

FINDING SIMILAR OR DIVERSE SOLUTIONS IN ANSWER SET
PROGRAMMING: THEORY AND APPLICATIONS

Halit Erdođan

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

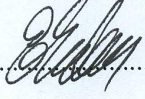
Sabancı University

August, 2011

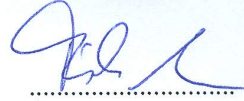
FINDING SIMILAR OR DIVERSE SOLUTIONS IN ANSWER SET
PROGRAMMING: THEORY AND APPLICATIONS

Approved by:

Assist. Prof. Dr. Esra Erdem
(Dissertation Supervisor)

.....


Dr. Michael Fink

.....



Assist. Prof. Dr. Volkan Patođlu

.....


Assoc. Prof. Dr. Uđur Sezerman

.....


Assoc. Prof. Dr. Berrin Yanıkođlu

.....


Date of Approval: 14/07/2011

© Halit Erdoğan 2011

All Rights Reserved

ÇÖZÜM KÜMESİ PROGRAMLAMA'DA BENZER YA DA FARKLI ÇÖZÜMLER BULMA: TEORİ VE UYGULAMALARI

Halit Erdoğan

Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, 2011

Tez Danışmanı: Esra Erdem

Özet

Birçok hesaplama probleminde ana amaç iyi tanımlanmış ölçütlere uygun en iyi çözümü (örneğin, en çok tercih edilen ürün yapısını, en kısa planı, en cimri filojeni) bulmaktır. Öte yandan, birçok gerçek uygulamada daha iyi karar verebilmek için bir küme birbirine benzer veya birbirinden farklı iyi çözümler hesaplamak istenebilir. Özellikle, üzerinde çalışılan problemin birçok iyi çözümü olabilir ve kullanıcılar birkaç çözümü inceleyerek birini seçmek isteyebilir; bu durumda, birbirine benzer veya birbirinden farklı iyi çözümler bulmak faydalı olur. Ayrıca, birçok uygulamada kullanıcılar optimizasyon probleminin formülasyonunda olmayan başka kriterleri de göz önünde bulundururlar; bu durumda, daha önceden belirlenmiş belirli bir çözüm kümesine yakın ya da uzak birkaç iyi çözüm bulmak faydalı olabilir.

Bu motivasyon ile bu tezde Çözüm Kümesi Programlama'da (ÇKP) benzer/farklı (yakın/uzak) çözümlerin hesaplanması ile alakalı çeşitli problemleri belirleyip, bu problemleri çözmek için çeşitli yeni hesaplama yöntemleri geliştirdik. Bu yöntemlerden bir tanesinde ÇKP çözümlerinden birinin algoritmasını değiştirerek, birçok ÇKP uygulaması için kullanışlı olabilecek yeni bir ÇKP çözümleri (CLASP-NK) geliştirdik. Bu yöntemlerin uygulanabilirliğini ve etkinliğini filojeni çıkarımı, planlama ve biyomedikal sorgu cevaplama alanlarında gösterdik. Elde ettiğimiz ümit verici deneysel sonuçlar neticesinde, bu alanlardaki uzmanlar tarafından kullanılacak yazılımlar geliştirdik.

FINDING SIMILAR OR DIVERSE SOLUTIONS IN ANSWER SET PROGRAMMING: THEORY AND APPLICATIONS

Halit Erdoğan

Computer Science and Engineering, Master's Thesis, 2011

Thesis Supervisor: Esra Erdem

Abstract

For many computational problems, the main concern is to find a best solution (e.g., a most preferred product configuration, a shortest plan, a most parsimonious phylogeny) with respect to some well-described criteria. On the other hand, in many real-world applications, computing a subset of good solutions that are similar/diverse may be desirable for better decision-making. For one reason, the given computational problem may have too many good solutions, and the user may want to examine only a few of them to pick one; in such cases, finding a few similar/diverse good solutions may be useful. Also, in many real-world applications the users usually take into account further criteria that are not included in the formulation of the optimization problem; in such cases, finding a few good solutions that are close to or distant from a particular set of solutions may be useful.

With this motivation, we have studied various computational problems related to finding similar/diverse (resp. close/distant) solutions with respect to a given distance function, in the context of Answer Set Programming (ASP). We have introduced novel offline/online computational methods in ASP to solve such computational problems. We have modified an ASP solver according to one of our online methods, providing a useful tool (CLASP-NK) for various ASP applications. We have showed the applicability and effectiveness of our methods/tools in three domains: phylogeny reconstruction, AI planning, and biomedical query answering. Motivated by the promising results, we have developed computational tools to be used by the experts in these areas.

Acknowledgements

I wish to express my gratitude to

- Esra Erdem for her invaluable supervision,
- my thesis committee for their reviews and suggestions,
- The Scientific and Technological Research Council of Turkey (TUBITAK) for the BIDEB scholarship that provided me the necessary financial support throughout my master's studies,
- all my friends from Sabancı University for their motivation and endless friendship,
- last, but not the least, my family for their unconditional love, support and persistent confidence in me.

Parts of this thesis are supported by TUBITAK Grants 107E229 and 108E229.

Contents

1	Introduction	1
2	Answer Set Programming	5
2.1	Programs	5
2.2	Representing a Problem in ASP	7
2.3	Example: Representing the c -Clique Problem in ASP	9
2.4	Finding a Solution using an Answer Set Solver	9
2.5	Applications of ASP	11
2.6	CLASP	12
3	Finding Similar/Diverse Solutions in ASP	15
3.1	Computational Problems	15
3.2	Computing n k -Similar/Diverse Solutions	16
3.2.1	Offline Method	16
3.2.2	Online Method 1: Reformulation	17
3.2.3	Online Method 2: Iterative Computation	19
3.2.4	Online Method 3: Incremental Computation	20
3.3	Computing k -Close/Distant Solution	21
3.4	Computing Similar/Diverse Weighted Solutions	21
4	Finding Similar/Diverse Phylogenies	25
4.1	Phylogeny Reconstruction Problem	26
4.2	Distance Measures for Phylogenies	27
4.2.1	Nodal Distance of Two Phylogenies	28
4.2.2	Descendant Distance of Two Phylogenies	30
4.2.3	Distance of a Set of Phylogenies	31
4.3	Computing n k -Similar/Diverse Phylogenies	32
4.4	Experimental Results	35
4.5	Computational Tools	37
4.5.1	PHYLOCOMPARE-ASP	37
4.5.2	PHYLORECONSTRUCTN-ASP	38

5	Finding Similar/Diverse Plans	42
5.1	Problem Description	42
5.2	Computing Similar/Diverse Plans	44
5.3	Experimental Results	48
6	Finding Similar/Diverse Genes	51
6.1	BIOQUERY-ASP	52
6.2	Computing Similar/Diverse Genes	54
6.3	Experimental Results	55
7	Related Work	57
8	Conclusion	59
A	ASP Formulations	62

List of Figures

2.1	Representation of the c -clique problem in ASP.	10
2.2	Representation of a sample undirected graph.	11
3.1	Offline Method for computing n k -similar solutions.	17
3.2	Online Methods for computing n k -similar solutions.	17
3.3	Computing n k -similar solutions, with Online Method 1.	18
3.4	ASP formulation that computes n distinct c -cliques.	19
3.5	ASP formulation of the Hamming distance between two cliques.	19
3.6	A constraint that forces the distance among any two solutions is less than or equal to k	19
3.7	Computing n k -similar solutions, with Online Method 2. Initially $S = \emptyset$. In each run, a solution is computed and added to S , until $ S = n$. The distance function and the constraints in the program ensure that when we add the computed solution to S , the set stays k -similar.	20
3.8	Computing n k -similar solutions, with Online Method 3. CLASP-NK is a modification of the ASP solver CLASP, that takes into account the distance function and constraints while computing an answer set in such a way that CLASP-NK becomes biased to compute similar solutions. Each computed solution is stored by CLASP-NK until a set of n k -similar solutions is computed.	21
4.1	A phylogeny for the species a, b, c, d.	28
4.2	Two phylogenies $P_1 = (a, (b, c))$ and $P_2 = (b, (a, c))$	29
4.3	A screen shot of PHYLOCOMPARE-ASP where the user enters four phylogenies in newick format.	39
4.4	PHYLOCOMPARE-ASP computes a set of 3 phylogenies with the minimum total distance among the given phylogenies shown in Figure 4.3.	40
5.1	Blocks World problem.	48
6.1	System overview of BIOQUERY-ASP.	52

6.2	A screenshot of BIOQUERY-ASP. Users construct queries with the help of the intelligent user interface.	54
A.1	A reformulation of the phylogeny reconstruction program of Brooks et al., to find n distinct phylogenies: Part 1	62
A.2	A reformulation of the phylogeny reconstruction program of Brooks et al., to find n distinct phylogenies: Part 2	63
A.3	A formulation of the nodal distance function D_n in ASP.	64
A.4	An ASP formulation of the descendant distance function D_l for two phylogenies.	65
A.5	An ASP formulation of the distance function Δ_D for a set of phylogenies, and the constraints for k -similarity.	66
A.6	Blocks World Formulation.	66
A.7	A reformulation of the Blocks World program shown in Fig. A.6, to compute n distinct plans.	67
A.8	An ASP formulation of the Hamming distance D_h for two plans.	68
A.9	An ASP formulation of the distance Δ_h for a set of plans and the constraint for k -similarity.	68

List of Tables

2.1	Applications of ASP.	12
2.2	ASP solvers.	13
4.1	In order to compute the nodal distance $D_n(P_1, P_2)$ between the phylogenies $P_1 = (a, (b, c))$ and $P_2 = (b, (a, c))$ shown in Figure 4.2, we compute the nodal distances of the pairs of leaves, $\{a, b\}$, $\{a, c\}$ and $\{b, c\}$, and take the sum of the differences. In this case the distance between P_1 and P_2 is 2.	29
4.2	In order to compute the descendant distance $D_l(P_1, P_2)$ between the phylogenies $P_1 = (a, (b, c))$ and $P_2 = (b, (a, c))$ shown in Figure 4.2, for each depth level, we multiply the number of vertices that have different descendants with the weight of that depth level. Then, we add up the products to find the total distance between P_1 and P_2 . The descendant distance between P_1 and P_2 is 4.	31
4.3	Computing similar/diverse phylogenies using the nodal distance Δ_n	36
4.4	Computing similar/diverse phylogenies using the descendant distance Δ_l	37
5.1	Computing similar/diverse plans for the blocks world problem. OM denotes “Out of memory.”	50
6.1	The retrieved relations among biomedical concepts.	53
6.2	Experimental results for answering queries Q1 and Q2.	56

Chapter 1

Introduction

For many computational problems, the main concern is to find a best solution (e.g., a most preferred product configuration, a shortest plan, a most parsimonious phylogeny) with respect to some well-described criteria. On the other hand, in many real-world applications, computing a subset of good solutions that are similar/diverse may be desirable for better decision-making. For one reason, the given computational problem may have too many good solutions, and the user may want to examine only a few of them to pick one; in such cases, finding a few similar/diverse good solutions may be useful. Also, in many real-world applications the users usually take into account further criteria that are not included in the formulation of the optimization problem; in such cases, finding a few good solutions that are close to or distant from a particular set of solutions may be useful. Here are some examples from several domains in which computing a subset of similar/diverse solutions could be useful. Consider, for instance, the problem of generating grid puzzles as in [104]. The authors introduce methods to generate puzzles with different difficulty levels automatically. For each difficulty level, it is desirable to generate many puzzles that are as diverse as possible, since users prefer to solve very different puzzles even if they have the same difficulty. As another example, consider a variation of the scenario in [57] about product advisor systems where we want to develop a system which recommends users products (e.g., cars) based on their preferences and constraints. Suppose that there are many products each of which suits a user's preferences. In such a case, instead of recommending all those products to the user, it is desirable to suggest a set of few products that are as diverse as possible. If the user likes one particular product, then the system may recommend a set of similar products to the selected one.

Motivated by such examples, we have studied various computational problems related to computing similar/diverse solutions in the context of Answer Set Programming (ASP) [74]. We have introduced general offline/online methods in ASP to find similar/diverse solutions. Then, we have applied these methods to specific domains such as phylogeny reconstruction, planning, and query answering.

In ASP, a combinatorial search problem is represented as a “program” whose models (called “answer sets”) correspond to the solutions. The answer sets for the given program can be computed by special systems called answer set solvers, such as SMOBELS [83], DLV [70], CMOBELS [54] and CLASP [49]. Due to the expressive formalism of ASP that allows us to represent, e.g., negation, defaults, aggregates, recursive definitions, and due to the continuous improvements of the efficiency of the solvers, ASP has been used in a wide-range of knowledge-intensive applications from different fields. For many of these applications, finding similar/diverse solutions (and thus the methods we have developed for computing similar/diverse solutions in ASP) could be useful.

The main contributions of this thesis can be summarized as follows.

