum of the chosen sites is below the Cost limit. Then we choose a terminal
node as source and write the expression

\[ x_0 + y_0 + t_N = \text{Size} \]

where \( t_N \) is the terminal chosen as source and \( \text{Size} \) is the number of sites. Now we need to write the rule

\[ y_{ij} \leq nx_j \]

so what we do is for each site \( j \) we find all the edges that go to \( j \) and write a line with the rule for each
of those edges. Next we have to write the flow conservation rules. For that we go through each site \( N \)
and sum the flow from sites that go to site \( N \), we subtract \( x_N \) which represents the inclusion of the site
and if it is included it consumes a unit of flow, and then we subtract the flow for every site to which \( N \)
is linked to. The last constraint is only related to the flow absorption being the same as the flow injected
in the system. For that we write

\[ x_1 + x_2 + \ldots + x_N - y_{0s} \]

where \( s \) is the source chosen before. The next step is to write the bounds which are

\[ y_{ji} \geq 0 \]

to say that the flow in any edge must be either zero or a positive number. To say that the source has a
flow that is greater or equal to zero we write the expression

\[ x_0 \geq 0 \]

and to force the inclusion of the terminal nodes we use the expression bellow.

\[ x_t = 1 \]

As a last step we indicate that the \( x_N \) variables always take binary values by writing \textit{binary} and the adding each of the sites.

### 5.2 Instance Selection

To test the performance of the program we have to select which instances we would like to test. Considering first the main goal of our work, we will describe the choices for the Wildlife corridor problem. Dilkina et al. [9] use areas with 100 sites and 3 terminal nodes, so as for the number of terminal nodes we decided to start from the same number. However, we will generate smaller instances with 81 sites (9 by 9) and bigger instances with 121 sites (11 by 11) to have a broader amount of results. To test the influence of the costs and utilities we also generated two types of instances for each size, one with a range from 1 to \( \sqrt{N} \), where \( N \) is the total amount of sites, and another from 1 to \( \frac{N}{3} \) rounded to the nearest integer, with \( N \) still representing the total amount of sites. So for our 9 by 9 sample with 81 sites we would have instances with the range 1 to 9, and instances with the range 1 to 27. We also generate 10 instances for each type, so we can average the obtained values for a more consistent result. For each instance we will test first with the minimum cost obtained using the program described in 4.1.3. We will then use increments of 10% over that base value up to a maximum of a 90% increment, which increases the difficulty of finding an optimal solution considering that we have a wider range of solutions available.

For the finding the minimum connected cover variant we will generate two types of instances: one where we have 10 distinct species repeated 20 times each, and another where we have 20 species each repeated 10 times. We will generate square samples with 7, 8, 9 and 10 lines and columns, and allow a maximum of 10 species in each site.

The chosen instances are summarized in table 5.1.

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

Table 5.1: Instance selection summary
5.3 Results

We will start by comparing the ground size between the two approaches developed for the wildlife corridor problem, explained on sections 4.1.1 and 4.1.2. Since the time of grounding for any instance was less than 0.02 seconds, we will consider it irrelevant in the future time considerations. Also, with such a small value for time, we will only compare the size of the ground files. Since the ground files have a similar size between the different cost limits, table 5.2 shows the average size for the case using the minimum cost. The 4 instances are placed on the left column, where short and long represents whether we used the small range of values for the costs and utilities or the long range of values, respectively. On the second column we have the size of the instances made by the first approach while on the third column we can find the size for the second approach. Both results are in Megabytes. On the rightmost column we have the relative size of the files from the second approach in comparison to the first, in percentage. That number is given by \[ \frac{\text{Second Approach}}{\text{First Approach}} \times 100 \] and allows a faster visualization of the gain or loss using the second approach.

<table>
<thead>
<tr>
<th>Instance</th>
<th>First Approach (MB)</th>
<th>Second Approach (MB)</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 by 9 short</td>
<td>0.85</td>
<td>0.49</td>
<td>58%</td>
</tr>
<tr>
<td>9 by 9 long</td>
<td>2.33</td>
<td>1.41</td>
<td>61%</td>
</tr>
<tr>
<td>10 by 10 short</td>
<td>1.51</td>
<td>0.84</td>
<td>56%</td>
</tr>
<tr>
<td>10 by 10 long</td>
<td>4.61</td>
<td>2.71</td>
<td>59%</td>
</tr>
</tbody>
</table>

