INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Protein Design using Answer Set Programming

João Filipe Rosado Gouveia

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. Miguel Mourão Fialho Bugalho

October 2012

"It is far, far easier to make a correct program fast than it is to make a fast program correct."
Herb Sutter, Andrei Alexandrescu,
C++ Coding Standards, Addison-Wesley, 2005

Abstract

Different proteins have different functions determined by its structure. Proteins are a sequence of amino acids folded into a three-dimensional structure. Each amino acid is formed by an amine group, a carboxyl group and a side-chain. The side-chain is specific for each amino acid and each amino acid can have numerous possible side-chain conformations, called rotamers. A protein has an energy associated and this energy depends on the amino acids and side-chain that form the protein. Predicting the set of amino acids and respective side-chain that minimizes the total energy of a protein is therefore a very important problem, called protein design.

In this work we develop a program to solve the protein design problem using Answer Set Programming. Answer Set Programming (ASP) is an approach to declarative solving problems. This work describes the ASP program implemented to solve protein design problems, using the ASP grounder *gringo* and the ASP solver *clasp* to search for the answer sets of the program.

Two approaches of the protein design problem were considered: one considered that the amino acids of the protein to design are kept fixed; the other considered that the amino acids are not kept fixed and therefore the amino acids and respective side-chains must be determined.

In this work were made two implementations in ASP for the protein design problem. One is a simple codification and the other uses a multi-criteria optimization. Moreover, there were implemented three algorithms of dead-end elimination (DEE): Original DEE; Simple Goldstein; and Simple Split.

Keywords

Protein
Protein Design
Answer Set Programming
Optimization
Search

Resumo

Proteínas diferentes têm diferentes funções fortemente determinadas pela sua estrutura. As proteínas são constituídas por uma sequência de aminoácidos que se dobram numa estrutura tridimensional. Cada aminoácido é formado por um grupo amino, um grupo carboxílico e uma cadeia lateral. A cadeia lateral é específica para cada aminoácido e pode ter muitas configurações, chamadas rotamers. Uma proteína tem uma energia associada que depende dos aminoácidos e respectivas cadeias laterais que formam a proteína. Prever o conjunto de aminoácidos e as respectivas cadeias laterais que minimizam a energia total de uma proteína é, portanto, um problema muito importante e é chamado de desenho de proteínas.

Neste trabalho desenvolveu-se um programa para resolver o problema de desenho de proteínas usando answer set programming (ASP). Answer set programming é uma abordagem declarativa para resolver problemas. Este trabalho descreve o programa em ASP implementado para resolver problemas de desenho de proteínas, usando o ASP grounder *gringo* e o ASP solver *clasp* para encontrar os conjuntos de resposta do programa.

Foram consideradas duas abordagens ao problema: numa considerámos que os aminoácidos de uma proteína são mantidos fixos; na outra considerámos que é preciso determinar tanto os aminoácidos da proteína a desenhar como as respectivas cadeias laterais.

Neste trabalho foram feitas duas implementações em ASP para o problema de desenho de proteínas. Uma delas é uma codificação simples e a outra é uma codificação que usa múltiplos critérios de optimização. Também foram implementados algoritmos de dead-end elimination (DEE): Original DEE; Simple Goldstein; e Simple Split.

Palavras-Chave

Proteína
Desenho de Proteínas
Answer Set Programming
Optimização
Procura

Acknowledgements

I have dedicated much time in this work. I know this thesis would be different without the support of so many people. I would like to thank all that have been around me and supported me during this work.

In particular I would like to thank Prof.ª Inês Lynce for the extraordinary orientation and for giving me hope in times of despair. Thanks for listen to all of my endless stories about the difficulties of this work.

I would like to thank Prof. Miguel Bugalho for the great help, clarifying me so many doubts about proteins. Without him I would not have understood so many concepts about proteins.

This work is part of the project ASPEN (PTDC/EIA-CCO/110921/2009), therefore I would also like to thank the Fundação para a Ciência e Tecnologia (FCT) for the support given.

A special thanks to my friends Tiago Almeida and Rodrigo Santos, who had to listen to me for hours. Thanks for the patience! Thanks for all the ideas for my work. Thanks for being there, every day, and for making me happy with so many jokes.

Thanks to my family, who were there for me when I needed. Thanks for giving me time and space and for being so understanding.

And thanks to all of my friends. Thanks for providing such wonderful times. I would not be the same without them.

João Filipe Gouveia
Sintra, 2012

Contents

List of Figures

List of Tables

1 Introduction

Proteins are biochemical compounds that take an important role in living cells. There are many different types of proteins each having a different function in the world. Some of the proteins types are enzymes, antibodies, structural components, receptors and transporters. The function of a protein is determined by its structure, therefore it is important to determine the best protein structure for a specific function. The study of the protein structure and the best structure of a protein for a given function can be useful in numerous areas. Some of these areas that can use this knowledge are, for instance, the pharmaceutical area where the understanding of some diseases can be improved or even to design new drugs or improve the existing ones.

There are many problems related to proteins. Some of these problems are structure prediction, protein design, energy calculation and mutation prediction. In this work we will focus on the protein design problem. The protein design problem consists in determining the best components of a protein for a given protein structure. This way it is possible to design the best protein for a specific function given that the function of a protein greatly depends on its structure.

1.1 Contributions

The goal of this work was to develop a program that solves the protein design problem, i.e., that determines the best components of a protein to have a given function. There are many approaches that use different methods to solve the protein design problem. The objective of this work was to use a different approach from the existing ones. In this work we will develop a program to solve protein design problems, which are *NP-hard*[38], using answer set programming.

Answer Set Programming (ASP) is a very recent approach to declarative problem solving[35]. Difficult search problems, like protein design problems, can be solved using ASP. ASP is based on logic, hence ASP solvers are applied to logic programs.

As the protein design problem is a search problem, our goal is to verify the capacity of ASP by applying it to the protein design problem.

This work contributes with a different approach of the protein design problem, by using ASP, as mentioned before. This works also contributes to the development of ASP by verifying its capacity of solving this problem.

1.2 Outline

This document gives a detailed description of the work done, describing the protein design problem and the ASP language.

In chapter 2 is described the background of this work. It is described the components of the proteins, specifically amino acids. In this chapter is also detailed the structure of amino acids and the relevant concepts for the protein design problem. The section 2.2 describes the protein design problem and the existing software related to the protein design problem. The details of the protein design problem, the used energy functions and the factors commonly considered, and the search space of the protein design problem are described as well. Several available tools that are related to the protein design problem are presented in section 2.3. The software tools presented are SCWRL4, SHARPEN, Rosetta and ProtSAT.

As our work is the development of a program to solve protein design problems using ASP, chapter 3 will give a description of ASP. This chapter will detail the concepts associated with ASP and the ASP language, with some examples. The tools *gringo* and *clasp* will be described in section 3.3. An example of a problem codification is given in appendix A which describes an implementation of an ASP program to solve kakuro.

Chapter 4 will describe in detail the ASP programs developed to solve the protein design problem. In section 4.1 will be described the definition of the problem solved, detailing the two approaches considered. The two codifications made will be described in detail. The representation of the problem in ASP and each rule implemented in ASP will be described in this chapter. Section 4.2 will describe the dead-end elimination methods implemented. Dead-end elimination methods are commonly used in the protein design problem.

The results of this work will be described in chapter 5. The instances used in the tests made are described in detail in section 5.1. In this chapter are shown the results of the dead-end elimination methods implemented and the results of the two codifications made (described in chapter 4). The complete results can be seen at appendix D.

Finally, chapter 6 describes the conclusions of this work. In this chapter will also be mentioned the future work, in section 6.1.

2 Proteins

Proteins have a very important role in life. There are many different types of proteins each having a different function in the world. Some of the proteins types are enzymes, antibodies, structural components, etc. In this chapter will be described the components of the proteins and then will be described in particular the structure of amino acids and the connection between them. Moreover, it will be also described the protein design problem and related tools.

2.1 Protein Structure

Proteins are biological compounds that are formed by a sequence of *amino acids*, also called residues. The sequence of amino acids in a protein defines the three-dimensional structure into which the protein folds and this structure is called the *backbone*. Each protein has a specific function and that function greatly depends on its structure.

There are 22 standard amino acids and each amino acid has an amine and a carboxyl functional groups that are common to every amino acid and a *side-chain* that varies. The side-chain is specific for each amino acid, varying in physical properties.

Figure 2.1[1] illustrates the general structure of an amino acid. In Figure 2.1 is indicated the amine group (NH_2) and the carboxyl group (COOH). The 'R' in Figure 2.1 represents the side-chain of the amino acid. The carbon atom next to the carboxyl group is called α-carbon. Table 2.1 shows the 20 essentials amino acids and respective codes. The amino acids *Selenocysteine* and *Pyrrolysine* are not represented in this table.

Each pair of amino acids in the sequence of amino acids in the protein forms a *peptide bond* and these bonds form the backbone of the protein. The *peptide bond* is the bond between the amine group of one amino acid and the carboxyl group of the other amino acid. When the peptide bond is formed, a molecule of water (H_2O) is also formed. Figure 2.2[2] illustrates a peptide bond between two amino acids.

Each amino acid has two *dihedral angles* called Φ and Ψ. A *dihedral angle* is measured on a sequence of four atoms where the first three form a plan and the last three form another plan. The angle between the two plans is the dihedral angel as shown in Figure 2.3[3].

[1]Adapted from
`http://www.protocolsupplements.com/Sports-Performance-Supplements/wp-content/uploads/2009/06/amino-acid-mcat1.png`

[2]`http://upload.wikimedia.org/wikipedia/commons/thumb/6/6d/Peptidformationball.svg/400px-Peptidformationball.svg.png`

[3]`http://www.structuralbioinformatics.com/images/chap7/Fig7_5%20Sign%20of%20the%20dihedral%20angle.jpg`

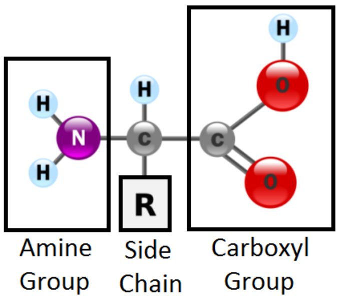


Figure 2.1: Amino Acid Structure

Name	Code		# Dihedral Angles
	3-Letter	1-Letter	
Alanine	Ala	A	0
Arginine	Arg	R	4
Asparagine	Asn	N	2
Aspartic Acid	Asp	D	2
Cysteine	Cys	C	2
Glutamic Acid	Glu	E	3
Glutamine	Gln	Q	3
Glycine	Gly	G	0
Histidine	His	H	2
Isoleucine	Ile	I	2
Leucine	Leu	L	2
Lysine	Lys	K	4
Methionine	Met	M	3
Phenylalanine	Phe	F	2
Proline	Pro	P	3
Serine	Ser	S	2
Threonine	Thr	T	2
Tryptophan	Trp	W	2
Tyrosine	Tyr	Y	3
Valine	Val	V	1

Table 2.1: List of Amino Acids

Figure 2.4[4] illustrates the Φ and Ψ angles of two amino acids bonded by a peptide bond. For example, the Ψ angle shown in Figure 2.4 is the angle formed by two plans. The first plan contains the atoms N, $C\alpha$ and C of *Amino Acid 1*. The second plan contains the atoms $C\alpha$ and C of *Amino Acid 1* and the atom N of *Amino Acid 2*. Notice that Figure 2.4 illustrates the Ψ angle of *Amino Acid 1* and the Φ angle of *Amino Acid 2* but each amino acid has the two dihedral angles.

The set of dihedral angles Φ and Ψ of all amino acids in a protein and the peptide bonds define the backbone of the protein.

The side-chain of an amino acid can have up to four dihedral angles, χ_1, χ_2, χ_3 and χ_4. Not all amino acids have the four degrees of freedom. Figure 2.5 shows part of a sequence of amino acids of a protein that illustrates the different degrees of freedom of some amino acids[36]. Table 2.1 contains the number

[4]Adapted from `https://wiki.cmbi.ru.nl/images/5/5d/Phipsi.jpg`

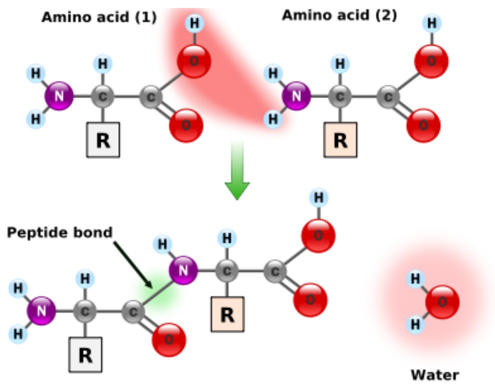


Figure 2.2: Peptide Bond

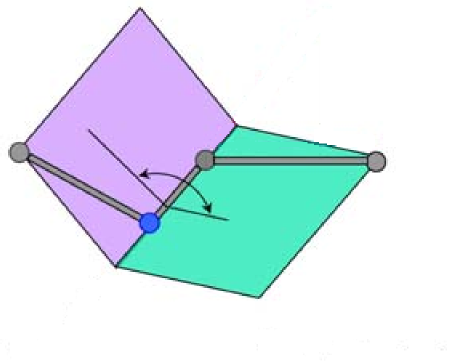


Figure 2.3: Dihedral Angle

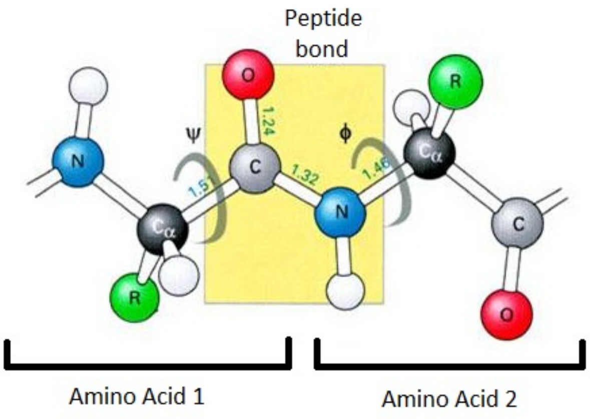


Figure 2.4: Φ and Ψ dihedral angles

of dihedral angles side-chain of each of the 20 essential amino[5].

In Figure 2.5 the position 7 of the protein has the amino acid *Valine* (V) which has only one dihedral angle (χ_1), the position 8 has *Arginine* (R) which has four dihedral angles (χ_1, χ_2, χ_3, χ_4) and the position 10 has *Glutamic Acid* (E) which has three dihedral angles (χ_1, χ_2, χ_3).

Each conformation of the side-chain of an amino acid is called a *rotamer*[6, 5]. There are infinite conformations, or rotamers, for each amino acid side-chain since the dihedral angles can be given with arbitrary precision. It is commonly considered only a representative discrete set of dihedral angles and this set is determined by a statistical analysis of the conformations that occur in nature[8]. It is also common to group the set of dihedral angles in three subsets:

- *gauche*$^+$$(-120° \leq \chi < 0°)$;
- *gauche*$^-$$(0° \leq \chi < 120°)$; and
- *trans*$(120° \leq \chi < 240°)$.

Figure 2.6 shows nine representative conformations of the amino acid *Valine* (V) which only has one dihedral angle (χ_1)[36].

[5]Dunbrack rotamer library - `http://dunbrack.fccc.edu/bbdep2010/`

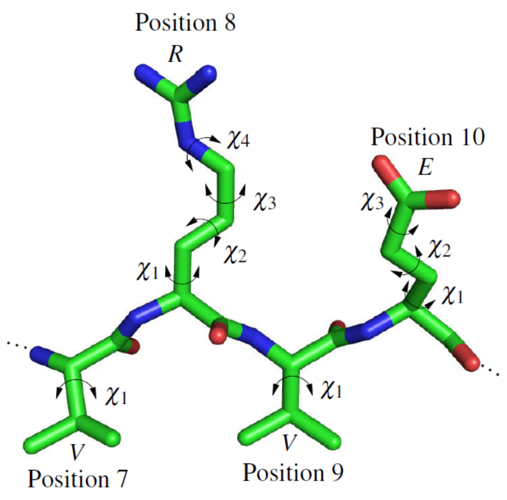


Figure 2.5: Sequence of amino acids

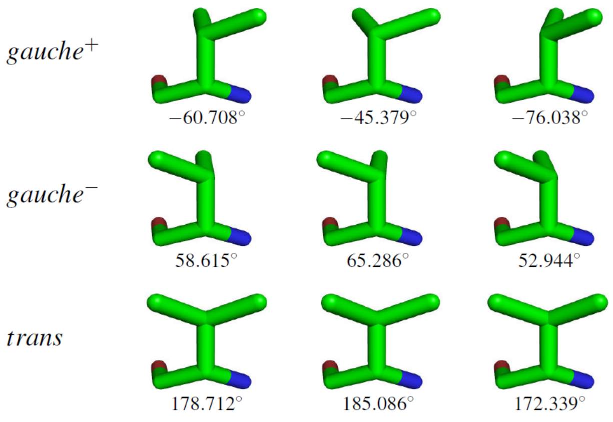


Figure 2.6: Rotamers of *Valine* (V)

2.2 Protein Design

Each protein has a specific function that is greatly determined by its structure. Hence, it is very relevant being able to determine the protein structure and respective amino acids for a specific function. Predicting the sequence of amino acids that form a protein with a specific function can be very useful in different areas such as therapeutics, treatment of diseases, the digestive process of animals or even the production of biofuels.

In what follows it will be described the protein design problem. Next, it will be described some of the existing tools related to the protein design problem.

2.2.1 Protein Design Problem

The protein design problem consists in determining a set of amino acids and respective side-chain conformations to form a protein. The set of amino acids of the protein folds into a well-defined three-dimensional structure[21]. As mentioned before this structure greatly determines the function of the protein. It is relevant to design a protein for a specific function, i.e., determining the set of amino acids that folds into a specific structure.

Many computational methods have been developed to solve the protein design problem[5, 21, 23, 18, 19].

The objective of protein design is to determine the set of amino acids that folds into a defined backbone structure minimizing the energy of the protein.

The set of amino acids of a protein folds the protein into a well-defined three-dimensional structure as said before. The process of folding depends on several factors such as the solvent, the temperature, the concentration of salts, etc. Usually the folded protein minimizes the number of hydrophobic amino acids exposed to water[7]. Therefore, most of the amino acids in the external part of the protein are hydrophilic amino acids and the amino acids near the core of the protein are hydrophobic amino acids.

In the protein design problem is common to keep the backbone of the protein fixed and determine the best side-chain conformation of each amino acid. So the structure of the protein, the amine and carboxyl groups of each amino acid are fixed and given as input. The problem therefore becomes to determine the set of rotamers that minimizes the total energy of a protein, i.e., identify the global minimum energy conformation (GMEC). Hence, protein design becomes an optimization problem.

The energy of a protein can be given by

$$E_{protein} = E_{backbone} + \sum_{i} E\left(r(i)\right) + \sum_{i} \sum_{j,j<i} E\left(r(i), r(j)\right)$$

where $E_{backbone}$ is the energy of the backbone, $E\left(r(i)\right)$ is the energy of the interaction between the rotamer $r(i)$ at position i and the backbone of the protein, and $E\left(r(i), r(j)\right)$ is the energy of the interaction between rotamer $r(i)$ at position i and rotamer $r(j)$ at position j[36]. The interaction energy between two rotamers is the sum of the interaction energies over all atoms of one rotamer with all atoms of the other rotamer[25].

Given that it is common to keep the backbone fixed, the term related to the energy of the backbone becomes irrelevant for the optimization problem and the energy function can be expressed as[44]:

$$E = \sum_{i} E\left(r(i)\right) + \sum_{i} \sum_{j,j<i} E\left(r(i), r(j)\right)$$

There are two types of energy functions[24]. One type of energy function is obtained from a fundamental analysis of the forces between particles, based on physical laws. The other type of energy function is called statistical effective energy function and is based on data derived from known protein structures.

There are several factors that can be taken into account to compute the energy functions. Some of the considered factors are force-fields, van der Waals potential, hydrogen bonding, electrostatics, internal coordinate terms, solvent interaction and entropy[17]. Not all of the methods used to compute energies take all possible factors into account. This happens because considering all factors is computationally heavy. The computation of the energy terms has a great impact in protein design. Energy functions in protein design are also called *scoring functions*.

The protein design problem, which is *NP-hard*[38], has an enormous solution space and therefore many approaches were used to try to solve this problem. A protein design problem with m positions (amino acids of the protein) and n rotamers per residue (or amino acid side-chain conformations), considering the 22 standard amino acids results in a solution space of $(22 \times n)^m$[5, 44]. If there are n rotamers per residue and 22 different amino acids then there are $22 \times n$ possible rotamers in each position of the protein. If the protein has m positions, i.e., if the protein is formed by a sequence of m amino acids, then there are $(22 \times n)^m$ possibilities. For example, a protein consisting in 100 amino acids and an average of 10 rotamers per amino acid results in $(22 \times 10)^{100} \approx 10^{234}$ possibilities.

Many approaches have been developed to obtain the optimal or near optimal solutions for the protein design problem. These approaches use different methods such as the Monte Carlo search algorithm[23], dead-end elimination[18, 16], genetic algorithms[21, 7] and the branch-and-terminate algorithm[19].

There is a trade-off between accuracy and speed when finding optimal or near optimal solutions[44]. The stochastic algorithm Monte Carlo relies on probabilities and therefore it is not possible to be sure that the solution obtained is optimal. The dead-end elimination method is deterministic and if it converges then the solution is optimal, but it is generally slower than Monte Carlo.

Note that it is possible that an optimal solution for a protein design problem is not optimal when tested experimentally due to approximations made and to possible inaccuracies in the energy function.

2.2.2 Tools

Many approaches using different methods have been developed to solve the protein design problem. There are also some tools to help solving the protein design problem. Next will be described some of the existing tools related to the protein design problem. Specifically we will describe the protein data bank and a rotamer library.