- We have described mainly two kinds of computational problems, namely n k -SIMILAR SOLUTIONS (resp. n k -DIVERSE SOLUTIONS) and k -CLOSE SOLUTION (resp. k -DISTANT SOLUTION), related to finding similar/diverse solutions of a given problem, in the context of ASP. Both kinds of problems take as input an ASP program \mathcal{P} that describes a problem, a distance measure Δ that maps a set of solutions of the problem to a nonnegative integer, and two nonnegative integers n and k .
 - n k -SIMILAR SOLUTIONS (resp. n k -DIVERSE SOLUTIONS) asks for a set S of size n that contains k -similar (resp. k -diverse) solutions, i.e., $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$).
 - k -CLOSE SOLUTION (resp. k -DISTANT SOLUTION) asks, given a set S of n solutions, for a k -close (resp. k -distant) solution s ($s \notin S$), i.e., $\Delta(S \cup \{s\}) \leq k$ (resp. $\Delta(S \cup \{s\}) \geq k$).
- We have introduced four methods to compute a set of n k -similar (resp. k -diverse) solutions to a given problem.
 - Offline Method computes all solutions in advance using ASP and then finds similar (resp. diverse) solutions using some clustering methods, possibly in ASP as well.
 - Online Method 1 reformulates the given program to compute n -distinct solutions and formulates the distance function as an ASP program, so that all n k -similar (resp. k -diverse) solutions can be extracted from an answer set for the union of these ASP programs.
 - Online Method 2 does not modify the ASP encoding of the problem, but formulates the distance function as an ASP program, so that a unique k -close (resp. k -distant) solution can be extracted from an answer set for the union of

these ASP programs and previously computed solutions; by iteratively computing k -close (resp. k -distant) solutions one after other, we can compute online a set of n k -similar (or k -diverse) solutions.

- Online Method 3 does not modify the ASP encoding of the problem, and does not formulate the distance function as an ASP program, but it modifies the search algorithm of an ASP solver, in our case CLASP [49], to compute all n k -similar (or k -diverse) solutions incrementally at once. The distance function is implemented in C++; in that sense, Online Method 3 allows for finding similar/diverse solutions when the distance function cannot be defined in ASP. Since the solutions are computed incrementally by a branch-and-bound like algorithm, Online Method 3 requires a heuristic function to estimate the distance function.
- We have illustrated the applicability of these approaches on three sorts of problems: phylogeny reconstruction, planning, and biomedical query answering.

- For phylogeny reconstruction, we have defined novel distance measures for a set of phylogenies, described how the offline method and the online methods are applied to find similar/diverse phylogenies, and compare the efficiency and effectiveness of these methods on the family of Indo-European languages studied in [12].

Since there is no phylogenetic system that helps experts analyze phylogenies by comparing them, this particular application of our methods also plays a significant role in phylogenetics. Therefore, we have developed two tools:

- * PHYLOCOMPARE-ASP helps users analyze the given phylogenies by computing their distance matrix and by grouping them with respect to their similarity/diversity.
- * PHYLORECONSTRUCTN-ASP computes similar/diverse set of phylogenies from a given matrix about the shared traits of the species.

Both these two tools are integrated into the phylogenetics system PHYLO-ASP [36].

- For planning, we have considered the action-based Hamming distance of [99] to measure the distance among plans, and compare the efficiency and effectiveness of the offline method and the online methods on some Blocks World problems.
- For answering queries about similar/diverse genes, we have considered the distance measure for genes introduced in [108]. Since there is no such system that can answer complex queries related to similar/diverse genes, we have

integrated our method into the query answering system BIOQUERY-ASP [38]. This system is useful for crucial research such as drug discovery.

In each application above, we have analyzed the complexity of computing the distance function. Also, to estimate the distance functions, we have introduced novel heuristic functions and proved their admissibility.

Outline of the rest of the thesis is as follows. In Chapter 2, we give preliminaries for answer set programming along with a summary of ASP applications and ASP solvers. We describe the computational problems and offline/online methods to solve these problems in Chapter 3. Then, in Chapter 4, we show the applicability of our methods on similar/diverse phylogeny reconstruction problem, along with the description of the software systems PHYLOCOMPARE-ASP and PHYLORECONSTRUCTN-ASP. In Chapter 5, we show the applicability of our approaches to the planning problems and compared the efficiency of the methods on a Blocks World domain. In Chapter 6, we describe the applicability of Online Method 3 to compute similar/diverse genes, as a part of BIOQUERY-ASP. After that, we summarize related work in Chapter 7 and conclude the thesis in Chapter 8 by providing a summary of our contributions and their significance and by discussing possible future research directions.

Chapter 2

Answer Set Programming

Answer Set Programming [74, 4] is a declarative programming paradigm oriented towards, primarily NP-Hard, knowledge-intensive search problems. The idea is to represent a problem as a “program” whose models (called “answer sets” [52]) correspond to the solutions. The answer sets for the given program can be computed by special systems called answer set solvers. ASP is similar to SAT solving [7] in the sense that both paradigms are for solving problems declaratively using propositional formulas, but ASP has a more expressive input language and different semantics. In particular ASP allows recursive definitions such as transitive closure and nonmonotonic negation. In addition, a range of special constructs, such as aggregates and weight constraints are supported by various ASP solvers. Due to the continuous improvement of the ASP solvers and expressive representation language, ASP has been applied to a wide range of areas.

In the following, we explain the syntax and semantics of ASP programs. Then we briefly overview, by providing specific examples, how computational problems can be represented as an ASP program and solved using ASP solvers. After that, we give a comprehensive list of applications that use ASP. Then we explain the answer set solver CLASP and its algorithm to find answer sets.

2.1 Programs

Syntax ASP programs are composed of three sets namely *constant symbols*, *predicate symbols*, and *variable symbols* where intersection of constant symbols and variable symbols is empty. The basic elements of the ASP programs are *atoms*. An atom $p(\vec{t})$ is composed of a predicate symbol $p \in \mathcal{P}$ and *terms* $\vec{t} = t_1, \dots, t_k$ where each t_i ($1 \leq i \leq k$) is either a constant or a variable. A *literal* is either an atom $p(\vec{t})$ or its negated form $\text{not } p(\vec{t})$.

An ASP program is composed of a finite set of *rules* of the form:

$$A \leftarrow A_1, \dots, A_k, \text{not } A_{k+1}, \dots, \text{not } A_m \quad (2.1)$$

where $m \geq k \geq 0$ and each A_i is an atom; whereas, A is an atom or \perp .

For a rule r of the form (2.1), A is called the *head* of the rule and denoted by $H(r)$. The conjunction of the literals $A_1, \dots, A_k, \text{not } A_{k+1}, \dots, \text{not } A_m$ is called the *body* of r . The set $\{A_1, \dots, A_k\}$ of atoms (called the positive part of the body) is denoted by $B^+(r)$, and the set $\{A_{k+1}, \dots, A_m\}$ of atoms (called the negative part of the body) is denoted by $B^-(r)$, and all the atoms in the body are denoted by $B(r) = B^+(r) \cup B^-(r)$.

We say that a rule r is a *fact* if $B(r) = \emptyset$, and we usually omit the \leftarrow sign; furthermore, we say that a rule r is a *constraint* if the head of r is \perp , and we usually omit the \perp sign.

Semantics (Answer Sets) Answer sets of a program are defined over *ground programs*. We call an atom, rule, or program *ground*, if it does not contain any variables. The set \mathcal{U}_Π represents all the constants in Π , and the set \mathcal{B}_Π represents all the ground atoms constructible from atoms in Π with constants in \mathcal{U}_Π . Given a program Π , $Ground(\Pi)$ denotes the set of all the ground rules which are obtained by substituting each variable in the rule with the set of all possible constants in \mathcal{U}_Π .

Given a program Π , a subset I of \mathcal{B}_Π is called an *interpretation* for Π . A ground atom p is true with respect to an interpretation I if $p \in I$; otherwise, it is false; similarly, a set S of atoms is true (resp. false) with respect to I if each atom $p \in S$ is true (resp. false) with respect to I . An interpretation I *satisfies* a ground rule r , if $B^+(r)$ is true and $B^-(r)$ is false whenever $H(r)$ is true with respect to I . An interpretation I is called a *model* of a program Π if it satisfies all the rules in Π .

The *reduct* Π^I of a program Π with respect to an interpretation is defined as follows:

$$\Pi^I = \{H(r) \leftarrow B^+(r) \mid r \in Ground(\Pi) \text{ s.t. } I \cap B^-(r) = \emptyset\}$$

An interpretation I is an *answer set* for a program Π , if it is a subset-minimal model for Π^I , and $AS(\Pi)$ denotes the set of all the answer sets of a program Π .

For example, consider the following program Π_1 :

$$p \leftarrow \text{not } q \tag{2.2}$$

and take an interpretation $I = \{p\}$. The reduct Π_1^I is as follows:

$$p \tag{2.3}$$

I is a model of the reduct (2.3). Let's take a strict subset I' of I which is \emptyset . Then reduct $\Pi_1^{I'}$ is again equal to (2.3); however, I' does not satisfy (2.3); therefore, $I = \{p\}$ is a subset-minimal model; hence an answer set of Π_1 . Note also that $\{p\}$ is the only answer set of Π .

The *not* in the ASP programs is called negation as failure and is different from classical negation in SAT in terms of its nonmonotonicity. Let the *conclusion* of a program be

the intersection of its all answer sets. In order to understand the nonmonotonicity of ASP programs, we need to observe the changes in the conclusion of programs when we extend them.

Consider the following program Π_2 :

$$\begin{aligned} p &\leftarrow \text{not } q \\ q &\leftarrow \text{not } p \end{aligned} \tag{2.4}$$

Note that Π_2 has one extra rule compared to Π_1 and has two answer sets $\{p\}$ and $\{q\}$. Adding a rule to the program Π_1 decreases the size of its conclusion from $\{p\}$ to \emptyset . Now, consider that we add a constraint to Π_2 and obtain the following program Π_3 :

$$\begin{aligned} p &\leftarrow \text{not } q \\ q &\leftarrow \text{not } p \\ &\leftarrow p \end{aligned} \tag{2.5}$$

Π_3 has a single answer set $\{q\}$. Note that the size of the conclusion of Π_2 increases from \emptyset to $\{q\}$ when we add the new constraint. We can observe that when we extend an ASP program by adding new rules, the change in the size of its conclusion is neither monotonic nor anti-monotonic. This is why the semantics of ASP is considered to be nonmonotonic unlike SAT.

2.2 Representing a Problem in ASP

The idea of ASP [74] is to represent a computational problem as a program whose answer sets correspond to the solutions of the problem, and to find the answer sets for that program using an answer set solver.

When we represent a problem in ASP, two kinds of rules play an important role: those that “generate” many answer sets corresponding to “possible solutions”, and those that can be used to “eliminate” the answer sets that do not correspond to solutions. Rules (2.4) are of the former kind: they generate the answer sets $\{p\}$ and $\{q\}$. Constraints are of the latter kind. For instance, adding the constraint

$$\leftarrow p$$

to program (2.4) as in (2.5) eliminates the answer sets for the program that contains p .

In ASP, we use special constructs of the form

$$\{A_1, \dots, A_n\}^c \tag{2.6}$$

(called *choice expressions*), and of the form

$$l \leq \{A_1, \dots, A_m\} \leq u \quad (2.7)$$

(called *cardinality expressions*) where each A_i is an atom and l and u are nonnegative integers denoting the “lower bound” and the “upper bound” [94]. Programs using these constructs can be viewed as abbreviations for normal nested programs defined in [43]. For instance, the following program

$$1 \leq \{p, q\}^c \leq 1 \leftarrow$$

stands for program (2.4). The constraint

$$\leftarrow 2 \leq \{p, q, r\}$$

stands for the constraints

$$\begin{aligned} &\leftarrow p, q \\ &\leftarrow p, r \\ &\leftarrow q, r. \end{aligned}$$

Expression (2.6) describes subsets of $\{A_1, \dots, A_n\}$. Such expressions can be used in heads of rules to generate many answer sets. For instance, the answer sets for the program

$$\{p, q, r\}^c \leftarrow \quad (2.8)$$

are arbitrary subsets of $\{p, q, r\}$. Expression (2.7) describes the subsets of the set $\{A_1, \dots, A_m\}$ whose cardinalities are at least l and at most u . Such expressions can be used in constraints to eliminate some answer sets.

For instance, adding the constraint

$$\leftarrow 2 \leq \{p, q, r\}$$

to program (2.8) eliminates the answer sets for (2.8) whose cardinalities are at least 2. Adding the constraint

$$\leftarrow \text{not } (1 \leq \{p, q, r\}) \quad (2.9)$$

to program (2.8) eliminates the answer sets for (2.8) whose cardinalities are not at least 1.

We abbreviate the rules

$$\begin{aligned} &\{A_1, \dots, A_m\}^c \leftarrow \text{Body} \\ &\leftarrow \text{not } (l \leq \{A_1, \dots, A_m\}) \\ &\leftarrow \text{not } (\{A_1, \dots, A_m\} \leq u) \end{aligned}$$

by

$$l \leq \{A_1, \dots, A_m\}^c \leq u \leftarrow \text{Body}.$$

For instance, rules (2.8), (2.9) and $\leftarrow \text{not } (\{p, q, r\} \leq 1)$ can be written as

$$1 \leq \{p, q, r\}^c \leq 1 \leftarrow$$

whose answer sets are the singleton subsets of $\{p, q, r\}$.

2.3 Example: Representing the c -Clique Problem in ASP

A *clique* in an undirected graph is a set of vertices that are pairwise adjacent. Given an undirected graph the c -clique problem is to decide whether a clique of size c exists. Consider, for instance, the use of the generate-and-test representation methodology above to represent the c -clique problem in ASP. Consider that we want to find a clique of size c . A solution can be described by a set of atoms of the form $\text{clique}(i)$; including $\text{clique}(i)$ in the set indicates that the i^{th} vertex is in a clique of size c .