Table 5.2: Ground size for two wildlife corridor approaches

We have confirmed that the ground size is indeed smaller for the second approach, which uses only around 60% of the size to represent the same instance. Now we will compare our approach with the Single Commodity Flow approach used to solve the wildlife corridor problem. The table of results for the instances referred in the previous section can be seen in table 5.3. The cost limit increment is represented by the % value in the second column, where 100% is the minimum cost that is needed for a solution. The time is given in seconds, and at the left of each time value we have the number of instances out of 10 of each type that were not solved in the given time, 720 seconds (12 minutes). 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 gringo 3.0.4. The time is given in seconds and is the average of the times for the 10 instances of each type. Whenever a solution was not given in due time, we added 720 as the time value of that instance to allow a better comparison of the average values.
<table>
<thead>
<tr>
<th>Instance</th>
<th>% min cost</th>
<th>First approach</th>
<th>Second approach</th>
<th>CPLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># Time-out</td>
<td>Time (s)</td>
<td># Time-out</td>
</tr>
<tr>
<td>9x9 short</td>
<td>100%</td>
<td>0</td>
<td>1.8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>110%</td>
<td>0</td>
<td>2.1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>120%</td>
<td>0</td>
<td>2.8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>130%</td>
<td>0</td>
<td>5.1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>140%</td>
<td>0</td>
<td>13.2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>150%</td>
<td>0</td>
<td>45.2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>160%</td>
<td>1</td>
<td>170.6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>170%</td>
<td>2</td>
<td>281.4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>180%</td>
<td>4</td>
<td>364.9</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>190%</td>
<td>5</td>
<td>485.6</td>
<td>5</td>
</tr>
<tr>
<td>9x9 long</td>
<td>100%</td>
<td>0</td>
<td>4.3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>110%</td>
<td>0</td>
<td>5.6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>120%</td>
<td>0</td>
<td>9.2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>130%</td>
<td>0</td>
<td>18.8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>140%</td>
<td>0</td>
<td>61.8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>150%</td>
<td>1</td>
<td>87.3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>160%</td>
<td>1</td>
<td>103.6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>170%</td>
<td>1</td>
<td>135.1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>180%</td>
<td>2</td>
<td>180.9</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>190%</td>
<td>2</td>
<td>215.0</td>
<td>2</td>
</tr>
<tr>
<td>10x10 short</td>
<td>100%</td>
<td>0</td>
<td>6.4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>110%</td>
<td>0</td>
<td>5.7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>120%</td>
<td>0</td>
<td>10.5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>130%</td>
<td>0</td>
<td>25.6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>140%</td>
<td>0</td>
<td>92.6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>150%</td>
<td>2</td>
<td>229.2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>160%</td>
<td>3</td>
<td>337.5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>170%</td>
<td>5</td>
<td>390.8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>180%</td>
<td>5</td>
<td>434.1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>190%</td>
<td>6</td>
<td>503.4</td>
<td>6</td>
</tr>
<tr>
<td>10x10 long</td>
<td>100%</td>
<td>0</td>
<td>21.5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>110%</td>
<td>0</td>
<td>56.4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>120%</td>
<td>0</td>
<td>159.4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>130%</td>
<td>3</td>
<td>331.1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>140%</td>
<td>6</td>
<td>464.7</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>150%</td>
<td>6</td>
<td>527.4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>160%</td>
<td>8</td>
<td>598.2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>170%</td>
<td>8</td>
<td>622.8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>180%</td>
<td>8</td>
<td>673.0</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>190%</td>
<td>10</td>
<td>time-out</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.3: Time comparison between ASP and CPLEX
With table 5.3 we can see that the second implementation is better than the first in most of the results, which shows that the smaller grounding was relevant in most cases. In the end it solves four more instances than the first approach. However it is important to refer that there are some particular cases in which the first approach ended up solving an instance that the second could not. This can be explained with the fact that since the grounding is different, the solver might have chosen better search paths due to the abundance of ground terms, which in a few cases can lead to a faster solution. This cannot be directly controlled, but since in most cases the second approach had better times and it solved more instances in total, we consider it to be a better version of the program.

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. Choosing a sample as an example of the 10 by 10 with short costs and utilities, we show in the graph 5.2 the amount of choices that the solver had to do relatively to the cost.