The Protein Data Bank

The Protein Data Bank (PDB)[6] was created in 1971. PDB is a single worldwide archive of structural data of biological macromolecules[3]. In PDB we can find the description of the structure of a very large number of proteins. This structure description can be downloaded as a PDB file, the most commonly used format, or as a XML file, mmCIF file, FASTA sequence and others. These files with the structure of a protein contain the amino acids that form the protein, their position in the protein, the side chain of the amino acids and the three-dimensional coordinates of each atom of the residues. The files also contain the list of contributors of the protein and the experimental technique used to determine the structure. Anyone can submit a structure to the public archive, although each structure will be validated before

[6]`http://www.pdb.org`

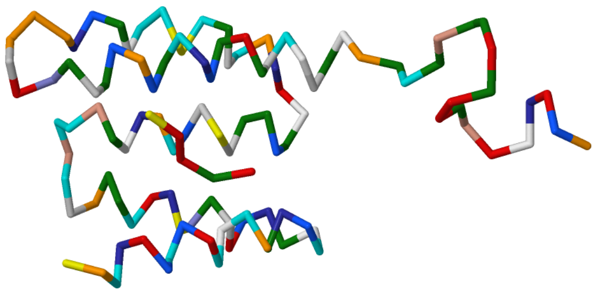


Figure 2.7: Protein 2L6J backbone

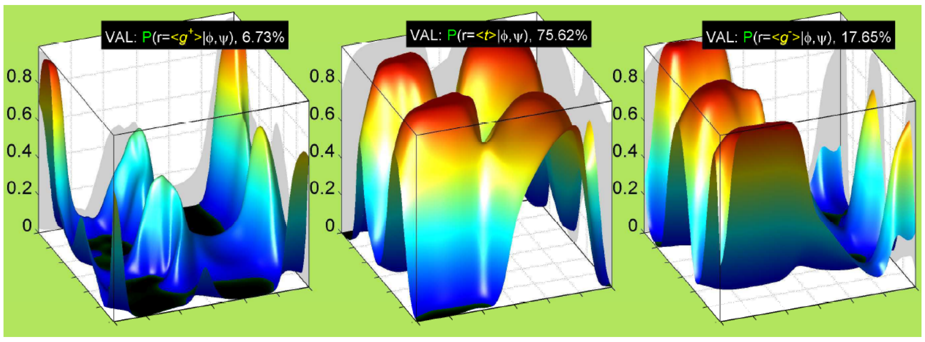


Figure 2.8: Valine rotamers probabilities

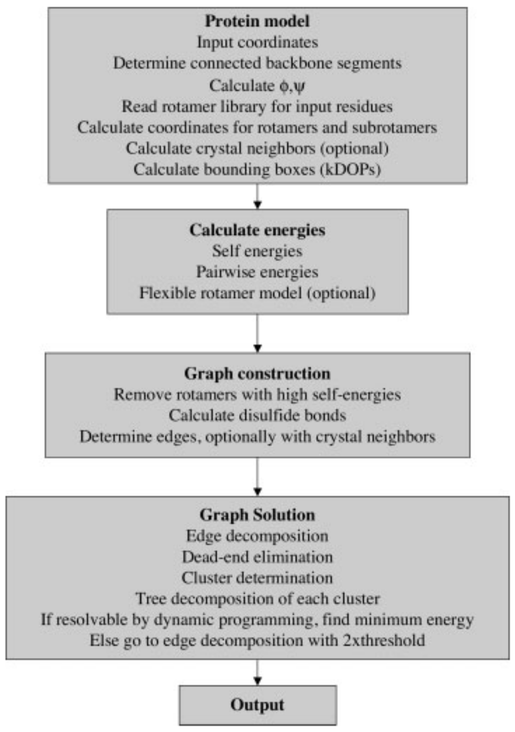
becoming part of the archive. In the website of the Protein Data Bank it is possible to see an image with the model of a protein and in most cases it is also possible to interact with a three-dimensional model.

Figure 2.7 shows the backbone of the protein *2L6J* that can be seen in the protein data bank website[7]. Each color represents an amino acid. Only the backbone, i.e., the amine group and the carboxyl group of each amino acid are represented in the figure (the side-chains of the amino acids are not represented).

Backbone-Dependent Rotamer Library

Numerous programs of protein design use discrete sets of rotamers. A library of rotamers is therefore very important. Since 1993, a backbone-dependent rotamer library that is used in protein structure determination, prediction and design has been available[40]. It is also available a 2010 version of the Dunbrack backbone-dependent rotamer library[8]. This backbone-dependent rotamer library uses known protein structures and a Bayesian formalism to build probabilities graphics for each amino acid. This library consists in rotamers frequencies, mean dihedral angles, and variances as a function of the backbone dihedral angles Φ and Ψ. A kernel density estimation is used in order to produce smooth and continuous estimates of the rotamers probabilities. Figure 2.8 shows the probabilities of the rotamers dihedral angles range of the Valine (VAL) amino acid as a function of the Φ and Ψ dihedral angles. These graphics can be seen in the backbone-dependent rotamer library [9].

[7] `http://www.pdb.org/pdb/explore/explore.do?structureId=2L6J`
[8] `http://dunbrack.fccc.edu/bbdep2010/`
[9] `http://dunbrack.fccc.edu/bbdep2010/Components/05PercSmoothed/VAL/05PercSmoothed.VAL.RotaProb.pdf`



Protein model
Input coordinates
Determine connected backbone segments
Calculate φ,ψ
Read rotamer library for input residues
Calculate coordinates for rotamers and subrotamers
Calculate crystal neighbors (optional)
Calculate bounding boxes (kDOPs)
Calculate energies
Self energies
Pairwise energies
Flexible rotamer model (optional)
Graph construction
Remove rotamers with high self-energies
Calculate disulfide bonds
Determine edges, optionally with crystal neighbors
Graph Solution
Edge decomposition
Dead-end elimination
Cluster determination
Tree decomposition of each cluster
If resolvable by dynamic programming, find minimum energy
Else go to edge decomposition with 2xthreshold
Output

Figure 2.9: SCWRL4 Architecture

2.3 Related Work

In the recent past have been developed programs that are related to the protein design problem. In this section will be described some of these programs, in particular SCWRL4, SHARPEN, Rosetta and ProtSAT.

SCWRL4

SCWRL4[22] is a program used for prediction of protein side-chain conformations, a simplification of the protein design problem where the residues of a protein are kept fixed and only the side-chain conformations are predicted. SCRWL4 does not guarantee a global minimum of the energy function when determining the side-chain conformation of a protein. Instead, SCWRL4 tries to quickly produce a near optimal result.

Figure 2.9 illustrates the architecture of SCWRL4[22]. The SCWRL4 architecture can be divided in five modules as shown in the figure.

In the first module of the SCWRL4 architecture the input is processed. The input of SCWRL4 is a PDB file containing four backbone atoms N, $C\alpha$, C and O per residue (atoms illustrated in Figure 2.4). The residues given as input are kept fixed and only the rotamer for that residue is determined. The dihedral angles Φ and Ψ are calculated from the coordinates of each atom of the residues of the protein given as input. For each residue, the Dunbrack backbone-dependent library is read to obtain the set of possible rotamers. Then, the side-chain coordinates are built for all rotamers. A collision detection

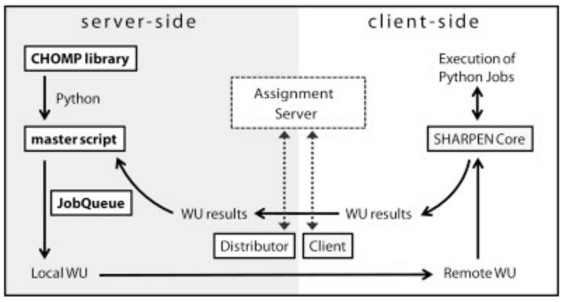


Figure 2.10: SHARPEN Architecture

algorithm is used to eliminate impossible pair of rotamers.

In the second module, the energy values for each rotamer and pair of rotamers are calculated. The energy function used takes into account the repulsive and attractive van der Waals terms and hydrogen bonding terms. The energy terms are calculated for the set of possible rotamers.

The third module is the graph construction. In this module is created an interaction graph that represents the side-chain placement problem. In this graph, vertices represent residues and each edge represents the existence of at least one pair of rotamers (one from each vertex connected by the edge) with nonzero interaction.

Afterwards, the solution of the graph is determined. The edge decomposition method and dead-end elimination are then applied to the interaction graph which has been constructed to find a solution within a threshold.

In the last module is returned the output. SCWRL4 produces as output a PDB file with the whole protein model identified. The PDB file therefore contains the three-dimensional coordinates of each atom of the designed protein.

SHARPEN

SHARPEN is a scalable distribution of an open-source library for protein design and structure prediction[33].

Figure 2.10 shows the architecture of SHARPEN[33] which is a server-client architecture.

The core of SHARPEN (Systematic Hierarchical Algorithms for Rotamers and Proteins on an Extended Network) is called CHOMP and is deployed in the server-side. CHOMP (Combinatorial Hierarchical Optimization for Macromolecular Problems) is an open-source software library directed to the optimization of macromolecular structures. The CHOMP library implements several functions and modules to be used by the users. CHOMP provides data representations for atoms, amino acids and rotamers. CHOMP also provides modular combinatorial optimization methods such as Monte Carlo and Simulated Annealing to solve protein design and structure prediction problems.

There are two energy functions implemented in the library: an all-atom energy function and a force field function.

To solve the protein design and structure prediction problems is used an interaction graph to perform combinatorial optimizations.

SHARPEN receives as input a python script called *master script* which uses the CHOMP library and is ran in the server-side of the software.

SHARPEN has its core in the client-side of the software and makes the connection between the client and the CHOMP library in server-side. SHARPEN uses a script called JobQueue for splitting the *master script* into several "Jobs" (work units). The JobQueue script distributes the several work units created through the client-server communication system called *Distributor* to clients. Then the JobQueue receives the results of the work units from the clients.

This architecture allows users to deploy the same script (*master script*) on a single processor, multiple processors on the same machine, a local heterogeneous cluster, or a large-scale distributed computing network, improving the scalability of the protein design and structure prediction problems.

Rosetta

Rosetta[39] is a well known open-source program. The Rosetta program can be used in several problems related to proteins such as structure prediction, protein design, protein-protein docking and loop modeling. This program can also score a given protein, i.e., it can calculate the energy of a given protein. Rosetta uses the Dunbrack backbone-dependent rotamer library, described before, to restrict the possible conformations to a discrete set when solving protein design problems for example.

The energy function used by Rosetta takes into account several factors making a more precise scoring of proteins. Some of these factors are van der Waals potential, solvation effects, hydrogen bonding and electrostatics.

One module of the Rosetta program that solves the protein design problem is RosettaDesign[32]. RosettaDesign takes as input the backbone coordinates of the target protein structure in PDB-format and a specification of which residues to design. Therefore, RosettaDesign keeps fixed the structure of the proteins and determines the set of residues and respective rotamers that minimize the total energy of the protein. Note that Rosetta can determine only the side-chain conformation of a protein, can determine the residues and respective side-chain and can design only specific positions of a protein, depending on the specification given as input.

The backbone-dependent rotamer library is used to determine the discrete set of rotamers that will be considered for the given protein. The RosettaDesign module uses a Monte Carlo optimization with simulated annealing to find a solution.

The output of RosettaDesign is a file with the three-dimensional coordinates of each atom of all amino acids and respective side-chain in PDB-format along with the energies of the redesign protein.

Rosetta current version is 3.4[10].

[10]`http://www.rosettacommons.org/`

ProtSAT

In 2009 was developed a SAT-based program for protein design called ProtSAT[36]. The objective of ProtSAT is to identify the global minimum energy conformation (GMEC) for a given protein structure. ProtSAT considers 20 essential amino acids (the amino acids in table 2.1). ProtSAT also considers a discrete set of possible rotamers. The backbone of the protein is kept rigid and is the input data of the program. The energy function considered for the protein design is

$$E_{protein} = E_{backbone} + \sum_{i} E\left(r(i)\right) + \sum_{i} \sum_{j,j<i} E\left(r(i), r(j)\right)$$

The energy values are pre-computed and then is applied a dead-end elimination method. The objective of the dead-end elimination is to eliminate rotamers that cannot be in the GMEC. For example, if for a given position i in the protein a rotamer a is always energetically more favorable than rotamer b then the rotamer b can be removed from the set of possible rotamers for position i. After using the dead-end elimination method, a SAT encoding is made. First, it is defined that each position of the protein has exactly one rotamer. Then the constraints obtained by dead-end elimination are added. For example if a pair of rotamers (a, b) is eliminated, meaning that the protein cannot have both rotamers a and b at the same time in its structure, then the clause $(\neg a \vee \neg b)$ is added.

ProtSAT makes a hierarchical decision. For each position of the protein, first is assigned the amino acid for that position, then it is determined the range of the dihedral angle χ_1 ($gauche^+$, $gauche^-$, $trans$). The range of the dihedral angles χ_2, χ_3 and χ_4 are determined as well. Finally the exact rotamer is determined for that position, taking into account the amino acid and the range of the dihedral angles already assigned.

Figure 2.11 illustrates the search tree for a position i[36]. In this figure are shown three branches of a search tree as an example. In the first level of the tree is chosen the amino acid for position i. In the second level is defined the range of the dihedral angle χ_1, the third level is χ_2 and so on. The leaf level contains the exact rotamers that can be part of the solution. In the figure, χ_x represents that the range of the dihedral angle of χ_x is $gauche^+$, $\tilde{\chi}_x$ represents that the range is $gauche^-$, and $\tilde{\tilde{\chi}}_x$ represents that the range is $trans$. In the left branch of the tree is chosen Alanine (A) to occupy the position i of the protein. Alanine only has one rotamer so there is only one possible rotamer (r_k) in this branch of the search tree. The second branch illustrated shows that the amino acid chosen is Lysine (K), then is chosen $gauche^-$ as the range for χ_1, $gauche^+$ for χ_2, $gauche^-$ for χ_3 and $gauche^+$ for χ_4. At the leaf level, the rotamer r'_k is chosen for position i. The right branch of the tree shows the possibility of the Valine (V) amino acid for position i. As Valine only has one degree of freedom that is χ_1, only χ_1 has different ranges in the search tree. In the figure is shown the branch that corresponds to the attribution of $gauche^+$ to the range of the dihedral angle χ_1 of the amino acid Valine. There are three possible rotamers for Valine with $\chi_1 = gauche^+$ (r''_k). Notice that only one leaf of the entire search tree can be in the solution of the

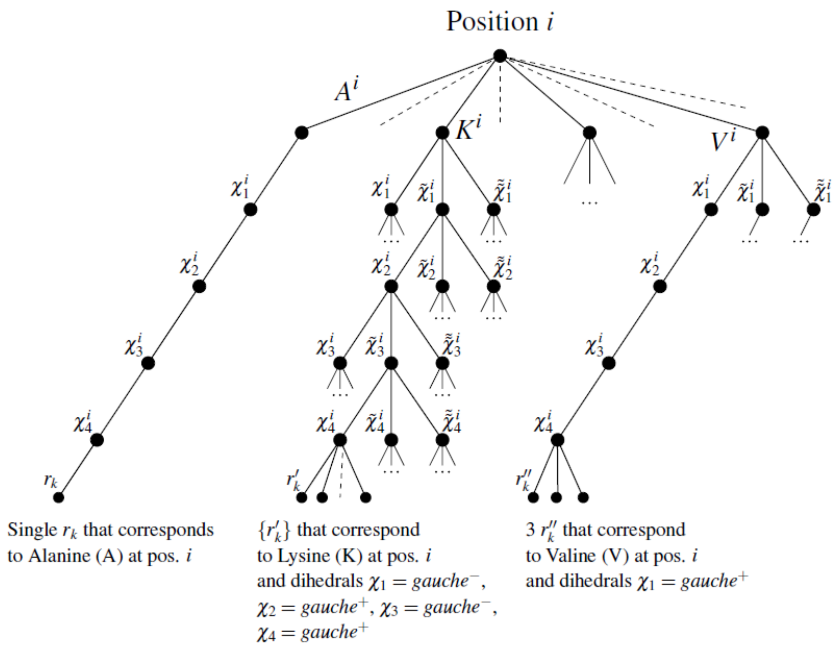

Position i
A^i
K^i
V^i
χ_1^i
$\tilde{\chi}_1^i$
$\tilde{\tilde{\chi}}_1^i$
χ_2^i
$\tilde{\chi}_2^i$
$\tilde{\tilde{\chi}}_2^i$
χ_3^i
$\tilde{\chi}_3^i$
$\tilde{\tilde{\chi}}_3^i$
χ_4^i
$\tilde{\chi}_4^i$
$\tilde{\tilde{\chi}}_4^i$
r_k
r'_k
r''_k
Single r_k that corresponds to Alanine (A) at pos. i
$\{r'_k\}$ that correspond to Lysine (K) at pos. i and dihedrals $\chi_1 = gauche^-$, $\chi_2 = gauche^+$, $\chi_3 = gauche^-$, $\chi_4 = gauche^+$
3 r''_k that correspond to Valine (V) at pos. i and dihedrals $\chi_1 = gauche^+$

Figure 2.11: ProtSAT search tree

protein design problem. Each position of the protein can only have one rotamer.

3 Answer Set Programming

In this chapter will be described in detail Answer Set Programming, starting with the general idea and the logic concepts associated with it. Then will be described the Answer Set Programming language with some examples for a better understanding. In the end of this section we will describe two tools commonly used.

3.1 Answer Set Programming Concepts

Answer Set Programming (ASP) is an approach to declarative problem solving. ASP is commonly applied to difficult, NP-Hard, search problem. The idea is to create a program that is the definition of the problem, the constraints and facts, instead of creating a program that is an algorithm to solve the problem. Knowledge representation and nonmonotonic reasoning, logic programming with negation, databases and Boolean constraint solving are the roots of ASP[12]. In the ASP approach, a problem is solved by representing the problem as a logic program such that the solutions of the program correspond to the solutions of the problem. An ASP solver can then be applied to the logic program to compute (i.e. search for) the solutions.

An ASP *program*[29] is a set of *rules*. A *rule* is a representation of some knowledge, has a head and a body and is represented as

$$Head \leftarrow Body.$$

If the rule has no head, then is in the form

$$\leftarrow Body.$$

and is a constraint. For example, if we have the rule

$$\leftarrow p.$$

it means that p must not hold.

If the rule has no body it is considered a fact and the symbol '$\leftarrow$' is dropped. The rule

$$p.$$

means that p must hold.

In general, a *rule* has the following form

$$L_1, L_2, ..., L_k, not\ L_{k+1}, ..., not\ L_l \leftarrow L_{l+1}, ..., L_m, not\ L_{m+1}, ..., not\ L_n.$$

where L_i is a literal and *not* L_i is the negation of the literal L_i. The concept associated with the symbol *not* is negation as failure, i.e., a literal L_i is considered to be false if there is nothing that says that L_i is true.

If the head of a rule has cardinality greater than 1 then is a disjunctive rule. A program with disjunctive rules is considered a disjunctive program. A normal logic program has only rules without head or rules where the head has only one positive literal and no negative literals.

Logic programs can have several answer sets. An *answer set*[29, 14, 2], or stable model, of a logic program is a set of literals that satisfies all the rules of the program, i.e., that make all the rules of the program true. For instance, if we consider the program with the following rules

$$\begin{aligned} p &\leftarrow q. \\ q &\leftarrow not\ r. \end{aligned}$$

This program has one stable model that is $\{q, p\}$. Notice that *not r* it is not included in the stable model because only positive literals are considered in the answer set. Therefore a stable model is the answer of a logic program, which contains the literals that must be true so that there are no conflicts between the rules of the program.

A more formal definition of a stable model (or stable set) is presented by Gelfond et al[14]:

Definition 1 ***(Stable Set)***
Let Π be a logic program. For any set M of atoms from Π, let Π_M be the program obtained from Π by deleting

1. *Each rule that has a negative literal $\neg B$ in its body with $B \in M$, and*

2. *All negative literals in the bodies of the remaining rules.*

Clearly, Π_M is negation-free, so that Π_M has a unique minimal Herbrand model. If this model coincides with M, then we say that M is a stable set *of Π.*

Most ASP solvers use grounders as front-end. A grounder is a tool that for a given logical program computes an equivalent variable-free version of the program (a ground program), i.e., replaces the variables in the rules with all possible instantiations. The grounders *lparse*[1] and *gringo*[2] are the most commonly used grounders for ASP. Figure 3.1 shows the general process of solving an ASP program, i.e., shows

[1]`http://www.tcs.hut.fi/Software/smodels/`
[2]`http://potassco.sourceforge.net/`

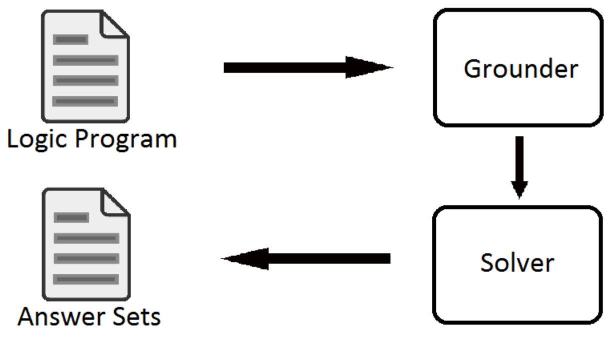


Figure 3.1: General solving process

the input (logic program) accepted by the grounder which gives its results to the solver and the solver outputs the answer sets of the input program.

Answer set programming can be applied not only to decision problems but also to optimization problems. There are two functions in ASP for optimization problems: *minimize* and *maximize*. It is possible to optimize by two different types of criteria. It is possible to optimize by the number of instances of specific predicates; for instance, it is possible to minimize the number of instances of the predicate `x`. It is also possible to optimize by weights, i.e., it is possible to assign a weight to each predicate and then optimize the sum of the weights of the instances of the predicates[3]. Therefore, it is possible to compute the optimal solution, if there is one, applying ASP to optimizations problems. If the problem does not have a solution, then the information that the program is unsatisfiable is returned. In ASP, the maximizations can be transformed in minimizations and the negative weights in positive weights, so that the solving process can be more efficient[10, 41].

Answer set programming is very flexible and it is possible to have multi-criteria optimization[10]. This is good for problems that have multiple criteria for optimization, for example, a problem with cost and rewards where it is desirable to minimize the cost and maximize the rewards. It is also possible to assign different priorities to different optimization criteria, hence allowing ASP to solve more different types of problems. Although it is possible to assign the same priority to different optimization criteria, this must be done carefully. Assigning the same priority may not lead to the expected results. After transforming maximizations in minimizations and transforming negative values in positive ones, ASP will sum the optimization values if there is more than one optimization criterion with the same priority. Hence, the optimization values of different optimization criteria with the same priority should have the same unit. For example, if there are two optimization criteria with the same priority, one regarding Euro values and other Great Britain Pounds, the sum of the two may not lead to the desired result. It is necessary to adjust the optimization values.

[3]`http://heanet.dl.sourceforge.net/project/potassco/potassco_guide/2010-10-04/guide.pdf`

3.2 Answer Set Programming Language

The input languages of *lparse* and *gringo*, the two grounders mentioned before, are very similar to the *Prolog*[4] language. For example the rule

$$p \leftarrow q.$$

Is written in *lparse* or *gringo* as

```
p :- q.
```

In ASP it is common to used predicates. Predicates in ASP are identical to the ones in *Prolog.* A *predicate* is a literal of the form:

$$p(n_1, n_2, ..., n_n).$$

where n_i is an argument of the predicate p. In ASP the name of the predicates cannot start with a capital letter.

The arguments of a predicate can start with capitalized letters or lower letters. Arguments starting with capitalized letters mean that they are not instantiated. Arguments starting with lower letters mean that they are instantiated with objects of the world.

ASP is considered a closed world, i.e., only the known facts are considered to be true and only the objects that appear in some fact are considered to exist. Everything else is considered to be false or do not exist.

For a better understanding of the language, we will show a simple example of predicates, rules and ASP program. We will define the following predicate for this example:

```
fatherOf(X,Y).
```

The predicate `fatherOf` represents a father-son relationship and indicates that `X` is the father of `Y`. Notice that the arguments start with a capitalized letter, therefore they are not instantiated.

An instance of this predicate could be, for example

```
fatherOf(john, peter).
```

meaning that '`john`' is the father of '`peter`'. As '`john`' and '`peter`' are instances, the arguments of the predicate `fatherOf` start in lower case letters as said before. In this case, '`john`' and '`peter`' are considered the objects of the world.