The “generate” part of our program will be:

$$c \leq \{\text{clique}(v_1), \text{clique}(v_2), \dots, \text{clique}(v_{|V|})\}^c \leq c \quad (v_i \in V, 1 \leq i \leq |V|) \quad (2.10)$$

(exactly c vertex for a clique). The “test” part consists of the constraints expressing that each member of a clique will be adjacent:

$$\leftarrow \text{clique}(v), \text{clique}(v'), \text{not } \text{edge}(v, v') \quad (v \neq v') \quad (2.11)$$

Every answer set of the program consisting of the rules (2.10) \cup (2.11) describes a clique of size c in a given graph.

2.4 Finding a Solution using an Answer Set Solver

Once we represent a computational problem as a program whose answer sets correspond to the solutions of the problem, we can use an answer set solver to compute the solutions of the problem. To present a program to an answer set solver, like CLASP, we need to make some syntactic modifications.

The syntax of the input language of CLASP is more limited in some ways than the class of programs defined above, but it includes many useful special cases. For instance,

```

% Generate a candidate set of c vertices
c{clique(V) : vertex(V)}c.
% Ensure that the candidate set corresponds to a clique
:- clique(V1), clique(V2), not edge(V1,V2), V1 != V2.

```

Figure 2.1: Representation of the c -clique problem in ASP.

the head of a rule can be an expression of one of the forms

$$\begin{aligned}
& \{A_1, \dots, A_n\}^c \\
& l \leq \{A_1, \dots, A_n\}^c \\
& \{A_1, \dots, A_n\}^c \leq u \\
& l \leq \{A_1, \dots, A_n\}^c \leq u
\end{aligned}$$

but the superscript c and the sign \leq are dropped. The body can contain cardinality expressions but the sign \leq is dropped.

In the input language of CLASP, $:-$ stands for \leftarrow , and each rule is followed by a period.

Variables in a program are represented by strings whose initial letter is capitalized. The constants and predicate symbols, on the other hand, start with a lowercase letter. For instance, the program Π_n

$$p_i \leftarrow \text{not } p_{i+1} \quad (1 \leq i \leq n)$$

can be presented to CLASP as follows:

```

index(1..n).
p(I) :- not p(I+1), index(I).

```

Here `index` is a “domain predicate” used to describe the range of variable I .

Variables can be also used “locally” to describe the list of formulas in a cardinality expression. For instance, the rule

$$1 \leq \{p_1, \dots, p_n\} \leq 1$$

can be expressed in CLASP as follows

```

index(1..n).
1{p(I) : index(I)}1.

```

For instance, the program consisting of the rules (2.10) \cup (2.11) describing the c -clique problem can be presented to CLASP as in Figure 2.1.

The expression `{clique(V) : vertex(V)}` is an abbreviation for `{clique(v_1), clique(v_2), ...}` for each vertex $v_i \in V$. To use this program, we can combine it with

```

vertex(1..4) .
edge(1,2) .
edge(2,3) .
edge(3,1) .
edge(X,Y) :- edge(Y,X) .

```

Figure 2.2: Representation of a sample undirected graph.

a description of a graph as shown in Figure 2.2. The first rule indicates that the input graph has four vertices. Subsequent rules represent the edges in the graph, and the last rule ensures the symmetricity of the edges (i.e., the graph is undirected). CLASP finds the following answer set where $c = 3$ for the union of these programs:

$$\{vertex(1), vertex(2), vertex(3), vertex(4), \\ edge(1,2), edge(2,1), edge(2,3), edge(3,2), edge(3,1), edge(1,3), \\ clique(1), clique(2), clique(3)\}$$

The `vertex` and `edge` atoms correspond to the given graph and the `clique` atoms correspond to a clique in the graph. We can understand from this answer set that the set $\{1, 2, 3\}$ of vertices corresponds to a clique of size three in the given graph.

2.5 Applications of ASP

Due to the continuous improvements in efficiency of answer set solvers and its expressive representation language, ASP has been applied to a wide range of areas in science. Here are some examples:

- *Decision Support Systems:* An ASP-based system was developed to solve planning and diagnostic tasks related to the operation of the space shuttle [84].
- *Automated Product Configuration:* A web-based commercial system¹ uses the ASP-based product configurator technology [102].
- *Semantic Web:* ASP-based semantic web applications provide advanced reasoning which require declarative methods to describe user preferences [17, 34, 101]. With the growing interest in the semantic web applications, there is a continuous improvement in the ASP tools for the semantic web.

Table 2.1 contains references for ASP applications in other fields.

¹<http://www.variantum.com/en/>

Table 2.1: Applications of ASP.

Area	Refernces
planning	[25] [72] [97]
theory update/revision	[64]
preferences	[92] [11]
diagnosis	[32] [3] [29]
learning	[90]
description logics and semantic web	[17] [34] [101]
probabilistic reasoning	[5]
data integration and question answering	[1] [69]
multi-agent systems	[97] [98] [106]
wire routing	[40] [27]
decision support systems	[84]
bounded model checking	[59]
game theory	[78] [107]
logic puzzles	[44]
phylogenetics	[30] [15] [39] [36]
systems biology	[103]
combinatorial auctions	[6]
haplotype inference	[37] [105]
systems biology	[103] [45] [91] [51]
automatic music composition	[10] [9]
verification of cryptographic protocols	[24]
assisted living	[80] [81]
context	[29]

2.6 CLASP

Since ASP is applied to many areas of science successfully, there is a growing interest in developing and optimizing answer set solvers. There exists several ASP solvers which have been developed and maintained by different universities. Table 2.2 lists some of the available ASP solvers.

In our experiments and systems, we used the ASP solver CLASP since it is open-source and the winner of the ASP Competitions 2009 and 2010. In the following, we describe the answer set solver CLASP and its algorithm for computing answer sets.

CLASP is a conflict-driven answer set solver [47, 49, 48]. CLASP finds an answer set for a program in two stages: first it gets rid of the schematic variables using a “grounder”, like GRINGO², and then it finds an answer set for the ground program using a DPLL-like [23] branch-and-bound algorithm (outlined in Algorithm 1). CLASP goes through three main steps to find an answer set. In the PROPAGATION step, it decides the literals that

²<http://potassco.sourceforge.net/>

Table 2.2: ASP solvers.

Name	Year	University	Reference
SMODELS	1996	Helsinki University of Technology	[83]
DLV	1997	Vienna Technical University	[70]
CMODELS	2002	University of Texas-Austin	[54]
ASSAT	2003	Hong Kong University of Science and Technology	[76]
PBMODELS	2005	University of Kentucky	[77]
CLASP	2006	University of Potsdam	[47]

have to be included in the answer set due to the current assignment and conflicts. In the RESOLVE-CONFLICT step, it tries to resolve the conflicts encountered in the previous step. If there is a conflict, then CLASP learns it and does backtracking to an appropriate level. Learning a conflict helps CLASP prevent redundant search. If there is no conflict and the currently selected literals do not represent an answer set, then, in SELECT, CLASP selects a new literal based on several heuristics to continue search.

Algorithm 1 CLASP

Input: An ASP program Π

Output: An answer set A for Π

$A \leftarrow \emptyset$ // current assignment of literals

$\nabla \leftarrow \emptyset$ // set of conflicts

while No Answer Set Found **do**

 PROPAGATION(Π, A, ∇) // propagate literals

if There is a conflict in the current assignment **then**

 RESOLVE-CONFLICT(Π, A, ∇) // learn and update conflicts, and backtrack

else

if Current assignment does not yield an answer set **then**

 SELECT(Π, A, ∇) // select a literal to continue search

else

return A

end if

end if

end while

CLASP's algorithm differs from DPLL in some aspects. First, DPLL is designed to solve SAT problems whereas CLASP is for ASP programs and solutions to SAT may not correspond to the answer sets of the problems [76]. Consider for instance the following program:

$$p \leftarrow q \tag{2.12}$$

The answer set of this program is \emptyset . This program can be translated into the following

SAT program:

$$\neg p \vee q \tag{2.13}$$

Models of this SAT problem are \emptyset and $\{p\}$. As can be seen from this example, there is no one-to-one correspondence between SAT models and answer sets. However, there is a close relation between these two paradigms. CLASP exploits this relationship by using loop formulas [76] and Clark completion [20] to solve ASP programs with local compilations to SAT formulas; then uses DPLL search over these local inferences. Second, CLASP enhances the DPLL search with concepts from constraint processing such as Nogoods [89] and other heuristics from SAT such as literal watching [82].

Chapter 3

Finding Similar/Diverse Solutions in ASP

For many computational problems, the main concern is to find a best solution (e.g., a most preferred product configuration, a shortest plan, a most parsimonious phylogeny) with respect to some well-described criteria. On the other hand, in many real-world applications, there are multiple solutions to a given problem. In such cases, one may be interested in computing a solution, some of the solutions, or all the solutions to the given problem. When the solution space is large, computing only one solution might not be desirable. On the other hand, computing all the solutions might be intractable because of the large number of solutions. Therefore, users may be interested in computing a set of few “informative” solutions to work on. With this motivation, we are interested computing

- a set of similar/diverse solutions, and
- a solution that is close/distant to a given set of solutions.

In the following, we introduce the main computational problems related to computing similar/diverse solutions in ASP and offline/online methods to solve these problems.

3.1 Computational Problems

We are mainly interested in the following problems related to computation of a similar/diverse collection of solutions:

n k-SIMILAR SOLUTIONS (resp. *n k*-DIVERSE SOLUTIONS)

Given an ASP program \mathcal{P} that formulates a computational problem P , a distance measure Δ that maps a set of solutions for P to a nonnegative integer, and two nonnegative integers n and k , find a set S of n solutions for P such that $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$).

k-CLOSE SOLUTION (resp. *k*-DISTANT SOLUTION)

Given an ASP program \mathcal{P} that formulates a computational problem P , a distance

measure Δ that maps a set of solutions for P to a nonnegative integer, a set S of solutions for P , and a nonnegative integer k , find a solution s ($s \notin S$) for P such that $\Delta(S \cup \{s\}) \leq k$ (resp. $\Delta(S \cup \{s\}) \geq k$).

For instance, consider the ASP program $\mathcal{P} = (2.10) \cup (2.11)$ that describes the c -clique problem for a given graph and nonnegative integer c . By providing this ASP program to an ASP solver, one can compute many cliques for the same input graph. In such a case, one may be interested in computing a set of similar or diverse cliques in the given graph. Suppose that the similarity of a set of cliques is defined by some distance measure Δ . Then finding a set of 3 cliques whose distance is at least 20 is an instance of n k -DIVERSE SOLUTIONS where $n = 3$ and $k = 20$. On the other hand, we may already have two cliques C_1 and C_2 and we may want to compute a clique whose distance from $\{C_1, C_2\}$ is at most 10; this problem is an instance of k -CLOSE SOLUTION where $k = 10$.

Complexities of the decision versions of these problems are NP-Complete under reasonable assumptions [31]. In [31], we have also defined various decision/optimization problems which are variations of these problems and presented algorithms to solve them.

3.2 Computing n k -Similar/Diverse Solutions

To compute a set of n solutions whose distance is at most (resp. at least) k , we introduce an offline method and three online methods. Offline Method computes all solutions in advance and finds a set of n k -similar (resp. k -diverse) solutions afterwards. On the other hand, the online methods find a set of n k -similar (resp. k -diverse) solutions on the fly. We denote the given ASP program \mathcal{P} with `Solve.lp`; in other words, `Solve.lp` describes a solution to the given problem P . Online Method 1 modifies this program to find n k -similar (resp. k -diverse) solutions; whereas, other methods use this program as it is.

Overviews of Offline Method and Online Methods are given in Figures 3.1 and 3.2 respectively. In the following, we describe each method in detail. Although we generally consider n k -similar solutions, the methods are applicable to computing n k -diverse solutions as well.

3.2.1 Offline Method

In the offline method, we compute the set S of all the solutions for P in advance using the ASP program `Solve.lp`, with an existing ASP solver. Then, we use some clustering methods to find similar solutions in S . The idea is to form clusters of n solutions, measure the distance of each cluster, and pick a cluster whose distance is less than or equal to k .

We can compute clusters of n solutions whose distance is at most k by means of solving a graph problem: build a complete graph G whose nodes correspond to the solutions

Figure 3.1: Offline Method for computing n k -similar solutions.

Method	Offline Method
Distance Function	ASP
Approach	Compute all the solutions in advance, and find a cluster of size n whose distance is at most k among those solutions
ASP Solver	CLASP

Figure 3.2: Online Methods for computing n k -similar solutions.

Method	Online Method 1 (Reformulation)	Online Method 2 (Iterative Computation)	Online Method 3 (Incremental Computation)
Distance Function	ASP	ASP	C++
Approach	Reformulate <i>Solve.lp</i> to compute n k similar solutions at once	Compute n k -similar solutions iteratively using <i>Solve.lp</i>	Modify the search algorithm of CLASP to compute n k -solutions at once
ASP Solver	CLASP	CLASP	CLASP-NK

in S and edges are labeled by distances between the corresponding solutions; and decide whether there is a clique C of size n in G whose weight (i.e., the distance of the set of solutions denoted by the weight of the clique) is less than or equal to k . The set of vertices in the clique represents n k -similar solutions.