![Figure 5.2: Graph representing the choices and conflicts in a 10 by 10 instance](image)

As we can see the amount of choices increases exponentially with the budget, which leads to the huge time increment. The ASP solver has to go through a huge search space and it naturally takes a great amount of time. 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 can not 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 5.4 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>0.20</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 5.4: 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. However, the connectivity constraint must be included,
so we do have to keep it in the program.
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 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 to a certain point, the ASP encoding takes longer than the CPLEX implementation. However, we do consider that the work developed in this thesis 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 by detailing the grounding of the rules, 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.
### A.1 ASPviz primitives

In table A.1 we can found the most primitives available in an ASPviz program:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(X,Y) )</td>
<td>Refers to a position on the screen.</td>
</tr>
<tr>
<td>( brush(Name) )</td>
<td>Defines the brush type to be used.</td>
</tr>
<tr>
<td>( brush_width(Name,Width) )</td>
<td>Defines the brush Width in pixels.</td>
</tr>
<tr>
<td>( brush_color(Name,Colour) )</td>
<td>Defines the brush Colour.</td>
</tr>
<tr>
<td>( font(Type) )</td>
<td>Defines the font to be used.</td>
</tr>
<tr>
<td>( font_style(Type,Style) )</td>
<td>Defines the font style, for example, like bold.</td>
</tr>
<tr>
<td>( font_color(Type,Colour) )</td>
<td>Defines the font colour.</td>
</tr>
<tr>
<td>( color(Name,rgb(X,Y,Z)) )</td>
<td>Defines a colour, given the RGB component.</td>
</tr>
<tr>
<td>( draw_line(BrushName,From,To) )</td>
<td>Draws a line using the brush given in ( BrushName ) between the given positions.</td>
</tr>
<tr>
<td>( draw_text(Font,c,c,Position,Text) )</td>
<td>Draws ( Text ) on the given ( Position ), using the ( Font ) as font style.</td>
</tr>
<tr>
<td>( draw_rect(Brush,Position,Width,Height) )</td>
<td>Draws a rectangle from the given ( Position ), with the specified ( Width ) and ( Height ) (in pixels).</td>
</tr>
<tr>
<td>( draw_ellipse(Brush,Position,Width,Height) )</td>
<td>Draws an ellipsis centred in the given ( Position ), with the specified ( Width ) and ( Height ) (in pixels).</td>
</tr>
<tr>
<td>( fill_rect(Brush,Colour,Position,Width,Height) )</td>
<td>Draws a filled rectangle with a given ( Colour ) rectangle from the given ( Position ).</td>
</tr>
</tbody>
</table>

Table A.1: ASPviz primitives
A.2 Representing with ASPViz

The problem we are solving uses an area and we want to find a subset of that area that meets the
requirements. Having said that, we consider that having a way to represent that solution would be
useful, since it gets much easier to understand which sites were selected and which sites were not. In
order to do that we will use ASPViz. Considering the drawing of the solution, it is useful to have a way
to represent the position of each site in a grid that represents the area. Since the area design is known,
the position can be given as input of our program. In order to do that we use the predicate

\[ \text{position}(S, X, Y). \]

where S is the site that is in the position (X,Y). The way we consider the X and Y is represented in figure
A.1 where X=1 represents the leftmost site and Y=1 represents the topmost site.

![Figure A.1: Empty grid representing the axis](image)

Having such information allows the correct representation of the solution when we consider the chosen
sites. First, we start by defining some basic brushes and colours so we actually draw the solution. To
draw the grid itself we will use the standard brush with a black colour, so we can have visible borders on
the sites. These definitions are done with lines A.1 to A.3.

\[ \text{brush}(\text{standard}). \]  \hspace{1cm} (A.1)

\[ \text{brush\_color}(\text{standard}, \text{black}). \]  \hspace{1cm} (A.2)

\[ \text{brush\_width}(\text{standard}, 1). \]  \hspace{1cm} (A.3)