Now consider the rule

```
father(X) :- fatherOf(X,Y).
```

The above rule introduces a new predicate `father` that indicates that `X` is a father. This rule represents

[4]`http://www.swi-prolog.org/`

that if `X` is the father of `Y` then `X` is a father. The `X` argument of the predicate `fatherOf` matches the `X` argument of the `father` predicate. The value of `X` is the same in both predicates. In this rule, `Y` can be discarded because its value is irrelevant. Therefore the rule can be rewritten as

```
father(X) :- fatherOf(X,_).
```

The '_' symbol represents any value.

The following program:

```
fatherOf(john,peter).
father(X) :- fatherOf(X,_).
```

has one stable model:

```
Answer: 1
Stable Model: fatherOf(john,peter) father(john)
```

One significant difference between ASP and *Prolog* is that ASP accepts *choice rules*[30]. An example of a choice rule is

```
{s,t} :- p.
```

This rule means that if `p` holds then the atoms `s` and `t` can be included in the stable model arbitrarily. We will give an example of the use of choice rules. Consider the following program written in ASP:

```
{eatCake, eatChocolate} :- goToParty.
goToParty :- not occupied.
```

This program intuitively represents that if we go to a party we can eat cake and/or chocolate or do not eat nothing at all (first rule); it is our choice. To go to a party we must not be occupied (second rule). This program has four answer sets:

```
Answer: 1
Stable Model: goToParty
Answer: 2
Stable Model: eatCake goToParty
Answer: 3
Stable Model: eatChocolate goToParty
Answer: 4
Stable Model: eatCake eatChocolate goToParty
```

The answer sets of the program represent that, once nothing says we are occupied, then we can go to the party. Therefore, all answer sets contain the literal '`goToParty`'. As we go to the party we can choose not to eat (Answer 1), eat only cake (Answer 2), eat only chocolate (Answer 3) or eat cake and chocolate (Answer 4).

It is possible to include numerical bounds on a choice rule. A number immediately before the choice rule represents the lower bound and a number immediately after the choice rule represents de upper bound. So if we have `{eatCake, eatChocolate}1` this means that the answer sets must include at most 1 atom of the set `{eatCake, eatChocolate}`, i.e., an answer set can contain `eatCake` or `eatChocolate` or none, but cannot contain both. If we have `1{eatCake, eatChocolate}` this means that the answer must include at least 1 atom of the set `{eatCake, eatChocolate}`, i.e., an answer set can contain `eatCake` and/or `eatChocolate`, and so must have at least one of them.

Another difference between ASP and *Prolog* is that ASP has two types of negations: strong negation and weak negation. The weak negation is negation as failure, i.e., a literal is considered to be false if nothing says that it is true. On the other hand, in strong negation a literal is considered false if that literal is specifically defined as false. In ASP the weak negation is represented by '`not`' and the strong negation by '`-`'. An example of each type of negation in ASP is shown next.

```
use_hat :- not raining.
use_hat :- -raining.
```

The first rule illustrates the use of negation as failure (weak negation). This rule represents that we use a hat if we can assume that it is not raining. The fact that it is raining is an unknown information and therefore is considered to be false. The second rule represents strong (or "classical") negation and indicates that we use a hat if we know for sure that it is not raining. Notice that in this rule being able to assume that it is not raining is not enough to use a hat.

An example of an ASP encoding is in appendix A. In this appendix is described in detail the encoding of a very well known game called kakuro.

3.3 Tools

There are several answer set programming solvers. In 1995 was developed *smodels*[5][41]. *Smodels* is used with *lparse* as grounder for the logic program. Another ASP solver is *dlv*[27] which became available in 1997. A difference between *smodels* and *dlv* is that *dlv* implements DLP (Disjunctive Logic Programming) which allows logic programs to have disjunctions in the rules' head. There are some ASP solvers that are SAT-based, i.e., based on SAT solvers. *Assat*[31], made available in 2002, is one SAT-based ASP solver. Another SAT-based ASP solver is *cmodels*[15, 28]. *Cmodels* uses the SAT solver *simo* as a search engine. *Cmodels* also computes answer sets to disjunctive logic programs and for logic programs with

[5]`http://www.tcs.hut.fi/Software/smodels/`

choice rules. In 2007 was created *clasp*[13, 12], a conflict-driven ASP solver. This solver is used with *lparse* or *gringo* as grounder. *Clasp* with *gringo*[11] as grounder will be used in this work.

Gringo[11] works as a grounder for *clasp*, as said before. A grounder is a tool that computes a variable-free program (ground program) for a given logical program given as input, i.e., makes a version of the initial program with all the rules instantiated. *Gringo* includes some aggregate functions allowing more flexibility to solve problems. Some of these functions are `#count`, `#sum`, `#minimize`, `#maximize`, `#even`, `#odd`, etc. The `#count` function corresponds to the syntax used in the choice rules (or cardinality constraints). So when we write `l{...}u` we are defining that the count of the literals inside brackets are greater than or equal to `l` and lesser than or equal to `u`. The same happens with the `#sum` function and the weight constraints. It is possible to associate weights with literals and use them in constraints. To associate a weight with a literal is used the symbol '='. If we write `l[...]u` we are defining that the sum of weights of literals inside square brackets is bounded by the lower bound `l` and the upper bound `u`. Consider the following rule

```
1[p(X) = X)]7.
```

In this rule, the '=' symbol is an attribution of weight. The rule is associating the weight `X` with literal `p(X)`, where `X` is the argument of `p`. The square brackets in the rule mean that the sum of weights of the instances of the predicate `p` must be between 1 and 7.

Other very important functions included in gringo are the `#maximize` and `#minimize` functions. These functions allow solving optimization problems. It is possible to maximize (or minimize) the count of a predicate, the sum of certain weights, and so on. It is also possible to have more than one optimization criterion[10] and give different weights to each criterion. An example of a maximization rule is:

```
#maximize[p(X)=X].
```

which means that the objective is to maximize the sum of weights of the predicate `p`, where `p(X)` has weight `X`.

The tool *clasp*[13, 12] is an ASP solver. *Clasp* uses two concepts of two great areas of artificial intelligence[13]:

1. The concept of conflict-driven learning from SAT
2. The concept of nogoods from Constraint Satisfaction Problems (CSP).

Clasp is different from the other ASP solvers because it was originally designed, created and optimized for conflict-driven ASP solving, using the concept of nogoods. The architecture of clasp is shown in figure 3.2[13].

Clasp is a highly configurable ASP solver. It provides some options for the user to configure the solver[9]. An example of these options is the `--heuristic` option. This option allows to choose

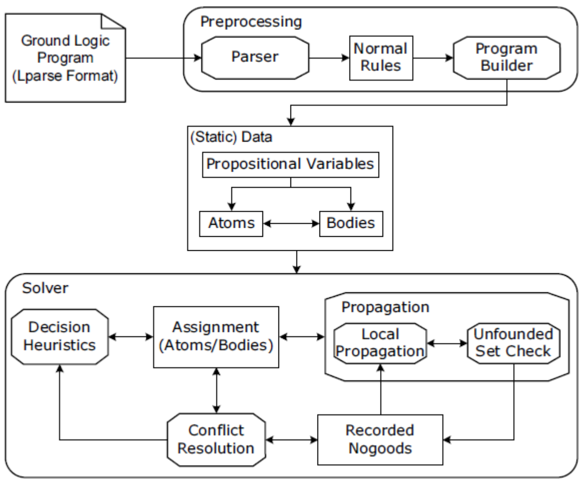


Figure 3.2: Clasp's Architecture

which heuristic to be used by *clasp*. Another option is `--trans-ext` which determines whether choice rules or cardinality and weight constraints are transformed into normal rules during the parsing phase. `--sat-prepro` is a very useful option. This option allows doing a SAT-based preprocessing once the program is transformed into Boolean constraints. It is also possible to adjust the restart policy used by *clasp*. To change the restart policy, *clasp* provides the option `--restarts`. There are more options available and to see them one has to use the option `--help`.

There are some variants of *clasp* like *claspD*, *claspar*, *clingo*, etc[12]. *ClaspD* is oriented to NP^{NP} problems and accepts disjunctive logic programs. There is also a variant of *clasp* that is focused on parallelization. *Claspar* is a distributed version of clasp which uses parallel processing during search. There is a version of clasp that results from the union between *gringo*, the grounder, and *clasp*, the solver, called *clingo*. Some problems, like planning problems, require some iteration during the computation of the solutions. *Iclingo* works with incremental programs which have incremental variables.

4 Implementation

The protein design problem is a very interesting problem. On the other hand, answer set programming is a recent approach to declarative problem solving. Our goal is to solve a simplified version of the protein design optimization problem using answer set programming.

In this chapter will be presented the codifications made to solve the problem using answer set programming, more specifically a simple codification and alternative codification, one of them using two optimization criteria. In this chapter will also be described the implementation of Dead-End Elimination (DEE) methods using answer set programming and Java.

4.1 Problem Codification

Two approaches were developed to solve a simplification of the protein design problem. The protein design problem consists in determining the set of amino acids and respective side-chain that form a protein with a specific function, minimizing the total energy of the protein. In our case we made a simplification of the problem. It is considered that the backbone of the protein is fixed and only the side-chains are designed. It is also considered only a discrete set of rotamers using the Dunbrack Backbone-Dependent Rotamer Library. These rotamers are given as input. For the energies is used the source code of Rosetta to compute the score of the interactions between each possible rotamer and the backbone and the interactions between each pair of rotamers at different positions. The positions to design is also given as input, therefore it is possible to design the entire protein or only a subset of specific positions.

Figure 4.1 illustrates the considered protein design problem. The figure shows that we use Rosetta to compute the interaction energies, using the PDB file of the protein. The PDB file is also used to define the positions of the protein to be designed. Figure 4.1 also shows that three types of data are given as input: the positions of the protein to be designed; the problem codification; and the interaction energies. The output is the set of rotamers that minimizes the total energy of the protein.

In a first approach was developed an ASP program to determine the optimal set of rotamers for a given protein. The program takes as input the backbone positions of the protein to design, the residues of each position and the energy values of all rotamers considered using the score function of rosetta and the rotamer library mentioned before. The program produces as output a side-chain conformation for each protein position designed. In this first approach only a side-chain prediction is made, i.e., the backbone of the given protein is kept fixed and the residues (or amino acids) of the protein received in the protein backbone are kept fixed as well. Only the best set of rotamers is determined.

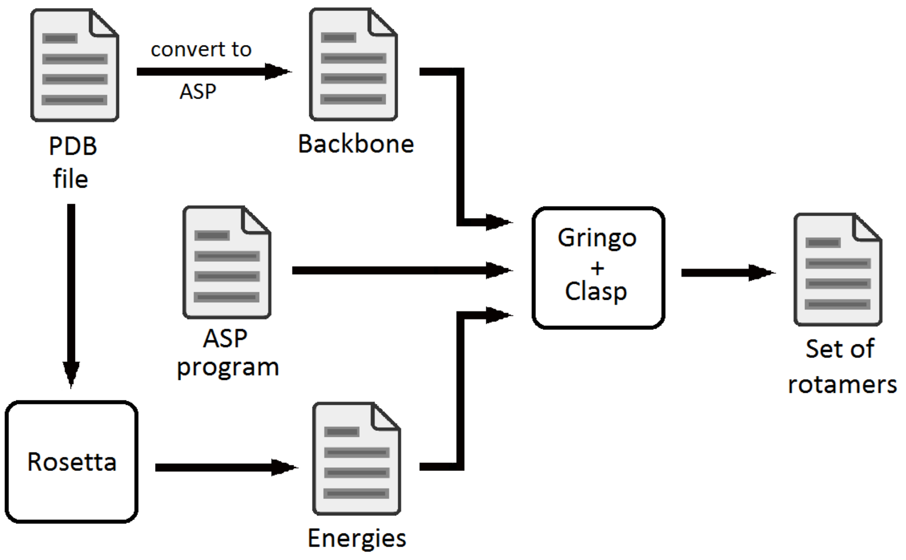


Figure 4.1: Schema of the considered problem

The second approach to the problem is similar to the first approach. The difference between the two approaches is that the second one does not keep the residues fixed. The backbone of the protein given as input is fixed and the amino acids and respective side-chains are flexible. The objective is to determine the set of amino acids and respective side-chain that minimizes the total energy of the protein.

The goal of the first approach is the same of the software SCWRL4. This approach can be seen as a sub-problem of the protein design problem where the amino acids are already determined. The second approach has the same objective of ProtSAT: predict amino acids and respective side-chain.

The minimization criterion for both approaches is the following total energy function:

$$E = \sum_{i} E\left(r(i)\right) + \sum_{i} \sum_{j,j<i} E\left(r(i), r(j)\right)$$

where $E\left(r(i)\right)$ is the energy of the interaction between the rotamer $r(i)$ at position i and the backbone of the protein, and $E\left(r(i), r(j)\right)$ is the energy of the interaction between rotamer $r(i)$ at position i and rotamer $r(j)$ at position j. As referred before these energy scores are computed by the score function of rosetta.

Clasp and *gringo* were used to solve the ASP program encoded in both approaches mentioned above. The use of these tools and the fact that the two approaches were developed using answer set programming are the main differences from the existent approaches. Using a different language, hence different tools, can improve the resolution of the protein design problem.

The two approaches are similar and the difference between them is the input. In the first approach the residues of the protein are given as input and in the second approach are not. Hence, it will be presented a single codification that works on both approaches.

All the code lines will be numerated so that they can be referred more than once.

4.1.1 Simple Codification

To encode this problem using answer set programming, it is first necessary to represent the input in predicates. It is assumed that the positions of the protein to design, the possible rotamers for each position and respective energy score for the interaction with the backbone of the protein and the energy scores for the interaction between each pair of possible rotamers at different positions are received as input. For the first approach, for which only side-chain conformation prediction is made, it is also received as input the residue (or amino acid) on each position to be design.

To represent the positions of the protein to be designed it is created the predicate `position` (4.1.1.1) with one argument.

`position(Pos).` (4.1.1.1)

The argument represents the position identifier and is a positive integer number.

For the representation of the possible rotamers is used the following predicate with four arguments.

`possibleRotamer(Pos,Id,Amin,Energy).` (4.1.1.2)

The first argument represents the position identifier for that rotamer, i.e. identifies the correspondent position of the protein in which that rotamer can be placed. The second argument is a unique identifier for that rotamer within the protein position. This is a positive integer number and is used, along with the position identifier, to identify uniquely the rotamer. Therefore, to each rotamer (or side-chain conformation) it is assigned an unique identifier. The third argument represents the residue correspondent to the rotamer and is represented by the three letter code (in lower case) shown in table 2.1. The fourth and last argument of the `possibleRotamer` predicate (4.1.1.2) corresponds to the energy score of the interaction between the rotamer and the backbone. This energy value must be an integer number.

To represent the energy scores of the interaction between two rotamers at different positions, is used the predicate `interEnergy` (4.1.1.3).

`interEnergy(Pos1,Id1,Pos2,Id2,Energy).` (4.1.1.3)

The first two arguments correspond to one rotamer identifier (position and unique identifier within the position), as mentioned in the `possibleRotamer` predicate (4.1.1.2). The third and fourth arguments identify, similarly to the firsts arguments, the other rotamer present in the interaction. Finally, the last argument is the energy score of the interaction between the two rotamers identified in the previous arguments. As in the `possibleRotamer` predicate, this value is an integer number. To avoid duplicated information, since the interaction between two rotamers are two-sided, only one-side interaction is represented (since they are equal) and it is always considered that the position of the first rotamer identified in the predicate is lesser than the position of the second rotamer. For example, if we want to represent the interaction between the first rotamer of position 1 of the protein (e.g. `possibleRotamer(1,1,5)`) and the first rotamer of position 2 (e.g. `possibleRotamer(2,1,-6)`), only the interaction from position 1 to position 2 is represented. Therefore, if that interaction has, for example, score 3, we will use the following predicate:

`interEnergy(1,1,2,1,3).` (4.1.1.4)

And **not**:

`interEnergy(2,1,1,1,3).` (4.1.1.5)

For the first approach, in which the residue for each position is given as input, it is used the predicate `residue` (4.1.1.6) with two arguments.

`residue(Amin,Pos).` (4.1.1.6)

The first argument is the three letter code for the amino acid (see table 2.1). The second argument corresponds to the position identifier of the protein.

With these predicates, all the input information is represented. However, we need more predicates to represent more information.

To identify all the possible amino acids that exist it is used the following predicate:

`aminoAcid(Amin).` (4.1.1.7)

where `Amin` is the three letter code of the amino acid.

It is also necessary to represent that a rotamer is part of the solution, i.e., that a specific rotamer belongs to the protein designed. For that purpose is used the predicate `rotamer` (4.1.1.8) with three arguments:

`rotamer(Pos,Id,Amin).` (4.1.1.8)

where `Pos` and `Id` identify the rotamer (position and identifier) and `Amin` represents the amino acid.

Given the need to represent that a rotamer is part of the final solution, it is necessary to represent that a specific energy score is part of the final solution. For this information is used the predicate `energy` (4.1.1.9).

`energy(Pos1,Id1,Pos2,Id2,Energy).` (4.1.1.9)

This predicate is very similar to the `interEnergy` (4.1.1.3) predicate mentioned before and like the `interEnergy` predicate only one side of the interaction is represented (`Pos1` must not be greater than `Pos2`). There are two differences between the `interEnergy` predicate and the `energy` predicate (4.1.1.9). One is that the `energy` predicate represents that the interaction energy belongs to the final solution. The second difference is that this predicate is used to represent the interaction between a rotamer and the backbone as well. This is made by repeating the identifier of the rotamer, i.e., if the pair (`Pos1`,`Id1`) is equal to pair (`Pos2`,`Id2`) which means it is an interaction between the rotamer identified by (`Pos1`,`Id1`) and the backbone. Notice that the `interEnergy` predicate can not have duplicated identifiers because it represents an interaction between two different rotamers at different positions.

At this point we have all the necessary information to represent the problem to solve. To solve our problem now it is necessary to represent the rules of the problem. We will now describe each rule individually so that the whole program can be understood.

The main idea is to define the conditions of the problem. The first thing needed is to define that each protein position to be designed must have exactly one residue (or amino acid). We can write this restriction in ASP as

```
1{residue(Amin,Pos) :  aminoAcid(Amin)}1 :- position(Pos).
```
(4.1.1.10)

The brackets in this rule, as mentioned in section 3.3, mean that there must be at least one predicate (of the ones inside the brackets) and at most one predicate. The predicate `residue` is the one limited by the boundaries. The symbol ’`:`’ defines the domain of the predicate before it. In this case, it means that the domain of the predicate `residue`, more precisely the argument `Amin`, is defined by the predicate `aminoAcid` (4.1.1.7). Hence, this rule means that for each predicate `position` (4.1.1.1) must exist exactly one predicate `residue` (4.1.1.6) in which the argument `Amin` must exist in the `aminoAcid` predicate. This rule is not necessary for the approach in which the residues of the protein are kept fixed and given as input.

It is necessary to define that each position must have one rotamer and only one. This rule is similar to the rule (4.1.1.10).

```
1{rotamer(Pos,Id,Amin) :  possibleRotamer(Pos,Id,Amin,_)}1 :- position(Pos).
```
(4.1.1.11)

This rules means that for each position there must exist one rotamer and that rotamer belongs to the set of possible rotamers (defined by the `possibleRotamer` predicate (4.1.1.2)).

With the rules (4.1.1.10) and (4.1.1.11) it is possible that the rotamer in a certain position is from a different amino acid than the residue of that position. So it is necessary to restrict the rotamers so that there are no contradictions. For this purpose is defined the following rule:

```
:- position(Pos), rotamer(Pos,Id,Amin1), residue(Amin2,Pos),
                                                 Amin1 != Amin2.
```
(4.1.1.12)

This rule means that there cannot exist a rotamer at a position (`Pos`) that belongs to a different amino acid (`Amin1`) than the residue of that position, i.e., the argument `Amin1` of the `rotamer` predicate cannot be different from the `Amin2` argument of the `residue` predicate for a specific position `Pos`.

With these three rules ((4.1.1.10), (4.1.1.11) and (4.1.1.12)) we can have a solution for our problem. We said that each position has one and only one residue, one and only one rotamer and that they belong to the same amino acid. However, our problem consists not only in giving a possible solution but also giving the solution with the minimum energy score. For this we need to define the energy contributions. It is necessary to define two types of energy contributions. The first one is the energy between a rotamer and a backbone. So it is necessary to say that if a rotamer belongs to the solution, then its energy score (interaction between that rotamer and the backbone of the protein) belongs to the solution too. For that it was defined the following rule:

```
energy(Pos,Id,Pos,Id,Energy) :- rotamer(Pos,Id,Amin),
                                possibleRotamer(Pos,Id,Amin,Energy).        (4.1.1.13)
```

This rule means that if there is a possible rotamer (represented by the `possibleRotamer` predicate (4.1.1.2)) and that rotamer is present in the solution (represented by the `rotamer` predicate (4.1.1.8)) then the energy score of the rotamer (the `Energy` argument in the `possibleRotamer` predicate) is present in the solution. The predicate `energy` (4.1.1.9), as said before, represents the energy scores present in the final solution.

The second type of interactions are the interactions between two rotamers. It is necessary to define that if two rotamers belong to the solution and they have an energy interaction then that energy score belongs to the solution. That is made by the rule (4.1.1.14).

```
energy(Pos1,Id1,Pos2,Id2,Energy) :- interEnergy(Pos1,Id1,Pos2,Id2,Energy),
                                    rotamer(Pos1,Id1,_), rotamer(Pos2,Id2,_).        (4.1.1.14)
```

This rule represents exactly what was said above. If there are two rotamers in the solution (the `rotamer` predicate) and they have an interaction energy (represented by the `interEnergy` predicate (4.1.1.3)) then the respective energy score also belongs to the solution. Note that once there are no interactions duplicated by the `interEnergy` predicate, there will be no duplicated interactions in the solution as well.

Finally, to extract the optimal solution, i.e., the solution with the minimal energy score, the optimization rule (4.1.1.15) needs to be defined.

```
#minimize[energy(_,_,_,_,Energy) = Energy].        (4.1.1.15)
```

This rule is to minimize de sum of the energies present on the solution, represented by the `energy` predicate(4.1.1.9) where each predicate has as weight the value of the `Energy` argument. The square brackets in the rule mean that the minimization criterion is the sum of the weights of the `energy` predicate, as referred in section 3.3.

With this set of rules ((4.1.1.10)-(4.1.1.15)) we can solve the problem where we are interested in the predicate `rotamer` which represents that the rotamer belongs to the final solution. The solution is the optimal solution once the minimizing rule guarantees that the solution has minimal total energy.

The complete program with all the rules mentioned in this section can be seen in appendix B.1.1.

Initial tests showed that this codification cannot solve our problem efficiently. Tests with 20 positions to design showed that it takes more than a day to solve, considering that the residues are not given as input (second approach) or it does not solve at all, running out of memory. Therefore we tried a different codification, using two optimization criteria.

4.1.2 Double Optimization

We had to try a different approach to the problem since the previous approach (4.1.1) was not enough.

The idea is to use two optimization criteria. In our problem there are two types of interactions, one between a rotamer and the backbone of the protein and another between two rotamers of different positions. Each of this interactions has an energy score, represented in the predicates described in section 4.1.1. These scores can be either positive or negative, therefore we can use this information to split an optimization criterion into two.