The weight of a clique (or the distance Δ of the solutions in the cluster) can be computed as follows: Given a function d to measure the distance between two solutions, let $\Delta(S)$ be the maximum distance between any two solutions in S . Then n k -similar solutions can be computed by Algorithm 2, where the graph G is built as follows: nodes correspond to solutions in S , and there is an edge between two nodes s_1 and s_2 in G if $d(s_1, s_2) \leq k$. Nodes of a clique of size n in this graph correspond to n k -similar solutions. Such a clique can be computed using the ASP formulation in Figure 2.1, or one of the existing exact/approximate algorithms discussed in [55].

Note that this method is sound and complete. On the other hand, however, when the solution space is very large, it might be intractable to compute all the solutions in advance and build a distance graph. In such a case, we may compute the distance graph of a tractable subset of all the solutions, and find n k -similar solutions among this subset. Although such an approach is not complete, it is still sound.

3.2.2 Online Method 1: Reformulation

Instead of computing all the solutions in advance as in the offline method, we can compute n k -similar solutions to the given problem P on the fly. First we reformulate the

Algorithm 2 Offline Method

Input: A set S of solutions, a distance function $d : S \times S \mapsto \mathbb{N}$, and two nonnegative integers n and k .

Output: A set C of n solutions whose distance is at most k .

$V \leftarrow$ Define a set of $|S|$ vertices, each denoting a unique solution in S ;

$E = \{\{v_i, v_j\} \mid v_i \neq v_j, v_i, v_j \text{ denote } s_i, s_j \in S, d(s_i, s_j) \leq k\}$;

$C \leftarrow$ Find a clique of size n in $\langle V, E \rangle$;

return C

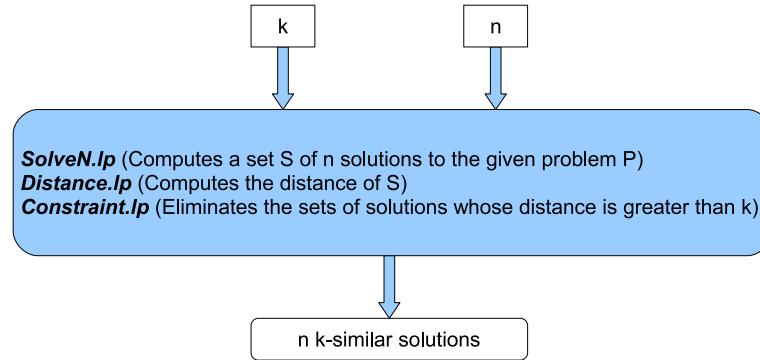


Figure 3.3: Computing n k -similar solutions, with Online Method 1.

ASP program `Solve.lp` in such a way to compute n -distinct solutions; let us call the reformulation as `SolveN.lp`. Such a reformulation can be obtained from `Solve.lp` as follows:

1. We specify the number of solutions: `solution(1..n)`.
2. In each rule of the program `Solve.lp`, we replace each atom $p(T_1, T_2, \dots, T_m)$ (except the ones specifying the input) with $p(N, T_1, T_2, \dots, T_m)$.
3. Add `solution(N)` to the body of each rule which is not safe¹.
4. Now we have a program that computes n solutions. To ensure that they are distinct, we add a constraint which expresses that every two solutions among these n solutions are different from each other.

Next we describe the distance function Δ as an ASP program, `Distance.lp`. In addition, we represent the constraints on the distance function (e.g., the distance of the solutions in S is at most k) as an ASP program `Constraint.lp`. Then we can compute n -distinct solutions for the given problem P that are k -similar, by one call of an existing ASP solver with the program `SolveN.lp` \cup `Distance.lp` \cup `Constraint.lp`, as shown in Figure 3.3. Let us give an example to illustrate Online Method 1.

¹The ASP grounder GRINGO expects rules to be safe, i.e., all variables that appear in a rule have to appear in some positive literal (a literal not preceded by *not*) in the body.

```

solution(1..n).
c{clique(S,X) : vertex(X)}c :- solution(S).
:- clique(S,X), clique(S,Y), not edge(X,Y), not edge(Y,X), X!=Y.
different(S1,S2) :- clique(S1,X), clique(S2,Y), S1 != S2, X != Y.
:- not different(S1,S2), solution(S1;S2), S1!=S2.

```

Figure 3.4: ASP formulation that computes n distinct c -cliques.

```

same(S1,S2,V) :- clique(S1,V), clique(S2,V), S1 < S2.
hammingDistance(S1,S2,c-H) :- H{same(S1,S2,V) : vertex(V)}H,
maximumDistance(H), S1 < S2.

```

Figure 3.5: ASP formulation of the Hamming distance between two cliques.

```

:- hammingDistance(S1,S2,H), H > k.

```

Figure 3.6: A constraint that forces the distance among any two solutions is less than or equal to k .

Example 1. Suppose that we want to compute n k -similar cliques in a graph. Assume that the similarity of two cliques is measured by the Hamming Distance: the distance between two cliques C and C' is equal to the number of different vertices, $|(C \setminus C') \cup (C' \setminus C)|$. The distance of a set S of cliques can be defined as the maximum distance among any two cliques in S .

The clique problem can be represented in ASP (`Solve.lp`) as in [74], also shown in Figure 2.1. We can obtain the `SolveN.lp` as described above. The reformulation (`SolveN.lp`) given in Figure 3.4. This reformulation computes n distinct cliques.

The Hamming Distance between any two cliques can be represented by the ASP program (`Distance.lp`) shown in Figure 3.5.

Finally, Figure 3.6 shows the constraint (`Constraint.lp`) that eliminates the sets whose distance is above k .

An answer set for the union of these three programs, `SolveN.lp` \cup `Distance.lp` \cup `Constraint.lp`, corresponds to n k -similar cliques.

3.2.3 Online Method 2: Iterative Computation

This method does not modify the given ASP program `Solve.lp` as in Online Method 1, but still formulates the distance function and the distance constraints as ASP programs. The idea is to find similar solutions iteratively, where the $\Delta(S)$ is always less than or equal to k after each new solution computed (Figure 3.7). Here n iterations lead to n solutions whose distance is at most k (i.e., n k -similar solutions).

Note that, like Offline Method and Online Method 1, this method is sound; however, unlike Offline Method and Online Method 1, it is not complete since the computation of a solution depends on the previously computed solutions. The method may not return

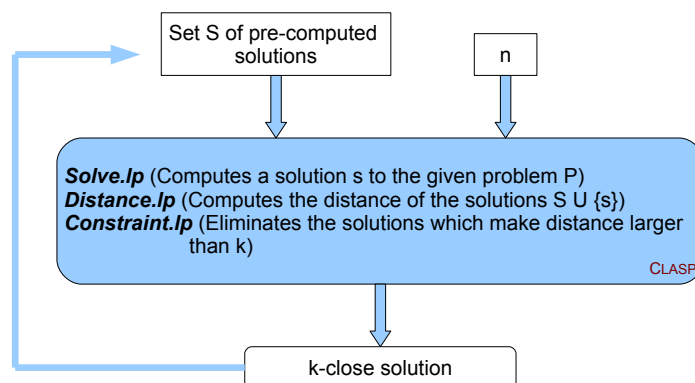


Figure 3.7: Computing n k -similar solutions, with Online Method 2. Initially $S = \emptyset$. In each run, a solution is computed and added to S , until $|S| = n$. The distance function and the constraints in the program ensure that when we add the computed solution to S , the set stays k -similar.

n k -similar solutions (even it exists) if the previously computed solutions comprise a bad solution set.

3.2.4 Online Method 3: Incremental Computation

This method is different from the other two online methods in the sense that it does not modify the ASP program `Solve.lp` describing the given computational problem P , it does not formulate the distance function Δ and the distance constraints as ASP programs. Instead, it modifies the search algorithm of an existing ASP solver in such a way that the modified ASP solver can compute n k -similar solutions (Figure 3.8). In this method, we modify the search algorithm of the ASP solver CLASP (Version 2.0.1) and the modified version is called CLASP-NK. The given distance measure Δ is implemented as a C++ program.

We modify CLASP’s algorithm as shown in Algorithm 3 to obtain CLASP-NK: the red parts show these modifications. To use CLASP-NK, one needs to prepare an options file, `NKoptions`, to describe the input parameters to compute n k -similar solutions, such as the values n and k , along with the names of predicates that characterize solutions and that are considered for computing the distance between solutions. Note that since an answer set (thus a solution) is computed incrementally in CLASP-NK, we cannot compute the distance between a partial solution and a set of solutions with respect to the given distance function Δ . Instead, one needs to implement a heuristic function to estimate a lower bound for the distance between any completion s of a partial solution with a set S of previously computed solutions. If this heuristic function is admissible then it does not underestimate the distance of $S \cup \{s\}$ (i.e., it returns a lower bound that is less than or equal to the optimal lower bound for the distance).

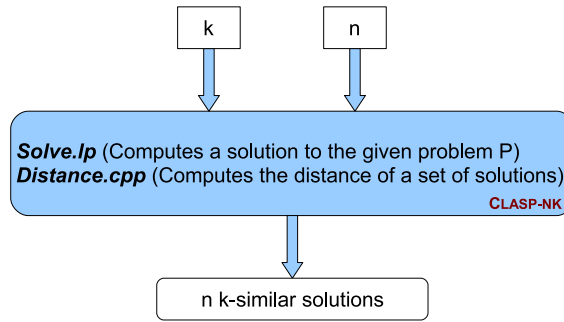


Figure 3.8: Computing n k -similar solutions, with Online Method 3. CLASP-NK is a modification of the ASP solver CLASP, that takes into account the distance function and constraints while computing an answer set in such a way that CLASP-NK becomes biased to compute similar solutions. Each computed solution is stored by CLASP-NK until a set of n k -similar solutions is computed.

Note that similar to Online Method 2, this method is also sound but not complete.

3.3 Computing k -Close/Distant Solution

We can solve the problem k -CLOSE SOLUTION utilizing the methods for n k -SIMILAR SOLUTIONS. For instance, we can modify Online Method 1 by modifying the ASP program \mathcal{P} (Solve.lp) that describes the computational problem P , by adding constraints, to ensure that the answer sets for \mathcal{P} characterize solutions for P except for the ones included in the given set S of solutions. Let us call the modified ASP program \mathcal{P}' . Next, we define a distance measure Δ' that maps a set of solutions for P to a nonnegative integer, in terms of the given measure Δ as follows: $\Delta'(X) = \Delta(S \cup X)$. Then, an answer set of \mathcal{P}' along with an ASP description of Δ' and a constraint that eliminates each solution X such that $\Delta(X) > k$, corresponds to k -close solution.

Alternatively, we can modify Online Method 2 by starting with a set S of solutions, then find a solution which is k -close to S .

Similarly, we can encode the solutions S into the DISTANCE-ANALYZE function of CLASP-NK; so that, DISTANCE-ANALYZE returns a lower bound for the distance between any completion of the partial solution and the solutions in S . Then, we can ask CLASP-NK to return one solution which will correspond to a k -close solution.

3.4 Computing Similar/Diverse Weighted Solutions

Although CLASP-NK is designed to compute similar/diverse solutions, it turns out that it could be useful to solve more general problems. We can consider DISTANCE-ANALYZE as a function that defines some preferences over answer sets. Using this function, we can ensure that the answer set solver computes answer sets that satisfy a preference function

Algorithm 3 CLASP-NK

Input: An ASP program Π , **nonnegative integers n , and k**

Output: **A set X of n solutions that are k similar (n k -similar solutions)**

```
 $A \leftarrow \emptyset$  // current assignment of literals
 $\nabla \leftarrow \emptyset$  // set of conflicts
 $X \leftarrow \emptyset$  // computed solutions
while  $|X| < n$  do
  PartialSolution  $\leftarrow A$ 
  LowerBound  $\leftarrow$  DISTANCE-ANALYZE( $X$ , PartialSolution) // compute a lower
  bound for the distance between any completion of a partial solution and the set of
  previously computed solutions
  PROPAGATION( $\Pi$ ,  $A$ ,  $\nabla$ ) // propagate literals
  if Conflict in propagation OR LowerBound  $> k$  then
    RESOLVE-CONFLICT( $\Pi$ ,  $A$ ,  $\nabla$ ) // learn and update conflicts, and backtrack
  else
    if Current assignment does not yield an answer set then
      SELECT( $\Pi$ ,  $A$ ,  $\nabla$ ) // select a literal to continue search
    else
       $X \leftarrow X \cup \{A\}$ 
       $A \leftarrow \emptyset$ 
    end if
  end if
end while
return  $X$ 
```

which is defined externally. More precisely, we can solve problems of the following sort studied in [15, 14]:

AT LEAST (resp. AT MOST) w -WEIGHTED SOLUTION: Given an ASP program \mathcal{P} that formulates a computational problem P , a weight measure ω that maps a solution for P to a nonnegative integer, and a nonnegative integer w , find a solution S for P such that $\omega(S) \geq w$ (resp. $\omega(S) \leq w$).

This problem asks for a single solution instead of a set of solutions; but this single solution should have a weight above/below some threshold. In order to solve this problem, we modified CLASP as in Algorithm 4, and call this modified version CLASP-W.