Line A.1 defines that we will use the standard brush type where line A.2 defines the brush colour as
black. Line A.3 defines the width of the brush, and we will use a brush of width 1, which is fairly visible
and adequate to our representation. Now we need to do two different things: first, we want to represent
the sites using a grid representation and then we want to represent the chosen sites so we can see the
connected path that was chosen as solution. The grid can be seen as a set of white filled squares with
black borders, so we can use line A.4 to represent it:
We use the position(_,X,Y) predicate to draw the grid since it gives us the relative position of every site. The name of the site itself is not important to draw the grid so we can use _ to ignore that information. The generic drawing predicate is the fill_rect(brush, color, p(X,Y), Width, Height). The predicate p(X,Y) represents a point in the ASPviz system. The ASPviz will draw a square considering the point p(X,Y) is the top left one and with the specified width and weight (which are the same in this case, since it is a square). Since we wanted to give our squares a width of 40 pixels, we need to say that the starting points are 40 pixels apart. For that reason, our predicate p(X,Y) uses the position value\(^1\) multiplied by 40, which corresponds to the distance that it should be apart from the previous point. Now we are able to draw some representation of the final solution. We will use blue circles to do that, since they are fairly visible and provide a visible representation of the solution. To do that we will use the fill_ellipse(Brush, Color, Point, W, H) predicate, that also uses a point, a width and a length like the previous one. The difference is that it draws the circle around the point, so we have to adjust the starting point to be in the middle of the squares on the grid. Line A.5 does that adjustment, adding the value of 20 pixels to the point in both X axis and Y axis, which is half of the square dimension.

Assuming that we have at least a solution, running this program with the ASP program will give us the grid of the area with the chosen sites filled in blue, since we specified that the sites where we draw the blue circle are only the ones which have a predicate chosen(N). Considering that all the information we use is the chosen sites and the position of each site, this solution will work for any of our approaches, since they all use the predicate chosen(N) to represent the solution of the problem. The example given in figure 4.1 will have the representation given in the figure A.2.

\[\text{fill\_rect(standard, white, } p((X - 1) \times 40, (Y - 1) \times 40), 40, 40) : \neg \text{position(_,X,Y).} \quad (A.4)\]

\[\text{fill\_ellipse(standard, blue, } p((X - 1) \times 40 + 20, (Y - 1) \times 40 + 20), 40, 40) : \neg \text{position(N, X, Y), chosen(N).} \quad (A.5)\]

\(^1\)Actually, it uses this value minus one, since the ASP representation starts in 1 but the referential for drawing in ASP starts in 0.
B.1 Wildlife corridor

The following code represents the input needed for the example in figure 4.1 found on section 4.1.

```
node(1). util(1,5). cost(1,3). position(1,1,1). edge(1,5). edge(1,6).
node(2). util(2,4). cost(2,3). position(2,2,1).
edge(2,6). edge(2,3). edge(2,7). edge(2,5).
node(3). util(3,1). cost(3,3). position(3,3,1).
edge(3,7). edge(3,4). edge(3,8). edge(3,6).
node(5). util(5,1). cost(5,2). position(5,1,2). edge(5,9). edge(5,6). edge(5,10).
node(6). util(6,5). cost(6,5). position(6,2,2).
edge(6,10). edge(6,7). edge(6,11). edge(6,9).
node(7). util(7,2). cost(7,2). position(7,3,2).
edge(7,11). edge(7,8). edge(7,12). edge(7,10).
node(8). util(8,1). cost(8,3). position(8,4,2). edge(8,12). edge(8,11).
node(9). util(9,4). cost(9,1). position(9,1,3). edge(9,13). edge(9,10). edge(9,14).
node(10). util(10,2). cost(10,3). position(10,2,3).
edge(10,14). edge(10,11). edge(10,15). edge(10,13).
node(11). util(11,4). cost(11,1). position(11,3,3).
node(12). util(12,5). cost(12,1). position(12,4,3). edge(12,16). edge(12,15).
node(13). util(13,3). cost(13,2). position(13,1,4). edge(13,14).
node(14). util(14,3). cost(14,1). position(14,2,4). edge(14,15).
terminal(1).
start(1).
terminal(11).
terminal(3).
```
B.2 Minimum connected cover