The optimization method in *clasp* does not accept negative values. Moreover, *gringo*, the most common grounder used with *clasp* (see section 3.1), normalizes all the values so that *clasp* can use those values during the search for the optimal solution. For this reason was tried a different approach where only positive scores were considered.

The idea is to split each predicate with energy score information in two, one with the non-negative scores and one with the negative scores. This way all the energy scores can be represented in positive numbers.

The rules described in section 4.1.1 must be adapted for this new model. The predicates representing the input information must also be reformulated so that they can support the identification of the positive and negative energies. In what follows will be described the predicates and rules reformulated.

To represent the positions of the protein to be designed it is used the predicate (4.1.1.1). Given that this predicate has no information about energies it does not change from the previous codification. However, to represent the possible rotamers considered, the predicate (4.1.1.2) has to be split into two, one for the rotamers with a non-negative interaction score with the backbone of the protein and another for the negative ones. Hence the `possibleRotamer` predicate will be replaced by the predicates (4.1.2.1) and (4.1.2.2).

```
possiblePRotamer(Pos,Id,Amin,Energy).                    (4.1.2.1)
```

```
possibleNRotamer(Pos,Id,Amin,Energy).                    (4.1.2.2)
```

The predicate `possiblePRotamer` has the same information as the `possibleRotamer` predicate(4.1.1.2) except for the fact that it only applies to rotamers with a non-negative energy score. The `possibleNRotamer` predicate is to represent the rotamers with a negative energy score. Therefore the `Energy` argument in the `possibleNRotamer` predicate is a positive integer number. The information about the energy being negative is already in the name of the predicate.

Other predicate that has information about energy scores is the `interEnergy` predicate (4.1.1.3) which represents the interaction energy between two rotamers at different positions. Similarly to the possibleRotamer predicate, this predicate will be split in two as well.

```
interPEnergy(Pos1,Id1,Pos2,Id2,Energy).          (4.1.2.3)

interNEnergy(Pos1,Id1,Pos2,Id2,Energy).          (4.1.2.4)
```

The interPEnergy predicate (4.1.2.3) represents a non-negative interaction between two rotamers at two different positions. This predicate, like (4.1.1.3), is used to represent the interaction between two rotamers and the position of the first rotamer (Pos1) is lesser than the position of the second rotamer (Pos2). There are no duplicated interactions. Analogously, the interNEnergy predicate (4.1.2.4) represents a negative interaction between two rotamers. In both predicates ((4.1.2.3) and (4.1.2.4)) the Energy argument is a positive integer and the information about whether is negative or not is in the name of the predicate.

The predicates (4.1.1.6), (4.1.1.7) and (4.1.1.8) described in section 4.1.1 remain unchanged for this codification. These predicates represent the residue in a specific position of the protein, a possible amino acid and a rotamer that belong to the final solution, respectively. These predicates do not have information about energy score and therefore remain unchanged.

The last predicate to be reformulated is the energy predicate (4.1.1.9) which represents an interaction score, either between two rotamers or a rotamer and the backbone, that belongs to the final solution. This predicate will also be split in two different predicates, one for non-negative scores and another for negative scores.

```
pEnergy(Pos1,Id1,Pos2,Id2,Energy).          (4.1.2.5)

nEnergy(Pos1,Id1,Pos2,Id2,Energy).          (4.1.2.6)
```

The predicates (4.1.2.5) and (4.1.2.6) represent, respectively, that a non-negative and a negative energy score belong to the final solution. Like in the possibleNRotamer predicate (4.1.2.2) and in the interNEnergy predicate (4.1.2.4) described above, the Energy argument in nEnergy predicate is a positive integer.

At this point we have all the necessary predicates necessary described and will now start to describe the rules used. Some rules remain unchanged from the simple codification (section 4.1.1) and we will refer them. Only the rules involving the modified predicates will be reformulated.

The rule (4.1.1.10) which says that each position has exactly one residue (or amino acid) remains unchanged as well as the rule (4.1.1.11) which says that each position has exactly one rotamer. The rule of consistency (4.1.1.12) that imposes that the rotamer and the residue at each position refer to the same amino acid also remains unchanged.

The rule that defines that if a rotamer belongs to the solution and has an energy score then that score also belongs to the solution (4.1.1.13) will be split in the following two rules:

```
pEnergy(Pos,Id,Pos,Id,Energy) :- rotamer(Pos,Id,Amin),
                                 possiblePRotamer(Pos,Id,Amin,Energy).
```
(4.1.2.7)

```
nEnergy(Pos,Id,Pos,Id,Energy) :- rotamer(Pos,Id,Amin),
                                 possibleNRotamer(Pos,Id,Amin,Energy).
```
(4.1.2.8)

The rule (4.1.2.7) means that if a rotamer with a non-negative energy score belongs to the solution then the non-negative energy score also belongs to the solution. Analogously, the rule (4.1.2.8) means that if a rotamer with a negative energy score belongs to the solution then the negative score also belongs to the solution.

The rule (4.1.1.14) described in section 4.1.1 will be split into two different rules.

```
pEnergy(Pos1,Id1,Pos2,Id2,Energy) :- interPEnergy(Pos1,Id1,Pos2,Id2,Energy),
                                     rotamer(Pos1,Id1,_), rotamer(Pos2,Id2,_).
```
(4.1.2.9)

```
nEnergy(Pos1,Id1,Pos2,Id2,Energy) :- interNEnergy(Pos1,Id1,Pos2,Id2,Energy),
                                     rotamer(Pos1,Id1,_), rotamer(Pos2,Id2,_).
```
(4.1.2.10)

The rule (4.1.2.9) means that if two rotamers belong to the solution and they have a non-negative interaction energy score then that score also belongs to the solution and the rule (4.1.2.10) means the same but for the negative interaction energies.

At this point we have all the rules described except the optimization rules. The idea of this implementation is to use multi-criteria optimization. For that we want to minimize all the positive energies and at the same time we want to maximize all the negative energies so that we can have the best solution, i.e., the solution with the lower total energy. We also want to minimize the positive energies and minimize the negative energies with the same priority. Only this way we can guarantee the optimal solution. It is not desirable neither to minimize first the positive energies and then within the minimum positive energies maximize the negative nor vice-versa. It is very important that both criteria have the same priority. As mentioned in the end of section 3.1, the maximization will be transformed in a minimization and the optimization values of the two criteria will be summed. This does not affect the optimality of the solution because the optimizations values (in this case, the energies values) have the same unit.

In this codification there are two optimization rules.

```
#minimize[pEnergy(_,_,_,_,Energy) = Energy @1].
```
(4.1.2.11)

```
#maximize[nEnergy(_,_,_,_,Energy) = Energy @1].
```
(4.1.2.12)

The optimization rule (4.1.2.11) is to minimize all the non-negative energy scores that belong to the solution, defined by the `pEnergy` predicate (4.1.2.5). The symbols '`@1`' at the end of the rule means that this optimization criterion has priority 1. Analogously the optimization rule (4.1.2.12) is to maximize all the negative energy scores that belong to the solution, defined by the `nEnergy` predicate (4.1.2.6). The symbols '`@1`' at the end of the rule mean that this optimization criterion has priority 1 so that both criteria have the same priority.

The complete program can be seen at appendix B.2.1.

With some initial tests this codification did not reveal more efficient than the codification described at 4.1.1. However we noticed that the main problem was the memory. Both codifications run out of memory very quickly for 20 positions due to the large number of possible combinations. It is therefore needed to eliminate some of the input that does not contribute to the best solution. This way we reduce the search space of the problem.

4.1.3 Alternative Codifications

There are some techniques to try to improve the efficiency of an ASP program. These techniques can be, for example, imposing an order in the attribution of variables or eliminating symmetries. Some of these techniques are described by Mancini et al. in [34].

We already eliminated symmetries by do not having duplicated information about interactions between two rotamers at different positions. However we can apply some techniques to try to improve the efficiency of our codifications.

We tried to impose some order in the attribution of the rotamers at each position. The idea is to design the protein from the lowest position identifier to the highest. Therefore, a rotamer for the first position must be defined before assigning a rotamer to the second position. This can be made by adding two simple rules to the program and eliminating the generation rule of rotamers (4.1.1.8).

```
1{rotamer(1,Id,Amin) : possibleRotamer(1,Id,Amin,_)}1 :- position(1).      (4.1.3.1)
```

```
1{rotamer(Pos,Id,Amin) : possibleRotamer(Pos,Id,Amin,_)}1 :- rotamer(Pos - 1,_,_).
                                                                             (4.1.3.2)
```

The first rule (4.1.3.1) means that the first position (position 1 of the protein) must have exactly one rotamer. That rotamer must be a possible rotamer for that position. The second rule (4.1.3.2) means that if there is one rotamer at a specific position (`Pos - 1`) then exactly one rotamer that is a possible rotamer for the next position (`Pos`) must exist.

The solver, with these rules, will assign first the rotamer of the first position, then the rotamer of the second position and so on until all the protein positions have a rotamer assigned. The rule (4.1.1.8) described in section 4.1.1 which says that each position must have exactly one rotamer is no longer useful.

As the positions of the protein to be designed may not be consecutive, the above rules have to be modified to consider that fact. A new predicate can be used to create an index of position so that it can be possible to identify the positions by consecutive numbers starting with '`1`'.

```
index(Index,Pos).                                                  (4.1.3.3)
```

The `index` predicate (4.1.3.3) represents an index for the positions of a protein where the `Index` argument refers to the index value and the `Pos` predicate refers to the position of the protein.

The rules (4.1.3.1) and (4.1.3.2) must be adapted to consider the design of non-consecutive positions using the `index` predicate.

```
1{rotamer(Pos,Id,Amin) : possibleRotamer(Pos,Id,Amin,_)}1 :- index(1,Pos).   (4.1.3.4)
```

```
1{rotamer(Pos2,Id,Amin) : possibleRotamer(Pos2,Id,Amin,_)}1 :- rotamer(Pos,_,_),
                              index(Index - 1,Pos), index(Index,Pos2).
                                                                     (4.1.3.5)
```

The rules (4.1.3.4) and (4.1.3.5) represent the same as the rules (4.1.3.1) and (4.1.3.2), respectively, but considering the `index` predicate (4.1.3.3).

Another technique that can be used is to design the protein in a hierarchically way. This means that first a residue must be assigned to a position before assigning a rotamer. Therefore the protein is designed hierarchically where it is designed from the most generic to the most specific, i.e., first it is assigned an amino acid to the positions, and then it is assigned the specific rotamer for each position within the amino acid chosen. Notice that this is only applicable to the second approach of the problem, where not only the rotamers but also the amino acids are designed as well. In the first approach the residues in each position are given as input and this technique does not apply.

To assign first the residue and then the rotamer the rule (4.1.1.8), described previously, was changed for the rule (4.1.3.6).

```
1{rotamer(Pos,Id,Amin) : possibleRotamer(Pos,Id,Amin,_)}1 :- position(Pos),
                                                                     (4.1.3.6)
                                               residue(Amin,Pos).
```

With this rule the rotamers at each position can only be assigned if the residue is already assigned. The consistency rule (4.1.1.12), which prevents a rotamer from belonging to a different amino acid that the respective residue, is no longer needed.

Unfortunately these techniques did not bring any improvement. With some tests we noticed that using these rules is more or less the same that do not using them. Our main problem lies on the large amount of input information and the great search space of the problem.

4.2 Dead End Elimination

As it was said before, the main problem at this point is the large amount of data and therefore the enormous search space of the protein design problem. It is necessary to eliminate some of the starting input to reduce the search space without losing the optimal solution. For this purpose were implemented *dead-end elimination* (DEE) algorithms.

The idea of *dead-end elimination* is to eliminate some rotamers that cannot be in the optimal solution [37, 18, 16]. There are several algorithms of dead-end elimination with different power of elimination. Three of those algorithms were implemented.

In this section will be described the algorithms implemented, the main idea of those algorithms and how they were implemented. First it will be described the answer set programming implementation and then the Java implementation.

4.2.1 Dead-End Elimination with ASP

The first approach made to the dead-end elimination algorithms used answer set programming.

The original DEE algorithm described in [37] is one of the most simple algorithms and it was implemented using answer set programming. The idea of this algorithm is to compare two rotamers at the same positions and eliminate one of them if possible. To do this, it is computed the worst case scenario for one rotamer and the best case scenario for the other rotamer and if the worst case of one rotamer is better than the best case of the other rotamer, the second rotamer can be eliminated. The worst case scenario of a rotamer is the sum of the highest energy interaction for each position different from the position of the rotamer. Analogously the best case scenario is the sum of the lowest energy interaction for each position. In other words, the worst case scenario for a specific rotamer is the sum of the energy scores interactions between the rotamer and each other rotamer at a different position. This scenario assumes that in each other position is the rotamer with the highest interaction energy with the first rotamer.

This dead-end elimination criterion can be described as follows. The rotamer i_r at position i can be eliminated if there is a rotamer i_t at the same position that satisfies:

$$E(i_r) + \sum_{j,j\neq i} \min_u E(i_r, j_u) > E(i_t) + \sum_{j,j\neq i} \max_u E(i_t, j_u) \tag{4.2.1.1}$$

where $E(i_r)$ is the energy score of the interaction between the rotamer i_r and the backbone of the protein and $E(i_r, j_u)$ is the energy interaction between rotamer i_r and rotamer j_u.

Now it will be described each rule used to implement the dead-end elimination criterion and each predicate used. The input information considered is the same as described in 4.1.1.

First, it is defined the best case scenario. The idea is to use a predicate that indicates which is the best case for a specific rotamer and a position different from the rotamers position. This is made with two rules, (4.2.1.2) and (4.2.1.3).

```
bestCase(Pos,Id,E,Pos2) :- E = #min[interEnergy(Pos,Id,Pos2,_,V)=V],
                Pos2>Pos, possibleRotamer(Pos,Id,_,_), position(Pos2),      (4.2.1.2)
                                         interEnergy(Pos,Id,Pos2,_,_).
```

```
bestCase(Pos,Id,E,Pos2) :- E = #min[interEnergy(Pos2,_,Pos,Id,V)=V],
                Pos2<Pos, possibleRotamer(Pos,Id,_,_), position(Pos2),      (4.2.1.3)
                                         interEnergy(Pos2,_,Pos,Id,_).
```

The `bestCase(Pos,Id,E,Pos2)` predicate in the above rules represents that the rotamer `Id` at position `Pos` has an energy score interaction `E` with a rotamer at position `Pos2` in the best case. The two rules above are very similar to each other. The first one (4.2.1.2) considers the positions (`Pos2`) greater than the position (`Pos`) of the rotamer. The second rule (4.2.1.3) considers the positions (`Pos2`) lesser than the position (`Pos`) of the rotamer. This is needed because there are no duplicated information about interactions between two rotamers, as mentioned in section 4.1.1. The `#min` function in the rules computes the minimum interaction energy (`V`) of the rotamer `Id` and a rotamer in position `Pos2`.

For the worst case scenario are used very similar rules except that it is desired the maximum interaction energies. The rules used are (4.2.1.4) and (4.2.1.5).

```
worstCase(Pos,Id,E,Pos2) :- E = #max[interEnergy(Pos,Id,Pos2,_,V)=V],
                Pos2>Pos, possibleRotamer(Pos,Id,_,_), position(Pos2),      (4.2.1.4)
                                         interEnergy(Pos,Id,Pos2,_,_).
```

```
worstCase(Pos,Id,E,Pos2) :- E = #max[interEnergy(Pos2,_,Pos,Id,V)=V],
                Pos2<Pos, possibleRotamer(Pos,Id,_,_), position(Pos2),      (4.2.1.5)
                                         interEnergy(Pos2,_,Pos,Id,_).
```

Now it is necessary to sum all the best and worst case scenario energy scores. For that are used the rules (4.2.1.6) and (4.2.1.7) to sum the best cases and the worst cases, respectively.

```
bestSum(Pos,Id,Sum+E) :- Sum = #sum[bestCase(Pos,Id,V,_)=V],
                                                                            (4.2.1.6)
                                    possibleRotamer(Pos,Id,_,E).
```

```
worstSum(Pos,Id,Sum+E) :- Sum = #sum[worstCase(Pos,Id,V,_)=V],
                                                                            (4.2.1.7)
                                    possibleRotamer(Pos,Id,_,E).
```

The two rules above sum all of the best cases and the worst cases, respectively, and the energy interaction score of the considered rotamer with the backbone of the protein. The `bestSum(Pos,Id,Sum+E)` predicate means that a rotamer `Id` at a position `Pos` has a total best case of `Sum+E` where `Sum` is the sum of all best cases for that rotamer and `E` is the energy score of the interaction between the rotamer and the backbone of the protein. The `worstSum` predicate is analogous.

Now that all the best and worst cases for each rotamer are defined, it is necessary to define the impossible rotamers. A rotamer is impossible if there is another rotamer at the same position with a worst case worse than the best case of the first rotamer.

```
        impossibleRotamer(Pos,Id) :- possibleRotamer(Pos,Id,_,_),
     possibleRotamer(Pos,Id2,_,_), Id!=Id2, bestSum(Pos,Id,Sum),        (4.2.1.8)
                                    worstSum(Pos,Id2,Sum2), Sum>Sum2.
```

The rule (4.2.1.8) defines that a rotamer is impossible (represented by the `impossibleRotamer` predicate) if it is a rotamer with a total energy of `Sum` in the best case and there is another rotamer at the same position `Pos` with a total energy of `Sum2` in the worst case and `Sum` is greater than `Sum2`.

The complete program can be seen in appendix C.

Although this codification works, it has the same problem as the other ASP codification for the protein design problem considered. The large amount of input data makes this program inefficient when compared to implementations made in different programming languages.

4.2.2 Dead-End Elimination with Java

To improve the efficiency of the dead-end elimination algorithms, we implemented three algorithms in Java. In this section will be described each algorithm implemented and a pseudo-code for each one will be presented.

Original DEE

The algorithm described in section 4.2.1 called original dead-end elimination was the first algorithm that we implemented in Java.

As it was said before, the idea of this algorithm is to compare two rotamers of the same position and eliminate, if possible, one of them. For each rotamer is computed the worst case scenario and the best case scenario (also explained in section 4.2.1). If the best case scenario of one rotamer has a total energy score greater than the total energy of the worst case of the other rotamer, then the first rotamer can be eliminated because the second one will always be better.

The pseudo-code for the original dead-end elimination algorithm is shown in algorithm 1.

Simple Goldstein DEE

A dead-end elimination algorithm that is more powerful than the previous one is called Simple Goldstein DEE [37, 18]. This algorithm generally eliminates more rotamers than the original dead-end elimination algorithm.

Algorithm 1: Original DEE

```
foreach position $i$ do
    foreach rotamer $r$ at $i$ do
        foreach rotamer $t$ at $i$, $t \neq r$ do
            $X = E(i_r)$                                    // store energy of rotamer $r$
            $Y = E(i_t)$                                    // store energy of rotamer $t$
            forall the other positions $j$, $j \neq i$ do
                forall the rotamers $u$ at position $j$ do
                    $Z = \min_u [E(i_r j_u)]$               // best case for rotamer $r$
                    $W = \max_u [E(i_t j_u)]$               // worst case for rotamer $t$
                end
                $X = X + Z$                                 // sum the best cases for rotamer $r$
                $Y = Y + W$                                 // sum the worst cases for rotamer $t$
            end
            if $X > Y$ then
                eliminate $i_r$ and break
            end
        end
    end
end
```

The idea of Simple Goldstein DEE is to compare two rotamers at the same position and eliminate one of them if the contribution to the total energy is always reduced by using the other rotamer. In other words, if the minimum difference of replacing one rotamer for an alternative rotamer is positive, then the first rotamer can be eliminated. The minimum difference being positive means that if the second rotamer replaces the first one than the total energy of the final protein is at least reduced by the value of the minimum difference. Therefore a rotamer is eliminated if there is another rotamer that can replace the first one and reduce the total energy of the protein.

Formally, this criterion states that a rotamer i_r can be eliminated if exists a rotamer i_t at the same position that satisfies:

$$E(i_r) - E(i_t) + \sum_{j, j \neq i} \{\min_u [E(i_r, j_u) - E(i_t, j_u)]\} > 0 \tag{4.2.2.1}$$

The pseudo-code in algorithm 2 [37] represents the simple Goldstein dead-end elimination algorithm that was implemented.

Simple Split DEE

Other dead-end elimination, more powerful than the original and the simple Goldstein algorithms, is the simple split dead-end elimination algorithm [37, 18].

The idea of this algorithm may not as easy to understand as the algorithms described before. The idea is that it is possible that a single rotamer may not be enough to eliminate other rotamer using the simple Goldstein algorithm. The simple split DEE uses partitions so that more than one rotamer can eliminate another rotamer. This is an alteration of the simple Goldstein algorithm and therefore the idea

Algorithm 2: Simple Goldstein

```
1  foreach position i do
2      foreach rotamer r at i do
3          foreach candidate rotamer t at i do
4              $X = E(i_r) - E(i_t)$                              // store energy difference
5              forall the other positions $j$, $j \neq i$ do
6                  forall the rotamers u at position j do
7                      $Y = \min_u [E(i_r j_u) - E(i_t j_u)]$     // minimum energy difference
8                  end
9                  $X = X + Y$                                    // sum of minimum differences
10             end
11             if $X > 0$ then
12                 eliminate $i_r$ and break
13             end
14         end
15     end
16 end
```

of minimum difference of replacing rotamers described in the simple Goldstein algorithm is present too.

The simple split algorithm consists in considering a rotamer at some position and consider a splitting position, different from the position of the rotamer considered. Then, for each rotamer of the splitting position, if there is another rotamer of the same position as the original rotamer with a positive minimum difference contribution, assuming that the rotamer of the splitting position belongs to the protein, the rotamer of the splitting position is marked. If all the rotamers of the splitting position are marked then the original rotamer can be eliminated. In other words if, for whatever rotamer in a specific position (splitting position), there is a rotamer in other position that can be replace by another rotamer (the total minimum difference of the replacement is positive) then the rotamer can be eliminated. Notice that it may not be the same rotamer that can replace the other one for all the rotamers at the splitting position.

This algorithm can be split in two parts: a first part where all the minimum difference contributions are stored for each pair of rotamers at the same position; and a second part where it is considered a splitting position and for each rotamer at the splitting position it is checked if exists some rotamer that can replace other rotamer.

The algorithm 3 [37] is the pseudo-code of the simple split dead-end elimination algorithm implemented.

The original DEE algorithm, the simple Goldstein DEE algorithm and the simple split DEE algorithm were the three algorithms implemented in Java. These algorithms were applied to the possible rotamers considered at each position in order to eliminate rotamers that cannot belong to the optimal solution so that the search space of the protein design problem is reduced.