CLASP-W is similar to CLASP-NK in the sense that the WEIGHT-ANALYZE function is called at each step of the search. However, WEIGHT-ANALYZE function only considers the current partial solution unlike DISTANCE-ANALYZE which considers also the previously computed solutions. Partial solution may extend to many complete solutions, the WEIGHT-ANALYZE function computes instead an upper bound (resp. a lower bound) for the weight of a solution that extends the current partial solution. Computing an exact upper bound (resp. a lower bound) might be hard and inefficient; therefore, one may be interested in implementing a heuristic function that computes an approximate upper

bound (resp. lower bound) for a solution. To guarantee to find a complete solution, the heuristic function should be admissible. In other words, the upper bound (resp. lower bound) computed by the heuristic function shall be greater (resp. less) than or equal to the exact upper bound (resp. lower bound). If this is not the case, then we have a risk of missing a solution.

Once the WEIGHT-ANALYZE function is defined to estimate the weight of a solution, we can check whether the estimated weight is less (resp. greater) than or equal to the given weight threshold w . If the upper bound (resp. the lower bound) computed by the heuristic function is already less (resp. greater) than the given weight threshold w , then there is no solution that can be characterized by the current assignment of literals and that has a weight greater (resp. smaller) than w . Therefore, the current assignment of literals can be set as conflict in that case. After setting an assignment as a conflict, CLASP-W learns that assignment and does backtracking and never selects those assignments in the further stages of the search.

Algorithm 4 CLASP-W

Input: An ASP program Π and a nonnegative integer w

Output: An answer set for Π , that describes an at least (resp. at most) w -weighted solution

```

 $A \leftarrow \emptyset$  // current assignment of literals
 $\nabla \leftarrow \emptyset$  // set of conflicts
while  $A$  does not represent an answer set do
    // propagate according to the current assignment and conflicts; update the current
    // assignment
    PROPAGATION( $\Pi$ ,  $A$ ,  $\nabla$ )
    // compute an upper (resp. lower) bound for the weight of a solution that contains  $A$ 
    // weight  $\leftarrow$  WEIGHT-ANALYZE( $A$ )
    // if the upper bound weight is less than the desired weight value  $w$ 
    // then no need to continue search to find an at least  $w$ -weighted solution
    if There is a conflict in propagation OR weight  $< w$  then
        RESOLVE-CONFLICT ( $\Pi$ ,  $A$ ,  $\nabla$ ) // learn and update the conflict set and do back-
        tracking
    end if
    if Current assignment does not yield an answer set then
        SELECT( $\Pi$ ,  $A$ ,  $\nabla$ ) // select a literal to continue search
    else
        return  $A$ 
    end if
end while
return false

```

We also defined a more general problem which is a combination of similar/diverse and weighted solutions in [15] as follows:

Algorithm 5 CLASP-NKW

Input: An ASP program Π and nonnegative integers w, n and k

Output: A set of n k -similar at least w -weighted solutions

```
 $A \leftarrow \emptyset$  // current assignment of literals  
 $\nabla \leftarrow \emptyset$  // set of conflicts  
 $X \leftarrow \emptyset$  // previously computed answer sets  
while  $|X| < n$  do  
  PROPAGATION( $\Pi, A, \nabla$ )  
   $weight \leftarrow$  WEIGHT-ANALYZE( $A$ ) // Related to CLASP-W  
   $distance \leftarrow$  DISTANCE-ANALYZE( $A, X$ ) // Related to CLASP-NK  
  if (There is a conflict in propagation) OR ( $weight < w$ ) OR ( $distance > k$ ) then  
    RESOLVE-CONFLICT ( $\Pi, A, \nabla$ )  
  end if  
  if Current assignment does not yield an answer set then  
    SELECT( $\Pi, A, \nabla$ )  
  else  
    return  $X \leftarrow X \cup A$   
  end if  
end while  
return  $X$ 
```

n k -SIMILAR (resp. k -DIVERSE) AT LEAST (resp. AT MOST) w -WEIGHTED SOLUTIONS: Given an ASP program \mathcal{P} that formulates a computational problem P , a weight measure ω that maps a solution for P to a nonnegative integer, a distance measure Δ that maps a set of solutions to a nonnegative integer, nonnegative integers w and k , decide whether a set S of n solutions for P exists such that $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$) and for each $s \in S$, $\omega(s) \geq w$ (resp. $\omega(s) \leq w$).

We modified the algorithm of CLASP as in Algorithm 5 to compute n k -similar (resp. k -diverse) at least (resp. at most) w -weighted solutions in ASP, this version is called CLASP-NKW. At each step of the search CLASP-NKW calls both WEIGHT-ANALYZE and DISTANCE-ANALYZE; so it ensures that any completion of the partial solution both has a weight of greater than or equal to w and the distance to the previously computed solutions smaller than or equal to k ; therefore, we can compute n k -SIMILAR AT LEAST w -WEIGHTED SOLUTIONS.

Chapter 4

Finding Similar/Diverse Phylogenies

Phylogenetic systematics developed by Willi Hennig [60, 61, 62] is the study of evolutionary relations among group of species (or “taxonomic units”). These relations can be modelled as a tree whose leaves represent species, internal vertices represent their ancestors and edges represent the genetic relationship among them. Such a tree is called a “phylogeny” (or a “phylogenetic tree”).

Phylogenetic systematics deals with the problem of reconstructing phylogenies based on the given traits of the species; so that, one can analyze how the given set of species evolve through time. This problem is important for research areas as disparate as genetics, historical linguistics, zoology, anthropology, archeology, etc.. For example, a phylogeny of parasites may help zoologists to understand the evolution of human diseases [13]; a phylogeny of languages may help scientists to better understand human migrations [109].

There are several software systems, such as PHYLIP [42], PAUP [100] or PHYLO-ASP [36], that can reconstruct a phylogeny for a set of taxonomic units, based on “maximum parsimony” [28] or “maximum compatibility” [18] criterion. With some of these systems, such as PHYLO-ASP, we can compute many good phylogenies (most parsimonious phylogenies, perfect phylogenies, phylogenies with highest number of compatible traits, etc.) according to the phylogeny reconstruction criteria. In such cases, in order to decide the most “plausible” ones, domain experts manually analyze these phylogenies, since there is no available phylogenetic system that can analyze/compare these phylogenies.

For instance, PHYLO-ASP computes 45 plausible phylogenies for the Indo-European languages based on the dataset of [12]. In order to pick the most plausible phylogenies, in [12], the historical linguist Don Ringe analyzes these phylogenies by trying to cluster them into diverse groups, each containing similar phylogenies. In such cases, having a tool that reconstructs similar/diverse solutions would be useful: with such a tool, an expert can compute (instead of computing all solutions) few most diverse solutions, pick the most plausible one, and then compute phylogenies that are close to this phylogeny.

In the following, we show how our methods for computing similar/diverse solutions can be used to compute similar/diverse phylogenies. Before that, we define the phylogeny reconstruction problem and some distance functions to measure the similarity/diversity of phylogenies.

4.1 Phylogeny Reconstruction Problem

There are two main approaches to reconstruct phylogenies: character-based and distance-based. Our approach is the character-based as in [87, 12]. In character-based phylogenetics, shared traits are “(qualitative) characters”. A character is a trait in which taxonomic units can instantiate a variety of ways. If a character is instantiated by a set of taxonomic units in the same way, then these taxonomic units are assigned the same “state” of the character.

There are two main criteria in character based phylogenetics: Maximum parsimony and maximum compatibility. In maximum parsimony [28], the aim is to minimize character state changes along the edges. In maximum compatibility [18], the aim is to maximize the number of “compatible” characters. Intuitively, a character is compatible if it evolves without backmutation¹ or parallel evolution.² We consider the latter criterion while reconstructing phylogenies.

Before we describe the problems related to weighted phylogenetic tree reconstruction, we need to introduce some definitions as in [12].

A *directed graph (digraph)* is an ordered pair $\langle V, E \rangle$, where V is a set and E is a binary relation on V . In a digraph $\langle V, E \rangle$, the elements of V are called *vertices*, and the elements of E are called the *edges* of the digraph. The *out-degree* of a vertex v is the number of edges (v, u) such that $u \in V$, and the *in degree* of v is the number of edges (u, v) such that $u \in V$. A digraph $\langle V', E' \rangle$ is a subgraph of a digraph $\langle V, E \rangle$ if $V' \subset V$ and $E' \subset E$.

In a digraph $\langle V, E \rangle$, a path from vertex u to a vertex u' is a sequence v_0, v_1, \dots, v_k of vertices such that $u = v_0$ and $u' = v_k$ and $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$. If there is a path from a vertex u to a vertex v , then we say that v is *reachable from* u . If V' is a subset of V , a path from u to v whose vertices belong to V' is a *path* from u to v in V' . If there exist a path from u to v in V' , v is *reachable from* u in V' .

A *rooted tree* is a digraph with a vertex of in-degree 0, called the *root*, such that every vertex different from the root has in-degree 1 and is reachable from the root. In a rooted tree, a vertex of out-degree 0 is called a *leaf*.

A *phylogeny* for a set of taxonomic units is a finite rooted binary tree $\langle V, E \rangle$ along

¹If a character evolves from one state to another and then back to the earlier state, then backmutation occurs in the evolution of that character.

²If a state appears independently in the different lines of descent, then parallel evolution occurs.

with two finite sets I and S and a function f from $L \times I$ to S , where L is the set of leaves of the tree. The set L represents the given taxonomic units, whereas the set V describes their ancestral units and the set E describes the genetic relationships between them. The elements of I are usually positive integers (“indices”) that represent, intuitively, qualitative characters, and elements of S are possible states of these characters. The function f “labels” every leaf v by mapping every index i to the state $f(v, i)$ of the corresponding character in that taxonomic unit.

A character $i \in I$ is *compatible* with a phylogeny (V, E, L, I, S, f) if there exist a function $g : V \times \{i\} \rightarrow S$ such that

- For every leaf v of the phylogeny, $g(v, i) = f(v, i)$
- For every $s \in S$ if the set

$$V_{is} = \{x \in V : g(x, i) = s\}$$

is nonempty, then the digraph $\langle V, E \rangle$ has a subgraph with the set V_{is} of vertices that is a rooted tree.

A character is *incompatible* with a phylogeny if it is not compatible with that phylogeny.

Consider the example (Figure 4.1) given in [12]. Character 2 is compatible with the phylogeny: take g to be a function that maps every internal vertex to 1, and every leaf x to $f(x)$. The vertices labelled 1 by g form a tree; the vertices labelled 0 by g also form a tree. On the other hand, Character 1 is incompatible: there is no way of labelling the internal vertices of the tree so that the vertices labelled 1 form a tree and that the vertices labelled 0 form a tree.

The phylogeny reconstruction problem is defined as follows: Given the sets L, I, S , and the function f , build a phylogeny (V, E, L, I, S, f) with the minimum number of incompatible characters. In [12], the authors describe and solve this problem using ASP. In our experiments, we used this ASP program (as `Solve.lp`) to compute similar/diverse phylogenies.

4.2 Distance Measures for Phylogenies

The labellings of leaves denote the values of shared traits at those nodes. We consider distance measures that depend on topologies of phylogenies, therefore, while defining them we discard these labelings.

There are various measures to compute the distance between two phylogenies [85, 88, 63, 67, 22]. In the following, we first consider one of these domain-independent functions, the nodal distance measure [8], to compare two phylogenies; and then we define a distance

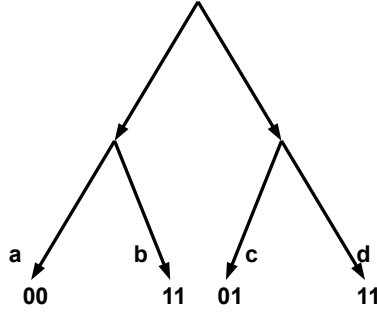


Figure 4.1: A phylogeny for the species a, b, c, d.

measure for a set of phylogenies based on the nodal distances of pairwise phylogenies, to show the applicability of our methods for finding n k -similar phylogenies. Then we define a novel distance function that measures the distance of two phylogenies, and a distance function that measures the distance of a set of phylogenies, taking into account some expert knowledge specific to evolution. With this measure we also show the effectiveness of our methods.

4.2.1 Nodal Distance of Two Phylogenies

The *nodal distance* $ND_P(x, y)$ of two leaves x and y in a phylogeny P is defined as follows: First, transform the phylogeny P to an undirected graph G where there is an undirected edge $\{i, j\}$ in the graph for each directed edge (i, j) in the phylogeny. Then $ND_P(x, y)$ is equal to the length of the shortest path between x and y in the undirected graph G . For example, consider the phylogeny, P_1 in Figure 4.2; the nodal distance between a and b is 3, whereas the nodal distance between b and c is 2. Intuitively, the nodal distance between two leaves in a phylogeny represents the degree of their relationship in that phylogeny.

Given two phylogenies P_1 and P_2 both with same set L of leaves, the *nodal distance* $D_n(P_1, P_2)$ of two phylogenies is calculated as follows:

$$D_n(P_1, P_2) = \sum_{x, y \in L} |ND_{P_1}(x, y) - ND_{P_2}(x, y)|.$$

Here the difference of the nodal distances of two leaves x and y represents the contribution of this pair of leaves to the distance between the phylogenies.