The following code represents the input needed for the example in figure 4.2 found on section 4.2.

```
node(1). position(1,1,1). edge(1,5). edge(1,2). edge(1,6).
node(2). position(2,1,2). edge(2,6). edge(2,3). edge(2,7). edge(2,5).
node(3). position(3,1,3). edge(3,7). edge(3,4). edge(3,8). edge(3,6).
node(4). position(4,1,4). edge(4,8). edge(4,7).
node(5). position(5,2,1). edge(5,9). edge(5,6). edge(5,10).
node(6). position(6,2,2). edge(6,10). edge(6,7). edge(6,11). edge(6,9).
node(7). position(7,2,3). edge(7,11). edge(7,8). edge(7,12). edge(7,10).
node(8). position(8,2,4). edge(8,12). edge(8,11).
node(9). position(9,3,1). edge(9,13). edge(9,10). edge(9,14).
node(12). position(12,3,4). edge(12,16). edge(12,15).
node(13). position(13,4,1). edge(13,14).
node(14). position(14,4,2). edge(14,15).
node(15). position(15,4,3). edge(15,16).
node(16). position(16,4,4).
species(1,c). species(1,d).
species(2,a). species(2,b).
species(3,a). species(3,b).
species(4,a). species(4,c).
species(5,a). species(5,b). species(5,c). species(5,d).
species(6,b). species(6,a).
species(7,a). species(7,d).
species(8,d). species(8,b).
species(9,a).
species(10,a). species(10,b).
species(10,f). species(11,a).
species(12,a). species(12,b).
species(13,e).
species(15,a).
species(16,e). species(16,f).

%This is only for the second approach
distinct_species(a). distinct_species(b). distinct_species(c).
distinct_species(d). distinct_species(e). distinct_species(f).
```
C.1 Wildlife corridor - first approach

The following code is the code in the first ASP implementation used to solve the wildlife corridor.

\[
\text{chosen}(N) : \text{node}(N).
\]

\[
\text{edge}(N,N2) :- \text{edge}(N2,N).
\]

\[
\text{poss}(N) :- \text{start}(N).
\]

\[
\text{poss}(N) :- \text{poss}(M), \text{edge}(N,M), \text{chosen}(M).
\]

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

\[
\text{sum}(1,C) :- \text{cost}(1,C), \text{chosen}(1).
\]

\[
\text{sum}(1,0) :- \neg \text{chosen}(1).
\]

\[
\text{sum}(N,C+C2) :- \text{sum}(N-1, C), \text{cost}(N,C2), \text{chosen}(N).
\]

\[
\text{sum}(N,C) :- \text{sum}(N-1, C), \neg \text{chosen}(N), \text{node}(N).
\]

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

\[
:- \text{chosen}(N), \neg \text{poss}(N).
\]

\[
:- \text{terminal}(N), \neg \text{chosen}(N).
\]

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

\[
\#\text{hide}.
\]

\[
\#\text{show chosen(\_)}.\]
C.2 Wildlife corridor - second approach

The following code is the code in the second ASP implementation used to solve the wildlife corridor.

\[
\text{chosen}(N) : \text{node}(N).
\]

\[
\text{edge}(N,N2) :- \text{edge}(N2,N).
\]

\[
\text{poss}(N) :- \text{start}(N).
\]

\[
\text{poss}(N) :- \text{poss}(M), \text{edge}(N,M), \text{chosen}(M).
\]

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

\[
\text{costC}(N,C) :- \text{cost}(N,C), \text{chosen}(N).
\]

\[
\text{sum}(C) :- C=\sum[\text{costC}(_,C2)=C2].
\]

\[
:- \text{sum}(C), C > \text{max}.
\]

\[
:- \text{chosen}(N), \text{not poss}(N).
\]

\[
:- \text{terminal}(N), \text{not chosen}(N).
\]

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

\[
\#\text{hide}.
\]

\[
\#\text{show chosen(\_)}.
\]
C.3 Minimum connected cover - first approach

The following code is the code in the first ASP implementation used to solve the minimum connected cover problem.

chosen(N) : node(N).
1start(N) : node(N)1.
:-start(N), not chosen(N).

edge(N,N2) :- edge(N2,N).
covered_species(X) :- chosen(N), species(N,X).
:- species(_,X), not covered_species(X).

poss(N) :- start(N).
poss(N) :- poss(M), edge(N,M), chosen(M).
:- chosen(N), not poss(N).

#minimize[chosen(N) : node(N)].
#hide.
#show chosen(_).
C.4 Minimum connected cover - second approach

The following code is the code in the second ASP implementation used to solve the minimum connected cover problem.

chosen(N) : node(N).
1start(N) : node(N)1.
:- start(N), not chosen(N).

edge(N,N2) :- edge(N2,N).
covered_species(X) :- chosen(N), species(N,X).
:- distinct_species(X), not covered_species(X).

poss(N) :- start(N).
poss(N) :- poss(M), edge(N,M), chosen(M).
:- chosen(N), not poss(N).

#minimize[chosen(N) : node(N)].
#hide.
#show chosen(_).
Bibliography