Algorithm 3: Simple Split

```
foreach position $i$ do
    foreach rotamer $r$ at $i$ do
        foreach candidate rotamer $t$ at $i$ do
            forall the other positions $j$, $j \neq i$ do
                forall the rotamers $u$ at position $j$ do
                    store $Y_{jt} = \min_u \left[E\left(i_r j_u\right) - E\left(i_t j_u\right)\right]$    // store minimum differences
                end
            end
        end
        foreach splitting position $k$                                       // split position $k$
        do
            set $elim_v = false \; \forall \; v$ at $k$                                  // initialization
            foreach competitor $t$ at $i$ do
                $X = E\left(i_r\right) - E\left(i_t\right)$                                  // store difference
                forall the other positions $j, j \neq i \neq k$ do
                    $X = X + Y_{jt}$                                         // sum minimum differences
                end
                foreach partition $v$ at $k$                       // rotamer $v$ at split position $k$
                do
                    if $\left(X + \left[E\left(i_r k_v\right) - E\left(i_t k v\right)\right]\right) > 0$ then
                        $elim_v = true$
                    end
                end
            end
            if $elim_v = true \; \forall \; v$ at $k$ then
                eliminate $i_r$ and break
            end
        end
    end
end
```

5 Results

To test the dead-end elimination algorithms and the codifications of the problem implemented we used 10 proteins, described with detail in next section (section 5.1). We tested the dead-end algorithms implemented using ASP and Java with the 10 proteins. We also tested the two codifications described in section 4.1 using the 10 proteins. First we tested without using any dead-end elimination algorithm. Afterwards, we tested the codifications using the 10 proteins in which we applied the simple split DEE algorithm (see section 16).

The tests were made using *gringo* version 3.0.4 as the ASP grounder and *clasp* version 2.1.0 as the ASP solver. Clasp was used with the default configuration. The tests were made on Intel Xeon 5160 machines (dual-cores with 3.00 GHz of clock speed, 4 MB of cache, 1333 MHz of FSB speed, and 4 GB of RAM each), running 64-bit versions of Linux 2.6.33.3-85.fc13. It was given a CPU time limit of 10800 seconds (3 hours) and a memory limit of 3000 Megabytes.

In this chapter will be described the results of the dead-end elimination algorithms and the results of the codifications of the problem implemented. It will start with a description of the instances used for the tests. Then the dead-end elimination results will be presented. Moreover, we will describe the results of the simple codification and the double optimization implementations.

5.1 Instances

As initials tests showed that ASP has difficulties with large amount of input data, we tried to constraint the instances used. For this purpose we only used proteins of the PDZ domain. PDZ domains are modules of interaction between proteins [4, 26, 43, 20].

The tests run used 10 proteins. The PDB id[1] of the proteins are: 1MFG, 1BE9, 2GZV, 2EGN, 2FNE, 1RZX, 1N7F, 1QAU, 1TP3 and 1I92.

To restrict the input information data in order to understand the capacities of the codifications made, only a subset of positions were designed for each protein. We choose to design, for each protein, 17, 15, 12 and 10 positions. The positions chosen are positions with a low solvent accessibility, i.e., positions near the core of the protein where most of the amino acids are hydrophobic, as mentioned in section 2.2.1. This way we can restrict the set of amino acids and consider only the hydrophobic amino acids. We consider that the hydrophobic amino acids are: A, C, F, I, L, M, V, W and Y (see table 2.1)[36].

[1] `http://www.pdb.org`

Protein	Positions
1MFG	7, 9, 15, 17, 19, 37, 39, 42, 48, 52, 58, 66, 74, 78, 85, 87, 89
1BE9	18, 20, 27, 29, 31, 40, 42, 45, 51, 57, 63, 71, 79, 83, 90, 92, 94
2GZV	28, 30, 38, 40, 42, 52, 54, 57, 63, 69, 75, 83, 91, 95, 102, 104, 106
2EGN	8, 10, 18, 20, 22, 40, 41, 44, 50, 55, 61, 69, 77, 81, 88, 90, 92
2FNE	29, 31, 38, 40, 42, 55, 57, 60, 66, 72, 78, 86, 94, 98, 105, 107, 109
1RZX	6, 8, 17, 19, 21, 40, 42, 45, 51, 57, 63, 71, 79, 83, 90, 92, 94
1N7F	9, 11, 18, 20, 22, 32, 34, 37, 43, 49, 55, 63, 71, 75, 82, 84, 86
1QAU	5, 7, 15, 17, 19, 28, 30, 33, 39, 45, 51, 59, 67, 71, 80, 82, 84
1TP3	18, 20, 27, 29, 31, 40, 42, 45, 51, 57, 63, 71, 79, 83, 90, 92, 94
1I92	7, 9, 16, 18, 20, 30, 31, 34, 40, 45, 51, 59, 67, 71, 78, 80, 82

Table 5.1: List of positions designed

The 17 positions designed for each position are shown in table 5.1. The 15 positions considered for each protein are the first 15 positions of the table 5.1, the 12 positions are the first 12 and the 10 positions are the first 10.

For the 1MFG protein we choose the same positions as in [36]. For the other 9 proteins we used two solvent accessibility prediction sites: SABLE[2] and SAS[3]. We crossed the information of the two sites and, comparing with 1MFG protein, we choose the positions with the lowest solvent accessibility prediction.

The 10 considered proteins were used considering 17, 15, 12 and 10 positions, as mentioned before. Moreover, for the second approach, described in section 4.1, in which the amino acids of the protein are not kept fixed, we considered two sets of amino acids. One set of amino acids considered contains all the 20 essential amino acids described in table 2.1. The other set of amino acids considered contains only the 9 hydrophobic amino acids mentioned before (A, C, F, I, L, M, V, W and Y). Therefore, to test the codifications and the DEE algorithms implemented, we considered the four set of positions. Moreover, for each set of positions we considered three set of amino acids: assuming that the amino acids are fixed (first approach described in section 4.1); the set of hydrophobic amino acids; and all the 20 essential amino acids (second approach).

For each protein, the PDB files were extracted from the Protein Data Bank. We used an adapted source code of *Rosetta* to compute the discrete set of possible rotamers given the PDB file of each protein and respective positions to be designed. Rosetta was also used to compute all the necessary energy interaction scores.

All the information data was converted to the predicates mentioned in chapter 4. The energy scores computed by Rosetta are not integer numbers. Therefore, as clasp only accepts integer numbers, we converted the scores to integers, considering 3 decimal places by multiplying each score by 1000. Hence, we have more precision computing the optimal solution. Previous versions of clasp (before version 2.1.0) have a sum limit of 999999999. Hence, we only considered 3 decimal places in the energy scores to avoid reaching the sum limit. Although in version 2.1.0 of clasp the sum limit increased to 2147483647, we continued considering only 3 decimal places for the same reason.

[2]`http://sable.cchmc.org/`
[3]`http://140.113.239.214/~weilun/index.php`

5.2 Dead-End Elimination Results

The Original DEE algorithm implemented using ASP and the Original DEE, Simple Goldstein and Simple Split algorithms implemented using Java were tested with the 10 proteins mention before. We considered the four sets of positions and the three sets of amino acids (the 20 essential, the hydrophobic and considering the amino acids of the proteins fixed). The Original and the Simple Goldstein algorithms were run one single time for each instance. The Simple Split algorithm was run iteratively until no further rotamers could be eliminated.

Table 5.2 shows the results for the Original DEE algorithm and for Simple Split algorithm considering 15 and 10 positions and considering the backbone fixed and hydrophobic amino acids. It contains the initial number of rotamers, the number of rotamers eliminated by each algorithm and the time in seconds for each approach. The full and more detailed results can be seen in appendix D.

In table 5.2 it is possible to see that for the Original DEE algorithm the Java approach is better than the ASP approach. It is also possible to see that in some cases the Simple Split algorithm is faster than the Original DEE algorithm implemented using ASP.

The Original DEE algorithm eliminates on average 11,93% of the rotamers with a standard deviation of 11,43%. The Simple Split algorithm eliminates 62,52% of the rotamers on average with a standard deviation of 17,77%.

5.3 Problem Codifications Results

One problem of clasp that can be verified looking at the results is the large amount of input data. In our case, a large amount of considered rotamers, and therefore a large amount of interactions, makes the search space too big. As this problem is an optimization problem, having a large search space leads to not returning the optimal solution due to the memory size or the CPU time limit imposed.

In the problem codifications tests we used the rotamers and interactions of each protein with and without using a DEE algorithm first. In other words, we tested the codifications using the possible rotamers given by Rosetta and also tested the codifications using the result of applying the Simple Split algorithm to the original rotamers. We choose to use the Simple Split algorithm because it is the most powerful, i.e., it is the algorithm that eliminates the largest number of rotamers.

For the instances where all the 20 essential amino acids were considered, was not possible to compute the optimal solution due to either the time limit of 3 hours or the memory size limit. For the instances designing 17 positions only for the approach in which the backbone of the protein is kept fixed, i.e., the amino acids in each position are given as input, were computed the optimal solution.

Table 5.3 shows the results for 15 and 10 positions. Only the instances considering the backbone fixed and the hydrophobic amino acids are represented. The table contains the time of the two codifications (simple codification - S.C. and double optimization - D.O.) without using DEE and using the Simple Split DEE algorithm in seconds. It also contains the number of initial rotamers considered (#R.).

Protein	Pos.	Amino Acids	# Initial Rotamers	Original DEE # Elim. Rotamers	Original DEE ASP time(s)	Original DEE Java time(s)	Simple Split DEE # Elim. Rotamers	Simple Split DEE Java time(s)
1MFG	15	Hydro.	410	12	7,878	1,264	321	7,633
		Fixed	34	10	0,013	0,138	19	0,151
	10	Hydro.	250	9	2,559	0,897	228	2,391
		Fixed	24	7	0,008	0,106	14	0,125
1BE9	15	Hydro.	435	24	9,517	1,206	373	7,344
		Fixed	35	8	0,015	0,130	20	0,151
	10	Hydro.	258	14	2,864	0,793	244	2,374
		Fixed	21	7	0,007	0,084	11	0,086
2GZV	15	Hydro.	456	21	10,417	1,236	355	10,601
		Fixed	41	11	0,020	0,157	26	0,210
	10	Hydro.	275	13	3,290	0,863	250	3,064
		Fixed	28	8	0,009	0,119	18	0,138
2EGN	15	Hydro.	413	42	7,692	1,151	321	7,148
		Fixed	35	13	0,013	0,128	20	0,149
	10	Hydro.	238	41	2,152	0,792	205	2,385
		Fixed	22	8	0,006	0,079	12	0,102
2FNE	15	Hydro.	438	38	9,678	1,170	321	8,945
		Fixed	39	8	0,018	0,150	22	0,179
	10	Hydro.	253	14	2,600	0,840	228	2,407
		Fixed	24	5	0,007	0,101	12	0,118
1RZX	15	Hydro.	404	17	7,117	1,123	337	5,640
		Fixed	45	12	0,025	0,234	30	0,253
	10	Hydro.	253	15	2,719	0,835	223	2,366
		Fixed	23	9	0,006	0,089	13	0,103
1N7F	15	Hydro.	402	27	7,329	1,192	339	6,303
		Fixed	38	9	0,018	0,139	23	0,172
	10	Hydro.	246	27	2,376	0,792	219	2,307
		Fixed	25	9	0,005	0,100	15	0,108
1QAU	15	Hydro.	421	26	8,360	1,211	311	8,653
		Fixed	37	8	0,018	0,144	19	0,184
	10	Hydro.	251	26	2,582	0,773	233	2,710
		Fixed	23	5	0,005	0,082	10	0,109
1TP3	15	Hydro.	447	17	10,183	1,269	346	9,451
		Fixed	40	12	0,021	0,173	25	0,193
	10	Hydro.	279	23	3,469	0,901	259	3,006
		Fixed	27	10	0,009	0,116	17	0,133
1I92	15	Hydro.	455	31	10,831	1,192	318	10,095
		Fixed	53	16	0,037	0,234	38	0,289
	10	Hydro.	283	33	3,717	0,883	225	3,526
		Fixed	36	11	0,016	0,130	26	0,151

Table 5.2: Dead-End Elimination Results for Original and Simple Split DEE

Protein	Pos.	Amino Acids	Without DEE			With DEE				
			#R.	S.C. time	D.O. time	#R.	S.C. time	D.O. time	S.C.+DEE time	D.O.+DEE time
1MFG	15	Hidro.	410	—	—	89	679,26	620,26	686,89	627,89
		Fixed	34	0,01	0,01	15	0,00	0,00	0,15	0,16
	10	Hidro.	250	—	—	22	0,00	0,00	2,39	2,39
		Fixed	24	0,01	0,00	10	0,00	0,00	0,13	0,13
1BE9	15	Hidro.	435	—	—	62	1,72	1,78	9,06	9,12
		Fixed	35	0,01	0,01	15	0,00	0,00	0,15	0,15
	10	Hidro.	258	—	—	14	0,00	0,00	2,38	2,38
		Fixed	21	0,00	0,00	10	0,00	0,00	0,09	0,09
2GZV	15	Hidro.	456	—	—	101	—	—	—	—
		Fixed	41	0,02	0,02	15	0,00	0,00	0,21	0,21
	10	Hidro.	275	—	—	25	0,01	0,00	3,08	3,07
		Fixed	28	0,01	0,01	10	0,00	0,00	0,14	0,14
2EGN	15	Hidro.	413	—	—	92	280,75	279,18	287,90	286,33
		Fixed	35	0,01	0,01	15	0,00	0,00	0,15	0,15
	10	Hidro.	238	—	—	33	0,02	0,02	2,40	2,40
		Fixed	22	0,00	0,00	10	0,00	0,00	0,10	0,10
2FNE	15	Hidro.	438	—	—	117	4976,56	4816,04	4985,50	4824,99
		Fixed	39	0,02	0,02	17	0,00	0,00	0,18	0,18
	10	Hidro.	253	—	—	25	0,00	0,00	2,41	2,41
		Fixed	24	0,01	0,00	12	0,00	0,00	0,12	0,12
1RZX	15	Hidro.	404	—	—	67	36,04	42,31	41,68	47,95
		Fixed	45	0,02	0,01	15	0,00	0,00	0,26	0,25
	10	Hidro.	253	—	—	30	0,00	0,01	2,37	2,37
		Fixed	23	0,00	0,00	10	0,00	0,00	0,10	0,11
1N7F	15	Hidro.	402	—	—	63	66,04	63,81	72,34	70,11
		Fixed	38	0,03	0,03	15	0,00	0,00	0,17	0,18
	10	Hidro.	246	—	—	27	0,00	0,01	2,31	2,31
		Fixed	25	0,00	0,00	10	0,00	0,00	0,11	0,11
1QAU	15	Hidro.	421	—	—	110	2295,68	1943,80	2304,33	1952,45
		Fixed	37	0,03	0,05	18	0,00	0,00	0,19	0,19
	10	Hidro.	251	—	—	18	0,00	0,00	2,71	2,71
		Fixed	23	0,00	0,00	13	0,00	0,00	0,11	0,11
1TP3	15	Hidro.	447	—	—	101	463,06	394,99	472,51	404,44
		Fixed	40	0,01	0,01	15	0,00	0,00	0,20	0,20
	10	Hidro.	279	—	—	20	0,00	0,00	3,01	3,01
		Fixed	27	0,00	0,00	10	0,00	0,00	0,14	0,13
1I92	15	Hidro.	455	—	—	137	—	—	—	—
		Fixed	53	0,16	0,21	15	0,00	0,00	0,29	0,29
	10	Hidro.	283	—	—	58	0,46	0,47	3,99	3,99
		Fixed	36	0,00	0,01	10	0,00	0,00	0,15	0,15

Table 5.3: Problem Codifications Results with and without DEE

It is possible to see in table 5.3 that both codifications, the simple and the double optimization, do not differ much. It is also possible to see that using the DEE algorithm first, as it decreases the number of rotamers, allows to solve a larger number of instances. For the instances that are solved without DEE we can see that using DEE does not increase significantly the total time.

More detailed and complete results can be found in appendix D.

There are 120 instances (12 for each of the 10 proteins). Of these 120 instances, 40 consider the 20 essential amino acids, other 40 consider only hydrophobic amino acids and the remaining 40 consider the amino acids given as input. Table 5.4 shows the percentage of solved instances with and without using the Simple Split DEE algorithm.

	Amino Acids		
	All	Hydrophobic	Fixed
Without DEE	0% (0/40)	0% (0/40)	100% (40/40)
With DEE	0% (0/40)	70% (28/40)	100% (40/40)

Table 5.4: Percentage of Solved Instances

In appendix D it is possible to verify that the majority of the unsolved instances ran out of memory. Therefore, increasing the time limit imposed would not allow us to solve all the instances.

5.4 Discussion

In the codifications made, we give as output the set of rotamers that minimizes the total energy of the protein, i.e., the dihedral angles of the side-chain of the amino acids. Although most of the existent tools do not guarantee the optimal solution, it is desired to compare solutions in order to evaluate the quality of our solution. For that, it is needed a tool that converts the dihedral angles in three-dimensional coordinates, generating a PDB file with the design protein. This way it is possible to compute the total energy of the protein designed, by our approach and by the several existent tools, using the same tool to calculate the energy, allowing the comparison of the solutions. This is suggested as future work.

Although we cannot directly compare the solutions, we can verify the percentage of amino acids of the designed protein that matches the original PDB file of the protein. As we do not have the optimal solution for all the instances that consider the 20 essential amino acids, we only made the comparison for the instances that consider only the hydrophobic amino acids. Notice that for the instances that consider the backbone fixed, i.e., the amino acids of each designed position is given as input, the match is always 100%. Table 5.5 shows the number of matches for the instances in which only the hydrophobic amino acids are considered. Remember that for 17 positions the optimal solution was only computed for the instances which considered the amino acids fixed. In table 5.5 is possible to see that there are more than 50% matches in each instance.

Protein	# Positions		
	15	12	10
1MFG	12	9	8
1BE9	11	8	6
2GZV	—	10	9
2EGN	11	7	6
2FNE	13	10	9
1RZX	11	8	7
1N7F	10	10	8
1QAU	14	11	9
1TP3	14	11	9
1I92	—	7	7

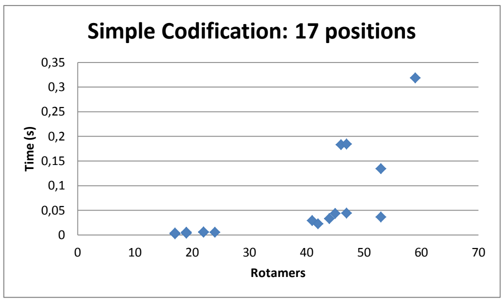
Table 5.5: Number of matched amino acids

We already mentioned that the large amount of data influences the time of computing the optimal solution. In fact the number of rotamers considered as input greatly influences the total time of grounding and solving.

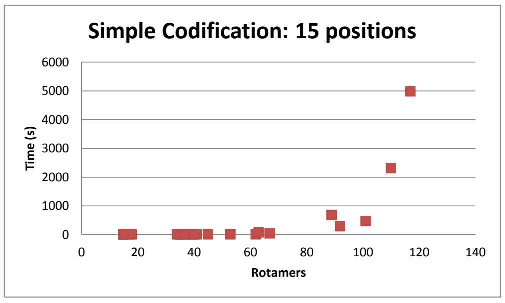
Figure 5.1 shows the relation between the number of rotamers and the time of computing the optimal solution. Figure 5.1(e) has a logarithmic time scale in order to better illustrate the results. It is possible to verify that the time increases exponentially with the number of rotamers given as input. Also, figure 5.1 shows that above around 110 rotamers is not possible to compute the optimal solution under 3 hours (the limit imposed).

In the beginning of this section we mentioned that we do not have a tool to transform the dihedral angles in three-dimensional coordinates in order to compare our solution with other existent tools. One of the main differences between our approach and the majority of the existent tools is that we guarantee the optimal solution for the given input information. Therefore, comparing our solution with the other tools may lead to wrong conclusions. It is necessary to consider that the other tools may not return the optimal solution. However, it would be interesting to verify how much better our solution is and try to understand if the quality of the solution justifies a possible difference of solving time.

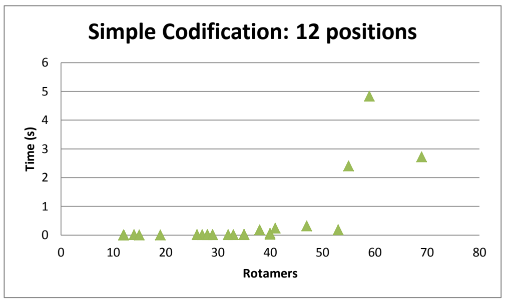
Despite of the fact that we cannot directly compare solutions, we can compare some results with ProtSAT[36], once that ProtSAT returns an optimal solution. We did not have access to ProtSAT, therefore we can only look to the results described in [36]. It is possible to see that ProtSAT can design 17 positions unlike our approach which could not design 17 positions except when considering that the amino acids of each position are given as input.



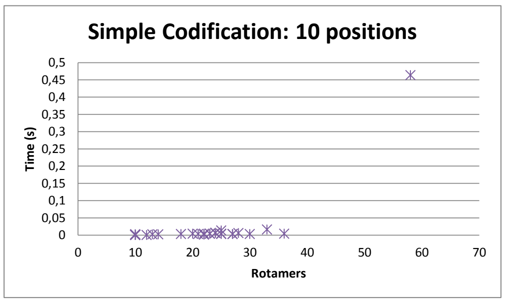
(a) 17 Positions



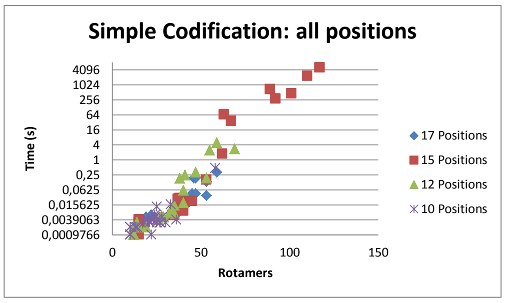
(b) 15 Positions



(c) 12 Positions



(d) 10 Positions



(e) All Sets of Positions

Figure 5.1: Graphs of Simple Codification Results

6 Conclusion

The main goal of this work is to verify the capacity of answer set programming when applied to the protein design problem. Answer set programming is commonly used in difficult search problems and can be used in optimization problems as well[35, 30, 10]. The protein design problem is a search problem with an optimization criterion: minimize the total energy of a protein. Therefore, our goal is to verify if ASP could be applied to solve efficiently the protein design problem.

Our work has two main differences from the existent tools. First, our approach guarantees the optimal solution for the given input information. Most of the existent tools do not guarantee the optimality of the solution. Second, our approach uses ASP, a language never used before to solve the protein design problem.

Two codifications were made for the protein design problem, described in chapter 4. The two approaches, as mentioned in chapter 5, do not differ much regarding the number of solutions and the solving time. We also implemented dead-end elimination methods[18, 16] using ASP and Java. A dead-end elimination method is a search problem. The goal of DEE is to find rotamers that cannot belong to the optimal solution of the protein design problem. Using two different languages allowed us to compare the efficiency of ASP in this search problem.

With this work we can conclude that ASP is not very good when there is a large amount of input information. Results showed that for more than around 110 rotamers no solution is given to the protein design problem (see figure 5.1). In figure 5.1(e) it is possible to verify that the solving time of ASP increases exponentially with the increasing of the number of rotamers.

Moreover, it is possible to see that the implementation of the Original DEE algorithm in ASP for a larger number of rotamers is worse than the implementation in Java of the same algorithm. In some cases, the Simple Split algorithm, which is more powerful than the Original DEE algorithm, if implemented in Java is faster than the Original DEE algorithm implemented in ASP.

However, it is also possible to conclude that, for a small number of rotamers, ASP is very fast. In the codifications of the protein design problem, results showed that for less 50 rotamers, it takes less than one second to compute the optimal solution. Regarding the Original DEE algorithm implemented, it is possible to see that for a small number of rotamers ASP is faster than Java.