Proposition 1. *Given two phylogenies P_1 and P_2 with same set L of leaves and the same leaf-labeling function, $D_n(P_1, P_2)$ can be computed in $O(|L|^2)$ time.*

Proof. In order to compute $D_n(P_1, P_2)$, we need to perform $\binom{|L|}{2}$ nodal distance computations where $|L|$ is the number of leaves. The nodal distance between each pair (x, y) of

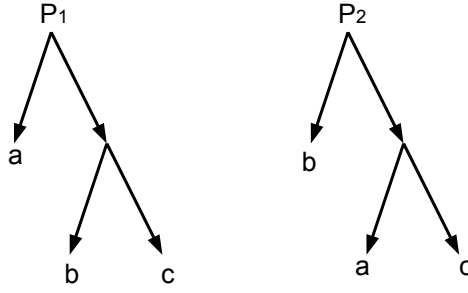


Figure 4.2: Two phylogenies $P_1 = (a, (b, c))$ and $P_2 = (b, (a, c))$.

Table 4.1: In order to compute the nodal distance $D_n(P_1, P_2)$ between the phylogenies $P_1 = (a, (b, c))$ and $P_2 = (b, (a, c))$ shown in Figure 4.2, we compute the nodal distances of the pairs of leaves, $\{a, b\}$, $\{a, c\}$ and $\{b, c\}$, and take the sum of the differences. In this case the distance between P_1 and P_2 is 2.

Pairs of leaves	Distance in P_1	Distance in P_2	Difference
$\{a, b\}$	3	3	0
$\{a, c\}$	3	2	1
$\{b, c\}$	2	3	1
Total distance			2

leaves in a tree T can be computed as $depth_T(x) + depth_T(y) - 2 \times depth_T(lca_T(x, y))$ where $lca_T(x, y)$ is the lowest common ancestor of x and y in T . Note that, once the lowest common ancestor of x and y is given, the computation of the nodal distance between x and y takes constant time. Therefore, the nodal distance between each pair of nodes in P_1 (resp. P_2) can be computed in $O(|L|^2)$ time.

In [56], the authors introduced an algorithm that finds the lowest common ancestor of two nodes in a tree in constant time after preprocessing the whole tree in linear time in the size of the number of nodes in that tree. Then, the lowest common ancestor of every two nodes in phylogeny P_1 (resp. P_2) can be computed in $O(2 \times |L| - 1) = O(|L|)$ time.

Therefore, the total time complexity of finding $D_n(P_1, P_2)$ is $O(|L|) + O(|L|^2) = O(|L|^2)$. \square

Table 4.1 shows an example of computing the nodal distance between two phylogenies. Here, the phylogenies are presented in the Newick format, where the sister sub-phylogenies are enclosed by parentheses. For instance, the first tree, P_1 , of Figure 4.2 can be represented in the Newick format as $(a, (b, c))$.

4.2.2 Descendant Distance of Two Phylogenies

Nodal distance measure computes the distance between two rooted binary trees and does not consider the evolutionary relations between nodes. In that sense, it is a domain-independent distance measure for comparing phylogenies. A distance measure that takes into account these relations might give more accurate results. Therefore, we define a new distance function based on our discussions with the historical linguist Don Ringe. In particular, we take into account the following domain-specific information in phylogenetics: the similarities of phylogenies towards their roots are more significant; and thus two phylogenies are more similar if the diversifications closer to their roots are more similar.

For each vertex v of a tree $T = \langle V, E \rangle$, let us define the descendants of x as follows:

$$desc_T(v) = \begin{cases} \{v\} & v \text{ is a leaf in } V \\ desc_T(u) \cup desc_T(u') & \text{otherwise } (v, u), (v, u') \in E, u \neq u' \end{cases}$$

and the depth of a vertex v as follows:

$$depth_T(v) = \begin{cases} 0 & v \text{ is the root of } T \\ 1 + depth_T(u) & \text{otherwise } (u, v) \in E. \end{cases}$$

To define the similarity of two phylogenies $T = \langle V, E \rangle$ and $T' = \langle V', E' \rangle$, let us first define the similarity of two vertices $v \in V$ and $v' \in V'$:

$$f(v, v') = \begin{cases} 1 & desc_T(v) \neq desc_{T'}(v') \\ 0 & \text{otherwise} \end{cases}$$

For every depth i ($0 \leq i \leq \min\{\max_{v \in V} depth_T(v), \max_{v' \in V'} depth_{T'}(v')\}$), let us also define a weight function $weight(i)$ that assigns a number to each depth i . The idea is to assign bigger weights to smaller depths so that two phylogenies are more similar if the diversifications closer to the root are more similar. This is motivated by the fact that reconstructing the evolution of languages closer to the root is more important for historical linguists.

Now we can define the similarity of two trees $T = \langle V, E \rangle$ and $T' = \langle V', E' \rangle$, with the roots R and R' respectively, at depth i ($0 \leq i \leq \min\{\max_{v \in V} depth(v), \max_{v' \in V'} depth(v')\}$), by the following measure:

$$\begin{aligned} g(0, T, T') &= weight(0) \times f(R, R') \\ g(i, T, T') &= g(i-1, T, T') + \\ &\quad weight(i) \times \sum_{x \in V, y \in V', depth_T(x)=depth_{T'}(y)=i} f(x, y), \quad i > 0 \end{aligned}$$

Table 4.2: In order to compute the descendant distance $D_l(P_1, P_2)$ between the phylogenies $P_1 = (a, (b, c))$ and $P_2 = (b, (a, c))$ shown in Figure 4.2, for each depth level, we multiply the number of vertices that have different descendants with the weight of that depth level. Then, we add up the products to find the total distance between P_1 and P_2 . The descendant distance between P_1 and P_2 is 4.

Depth	Weight P_1	Number of pairs of vertices that have different descendant sets
0 (root)	2	0
1	1	4
2	0	3
<hr/>		
Distance =	$2 \times 0 + 1 \times 4 + 0 \times 3 = 4$	

and the similarity of two trees as follows:

$$D_l(T, T') = g(\min\{\max_{v \in V} \text{depth}_T(v), \max_{v' \in V'} \text{depth}_{T'}(v')\}, T, T').$$

Proposition 2. *Given two trees P_1 and P_2 with same set L of leaves and the same leaf-labeling function, $D_l(P_1, P_2)$ can be computed in $O(|L|^3)$ time.*

Proof. Let v be the number of vertices in one tree, then v^2 is an upper bound for the number of the pairs that we can compare their descendants. Therefore, we have at most $O(v^2)$ comparisons.

Since the number of descendants is bounded by $|L|$ (after obtaining the descendants of each vertex by preprocessing in $O(v \cdot |L|)$ time), each comparison takes time $O(|L|)$.

Since $v = 2 \times |L| - 1$, $D_l(P_1, P_2)$ can be computed in $(2 \times |L| - 1)^2 \times |L|$ steps which is $O(|L|^3)$. \square

Table 4.2 shows an example of computing the distance between two trees shown in Figure 4.2.

4.2.3 Distance of a Set of Phylogenies

In the previous subsections, we defined distance functions for measuring the distance between two phylogenies. However, the problems that we defined in Section 3.1 require a distance function that measures the distance of a set of phylogenies. We can define the distance of a set of phylogenies based on the distances among pairwise phylogenies. For instance, the distance of a set S of phylogenies can be defined as the maximum distance among any two phylogenies in S .

Let D be one of the distance measures defined in the previous subsection. Then, to be able to find similar phylogenies, the distance of a set S of phylogenies (Δ_D) is defined as

follows:

$$\Delta_D(S) = \max\{D(P_1, P_2) \mid P_1, P_2 \in S\}.$$

To be able to find diverse phylogenies, the distance of a set S of phylogenies (Δ_D) is defined as follows:

$$\Delta_D(S) = \min\{D(P_1, P_2) \mid P_1, P_2 \in S\}.$$

4.3 Computing n k -Similar/Diverse Phylogenies

Analogous to the n k -similar (resp. diverse) solutions, we define the n k -similar (resp. diverse) phylogenies as follows:

n k -SIMILAR PHYLOGENIES (RESP. n k -DIVERSE PHYLOGENIES)

Given an ASP program \mathcal{P} that formulates a phylogeny reconstruction problem P , a distance measure Δ_D that maps a set of phylogenies for P to a nonnegative integer, and two nonnegative integers n and k , find a set S of n phylogenies such that $\Delta_D(S) \leq k$ (resp. $\Delta_D(S) \geq k$).

Recall that in order to compute n k -similar (resp. diverse) solutions we need an ASP program that computes a solution and a distance measure. We consider the ASP program `phylogeny-improved.lp` described in [12] as our main program that computes a phylogeny. We represent the nodal distance D_n (resp. the descendant D_l) of two phylogenies as the ASP program in Figure A.3 (resp. Figure A.4) in Appendix A. In addition, we consider the program in Figure A.5 that computes the total distance of a set of solutions with Δ_D and eliminates the ones whose total distance is greater than k .

For **Offline Method**, we compute all the phylogenies using `phylogeny-improved.lp`. Then we build a graph of phylogenies as in Section 3.2.1. Then, we use the ASP program in Figure 2.1 of Section 2.1 to find a clique of size n in the constructed graph. This clique corresponds to n k -similar phylogenies.

For **Online Method 1**, we reformulate the main program `phylogeny-improved.lp` to obtain a program that computes n distinct phylogenies as in Section 3.2.2. The reformulation is shown in Figures A.1 and A.2 in Appendix A.

For **Online Method 3**, we define a heuristic function to estimate a lower bound for the distance between any completion of a given partial phylogeny and a complete phylogeny.

Let P_c be any complete phylogeny, P_p be any partial phylogeny and L_p be the set of pairs of leaves that appear in P_p . Consider the nodal distance (Section 4.2.1) for comparing two phylogenies. Then we can define a lower bound as follows:

$$\mathcal{LB}_n(P_p, P_c) = \sum_{x,y \in L_p} |ND_{P_c}(x, y) - ND_{P_p}(x, y)|.$$

This lower bound does not overestimate the distance between a phylogeny and any completion of a partial phylogeny.

Proposition 3. *Given a partial phylogeny P_p and a complete phylogeny P_c , $\mathcal{LB}_n(P_p, P_c)$ is admissible.*

Proof. Let S'_p be a set of all completions of the partial phylogeny P_p . For every $P \in S'_p$, we need to prove that

$$\mathcal{LB}_n(P_p, P_c) \leq D_n(P, P_c)$$

holds.

Let $P_l \in \arg \min_{P \in S'_p} (D_n(P, P_c))$ be a completion with smallest distance. Then it will be enough to prove that

$$\mathcal{LB}_n(P_p, P_c) \leq D_n(P_l, P_c)$$

holds. If we replace \mathcal{LB}_n and D_n with their equivalents, the inequality will look like the following:

$$\sum_{x,y \in L_p} |ND_{P_c}(x, y) - ND_{P_p}(x, y)| \leq \sum_{x,y \in L} |ND_{P_l}(x, y) - ND_{P_c}(x, y)|$$

We can break the right hand side summation into two for L_p and $L \setminus L_p$ as follows:

$$\begin{aligned} & \sum_{x,y \in L_p} |ND_{P_c}(x, y) - ND_{P_p}(x, y)| \leq \\ & \sum_{x,y \in L_p} |ND_{P_l}(x, y) - ND_{P_c}(x, y)| + \sum_{(x,y) \in L^2 \setminus L_p^2} |ND_{P_l}(x, y) - ND_{P_c}(x, y)| \end{aligned}$$

The distance between x and y is the same for P_p and P_l where $x, y \in L_p$. Therefore, terms cancel each other and we have the following:

$$0 \leq \sum_{(x,y) \in L^2 \setminus L_p^2} |ND_{P_l}(x, y) - ND_{P_c}(x, y)|$$

Since the right hand side is a summation of absolute values, the inequality holds which completes the proof. \square

Similarly, we can define an upper bound for the differences of nodal distances measure as follows:

$$\mathcal{UB}_n(P_p, P_c) = \sum_{x,y \in L_p} |ND_{P_c}(x, y) - ND_{P_p}(x, y)| + \left(\binom{l}{2} - \binom{|L_p|}{2} \right) \times l.$$

where l denotes the number of leaves in the complete tree.

This upper bound does not underestimate the distance between a phylogeny and any completion of a partial phylogeny.

Proposition 4. *Given a partial phylogeny P_p and a complete phylogeny P_c , $\mathcal{UB}_n(P_p, P_c)$ is admissible.*

Proof of Proposition 4. Let S'_p be a set of all completions of the partial phylogeny P_p . For every $P \in S'_p$, we need to prove that

$$\mathcal{UB}_n(P_p, P_c) \geq D_n(P, P_c).$$

Let $P_u \in \arg \max_{p \in S'_p} (D_n(p, P_c))$ be a completion at largest distance. Then it will be enough to prove that

$$\mathcal{UB}_n(P_p, P_c) \geq D_n(P_u, P_c).$$

If we replace \mathcal{UB}_n and D_n with their definition, the inequality is

$$\sum_{x,y \in L_p} |ND_{P_c}(x, y) - ND_{P_p}(x, y)| + \left(\binom{l}{2} - \binom{|L_p|}{2} \right) \times l \geq \sum_{x,y \in L} |ND_{P_c}(x, y) - ND_{P_u}(x, y)|.$$

We can break the right hand side summation into two for L_p and $L \setminus L_p$ as follows:

$$\begin{aligned} & \sum_{x,y \in L_p} |ND_{P_c}(x, y) - ND_{P_p}(x, y)| + \left(\binom{l}{2} - \binom{|L_p|}{2} \right) \times l \geq \\ & \sum_{x,y \in L_p} |ND_{P_u}(x, y) - ND_{P_c}(x, y)| + \sum_{x,y \in L \setminus L_p} |ND_{P_u}(x, y) - ND_{P_c}(x, y)| \end{aligned}$$

The distance between x and y is same for P_p and P_u where $x, y \in L_p$. Terms cancel each other:

$$\left(\binom{l}{2} - \binom{|L_p|}{2} \right) \times l \geq \sum_{x,y \in L \setminus L_p} |ND_{P_u}(x, y) - ND_{P_c}(x, y)|.$$

The maximum nodal distance in a tree is equal to the number of leaves; therefore, each term in the right hand side of the inequality is at most l . Since, there are $\left(\binom{l}{2} - \binom{|L_p|}{2} \right)$ terms in the right hand side summation, $\left(\binom{l}{2} - \binom{|L_p|}{2} \right) \times l$ is greater than or equal to the summation. \square

As regards the descendants distance measure, we could not find a tight lower and upper bound. In our experiments, we consider that the lower bound (resp. upper bound) between a complete phylogeny and any completion of a partial phylogeny is 0 (resp. ∞).