ASP can be very good with search and optimization problems with a small amount of input information. When applied to the protein design problem, as it has a large amount of information regarding interactions between rotamers, ASP may not be the best approach.

6.1 Future Work

As future work, we suggest the development of a tool to convert the dihedral angles in three-dimensional coordinates, as mentioned in section 5.4. This tool would allow to transform the output of our approach in a PDB file. This PDB file would contain the three-dimensional coordinates of all the atoms of the protein designed. Therefore, it would be possible to compare our solution with the solutions of the existent protein design tools. Although most of the other approaches do not guarantee the optimal solution, it is important to compare the total energy of the protein designed by the different approaches. This can be made correctly using the PDB file of the designed protein to compute the total energy of the protein. Comparing the solutions, even considering that our approach guarantees the optimal solution, may lead to important conclusions. If our solution is better than the other solutions, i.e., the total energy of our solution is lower that the others, than it is interesting to verify if a possible difference of solving time justifies the quality of our solution. If, for some reason, the other solutions are better than ours, it is important to understand the reason. It can be because we may not be considering some kind of information. Therefore, we could improve our approach by considering a possible missing information.

In this work it is not considered any collision detection method between rotamers. The set of possible rotamers considered for each protein can have rotamers that collide with each other when present on the protein structure. It is not desired to design proteins with rotamers that collide because it is not a possible protein. The energy values for interaction between two rotamers have collisions in account, i.e., the interaction energy of two rotamers that collide with each other is very high compared to the other interaction values. Although it is probable that Dead-End Elimination methods eliminate rotamers that collide, it is possible that such rotamers are considered in the final solution (if they are not eliminated by the DEE algorithm). As future work we suggest the use of a tool that identifies pairs of rotamers that collide. This way we can add more restrictions to the ASP program and possibly increase the efficiency of the program.

As ASP revealed to be fast for small number of rotamers considered, therefore the use of different tools to eliminate rotamers or add restrictions to the ASP program can improve the efficiency of the program. The development of hybrid programs that use tools such as Dead-End Elimination methods and collision detection methods in order to eliminate rotamers and add restrictions and then use ASP to find the optimal solution may lead to better results.

Clasp, the ASP solver used in this work, has some internal options that can be modified. In this work we used the default configuration. However, as future work, we suggest to study the impact of each option in this particular approach. It is possible to change the restart policy. The heuristic used by clasp during the search of the solution, as well as using SAT preprocessing or transform the choice rules into normal rules can be modified as well. Although we planned to change and test different options configurations, due to the lack of time, this was not made. Therefore, as future work it is important to test different configurations to verify if our approach can be more efficient.

A Kakuro using ASP

Good examples to be used with ASP are games. There are many games that can be viewed as Constraint Satisfaction Problems (CSP) and can be encoded in ASP. Next it will be presented a codification for Kakuro, a well known game.

Kakuro is a game considered to be the mathematical version of crosswords. Kakuro consists in a grid with white cells, black cells and hints. The hints can be in a right diagonal of a cell or in a left diagonal of a cell. A hint in a left diagonal means that the sum of the consecutive white cells vertically above the hint must sum the number given in the hint. A hint in a right diagonal means that the sum of the horizontally consecutive white cells must be the number given in the hint. The objective of the game is to put a number in each white cells so that the hints can be verified. This game has only three simple rules:

1. Each white cell must have exactly one integer number between 1 and 9.
2. All numbers in a consecutive group of white cells (vertically or horizontally) must be different.
3. The sum of consecutive white cells must be equal to the respective hint.

Figure A.1 illustrates an instance of a kakuro game and the respective solution.

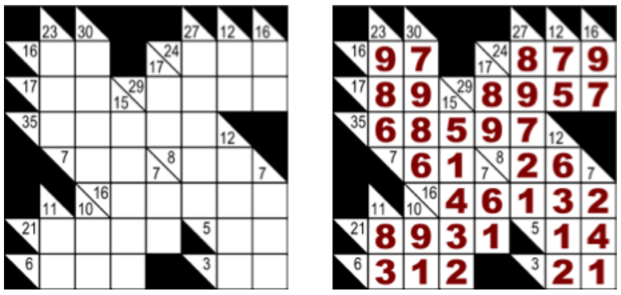


Figure A.1: Kakuro instance and solution

A simple ASP program to solve kakuro is described next. To solve kakuro using ASP, first we define the facts that will be used. The facts represent the specific information about a kakuro instance. Several predicates are defined to represent the information of a kakuro grid. To represent each cell of the grid we use the predicate `cell(X,Y)` where `X,Y` are coordinates representing the line of the grid (`X`) and the column of the grid (`Y`). To define a group of consecutive white cells (horizontally or vertically) we define the predicate `group(Id,X,Y)` meaning that the cell with coordinates `(X,Y)` belongs to group `Id`. The black cells of the grid are represented by the predicate `black(X,Y)`, where `X,Y` are the coordinates of the black cell. Finally, to represent all the information of an instance of a kakuro game, we define the predicates `leftDiag(Sum,X,Y)` and `rightDiag(Sum,X,Y)` meaning that a cell with coordinates `X,Y` is a hint on a left diagonal or right diagonal, respectively, with the value `Sum`.

To solve kakuro we also define two auxiliary predicates that indicates the possible values of the white cells and how much can a group of consecutive white cells sum. The predicates used are `val(1..9)` indicating that possible values of a cell are between 1 and 9, and `sum(1..45)` indicating that a sum of consecutive white cells can be between 1 and 45.

Once the facts are represented, we define rules to represent the constraints of the game. To represent that each white cell must have exactly one number we define the following rule:

```
1{num(N,X,Y):val(N)}1 :- cell(X,Y),not diag(X,Y),not black(X,Y).
```

In this rule we define the predicate `num(N,X,X)` meaning that the cell of coordinates `X,Y` has the number `N`. The rule indicates that each cell that is not a diagonal and is not black, hence is white, must have at least one number and at most one number, i.e. exactly one number, and the number must be a possible value.

The predicate `diag(X,Y)` meaning that the cell `(X,Y)` has a diagonal is defined by the two following rules:

```
diag(X,Y) :- leftDiag(Sum,X,Y),sum(Sum),cell(X,Y).
diag(X,Y) :- rightDiag(Sum,X,Y),sum(Sum),cell(X,Y).
```

These rules represent that if a cell has a left diagonal then has a diagonal and the same holds for the right diagonal.

To add some consistency to the program we also define that diagonals and black cell cannot have any number. This is guaranteed by the following rules:

```
:- num(Val,X,Y),val(Val),cell(X,Y),black(X,Y).
:- num(Val,X,Y),val(Val),cell(X,Y),diag(X,Y).
```

To define the second rule of kakuro that says that the numbers in a consecutive group of white cells must be all different we define rules that represent that two different cells of the same group of consecutive white cells must not have the same number. This is encoded with the rules:

```
:- num(Val,X1,Y1),num(Val,X2,Y2),val(Val),X2!=X1,Y1=Y2,
   group(Id1,X1,Y1),group(Id2,X2,Y2),Id1=Id2.
:- num(Val,X1,Y1),num(Val,X2,Y2),val(Val),X2=X1,Y1!=Y2
   group(Id1,X1,Y1),group(Id2,X2,Y2),Id1=Id2.
```

The third rule of kakuro says that a consecutive group of white cells (horizontally or vertically) must sum the value of the respective hint. To encode this rule we define the predicates `hSum` and `vSum` that represents the horizontal and vertical sum, respectively. The sum of a consecutive white cell is made recursively. For a horizontal sum we sum the cells from the right to the left so that the cell next to the respective hint, which will be on the left of the group, has the sum of all the consecutive white cells. For the vertical sum, it is analogous, hence the sum is made from the bottom to top.

To sum a horizontal group of consecutive cells we say that the sum at each cell is the sum of the number in that cell with all numbers at the right of that cell. If the cell is the last of the consecutive white cells we say that its sum is equal to the number in it. Notice that for a cell to be the last in a horizontal group, the cell on its right must be a diagonal or a black cell or not to exist. The following rules are then defined to the horizontal sum:

```
HSum(Val,X,Y) :-  cell(X,Y),num(Val,X,Y),val(Val),not cell(X,Y+1).
HSum(Val,X,Y) :-  cell(X,Y),num(Val,X,Y),val(Val),black(X,Y+1).
HSum(Val,X,Y) :-  cell(X,Y),num(Val,X,Y),val(Val),diag(X,Y+1).
HSum(Sum+Val,X,Y) :-  hSum(Sum,X,Y+1),sum(Sum),cell(X,Y),
                      num(Val,X,Y),val(Val).
```

The `vSum` predicate is defined analogously.

To satisfy the third rule of kakuro it is needed to define that the sum in the first cell of a group of consecutive white cells must be equal to the respective hint, i.e, their values cannot be different. This is achieved with the rules:

```
:-  hSum(Sum1,X,Y+1),sum(Sum1),cell(X,Y),rightDiag(Sum2,X,Y),
    Sum1!=Sum2.
:-  vSum(Sum1,X+1,Y),sum(Sum1),cell(X,Y),leftDiag(Sum2,X,Y),
    Sum1!=Sum2.
```

With these rules any instance of a kakuro game can be solved.

Many different approaches can be made to solve this problem using answer set programming. The approach presented here is a simple one where we define the rules of the game and let the ASP solver find the solution. A different approach is made in [42]. The approach is more similar to the approach taken by humans to solve the game by hand. In [42] predicates are defined to represent the possible number for each cell and then impossible combinations are eliminated. If a cell has only one possible number then that number must be in that cell.

Different approaches to solve the same problem, not only kakuro, can be equally efficient or it is possible for one approach to be more efficient in solving smaller instances and the other more efficient in the larger instances.

B ASP Codifications for Protein Design

In this section will be shown the complete code for the Simple Codification and for the Double Optimization. For a better understanding it will be given an example of input and respective output for each codification.

B.1 Simple codification

B.1.1 Code

```
%Each position has exactly one amino acid
1{residue(Amin,Pos) : aminoAcid(Amin)}1 :- position(Pos).

%Each position has exactly one rotamer
1{rotamer(Pos,Id,Amin):possibleRotamer(Pos,Id,Amin,_)}1 :- position(Pos).


%There cannot exist a rotamer that do not match the amino acid
:- position(Pos), rotamer(Pos,Id,Amin1), residue(Amin2,Pos),
 Amin1 != Amin2.


%Each rotamer contributes with an energy interaction with the backbone
energy(Pos,Id,Pos,Id,Energy) :- rotamer(Pos,Id,Amin),
 possibleRotamer(Pos,Id,Amin,Energy).

%Each pair of rotamers contributes with an energy interaction between them
energy(Pos1,Id1,Pos2,Id2,Energy) :- interEnergy(Pos1,Id1,Pos2,Id2,Energy),
 rotamer(Pos1,Id1,_), rotamer(Pos2,Id2,_).

%Minimize the energy
#minimize[energy(_,_,_,_,Energy) = Energy].
```

```
%Only show the rotamers
#hide.
#show rotamer(_,_,_).
```

B.1.2 Example

This section illustrates a little example of input and output. The example do not correspond to a real case, it is just to show the codification of the input data and the output produced.

Input

```
aminoAcid(ala).
aminoAcid(val).

position(1).
position(2).

possibleRotamer(1,1,ala,3).
possibleRotamer(1,2,ala,-9).
possibleRotamer(2,1,val,5).
possibleRotamer(2,2,ala,-7).

interEnergy(1,1,2,1,6).
interEnergy(1,1,2,2,0).
interEnergy(1,2,2,1,1).
interEnergy(1,2,2,2,16).
```

Output

```
clasp\ version 2.1.0
Reading from stdin
Solving...
Answer: 1
rotamer(2,2,ala) rotamer(1,2,ala)
Optimization: 16
Answer: 2
rotamer(2,1,val) rotamer(1,2,ala)
Optimization: 13
Answer: 3
```

```
rotamer(2,2,ala) rotamer(1,1,ala)
Optimization: 12
OPTIMUM FOUND

Models      : 1
  Enumerated: 3
  Optimum   : yes
Optimization: 12
Time        : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
```

The optimization value of the optimal solution in this case is 12 because the conversions of the negative values were made. In fact, the value comes from the sum of the positive energies 3 and 0, from `rotamer(1,1,ala)` and from the interaction between the two rotamers of the solution respectively. The energy of `rotamer(2,2,ala)`, as it is a negative value, after the transformation, has the absolute value of the negative interactions that do not belong to the solution. In this case as value 9. Therefore, the optimization value for the solution is 12 (3+0+9).

B.2 Double Optimization Codification

B.2.1 Code

```
%Each position has exactly one amino acid
1{residue(Amin,Pos) : aminoAcid(Amin)}1 :- position(Pos).

%Cada posicao tem um e um so rotamer
1{rotamer(Pos,Id,Amin) : possiblePRotamer(Pos,Id,Amin,_),
 rotamer(Pos,Id,Amin) : possibleNRotamer(Pos,Id,Amin,_)}1
  :- position(Pos).



%Each position has exactly one rotamer
:- position(Pos), rotamer(Pos,Id,Amin1), residue(Amin2,Pos),
 Amin1 != Amin2.



%Each rotamer contributes with an energy interaction with the backbone
```

```
%positive
pEnergy(Pos,Id,Pos,Id,Energy) :- rotamer(Pos,Id,Amin),
  possiblePRotamer(Pos,Id,Amin,Energy).
%negative
nEnergy(Pos,Id,Pos,Id,Energy) :- rotamer(Pos,Id,Amin),
  possibleNRotamer(Pos,Id,Amin,Energy).



%Each pair of rotamers contributes with an energy interaction between them
%positive
pEnergy(Pos1,Id1,Pos2,Id2,Energy) :- interPEnergy(Pos1,Id1,Pos2,Id2,Energy),
  rotamer(Pos1,Id1,_), rotamer(Pos2,Id2,_).
%negative
nEnergy(Pos1,Id1,Pos2,Id2,Energy) :- interNEnergy(Pos1,Id1,Pos2,Id2,Energy),
  rotamer(Pos1,Id1,_), rotamer(Pos2,Id2,_).

%Minimize the positive energy
#minimize[pEnergy(_,_,_,_,Energy) = Energy @1].
%Maximize the negative energy
#maximize[nEnergy(_,_,_,_,Energy) = Energy @1].

%Only show the rotamers
#hide.
#show rotamer(_,_,_).
```

B.2.2 Example

Now will be showed a the same example showed in previous section but adapted for the double optimization codification.

Input

```
aminoAcid(ala).
aminoAcid(val).

position(1).
position(2).

possiblePRotamer(1,1,ala,3).
```

```
possibleNRotamer(1,2,ala,9).
possiblePRotamer(2,1,val,5).
possibleNRotamer(2,2,ala,7).

interPEnergy(1,1,2,1,6).
interPEnergy(1,1,2,2,0).
interPEnergy(1,2,2,1,1).
interPEnergy(1,2,2,2,16).
```

Output

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
rotamer(2,1,val) rotamer(1,1,ala)
Optimization: 30
Answer: 2
rotamer(2,1,val) rotamer(1,2,ala)
Optimization: 13
Answer: 3
rotamer(2,2,ala) rotamer(1,1,ala)
Optimization: 12
OPTIMUM FOUND

Models      : 1
  Enumerated: 3
  Optimum   : yes
Optimization: 12
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
```

C Dead End Elimination Codification

In this section is shown the complete implementation of the Original Dead-End Elimination algorithm using ASP.

```
bestCase(Pos,Id,E,Pos2) :- E = #min[interEnergy(Pos,Id,Pos2,_,V)=V],
 Pos2>Pos, possibleRotamer(Pos,Id,_,_), position(Pos2),
 interEnergy(Pos,Id,Pos2,_,_).

bestCase(Pos,Id,E,Pos2) :- E = #min[interEnergy(Pos2,_,Pos,Id,V)=V],
 Pos2<Pos, possibleRotamer(Pos,Id,_,_), position(Pos2),
 interEnergy(Pos2,_,Pos,Id,_).



bestSum(Pos,Id,Sum+E) :- Sum = #sum[bestCase(Pos,Id,V,_)=V],
 possibleRotamer(Pos,Id,_,E).




worstCase(Pos,Id,E,Pos2) :- E = #max[interEnergy(Pos,Id,Pos2,_,V)=V],
 Pos2>Pos, possibleRotamer(Pos,Id,_,_), position(Pos2),
 interEnergy(Pos,Id,Pos2,_,_).

worstCase(Pos,Id,E,Pos2) :- E = #max[interEnergy(Pos2,_,Pos,Id,V)=V],
 Pos2<Pos, possibleRotamer(Pos,Id,_,_), position(Pos2),
 interEnergy(Pos2,_,Pos,Id,_).



worstSum(Pos,Id,Sum+E) :- Sum = #sum[worstCase(Pos,Id,V,_)=V],
 possibleRotamer(Pos,Id,_,E).
```

```
impossibleRotamer(Pos,Id) :- possibleRotamer(Pos,Id,_,_),
 possibleRotamer(Pos,Id2,_,_), Id!=Id2, bestSum(Pos,Id,Sum),
 worstSum(Pos,Id2,Sum2), Sum>Sum2.