We implement the admissible distance functions defined above, and give it to CLASP-NK along with the main phylogeny reconstruction program `phylogeny-improved.lp`.

In the following section, we show the experimental results that compare offline and online methods individually for similar/diverse phylogeny reconstruction.

4.4 Experimental Results

We performed experiments on reconstructing similar/diverse phylogenies based on the methods described in the previous section. In these experiments, we used the dataset assembled by Don Ringe and Ann Taylor [87] for reconstructing phylogenies for Indo-European languages. As in [12], to compute similar/diverse phylogenies, we considered the language groups Balto-Slavic (BS), Italo-Celtic (IC), Greco-Armenian (GA), Anatolian (AN), Tocharian (TO), Indo-Iranian (IIR), Germanic (GE), and the language Albanian (AL). While computing phylogenies, we also took into account some domain-specific information about these languages.

In our experiments, we considered the distance measures described in Section 4.2 as in Section 4.3.

In Tables 4.3 and 4.4, we present the results for the following computations: 2 most similar phylogenies, 2 most diverse phylogenies, 3 most similar phylogenies, 3 most diverse phylogenies, 6 most similar phylogenies with respect to the nodal distance and the descendant distance respectively. We solve these optimization problems by iteratively solving the corresponding problems (n k -SIMILAR/DIVERSE PHYLOGENIES). In the experiments, we consider the phylogenies with at most 17 incompatible characters. For each method, we present the computation time, the size of the memory used in computation, and the optimal value of k . All CPU times in the tables are in seconds, for a workstation with a 1.5GHz Xeon processor and 4x512MB RAM, running Ubuntu Server (Version 10.10). For Offline Method, Online Method 1, and Online Method 2 we used the ASP solver CLASP (Version 2.0.1), for Online Method 3, we used CLASP-NK. GRINGO (Version 3.0.3) is used as a grounder for both CLASP and CLASP-NK.

Let us first examine the results of experiments, considering the distance measure Δ_n , based on the nodal distance (Table 4.3).

Offline method first computes all the phylogenies and then finds similar/diverse phylogenies among them using ASP, as explained in Section 3.2.1. Offline method is more efficient, in terms of both computation time and memory, than the online methods. It computes all the answer sets using the projected solutions option of CLASP [50]; in other words, it computes all the phylogenies by projecting the answer sets onto edge atoms. Since there are only 8 different phylogenies (with at most 17 incompatible characters), computing all the phylogenies, building the graph of size 8, and finding a clique in this graph do not take so much time and space.

Let us compare the online methods. In terms of both computation time and memory size, Online Method 3 performs best, and Online Method 2 performs better than Online Method 1. These results conform with our expectations. Online Method 1 takes as input an ASP representation of computing n k -similar/diverse phylogenies, which is almost n times as large as the ASP program describing the phylogeny reconstruction problem used

Table 4.3: Computing similar/diverse phylogenies using the nodal distance Δ_n .

Problem	Offline Method	Online Methods		
		Reformulation	Iterative Comp.	Incremental Comp.
2 most similar ($k = 12$)	0.51 sec. 3MB $k = 12$	4.33 sec. 23MB $k = 12$	0.78 sec. 4MB $k = 12$	0.32 sec. 4MB $k = 12$
2 most diverse ($k = 32$)	0.51 sec. 3MB $k = 32$	3.29 sec. 19MB $k = 32$	0.59 sec. 10MB $k = 24$	0.37 sec. 4MB $k = 32$
3 most similar ($k = 15$)	0.51 sec. 3MB $k = 15$	28.14 sec. 70MB $k = 15$	1.43 sec. 10MB $k = 20$	0.5 sec. 4MB $k = 20$
3 most diverse ($k = 26$)	0.52 sec. 3MB $k = 26$	20.31 sec. 40MB $k = 26$	1.1 sec. 10MB $k = 26$	0.63 sec. 4MB $k = 26$
6 most similar ($k = 25$)	0.52 sec. 3MB $k = 25$	297.49 sec. 173MB $k = 25$	4.04 sec. 12MB $k = 25$	0.99 sec. 4MB $k = 25$

in other methods. Therefore, its computational performance may not be as good as the other online methods. Online Method 2 takes as input an ASP representation of phylogeny reconstruction, and an ASP representation of the distance measure, and then computes similar/diverse solutions by computing k -close/distant phylogeny n times. Since computing k -close/distant phylogeny from scratch is easy, this method provides a significant performance gain over Online Method 1. Online Method 3 computes n k -similar/diverse phylogenies with an incremental approach using CLASP-NK. It deals with the distance computation at the search level. In addition, it does not restart the search process to compute a new phylogeny; instead, it learns the conflicts caused by distance difference while computing a new phylogeny and backtracks to approximate levels to compute similar/diverse phylogenies. Therefore, it is better than Online Method 2.

Here both Offline method and Online Method 1 guarantee finding optimal solutions by iteratively solving the corresponding problems (n k -SIMILAR/DIVERSE PHYLOGENIES). On the other hand, Online Methods 2 and 3 compute similar/diverse phylogenies with respect to the first computed phylogeny, and thus may not find the optimal value for k , as observed in the computation of 3 most similar phylogenies.

Now, let us consider the distance measures Δ_l , based on preference over diversifications (Table 4.4): two phylogenies are more similar if the diversifications closer to the root are more similar. Here we consider the similarities of diversifications until depth 3 (inclusive). The results are similar with Table 4.3 in terms of computation time and memory.

Table 4.4: Computing similar/diverse phylogenies using the descendant distance Δ_l .

Problem	Offline Method	Online Methods		
		Reformulation	Iterative Comp.	Incremental Comp.
2 most similar ($k = 18$)	0.57 sec. 3MB $k = 18$	1.67 sec. 13MB $k = 18$	0.55 sec. 7MB $k = 18$	0.34 sec. 8MB $k = 18$
3 most diverse ($k = 20$)	0.57 sec. 3MB $k = 20$	3.87 sec. 21MB $k = 20$	0.91 sec. 7MB $k = 20$	0.63 sec. 8MB $k = 20$
6 most similar ($k = 18$)	0.57 sec. 3MB $k = 18$	32.68 sec. 68MB $k = 18$	2.26 sec. 9MB $k = 18$	0.97 sec. 8MB $k = 18$

In [12], after computing all 34 plausible phylogenies, the authors examine them manually and come up with three forms of tree structures, and then “filter” the phylogenies with respect to these tree structures. The phylogenies computed with our systems comply with this grouping. For example, while 2 most similar phylogenies are in the same group, 3 most diverse phylogenies are in different groups.

These results (in terms of computational efficiency and accuracy) show the effectiveness of our methods in phylogeny reconstruction: we can automatically compare many phylogenies in detail; therefore, we have developed tools to analyze and compute similar/diverse phylogenies. The features of these tools are explained in the next section.

4.5 Computational Tools

Motivated by the promising experimental results, we have developed computational tools called PHYLOCOMPARE-ASP and PHYLORECONSTRUCTN-ASP to be used by the experts in phylogenetics. PHYLOCOMPARE-ASP is useful to analyze phylogenies at hand by measuring their similarity/diversity and by grouping them. On the other hand, PHYLORECONSTRUCTN-ASP is useful to analyze a given set of species by reconstructing similar/diverse phylogenies. Since there were no such phylogenetic systems, our tools have fulfilled this need. We have integrated these tools in the phylogenetic system PHYLO-ASP [36]. In the following, we describe PHYLOCOMPARE-ASP and PHYLORECONSTRUCTN-ASP in detail.

4.5.1 PHYLOCOMPARE-ASP

Given a set of phylogenies, PHYLOCOMPARE-ASP is for computing a set of similar (resp. diverse) phylogenies among them. It takes as input:

- A set S of phylogenies (in Newick format),
- nonnegative integers n and k (optional),
- a distance function (Nodal Distance or Descendant Distance),
- a similarity/diversity option,

and outputs

- a set $S' \subseteq S$ of n phylogenies such that the distance of S' is at most (resp. at least) k (if k is provided),
- a set $S' \subseteq S$ of n phylogenies with the minimum (resp. maximum) distance (if k is not provided).

PHYLOCOMPARE-ASP utilizes the offline method to compute similar (resp. diverse) phylogenies in such a way that it builds a graph G whose nodes correspond to the phylogenies in S and edges are labelled by the distances between the corresponding phylogenies. In case k is provided, PHYLOCOMPARE-ASP finds a clique of size n in G , such that the distance of the set of phylogenies in the clique is less (resp. greater) than or equal to k , as in Algorithm 2. Such a clique is computed using the ASP program in Figure 2.1. In case k is not provided, PHYLOCOMPARE-ASP optimizes it using a similar program with additional optimization statements, such as `#minimize` and `#maximize` available in GRINGO [46].

Figure 4.3 shows a screen shot of the web interface of PHYLOCOMPARE-ASP which is for computing n phylogenies with the minimum (resp. maximum) distance. The user enters 4 phylogenies, and wants to compute a set of 3 phylogenies among them with the minimum total distance (with respect to the Nodal Distance). Figure 4.4 shows the result of the computation, where the distance matrix of the given phylogenies is provided as well as a set of 3 phylogenies with the minimum total distance.

4.5.2 PHYLORECONSTRUCTN-ASP

PHYLORECONSTRUCTN-ASP differs from PHYLOCOMPARE-ASP in the sense that it does not find similar (resp. diverse) phylogenies in a given set; instead, it reconstructs similar (resp. diverse) phylogenies from a given character-state matrix. PHYLORECONSTRUCTN-ASP takes as input

- a matrix M ,
- nonnegative integers n , k (optional), and c ,
- a distance function (Nodal Distance or Descendant Distance),

Phylogenies:

```
(A,(B,(C,D)))  
((A,B),(C,D))  
(B,(C,(D,A)))  
(D,(A,(B,C)))
```

n:

Similar/Diverse:

Distance Measure:

Figure 4.3: A screen shot of PHYLOCOMPARE-ASP where the user enters four phylogenies in newick format.

- a similarity/diversity option,

and outputs (in newick format)

- n k -similar (resp. k -diverse) phylogenies (if k is provided),
- n most similar (resp. most diverse) phylogenies (if k is not provided),

where each phylogeny has at most c incompatible characters. In matrix M , the rows represent the leaves L , columns represent the characters I and, for each $l \in L$ and $i \in I$, $M[l, i] = f(l, i)$. The problem of computing n most similar (resp. most diverse) phylogenies is an instance of n MOST SIMILAR (resp. MOST DIVERSE) SOLUTIONS defined in [31]. It is an optimization version of n k -SIMILAR (resp. k -DIVERSE) SOLUTIONS (k is minimized (resp. maximized)) where the aim is to compute a set of solutions with the minimum (resp. maximum) distance. If k is provided as an input, the system uses CLASP-NK to find n k -similar phylogenies, as shown in Section 4.3. If k is not provided then the system optimizes k and find n most similar (resp. most diverse) phylogenies. Algorithm 6 shows how PHYLORECONSTRUCTN-ASP computes n most similar phylogenies with respect to a given input. PHYLORECONSTRUCTN-ASP first transforms the given

RESULTS

Tree 1 : (A,(B,(C,D)))
Tree 2 : ((A,B),(C,D))
Tree 3 : (B,(C,(D,A)))
Tree 4 : (D,(A,(B,C)))

DISTANCE VALUES

Distance Matrix (first row and first column correspond to tree numbers) :

-	1	2	3	4
1	0	3	6	6
2	3	0	7	7
3	6	7	0	4
4	6	7	4	0

Maximum distance is 7.
Minimum distance is 3.

Set(s) of 3 phylogenies whose distance is minimum

{1,3,4}.

Figure 4.4: PHYLOCOMPARE-ASP computes a set of 3 phylogenies with the minimum total distance among the given phylogenies shown in Figure 4.3.

input matrix into an ASP representation. This is done by using the PHYLOANALYZE-ASP³ tool of PHYLO-ASP. Then the system performs a binary search between an upper bound and a lower bound for k . A natural lower bound for k is 0, since the distance cannot be negative. Similarly, since we need to minimize k , the maximum pairwise distance in a set of n phylogenies is used as an upper bound. CLASP is used to compute such a set of n different phylogenies. In each step of the binary search, the algorithm checks whether n k -similar phylogenies exist (using CLASP-NK) and sets the bounds of the search accordingly.