#hide.
#show impossibleRotamer/2.
```

D Complete Tables of Results

In this section will be shown the complete tables of results for the dead-end elimination algorithms implemented and for the two codifications made.

Tables D.1, D.2 and D.3 show the results for the Original DEE algorithm implemented in ASP. The times are in seconds and the grounder size in Megabytes. It also shows the number of eliminated rotamers.

Tables D.4, D.5 and D.6 contain the results for the Original, Simple Goldstein and Simple Split DEE algorithms implemented in Java. These tables shows the number of eliminated rotamers in each algorithm. The times are in seconds.

The complete results for the simple codification can be seen in tables D.7, D.8 and D.9. In this tables there were not applied any DEE algorithm to the instances. It is possible to see the number of rotamers considered in each instance, the grounder size in Megabytes and the times of grounding, solving and total, in seconds. The instances for which we do not computed the optimal solution in under 3 hours are represented with '`+3h.`'. The instances with '`mem.`' means that did not solve under the memory size limit.

Tables D.10, D.11 and D.12 show the results of the simple codification in which was applied the Simple Split DEE algorithm to each instance.

Tables D.13, D.14 and D.15, and tables D.16, D.17 and D.18, are similar to the mentioned above but for the Double Optimization codification. The first three tables consider instances where there were not applied DEE algorithms The last three tables show the results of the Double Optimization considering instances where were applied the Simple Split DEE algorithm.

Protein	Pos.	Amino Acids	# Elim. Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1MFG	17	All	39	44,7	1668,318	2,361	1670,679
		Hydro.	19	2,1	17,703	0,120	17,823
		Fixed	9	0,0	0,023	0,000	0,023
	15	All	40	24,9	808,308	1,341	809,649
		Hydro.	12	1,1	7,816	0,062	7,878
		Fixed	10	0,0	0,013	0,000	0,013
	12	All	34	13,8	431,006	0,766	431,772
		Hydro.	22	0,6	4,316	0,030	4,346
		Fixed	8	0,0	0,010	0,000	0,010
	10	All	25	8,3	249,912	0,446	250,358
		Hydro.	9	0,4	2,546	0,013	2,559
		Fixed	7	0,0	0,008	0,000	0,008
1BE9	17	All	38	42,5	1573,381	2,510	1575,891
		Hydro.	21	1,8	15,126	0,110	15,236
		Fixed	7	0,0	0,019	0,000	0,019
	15	All	34	27,1	929,879	1,630	931,509
		Hydro.	24	1,2	9,447	0,070	9,517
		Fixed	8	0,0	0,015	0,000	0,015
	12	All	53	14,3	476,855	0,880	477,735
		Hydro.	29	0,7	5,136	0,030	5,166
		Fixed	7	0,0	0,008	0,000	0,008
	10	All	18	9,9	327,966	0,610	328,576
		Hydro.	14	0,4	2,854	0,010	2,864
		Fixed	7	0,0	0,007	0,000	0,007
2GZV	17	All	22	55,8	2353,776	3,330	2357,106
		Hydro.	23	2,4	22,889	0,140	23,029
		Fixed	15	0,0	0,037	0,000	0,037
	15	All	22	31,8	1167,628	1,920	1169,548
		Hydro.	21	1,3	10,347	0,070	10,417
		Fixed	11	0,0	0,020	0,000	0,020
	12	All	30	17,7	645,332	1,080	646,412
		Hydro.	26	0,7	5,707	0,030	5,737
		Fixed	8	0,0	0,013	0,000	0,013
	10	All	21	10,8	373,509	0,700	374,209
		Hydro.	13	0,4	3,280	0,010	3,290
		Fixed	8	0,0	0,009	0,000	0,009
2EGN	17	All	30	41,0	1576,206	2,400	1578,606
		Hydro.	30	1,9	16,318	0,110	16,428
		Fixed	14	0,0	0,025	0,000	0,025
	15	All	60	24,4	820,727	1,460	822,187
		Hydro.	42	1,0	7,632	0,060	7,692
		Fixed	13	0,0	0,013	0,000	0,013
	12	All	103	10,9	328,585	0,670	329,255
		Hydro.	28	0,5	3,443	0,010	3,453
		Fixed	10	0,0	0,008	0,000	0,008
	10	All	77	7,1	207,857	0,410	208,267
		Hydro.	41	0,3	2,142	0,010	2,152
		Fixed	8	0,0	0,006	0,000	0,006

Table D.1: DEE with ASP - 1MFG, 1BE9, 2GZV, 2EGN

Protein	Pos.	Amino Acids	# Elim. Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
2FNE	17	All	41	49,1	1973,531	2,960	1976,491
		Hydro.	46	2,2	20,110	0,120	20,230
		Fixed	8	0,0	0,027	0,000	0,027
	15	All	35	29,2	1051,298	1,760	1053,058
		Hydro.	38	1,2	9,618	0,060	9,678
		Fixed	8	0,0	0,018	0,000	0,018
	12	All	72	14,1	455,842	0,850	456,692
		Hydro.	33	0,6	4,356	0,020	4,376
		Fixed	6	0,0	0,011	0,000	0,011
	10	All	48	9,0	287,701	0,560	288,261
		Hydro.	14	0,4	2,590	0,010	2,600
		Fixed	5	0,0	0,007	0,000	0,007
1RZX	17	All	19	47,9	1880,231	2,820	1883,051
		Hydro.	18	2,0	17,123	0,110	17,233
		Fixed	8	0,0	0,036	0,000	0,036
	15	All	19	27,8	980,342	1,660	982,002
		Hydro.	17	1,0	7,057	0,060	7,117
		Fixed	12	0,0	0,025	0,000	0,025
	12	All	40	12,3	410,918	0,740	411,658
		Hydro.	18	0,5	3,650	0,020	3,670
		Fixed	8	0,0	0,010	0,000	0,010
	10	All	15	10,0	346,099	0,600	346,699
		Hydro.	15	0,4	2,709	0,010	2,719
		Fixed	9	0,0	0,006	0,000	0,006
1N7F	17	All	67	41,4	1592,628	2,480	1595,108
		Hydro.	23	1,9	16,492	0,110	16,602
		Fixed	7	0,0	0,026	0,000	0,026
	15	All	84	22,8	737,767	1,360	739,127
		Hydro.	27	1,0	7,279	0,050	7,329
		Fixed	9	0,0	0,018	0,000	0,018
	12	All	38	13,8	463,031	0,860	463,891
		Hydro.	17	0,6	4,284	0,020	4,304
		Fixed	9	0,0	0,011	0,000	0,011
	10	All	64	8,2	249,685	0,510	250,195
		Hydro.	27	0,4	2,366	0,010	2,376
		Fixed	9	0,0	0,005	0,000	0,005
1QAU	17	All	33	45,5	1766,765	2,730	1769,495
		Hydro.	28	2,0	18,231	0,120	18,351
		Fixed	10	0,0	0,027	0,000	0,027
	15	All	34	25,4	864,903	1,610	866,513
		Hydro.	26	1,1	8,300	0,060	8,360
		Fixed	8	0,0	0,018	0,000	0,018
	12	All	44	12,8	413,096	0,760	413,856
		Hydro.	19	0,6	4,133	0,020	4,153
		Fixed	5	0,0	0,010	0,000	0,010
	10	All	80	8,7	278,254	0,550	278,804
		Hydro.	26	0,4	2,572	0,010	2,582
		Fixed	5	0,0	0,005	0,000	0,005

Table D.2: DEE with ASP - 2FNE, 1RZX, 1N7F, 1QAU

Protein	Pos.	Amino Acids	# Elim. Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1TP3	17	All	34	43,6	1613,248	2,550	1615,798
		Hydro.	20	1,9	15,614	0,110	15,724
		Fixed	9	0,0	0,028	0,000	0,028
	15	All	31	29,2	1023,314	1,760	1025,074
		Hydro.	17	1,3	10,123	0,060	10,183
		Fixed	12	0,0	0,021	0,000	0,021
	12	All	24	15,9	542,621	0,980	543,601
		Hydro.	24	0,7	5,927	0,030	5,957
		Fixed	10	0,0	0,012	0,000	0,012
	10	All	37	10,4	353,710	0,620	354,330
		Hydro.	23	0,5	3,459	0,010	3,469
		Fixed	10	0,0	0,009	0,000	0,009
1I92	17	All	22	45,0	1809,885	2,700	1812,585
		Hydro.	26	2,2	21,396	0,130	21,526
		Fixed	16	0,0	0,045	0,000	0,045
	15	All	27	26,3	933,011	1,580	934,591
		Hydro.	31	1,3	10,761	0,070	10,831
		Fixed	16	0,0	0,037	0,000	0,037
	12	All	48	13,4	470,347	0,860	471,207
		Hydro.	29	0,6	5,304	0,030	5,334
		Fixed	13	0,0	0,022	0,000	0,022
	10	All	18	9,7	349,429	0,630	350,059
		Hydro.	33	0,5	3,707	0,010	3,717
		Fixed	11	0,0	0,016	0,000	0,016

Table D.3: DEE with ASP - 1TP3, 1I92

Protein	Pos.	Amino Acids	Original DEE		Simple Goldstein		Simple Split	
			# Elim. Rotamers	Time(s)	# Elim. Rotamers	Time(s)	# Elim. Rotamers	Time(s)
1MFG	17	All	39	26,477	431	652,490	583	4536,752
		Hydro.	19	1,728	225	7,439	347	20,313
		Fixed	9	0,158	18	0,177	24	0,181
	15	All	40	14,030	505	287,440	787	1918,087
		Hydro.	12	1,264	190	3,465	321	7,633
		Fixed	10	0,138	17	0,133	19	0,151
	12	All	34	7,533	521	137,887	886	632,924
		Hydro.	22	1,059	154	2,143	273	3,663
		Fixed	8	0,116	14	0,112	16	0,131
	10	All	25	4,882	518	73,508	858	199,909
		Hydro.	9	0,897	132	1,601	228	2,391
		Fixed	7	0,106	12	0,110	14	0,125
1BE9	17	All	38	22,390	445	572,432	606	3885,199
		Hydro.	21	1,531	249	5,769	374	14,396
		Fixed	7	0,194	16	0,200	23	0,232
	15	All	34	14,854	497	315,710	777	2272,126
		Hydro.	24	1,206	220	3,642	373	7,344
		Fixed	8	0,130	16	0,132	20	0,151
	12	All	53	7,731	506	151,439	866	711,925
		Hydro.	29	0,976	169	2,190	311	3,977
		Fixed	7	0,118	12	0,121	15	0,129
	10	All	18	5,555	539	92,421	979	286,614
		Hydro.	14	0,793	151	1,374	244	2,374
		Fixed	7	0,084	9	0,088	11	0,086
2GZV	17	All	22	32,517	441	1272,201	612	7740,051
		Hydro.	23	1,787	215	9,877	372	31,211
		Fixed	15	0,249	18	0,275	36	0,303
	15	All	22	18,031	541	431,441	851	2741,559
		Hydro.	21	1,236	197	4,549	355	10,601
		Fixed	11	0,157	15	0,171	26	0,210
	12	All	30	9,687	545	219,122	917	1152,268
		Hydro.	26	1,046	175	2,563	297	5,220
		Fixed	8	0,139	12	0,141	23	0,159
	10	All	21	6,413	525	119,700	799	473,970
		Hydro.	13	0,863	150	1,661	250	3,064
		Fixed	8	0,119	10	0,123	18	0,138
2EGN	17	All	30	22,948	597	599,467	789	4538,185
		Hydro.	30	1,522	246	6,729	382	16,753
		Fixed	14	0,188	21	0,191	27	0,224
	15	All	60	12,824	686	275,586	971	1637,053
		Hydro.	42	1,151	212	3,291	321	7,148
		Fixed	13	0,128	17	0,129	20	0,149
	12	All	103	6,554	602	95,863	945	350,433
		Hydro.	28	0,858	166	1,678	242	3,855
		Fixed	10	0,112	11	0,116	14	0,132
	10	All	77	4,195	565	54,748	863	170,553
		Hydro.	41	0,792	140	1,248	205	2,385
		Fixed	8	0,079	9	0,084	12	0,102

Table D.4: DEE with Java - 1MFG, 1BE9, 2GZV, 2EGN

Protein	Pos.	Amino Acids	Original DEE		Simple Goldstein		Simple Split	
			# Elim. Rotamers	Time(s)	# Elim. Rotamers	Time(s)	# Elim. Rotamers	Time(s)
2FNE	17	All	41	28,208	428	801,896	586	5848,930
		Hydro.	46	1,713	255	8,235	358	26,148
		Fixed	8	0,216	16	0,238	26	0,271
	15	All	35	15,980	496	389,451	758	2127,102
		Hydro.	38	1,170	218	3,906	321	8,945
		Fixed	8	0,150	14	0,154	22	0,179
	12	All	72	7,866	527	149,279	859	642,600
		Hydro.	33	0,921	166	1,960	269	3,835
		Fixed	6	0,116	10	0,121	15	0,159
	10	All	48	5,183	539	84,241	913	271,897
		Hydro.	14	0,840	146	1,373	228	2,407
		Fixed	5	0,101	8	0,100	12	0,118
1RZX	17	All	19	26,683	519	689,014	713	4483,937
		Hydro.	18	1,576	226	6,815	364	16,584
		Fixed	8	0,270	19	0,262	31	0,287
	15	All	19	15,173	600	323,011	931	1707,972
		Hydro.	17	1,123	199	2,942	337	5,640
		Fixed	12	0,234	22	0,231	30	0,253
	12	All	40	7,000	513	133,068	856	671,037
		Hydro.	18	0,868	170	1,692	264	2,933
		Fixed	8	0,120	13	0,121	16	0,135
	10	All	15	5,707	501	105,130	759	536,278
		Hydro.	15	0,835	139	1,428	223	2,366
		Fixed	9	0,089	11	0,087	13	0,103
1N7F	17	All	67	22,902	577	603,960	810	4507,277
		Hydro.	23	1,540	221	6,939	349	19,582
		Fixed	7	0,216	15	0,252	22	0,308
	15	All	84	12,234	661	248,828	974	1341,552
		Hydro.	27	1,192	193	3,135	339	6,303
		Fixed	9	0,139	13	0,151	23	0,172
	12	All	38	7,658	545	150,875	930	575,559
		Hydro.	17	0,903	140	2,159	277	4,287
		Fixed	9	0,130	13	0,139	21	0,148
	10	All	64	4,855	497	78,362	823	273,124
		Hydro.	27	0,792	133	1,406	219	2,307
		Fixed	9	0,100	12	0,099	15	0,108
1QAU	17	All	33	25,928	473	697,268	625	4842,914
		Hydro.	28	1,643	222	7,778	344	20,606
		Fixed	10	0,222	13	0,258	25	0,319
	15	All	34	14,160	579	305,220	861	2061,548
		Hydro.	26	1,211	183	3,692	311	8,653
		Fixed	8	0,144	10	0,145	19	0,184
	12	All	44	7,176	535	136,677	903	732,509
		Hydro.	19	0,947	146	2,018	250	4,354
		Fixed	5	0,124	7	0,127	14	0,148
	10	All	80	5,142	529	82,623	919	265,242
		Hydro.	26	0,773	130	1,388	233	2,710
		Fixed	5	0,082	7	0,090	10	0,109

Table D.5: DEE with Java - 2FNE, 1RZX, 1N7F, 1QAU

Protein	Pos.	Amino Acids	Original DEE		Simple Goldstein		Simple Split	
			# Elim. Rotamers	Time(s)	# Elim. Rotamers	Time(s)	# Elim. Rotamers	Time(s)
1TP3	17	All	34	24,078	449	614,867	659	4927,256
		Hydro.	20	1,623	241	6,381	362	17,712
		Fixed	9	0,225	18	0,234	30	0,241
	15	All	31	15,435	517	356,142	813	2220,966
		Hydro.	17	1,269	214	4,230	346	9,451
		Fixed	12	0,173	19	0,165	25	0,193
	12	All	24	8,436	490	185,516	866	1079,788
		Hydro.	24	1,010	171	2,727	311	5,190
		Fixed	10	0,128	14	0,127	20	0,152
	10	All	37	6,000	485	112,576	850	426,576
		Hydro.	23	0,901	149	1,710	259	3,006
		Fixed	10	0,116	13	0,120	17	0,133
1I92	17	All	22	26,778	515	722,435	681	5997,457
		Hydro.	26	1,819	260	9,047	374	25,570
		Fixed	16	0,305	23	0,317	42	0,438
	15	All	27	14,909	623	314,557	975	1908,529
		Hydro.	31	1,192	219	4,240	318	10,095
		Fixed	16	0,234	22	0,256	38	0,289
	12	All	48	7,301	578	141,003	1041	521,726
		Hydro.	29	0,935	169	2,314	266	4,687
		Fixed	13	0,135	19	0,152	28	0,181
	10	All	18	5,596	550	101,603	933	324,018
		Hydro.	33	0,883	156	1,699	225	3,526
		Fixed	11	0,130	18	0,135	26	0,151

Table D.6: DEE with Java - 1TP3, 1I92

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1MFG	17	All	2608	163,4	42,408	mem.	—
		Hydro.	560	6,8	1,799	mem.	—
		Fixed	41	0,0	0,009	0,020	0,029
	15	All	1972	91,2	23,311	mem.	—
		Hydro.	410	3,4	0,903	+3h.	—
		Fixed	34	0,0	0,006	0,000	0,006
	12	All	1500	52,5	13,006	mem.	—
		Hydro.	314	1,9	0,498	+3h.	—
		Fixed	28	0,0	0,005	0,000	0,005
	10	All	1189	30,7	7,823	mem.	—
		Hydro.	250	1,2	0,309	+3h.	—
		Fixed	24	0,0	0,006	0,000	0,006
1BE9	17	All	2547	154,8	38,453	mem.	—
		Hydro.	529	6,1	1,553	+3h.	—
		Fixed	42	0,1	0,012	0,010	0,022
	15	All	2063	98,9	25,034	mem.	—
		Hydro.	435	3,8	1,005	+3h.	—
		Fixed	35	0,0	0,008	0,000	0,008
	12	All	1540	54,5	13,512	mem.	—
		Hydro.	330	2,1	0,564	+3h.	—
		Fixed	27	0,0	0,005	0,000	0,005
	10	All	1292	36,5	9,127	mem.	—
		Hydro.	258	1,3	0,327	+3h.	—
		Fixed	21	0,0	0,003	0,000	0,003
2GZV	17	All	2934	202,5	50,067	mem.	—
		Hydro.	613	8,0	2,050	mem.	—
		Fixed	53	0,1	0,014	0,120	0,134
	15	All	2247	115,0	29,281	mem.	—
		Hydro.	456	4,1	1,086	mem.	—
		Fixed	41	0,0	0,010	0,010	0,020
	12	All	1721	66,6	16,596	mem.	—
		Hydro.	352	2,3	0,631	+3h.	—
		Fixed	35	0,0	0,007	0,000	0,007
	10	All	1371	40,1	10,369	mem.	—
		Hydro.	275	1,4	0,371	+3h.	—
		Fixed	28	0,0	0,005	0,000	0,005
2EGN	17	All	2548	149,5	37,290	mem.	—
		Hydro.	546	6,2	1,643	mem.	—
		Fixed	44	0,1	0,013	0,020	0,033
	15	All	2005	89,2	22,904	mem.	—
		Hydro.	413	3,2	0,878	+3h.	—
		Fixed	35	0,0	0,006	0,000	0,006
	12	All	1371	40,0	10,438	mem.	—
		Hydro.	295	1,6	0,422	+3h.	—
		Fixed	26	0,0	0,004	0,000	0,004
	10	All	1118	26,0	6,779	mem.	—
		Hydro.	238	1,1	0,262	+3h.	—
		Fixed	22	0,0	0,004	0,000	0,004

Table D.7: Simple Codification Results without DEE - 1MFG, 1BE9, 2GZV, 2EGN

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
2FNE	17	All	2736	178,6	44,676	mem.	—
		Hydro.	577	7,2	1,885	mem.	—
		Fixed	45	0,1	0,013	0,030	0,043
	15	All	2144	106,6	26,913	mem.	—
		Hydro.	438	3,8	1,023	mem.	—
		Fixed	39	0,0	0,010	0,010	0,020
	12	All	1529	53,6	13,761	mem.	—
		Hydro.	316	1,9	0,511	+3h.	—
		Fixed	29	0,0	0,005	0,000	0,005
	10	All	1257	33,2	8,692	mem.	—
		Hydro.	253	1,2	0,306	+3h.	—
		Fixed	24	0,0	0,005	0,000	0,005
1RZX	17	All	2722	173,3	44,909	mem.	—
		Hydro.	554	6,5	1,693	mem.	—
		Fixed	53	0,1	0,016	0,020	0,036
	15	All	2121	100,7	25,021	mem.	—
		Hydro.	404	3,1	0,843	mem.	—
		Fixed	45	0,1	0,012	0,010	0,022
	12	All	1452	45,2	11,907	mem.	—
		Hydro.	299	1,7	0,444	+3h.	—
		Fixed	28	0,0	0,005	0,000	0,005
	10	All	1306	37,2	9,201	mem.	—
		Hydro.	253	1,3	0,311	+3h.	—
		Fixed	23	0,0	0,004	0,000	0,004
1N7F	17	All	2548	150,3	36,765	mem.	—
		Hydro.	544	6,2	1,645	mem.	—
		Fixed	46	0,1	0,013	0,170	0,183
	15	All	1934	83,0	21,526	mem.	—
		Hydro.	402	3,0	0,833	+3h.	—
		Fixed	38	0,0	0,009	0,020	0,029
	12	All	1539	52,8	13,142	mem.	—
		Hydro.	315	1,9	0,482	+3h.	—
		Fixed	33	0,0	0,008	0,000	0,008
	10	All	1192	29,9	7,647	mem.	—
		Hydro.	246	1,1	0,288	+3h.	—
		Fixed	25	0,0	0,003	0,000	0,003
1QAU	17	All	2643	165,3	42,039	mem.	—
		Hydro.	562	6,8	1,780	mem.	—
		Fixed	47	0,1	0,014	0,170	0,184
	15	All	2023	92,7	23,664	mem.	—
		Hydro.	421	3,4	0,937	mem.	—
		Fixed	37	0,0	0,007	0,020	0,027
	12	All	1470	46,9	11,994	mem.	—
		Hydro.	309	1,8	0,474	+3h.	—
		Fixed	29	0,0	0,005	0,000	0,005
	10	All	1223	32,0	8,290	mem.	—
		Hydro.	251	1,2	0,307	+3h.	—
		Fixed	23	0,0	0,004	0,000	0,004

Table D.8: Simple Codification Results without DEE - 2FNE, 1RZX, 1N7F, 1QAU

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1TP3	17	All	2582	158,3	39,623	mem.	—
		Hydro.	536	6,2	1,644	mem.	—
		Fixed	47	0,1	0,014	0,030	0,044
	15	All	2142	106,2	26,743	mem.	—
		Hydro.	447	4,0	1,066	mem.	—
		Fixed	40	0,0	0,009	0,000	0,009
	12	All	1627	60,3	15,200	mem.	—
		Hydro.	351	2,4	0,650	+3h.	—
		Fixed	32	0,0	0,007	0,000	0,007
	10	All	1339	38,6	9,872	mem.	—
		Hydro.	279	1,5	0,384	+3h.	—
		Fixed	27	0,0	0,003	0,000	0,003
1I92	17	All	2648	164,5	42,600	mem.	—
		Hydro.	593	7,5	1,918	mem.	—
		Fixed	59	0,1	0,018	0,300	0,318
	15	All	2055	96,0	24,474	mem.	—
		Hydro.	455	4,0	1,085	mem.	—
		Fixed	53	0,1	0,017	0,140	0,157
	12	All	1524	51,5	12,877	mem.	—
		Hydro.	335	2,1	0,538	+3h.	—
		Fixed	40	0,0	0,009	0,010	0,019
	10	All	1313	36,5	9,287	mem.	—
		Hydro.	283	1,5	0,392	+3h.	—
		Fixed	36	0,0	0,004	0,000	0,004

Table D.9: Simple Codification Results without DEE - 1TP3, 1I92

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1MFG	17	All	2025	98,8	24,876	mem.	—
		Hydro.	213	0,9	0,229	+3h.	—
		Fixed	17	0,0	0,003	0,000	0,003
	15	All	1185	31,1	8,045	mem.	—
		Hydro.	89	0,2	0,037	679,220	679,257
		Fixed	15	0,0	0,002	0,000	0,002
	12	All	614	8,2	2,019	mem.	—
		Hydro.	41	0,0	0,009	0,230	0,239
		Fixed	12	0,0	0,002	0,000	0,002
	10	All	331	2,2	0,535	+3h.	—
		Hydro.	22	0,0	0,001	0,000	0,001
		Fixed	10	0,0	0,000	0,000	0,000
1BE9	17	All	1941	90,5	22,928	mem.	—
		Hydro.	155	0,5	0,110	+3h.	—
		Fixed	19	0,0	0,005	0,000	0,005
	15	All	1286	36,7	9,532	mem.	—
		Hydro.	62	0,1	0,019	1,700	1,719
		Fixed	15	0,0	0,001	0,000	0,001
	12	All	674	9,7	2,354	mem.	—
		Hydro.	19	0,0	0,002	0,000	0,002
		Fixed	12	0,0	0,000	0,000	0,000
	10	All	313	1,9	0,465	+3h.	—
		Hydro.	14	0,0	0,002	0,000	0,002
		Fixed	10	0,0	0,000	0,000	0,000
2GZV	17	All	2322	128,5	32,351	mem.	—
		Hydro.	241	1,2	0,290	+3h.	—
		Fixed	17	0,0	0,002	0,000	0,002
	15	All	1396	42,9	11,371	mem.	—
		Hydro.	101	0,2	0,050	+3h.	—
		Fixed	15	0,0	0,003	0,000	0,003
	12	All	804	13,9	3,629	mem.	—
		Hydro.	55	0,1	0,015	2,380	2,395
		Fixed	12	0,0	0,002	0,000	0,002
	10	All	572	7,0	1,741	mem.	—
		Hydro.	25	0,0	0,003	0,010	0,013
		Fixed	10	0,0	0,001	0,000	0,001
2EGN	17	All	1759	73,9	18,193	mem.	—
		Hydro.	164	0,5	0,135	+3h.	—
		Fixed	17	0,0	0,003	0,000	0,003
	15	All	1034	23,2	6,022	mem.	—
		Hydro.	92	0,2	0,042	280,710	280,752
		Fixed	15	0,0	0,004	0,000	0,004
	12	All	426	3,7	0,928	mem.	—
		Hydro.	53	0,1	0,012	0,170	0,182
		Fixed	12	0,0	0,001	0,000	0,001
	10	All	255	1,3	0,321	+3h.	—
		Hydro.	33	0,0	0,006	0,010	0,016
		Fixed	10	0,0	0,002	0,000	0,002

Table D.10: Simple Codification Results with DEE - 1MFG, 1BE9, 2GZV, 2EGN

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
2FNE	17	All	2150	110,8	28,427	mem.	—
		Hydro.	219	1,0	0,241	+3h.	—
		Fixed	19	0,0	0,003	0,000	0,003
	15	All	1386	42,8	11,242	mem.	—
		Hydro.	117	0,3	0,065	4976,490	4976,555
		Fixed	17	0,0	0,002	0,000	0,002
	12	All	670	9,8	2,550	mem.	—
		Hydro.	47	0,0	0,011	0,300	0,311
		Fixed	14	0,0	0,003	0,000	0,003
	10	All	344	2,4	0,591	+3h.	—
		Hydro.	25	0,0	0,004	0,000	0,004
		Fixed	12	0,0	0,000	0,000	0,000
1RZX	17	All	2009	95,7	24,255	mem.	—
		Hydro.	190	0,7	0,174	+3h.	—
		Fixed	22	0,0	0,005	0,000	0,005
	15	All	1190	30,3	8,013	mem.	—
		Hydro.	67	0,1	0,023	36,020	36,043
		Fixed	15	0,0	0,002	0,000	0,002
	12	All	596	7,5	1,839	mem.	—
		Hydro.	35	0,0	0,005	0,010	0,015
		Fixed	12	0,0	0,001	0,000	0,001
	10	All	547	6,4	1,527	mem.	—
		Hydro.	30	0,0	0,003	0,000	0,003
		Fixed	10	0,0	0,000	0,000	0,000
1N7F	17	All	1738	71,6	17,627	mem.	—
		Hydro.	195	0,7	0,190	+3h.	—
		Fixed	24	0,0	0,005	0,000	0,005
	15	All	960	19,7	5,381	mem.	—
		Hydro.	63	0,1	0,018	66,020	66,038
		Fixed	15	0,0	0,000	0,000	0,000
	12	All	609	8,0	1,991	mem.	—
		Hydro.	38	0,0	0,007	0,170	0,177
		Fixed	12	0,0	0,000	0,000	0,000
	10	All	369	2,8	0,690	+3h.	—
		Hydro.	27	0,0	0,003	0,000	0,003
		Fixed	10	0,0	0,001	0,000	0,001
1QAU	17	All	2018	97,2	24,516	mem.	—
		Hydro.	218	0,9	0,238	+3h.	—
		Fixed	22	0,0	0,006	0,000	0,006
	15	All	1162	29,3	7,613	mem.	—
		Hydro.	110	0,2	0,058	2295,620	2295,678
		Fixed	18	0,0	0,002	0,000	0,002
	12	All	567	6,9	1,707	mem.	—
		Hydro.	59	0,1	0,018	4,810	4,828
		Fixed	15	0,0	0,002	0,000	0,002
	10	All	304	1,8	0,450	+3h.	—
		Hydro.	18	0,0	0,003	0,000	0,003
		Fixed	13	0,0	0,002	0,000	0,002

Table D.11: Simple Codification Results with DEE - 2FNE, 1RZX, 1N7F, 1QAU

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1TP3	17	All	1923	88,6	22,583	mem.	—
		Hydro.	174	0,6	0,149	+3h.	—
		Fixed	17	0,0	0,003	0,000	0,003
	15	All	1329	39,2	10,526	mem.	—
		Hydro.	101	0,2	0,051	463,010	463,061
		Fixed	15	0,0	0,004	0,000	0,004
	12	All	761	12,5	3,136	mem.	—
		Hydro.	40	0,0	0,009	0,050	0,059
		Fixed	12	0,0	0,001	0,000	0,001
	10	All	489	5,0	1,244	mem.	—
		Hydro.	20	0,0	0,004	0,000	0,004
		Fixed	10	0,0	0,002	0,000	0,002
1I92	17	All	1967	92,8	23,498	mem.	—
		Hydro.	219	0,9	0,239	+3h.	—
		Fixed	17	0,0	0,003	0,000	0,003
	15	All	1080	25,1	6,676	mem.	—
		Hydro.	137	0,3	0,091	+3h.	—
		Fixed	15	0,0	0,002	0,000	0,002
	12	All	483	4,8	1,227	mem.	—
		Hydro.	69	0,1	0,023	2,690	2,713
		Fixed	12	0,0	0,002	0,000	0,002
	10	All	380	2,9	0,713	mem.	—
		Hydro.	58	0,1	0,013	0,450	0,463
		Fixed	10	0,0	0,001	0,000	0,001

Table D.12: Simple Codification Results with DEE - 1TP3, 1I92

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1MFG	17	All	2608	163,4	42,029	mem.	—
		Hydro.	560	6,8	1,833	mem.	—
		Fixed	41	0,0	0,010	0,010	0,020
	15	All	1972	91,2	23,112	mem.	—
		Hydro.	410	3,4	0,883	+3h.	—
		Fixed	34	0,0	0,010	0,000	0,010
	12	All	1500	52,5	13,024	mem.	—
		Hydro.	314	1,9	0,482	+3h.	—
		Fixed	28	0,0	0,003	0,000	0,003
	10	All	1189	30,8	8,100	mem.	—
		Hydro.	250	1,2	0,296	+3h.	—
		Fixed	24	0,0	0,004	0,000	0,004
1BE9	17	All	2547	154,8	39,849	mem.	—
		Hydro.	529	6,1	1,584	+3h.	—
		Fixed	42	0,1	0,011	0,000	0,011
	15	All	2063	98,9	25,096	mem.	—
		Hydro.	435	3,8	1,026	+3h.	—
		Fixed	35	0,0	0,008	0,000	0,008
	12	All	1540	54,5	13,505	mem.	—
		Hydro.	330	2,1	0,553	+3h.	—
		Fixed	27	0,0	0,004	0,000	0,004
	10	All	1292	36,5	9,437	mem.	—
		Hydro.	258	1,3	0,309	+3h.	—
		Fixed	21	0,0	0,003	0,000	0,003
2GZV	17	All	2934	202,5	52,527	mem.	—
		Hydro.	613	8,0	2,132	mem.	—
		Fixed	53	0,1	0,015	0,140	0,155
	15	All	2247	115,0	30,135	mem.	—
		Hydro.	456	4,1	1,134	mem.	—
		Fixed	41	0,0	0,008	0,010	0,018
	12	All	1721	66,6	17,255	mem.	—
		Hydro.	352	2,3	0,602	mem.	—
		Fixed	35	0,0	0,010	0,000	0,010
	10	All	1371	40,1	10,824	mem.	—
		Hydro.	275	1,4	0,364	+3h.	—
		Fixed	28	0,0	0,005	0,000	0,005
2EGN	17	All	2548	149,5	38,045	mem.	—
		Hydro.	546	6,2	1,597	mem.	—
		Fixed	44	0,1	0,010	0,020	0,030
	15	All	2005	89,2	22,850	mem.	—
		Hydro.	413	3,2	0,829	+3h.	—
		Fixed	35	0,0	0,007	0,000	0,007
	12	All	1371	40,0	10,849	mem.	—
		Hydro.	295	1,6	0,403	+3h.	—
		Fixed	26	0,0	0,006	0,000	0,006
	10	All	1118	26,0	6,667	mem.	—
		Hydro.	238	1,1	0,249	+3h.	—
		Fixed	22	0,0	0,003	0,000	0,003

Table D.13: Double Optimization Results without DEE - 1MFG, 1BE9, 2GZV, 2EGN

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
2FNE	17	All	2736	178,6	47,006	mem.	—
		Hydro.	577	7,2	1,940	mem.	—
		Fixed	45	0,1	0,010	0,020	0,030
	15	All	2144	106,6	27,178	mem.	—
		Hydro.	438	3,8	1,035	mem.	—
		Fixed	39	0,0	0,012	0,010	0,022
	12	All	1529	53,6	13,729	mem.	—
		Hydro.	316	1,9	0,484	+3h.	—
		Fixed	29	0,0	0,006	0,000	0,006
	10	All	1257	33,3	8,793	mem.	—
		Hydro.	253	1,2	0,289	+3h.	—
		Fixed	24	0,0	0,002	0,000	0,002
1RZX	17	All	2722	173,3	45,181	mem.	—
		Hydro.	554	6,5	1,677	mem.	—
		Fixed	53	0,1	0,016	0,020	0,036
	15	All	2121	100,7	26,265	mem.	—
		Hydro.	404	3,1	0,801	mem.	—
		Fixed	45	0,1	0,012	0,000	0,012
	12	All	1452	45,2	12,139	mem.	—
		Hydro.	299	1,7	0,417	+3h.	—
		Fixed	28	0,0	0,005	0,000	0,005
	10	All	1306	37,2	9,555	mem.	—
		Hydro.	253	1,3	0,309	+3h.	—
		Fixed	23	0,0	0,002	0,000	0,002
1N7F	17	All	2548	150,3	38,246	mem.	—
		Hydro.	544	6,2	1,604	mem.	—
		Fixed	46	0,1	0,015	0,170	0,185
	15	All	1934	83,0	21,736	mem.	—
		Hydro.	402	3,0	0,802	+3h.	—
		Fixed	38	0,0	0,010	0,020	0,030
	12	All	1539	52,8	13,385	mem.	—
		Hydro.	315	1,9	0,484	+3h.	—
		Fixed	33	0,0	0,007	0,000	0,007
	10	All	1192	30,0	7,956	mem.	—
		Hydro.	246	1,1	0,273	+3h.	—
		Fixed	25	0,0	0,004	0,000	0,004
1QAU	17	All	2643	165,3	43,229	mem.	—
		Hydro.	562	6,8	1,782	mem.	—
		Fixed	47	0,1	0,014	0,160	0,174
	15	All	2023	92,7	24,098	mem.	—
		Hydro.	421	3,5	0,941	mem.	—
		Fixed	37	0,0	0,008	0,040	0,048
	12	All	1470	46,9	12,541	mem.	—
		Hydro.	309	1,8	0,483	+3h.	—
		Fixed	29	0,0	0,005	0,000	0,005
	10	All	1223	32,1	8,370	mem.	—
		Hydro.	251	1,2	0,292	+3h.	—
		Fixed	23	0,0	0,002	0,000	0,002

Table D.14: Double Optimization Results without DEE - 2FNE, 1RZX, 1N7F, 1QAU

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1TP3	17	All	2582	158,3	41,525	mem.	—
		Hydro.	536	6,2	1,606	mem.	—
		Fixed	47	0,1	0,013	0,030	0,043
	15	All	2142	106,2	26,938	mem.	—
		Hydro.	447	4,0	1,076	mem.	—
		Fixed	40	0,0	0,009	0,000	0,009
	12	All	1627	60,3	15,724	mem.	—
		Hydro.	351	2,4	0,638	+3h.	—
		Fixed	32	0,0	0,008	0,000	0,008
	10	All	1339	38,6	9,991	mem.	—
		Hydro.	279	1,5	0,357	+3h.	—
		Fixed	27	0,0	0,004	0,000	0,004
1I92	17	All	2648	164,5	42,089	mem.	—
		Hydro.	593	7,5	2,003	mem.	—
		Fixed	59	0,1	0,018	0,420	0,438
	15	All	2055	96,0	25,064	mem.	—
		Hydro.	455	4,0	1,074	mem.	—
		Fixed	53	0,1	0,017	0,190	0,207
	12	All	1524	51,5	12,781	mem.	—
		Hydro.	335	2,1	0,547	+3h.	—
		Fixed	40	0,0	0,009	0,010	0,019
	10	All	1313	36,5	9,511	mem.	—
		Hydro.	283	1,5	0,378	+3h.	—
		Fixed	36	0,0	0,009	0,000	0,009

Table D.15: Double Optimization Results without DEE - 1TP3, 1I92

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1MFG	17	All	2025	98,8	25,379	mem.	—
		Hydro.	213	0,9	0,225	+3h.	—
		Fixed	17	0,0	0,003	0,000	0,003
	15	All	1185	31,2	8,442	mem.	—
		Hydro.	89	0,2	0,039	620,220	620,259
		Fixed	15	0,0	0,004	0,000	0,004
	12	All	614	8,2	2,102	mem.	—
		Hydro.	41	0,0	0,007	0,200	0,207
		Fixed	12	0,0	0,003	0,000	0,003
	10	All	331	2,2	0,536	+3h.	—
		Hydro.	22	0,0	0,002	0,000	0,002
		Fixed	10	0,0	0,002	0,000	0,002
1BE9	17	All	1941	90,5	23,499	mem.	—
		Hydro.	155	0,5	0,113	+3h.	—
		Fixed	19	0,0	0,004	0,000	0,004
	15	All	1286	36,7	9,635	mem.	—
		Hydro.	62	0,1	0,018	1,760	1,778
		Fixed	15	0,0	0,001	0,000	0,001
	12	All	674	9,7	2,451	mem.	—
		Hydro.	19	0,0	0,002	0,000	0,002
		Fixed	12	0,0	0,002	0,000	0,002
	10	All	313	1,9	0,459	+3h.	—
		Hydro.	14	0,0	0,001	0,000	0,001
		Fixed	10	0,0	0,002	0,000	0,002
2GZV	17	All	2322	128,5	33,563	mem.	—
		Hydro.	241	1,2	0,287	+3h.	—
		Fixed	17	0,0	0,003	0,000	0,003
	15	All	1396	43,0	11,488	mem.	—
		Hydro.	101	0,2	0,046	+3h.	—
		Fixed	15	0,0	0,004	0,000	0,004
	12	All	804	13,9	3,580	mem.	—
		Hydro.	55	0,1	0,014	2,400	2,414
		Fixed	12	0,0	0,002	0,000	0,002
	10	All	572	7,0	1,730	mem.	—
		Hydro.	25	0,0	0,004	0,000	0,004
		Fixed	10	0,0	0,001	0,000	0,001
2EGN	17	All	1759	73,9	18,940	mem.	—
		Hydro.	164	0,5	0,124	+3h.	—
		Fixed	17	0,0	0,003	0,000	0,003
	15	All	1034	23,2	6,075	mem.	—
		Hydro.	92	0,2	0,041	279,140	279,181
		Fixed	15	0,0	0,002	0,000	0,002
	12	All	426	3,7	0,918	mem.	—
		Hydro.	53	0,1	0,012	0,180	0,192
		Fixed	12	0,0	0,003	0,000	0,003
	10	All	255	1,3	0,305	+3h.	—
		Hydro.	33	0,0	0,006	0,010	0,016
		Fixed	10	0,0	0,002	0,000	0,002

Table D.16: Double Optimization Results with DEE - 1MFG, 1BE9, 2GZV, 2EGN

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
2FNE	17	All	2150	110,8	28,360	mem.	—
		Hydro.	219	1,0	0,241	+3h.	—
		Fixed	19	0,0	0,003	0,000	0,003
	15	All	1386	42,8	11,429	mem.	—
		Hydro.	117	0,3	0,062	4815,980	4816,042
		Fixed	17	0,0	0,003	0,000	0,003
	12	All	670	9,8	2,504	mem.	—
		Hydro.	47	0,0	0,010	0,310	0,320
		Fixed	14	0,0	0,001	0,000	0,001
	10	All	344	2,4	0,577	+3h.	—
		Hydro.	25	0,0	0,003	0,000	0,003
		Fixed	12	0,0	0,002	0,000	0,002
1RZX	17	All	2009	95,7	24,802	mem.	—
		Hydro.	190	0,7	0,180	+3h.	—
		Fixed	22	0,0	0,005	0,000	0,005
	15	All	1190	30,4	8,192	mem.	—
		Hydro.	67	0,1	0,021	42,290	42,311
		Fixed	15	0,0	0,001	0,000	0,001
	12	All	596	7,5	1,852	mem.	—
		Hydro.	35	0,0	0,004	0,010	0,014
		Fixed	12	0,0	0,003	0,000	0,003
	10	All	547	6,4	1,543	mem.	—
		Hydro.	30	0,0	0,005	0,000	0,005
		Fixed	10	0,0	0,002	0,000	0,002
1N7F	17	All	1738	71,6	18,299	mem.	—
		Hydro.	195	0,7	0,184	+3h.	—
		Fixed	24	0,0	0,005	0,000	0,005
	15	All	960	19,7	5,405	mem.	—
		Hydro.	63	0,1	0,020	63,790	63,810
		Fixed	15	0,0	0,004	0,000	0,004
	12	All	609	8,0	2,061	mem.	—
		Hydro.	38	0,0	0,008	0,170	0,178
		Fixed	12	0,0	0,003	0,000	0,003
	10	All	369	2,8	0,696	+3h.	—
		Hydro.	27	0,0	0,005	0,000	0,005
		Fixed	10	0,0	0,001	0,000	0,001
1QAU	17	All	2018	97,2	25,171	mem.	—
		Hydro.	218	0,9	0,236	+3h.	—
		Fixed	22	0,0	0,004	0,000	0,004
	15	All	1162	29,4	7,926	mem.	—
		Hydro.	110	0,2	0,057	1943,740	1943,797
		Fixed	18	0,0	0,002	0,000	0,002
	12	All	567	6,9	1,691	mem.	—
		Hydro.	59	0,1	0,017	4,660	4,677
		Fixed	15	0,0	0,003	0,000	0,003
	10	All	304	1,8	0,441	+3h.	—
		Hydro.	18	0,0	0,003	0,000	0,003
		Fixed	13	0,0	0,002	0,000	0,002

Table D.17: Double Optimization Results with DEE - 2FNE, 1RZX, 1N7F, 1QAU

Protein	Pos.	Amino Acids	# Rotamers	Grounder size(MB)	Grounder time(s)	Solver time(s)	Total time(s)
1TP3	17	All	1923	88,6	23,009	mem.	—
		Hydro.	174	0,6	0,148	+3h.	—
		Fixed	17	0,0	0,003	0,000	0,003
	15	All	1329	39,3	10,811	mem.	—
		Hydro.	101	0,2	0,048	394,940	394,988
		Fixed	15	0,0	0,002	0,000	0,002
	12	All	761	12,5	3,162	mem.	—
		Hydro.	40	0,0	0,009	0,060	0,069
		Fixed	12	0,0	0,002	0,000	0,002
	10	All	489	5,0	1,219	mem.	—
		Hydro.	20	0,0	0,004	0,000	0,004
		Fixed	10	0,0	0,001	0,000	0,001
1I92	17	All	1967	92,8	24,064	mem.	—
		Hydro.	219	0,9	0,233	+3h.	—
		Fixed	17	0,0	0,002	0,000	0,002
	15	All	1080	25,1	6,645	mem.	—
		Hydro.	137	0,3	0,088	+3h.	—
		Fixed	15	0,0	0,002	0,000	0,002
	12	All	483	4,8	1,193	mem.	—
		Hydro.	69	0,1	0,022	3,390	3,412
		Fixed	12	0,0	0,002	0,000	0,002
	10	All	380	2,9	0,721	mem.	—
		Hydro.	58	0,1	0,015	0,450	0,465
		Fixed	10	0,0	0,001	0,000	0,001

Table D.18: Double Optimization Results with DEE - 1TP3, 1I92

Bibliography

[1] Marcello Balduccini and Tran Cao Son, editors. *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *Lecture Notes in Computer Science*. Springer, 2011.

[2] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*, pages 11–39. Cambridge University Press, New York, NY, USA, 2003.

[3] Helen M. Berman, John D. Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

[4] Gabriel Birrane, Judy Chung, and John A. A. Ladias. Novel Mode of Ligand Recognition by the Erbin PDZ Domain. *Journal of Biological Chemistry*, 278(3):1399–1402, January 2003.

[5] B. I. Dahiyat and S. L. Mayo. Protein design automation. *Protein Science*, 5(5):895–903, May 1996.

[6] Bassil I. Dahiyat and Stephen L. Mayo. De novo protein design: Fully automated sequence selection. *Science*, 278(5335):82–87, 1997.

[7] J. R. Desjarlais and T. M. Handel. De novo design of the hydrophobic cores of proteins. *Protein Science*, 4(10):2006–2018, October 1995.

[8] R. L. Dunbrack and F. E. Cohen. Bayesian statistical analysis of protein side-chain rotamer preferences. *Protein Science*, 6(8):1661–1681, August 1997.

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Challenges in answer set solving. In Balduccini and Son [1], pages 74–90.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-criteria optimization in answer set programming. In John P. Gallagher and Michael Gelfond, editors, *ICLP (Technical Communications)*, volume 11 of *LIPIcs*, pages 1–10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[11] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In James P. Delgrande and Wolfgang Faber, editors, *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 345–351. Springer, 2011.

[12] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.

[13] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *LPNMR'07*, pages 260–265. Springer, 2007.

[14] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[15] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, April 2006.

[16] R F Goldstein. Efficient rotamer elimination applied to protein side-chains and related spin glasses. *Biophysical Journal*, 66(5):1335–1340, 1994.

[17] D. Gordon, S. Marshall, and S. Mayo. Energy functions for protein design. *Current Opinion in Structural Biology*, 9(4):509–513, August 1999.

[18] D. Benjamin Gordon, Geoffrey K. Hom, Stephen L. Mayo, and Niles A. Pierce. Exact rotamer optimization for protein design. *Journal of Computational Chemistry*, 24(2):232–243, 2003.

[19] D.B. Gordon and S.L. Mayo. Branch-and-terminate: a combinatorial optimization algorithm for protein design. *Structure*, 7(9):1089–1098, 1999.

[20] B Z Harris and W A Lim. Mechanism and role of pdz domains in signaling complex assembly. *Journal of Cell Science*, 114(Pt 18):3219–3231, Sep 2001.

[21] David T. Jones. De novo protein design using pairwise potentials and a genetic algorithm. *Protein Science*, 3:567–574, 1994.

[22] Georgii G. Krivov, Maxim V. Shapovalov, and Roland L. Dunbrack. Improved prediction of protein side-chain conformations with SCWRL4. *Proteins: Structure, Function, and Bioinformatics*, 77(4):778–795, December 2009.

[23] Brian Kuhlman and David Baker. Native protein sequences are close to optimal for their structures. *Proceedings of the National Academy of Sciences*, 97(19):10383–10388, September 2000.

[24] T. Lazaridis and M. Karplus. Effective energy functions for protein structure prediction. *Current Opinion in Structural Biology*, 10(2):139–145, April 2000.

[25] Andrew Leaver-Fay, Brian Kuhlman, and Jack Snoeyink. Rotamer-pair energy calculations using a trie data structure. In Rita Casadio and Gene Myers, editors, *WABI*, volume 3692 of *Lecture Notes in Computer Science*, pages 389–400. Springer, 2005.

[26] Ho-Jin J. Lee and Jie J. Zheng. PDZ domains and their binding partners: structure, specificity, and modification. *Cell communication and signaling : CCS*, 8(1):8+, May 2010.

[27] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2002.

[28] Yuliya Lierler. Cmodels for tight disjunctive logic programs. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 163–166. Universität Ulm, Germany, 2005.

[29] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[30] Vladimir Lifschitz. What is answer set programming? In *AAAI*, pages 1594–1597. AAAI Press, 2008.

[31] Fangzhen Lin and Yuting Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[32] Yi Liu and Brian Kuhlman. Rosettadesign server for protein design. *Nucleic Acids Research*, 34:235–238, 2006.

[33] Ilya V. Loksha, James R. Maiolo III, Cheng W. Hong, Albert H. Ng, and Christopher D. Snow. Sharpen - systematic hierarchical algorithms for rotamers and proteins on an extended network. *Journal of Computational Chemistry*, 30(6):999–1005, 2009.

[34] Toni Mancini, Davide Micaletto, Fabio Patrizi, and Marco Cadoli. Evaluating asp and commercial solvers on the csplib. *Constraints*, 13(4):407–436, 2008.

[35] Ilkka Niemelä. Answer set programming: A declarative approach to solving search problems. In Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, editors, *JELIA*, volume 4160 of *Lecture Notes in Computer Science*, pages 15–18. Springer, 2006.

[36] Noah Ollikainen, Ellen Sentovich, Carlos Coelho, Andreas Kuehlmann, and Tanja Kortemme. Sat-based protein design. In *ICCAD*, pages 128–135. IEEE, 2009.

[37] Niles A. Pierce, Jan A. Spriet, J. Desmet, and Stephen L. Mayo. Conformational splitting: A more powerful criterion for dead-end elimination. *Journal of Computational Chemistry*, 21(11):999–1009, 2000.

[38] Niles A. Pierce and Erik Winfree. Protein design is NP-hard. *Protein Engineering*, 15(10):779–782, 2002.

[39] Carol A. Rohl, Charlie E. M. Strauss, Kira M. S. Misura, and David Baker. *Protein Structure Prediction Using Rosetta*, volume 383 of *Methods in Enzymology*, pages 66–93. Elsevier, Department of Biochemistry and Howard Hughes Medical Institute, University of Washington, Seattle, Washington 98195, USA., 2004.

[40] Maxim V. Shapovalov and Roland L. Dunbrack Jr. A smoothed backbone-dependent rotamer library for proteins derived from adaptive kernel density estimates and regressions. *Structure*, 19(6):844 – 858, 2011.

[41] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[42] Tommi Syrjänen. On the practical side of answer set programming. In Balduccini and Son [1], pages 473–489.

[43] Raffi Tonikian, Yingnan Zhang, Stephen L. Sazinsky, Bridget Currell, Jung-Hua Yeh, Boris Reva, Heike A. Held, Brent A. Appleton, Marie Evangelista, Yan Wu, Xiaofeng Xin, Andrew C. Chan, Somasekar Seshagiri, Laurence A. Lasky, Chris Sander, Charles Boone, Gary D. Bader, and Sachdev S. Sidhu. A Specificity Map for the PDZ Domain Family. *PLoS Biology*, 6(9):e239+, September 2008.

[44] Christopher A. Voigt, D.Benjamin Gordon, and Stephen L. Mayo. Trading accuracy for speed: a quantitative comparison of search algorithms in protein sequence design. *Journal of Molecular Biology*, 299(3):789 – 803, 2000.