³<http://krr.sabanciuniv.edu/projects/Phylo-ASP/PhyloAnalyze-ASP/>

Algorithm 6 PHYLORECONSTRUCTN-ASP

Input: Input matrix M , nonnegative integers c and n .

Output: A set S of n most similar phylogenies.

$\mathcal{P} \leftarrow$ ASP program that describes a phylogeny for M which has at most c incompatible characters

$S \leftarrow$ Compute a set of n different phylogenies using CLASP with \mathcal{P}

$k \leftarrow$ Maximum pairwise distance in S

$UpperBound := k$

$LowerBound := 0$

while $UpperBound - LowerBound > 1$ **do**

$k := \lfloor (UpperBound + LowerBound)/2 \rfloor$

$S' \leftarrow$ Compute a set of n k -similar phylogenies using CLASP-NK with \mathcal{P}

if $|S'| = n$ **then** // n k -similar phylogenies exists

$S := S'$

$UpperBound \leftarrow$ Maximum pairwise distance in S

else // n k -similar phylogenies does not exist

$LowerBound := k$

end if

end while

if $|S| = n$ **then**

return S

else

return "No solution"

end if

Chapter 5

Finding Similar/Diverse Plans

Given an initial state, goal conditions, and a description of actions, planning is the problem of finding a sequence of actions (i.e., a plan) that would lead the initial state to a goal state. Planning is applied in various domains, such as robotics, web service composition, and genome rearrangement. In planning, it may be desirable to compute a set of similar or diverse plans in different situations. For instance, consider a variation of the example given in [99] in connection with modeling web service composition as a planning problem [79]: suppose that the web service engine computes a plan/composition; then it can compute a set of compositions similar to this particular one, so that if a failure occurs while executing one composition, an alternative composition which is less likely to be failing simultaneously can be used [19]. Alternatively, let us consider planning in the context of robotics in a dynamic environment with uncertainties. If the plan failure occurs, for instance, due to some collisions with an obstacle as in the scenarios presented in [16], the agent may want to find a plan that is distant from the previously computed plan so that it does not collide with the obstacle again.

Motivated by these examples, we study the problem of finding similar/diverse plans using answer set programming. In the following, we define the planning problem, a distance measure for plans and the n k -similar/diverse plans problem. Then, we show that how our methods for computing n k -similar/diverse solutions are useful for computing similar/diverse plans.

5.1 Problem Description

A *Planning domain* is a 5-tuple $\langle S, A, f, s_0, G \rangle$ where S is a finite set of states, A is a finite set of actions, $f : S \times A \rightarrow S$ is a state transition function, $s_0 \in S$ is the initial state, and $G \subseteq S$ is a set of goal states. Planning problem is defined as follows: Given S, A, f, s_0, G and a nonnegative integer l ; find a plan (i.e., a sequence of actions) of length at most l that transforms s_0 into a state $g \in G$.

A planning problem can be described as an ASP program [71, 73, 33] such that each answer set corresponds to a plan; therefore, we can apply our methods for finding similar/diverse solutions to find similar/diverse plans. In particular, we study the following instance of n k -SIMILAR SOLUTIONS (resp. n k -DIVERSE SOLUTIONS):

n k -SIMILAR PLANS (RESP. n k -DIVERSE PLANS)

Given an ASP program \mathcal{P} that formulates a planning problem P , a distance measure Δ_h that maps a set of plans for P to a nonnegative integer, and two nonnegative integers n and k , find a set S of n plans for P such that $\Delta_h(S) \leq k$ (resp. $\Delta_h(S) \geq k$).

In order to compute similar/diverse plans, we need to define a distance function Δ_h for a set of plans. The distance function may be specific to the planning domain we are interested in. On the other hand, there are domain-independent distance functions to measure the distance of two plans based on the hamming distance. We define the distance $\Delta_h(S)$ of a set S of similar plans as follows:

$$\Delta_h(S) = \max\{D_h(P_1, P_2) \mid P_1, P_2 \in S, |P_1| \leq |P_2|\}$$

Similarly, the distance $\Delta_h(S)$ of a set S of diverse plans is defined as follows:

$$\Delta_h(S) = \min\{D_h(P_1, P_2) \mid P_1, P_2 \in S, |P_1| \leq |P_2|\}$$

based on the action-based hamming distance D_h defined in [99] to measure the distance between two plans. Intuitively, $D_h(P_1, P_2)$ is the number of differentiating actions in each time step of two plans P_1 and P_2 . More precisely: let us denote a plan X of length l by a function act_X that maps every nonnegative integer i ($1 \leq i \leq l$) to the i 'th action of the plan X , and let us denote by $|X|$ the length of a plan X ; then the Hamming Distance $D_h(P_1, P_2)$ between two plans P_1 and P_2 such that $|P_1| \leq |P_2|$ can be defined as follows:

$$D_h(P_1, P_2) = |\{i \mid act_{P_1}(i) \neq act_{P_2}(i), 1 \leq i \leq |P_1|\}| + |P_2| - |P_1|$$

Proposition 5. *Given two plans P_1 and P_2 such that $|P_1| \leq |P_2|$, $D_h(P_1, P_2)$ can be computed in $O(|P_1|)$ time.*

Proof. In order to compute $D_h(P_1, P_2)$, we need to compare $act_{P_1}(i)$ and $act_{P_2}(i)$ for each i ($1 \leq i \leq |P_1|$). Therefore, this step takes $O(|P_1|)$ time. In addition, the difference $|P_2| - |P_1|$ can be calculated in constant time; hence the total time complexity of computing $D_h(P_1, P_2)$ is $O(|P_1|) + O(1) = O(|P_1|)$. \square

Example 2. *Suppose that a planning problem asks for a plan of length less than or equal to 7. Consider two plans, P_1 and P_2 , that are characterized by the functions act_{P_1} and*

act_{P_2} respectively, as follows:

$$act_{P_1}(1) = a_1 \quad act_{P_1}(2) = a_2$$

$$act_{P_1}(3) = a_3 \quad act_{P_1}(4) = a_4$$

$$act_{P_1}(5) = a_5 \quad act_{P_1}(6) = a_6$$

$$act_{P_2}(1) = a_1 \quad act_{P_2}(2) = a_2$$

$$act_{P_2}(3) = a_7 \quad act_{P_2}(4) = a_4$$

$$act_{P_2}(5) = a_3 \quad act_{P_2}(6) = a_5$$

$$act_{P_2}(7) = a_6$$

The distance $D_h(P_1, P_2)$ between P_1 and P_2 is 4 since the actions at time steps 3, 5 and 6 are different and P_2 has an additional action (at time step 7).

5.2 Computing Similar/Diverse Plans

We apply our methods for computing n k -similar/diverse solutions (Section 3.2) to compute n k -similar/diverse plans for the blocks world planning problem. In this problem, we have blocks on a table arranged in several towers. Each block is on top of either another block or the table. There is a single action called *move* which takes one block from top of a tower and puts it on top of another tower or on the table.

We take \mathcal{P} as the ASP formulation of the non-concurrent Blocks World¹ as in [35] to compute a plan of length at most l (Figure A.6 in Appendix A), together with an ASP description of the Blocks World instance given in Figure 5.1. ASP formulations of the distance functions D_h and $\Delta_h(S)$ for Blocks World are presented in Figures A.8 and A.9 in Appendix A.

For Online Method 1, we reformulate the main program to obtain a program that computes n distinct plans as in Section 3.2.2. The reformulation is shown in Figure A.7 in Appendix A.

To be able to apply our Online Method 3 with CLASP-NK to compute n k -similar plans of length at most l , we define a heuristic function \mathcal{LB}_h to estimate a lower bound for the distance between a plan P_c and any plan-completion of a “partial” plan P_p . Intuitively, a partial plan consists of parts of a plan. Let us characterize a partial plan P_p by a partial function act_{P_p} from $\{1, \dots, l\}$ to the set of actions; that is, act_{P_p} is a function from a subset of $\{1, \dots, l\}$ to the set of actions. A *plan-completion of a partial plan P_p* is a plan Y of length l' ($l' \leq l$) for the planning problem P such that act_Y is an extension of act_{P_p} to

¹Although we only consider non-concurrent Blocks World, it is also possible to perform experiments on a concurrent version by easily modifying the ASP formulation by removing the restriction for concurrency, and by defining a similar distance function for concurrent plans.

$\{1, \dots, l'\}$. Then we can define $\mathcal{LB}_h(P_p, P_c)$ for a partial plan P_p and a plan P_c as follows:

$$\begin{aligned} \mathcal{LB}_h(P_p, P_c) = & |\{i \mid act_{P_p}(i) \neq act_{P_c}(i), i \in \text{dom } act_{P_p}, 1 \leq i \leq |P_c|\}| \\ & + |\{i \mid i \in \text{dom } act_{P_p}, |P_c| < i \leq l'\}| \end{aligned}$$

In Example 2, consider a partial plan P_p characterized by the function act_{P_p} as follows:

$$\begin{aligned} act_{P_p}(2) = a_2 \quad act_{P_p}(4) = a_4 \\ act_{P_p}(5) = a_3 \quad act_{P_p}(7) = a_6 \end{aligned}$$

The lower bound $\mathcal{LB}_h(P_p, P_1)$ for the distance between any completion of P_p and P_1 is computed as follows:

$$\begin{aligned} \mathcal{LB}_h(P_p, P_1) &= |\{i \mid act_{P_p}(i) \neq act_{P_1}(i), i \in \text{dom } act_{P_p}, 1 \leq i \leq 6\}| \\ &\quad + |\{i \mid i \in \text{dom } act_{P_p}, 6 < i \leq 7\}| \\ &= |\{5\}| + |\{7\}| = 2. \end{aligned}$$

One completion of P_p is P_2 . Note that $\mathcal{LB}_h(P_p, P_1) \leq D_h(P_1, P_2)$. Indeed, the following proposition expresses that \mathcal{LB}_h does not overestimate the distance between P_c and any plan-completion X of P_p .

Proposition 6. *For a partial plan P_p and a plan P_c for the planning problem P , $\mathcal{LB}_h(P_p, P_c)$ is admissible.*

Proof of Proposition 6. Take any plan-completion X of the partial plan P_p . Consider two cases.

Case 1: $|X| \leq |P_c|$. Our goal is to prove that

$$\mathcal{LB}_h(P_p, P_c) \leq D_h(X, P_c).$$

By the definition of D_h , the distance between X and P_c is:

$$D_h(X, P_c) = |\{i \mid act_X(i) \neq act_{P_c}(i), 1 \leq i \leq |X|\}| + |P_c| - |X|.$$

Since X is a plan-completion of P_p and $|X| \leq |P_c|$, $\text{dom } act_{P_p} \subseteq \text{dom } act_{P_c}$; then, by the definition of \mathcal{LB}_h :

$$\mathcal{LB}_h(P_p, P_c) = |\{i \mid act_{P_p}(i) \neq act_{P_c}(i), i \in \text{dom } act_{P_p}\}|.$$

Since X is a plan-completion of P_p ,

$$\{i \mid act_{P_p}(i) \neq act_{P_c}(i), i \in \text{dom } act_{P_p}\} \subseteq \{i \mid act_X(i) \neq act_{P_c}(i), 1 \leq i \leq |X|\}.$$

Hence,

$$\mathcal{LB}_h(P_p, P_c) \leq |\{i \mid \text{act}_X(i) \neq \text{act}_{P_c}(i), 1 \leq i \leq |X|\}| + |P_c| - |X| = D_h(X, P_c).$$

Case 2: $|X| > |P_c|$. Our goal is to prove that

$$\mathcal{LB}_h(P_p, P_c) \leq D_h(P_c, X).$$

By the definition of D_h , the distance between X and P_c is:

$$D_h(P_c, X) = |\{i \mid \text{act}_X(i) \neq \text{act}_{P_c}(i), 1 \leq i \leq |P_c|\}| + |X| - |P_c|.$$

By the definition of \mathcal{LB}_h :

$$\begin{aligned} \mathcal{LB}_h(P_p, P_c) = & |\{i \mid \text{act}_{P_p}(i) \neq \text{act}_{P_c}(i), i \in \text{dom } \text{act}_{P_p}, 1 \leq i \leq |P_c|\}| + \\ & |\{i \mid l \geq i > |P_c|, i \in \text{dom } \text{act}_{P_p}\}|. \end{aligned}$$

Since X is a plan-completion of P_p , act_X extends act_{P_p} , and then

$$\begin{aligned} & \{i \mid \text{act}_{P_p}(i) \neq \text{act}_{P_c}(i), i \in \text{dom } \text{act}_{P_p}, 1 \leq i \leq |P_c|\} \\ & \subseteq \{i \mid \text{act}_X(i) \neq \text{act}_{P_c}(i), 1 \leq i \leq |X|\}. \end{aligned}$$

Since $|X| > |P_c|$,

$$|X| - |P_c| > |\{i \mid i \in \text{dom } \text{act}_{P_p}, |X| \geq i > |P_c|\}| = |\{i \mid i \in \text{dom } \text{act}_{P_p}, l \geq i > |P_c|\}|.$$

Hence,

$$\mathcal{LB}_h(P_p, P_c) \leq |\{i \mid \text{act}_X(i) \neq \text{act}_{P_c}(i), 1 \leq i \leq |X|\}| + |X| - |P_c| = D_h(P_c, X).$$

□

Similarly, to be able to apply our Online Method 3 with CLASP-NK to compute n k -diverse plans of length at most l , we define a heuristic function $\mathcal{UB}_h(P_p, P_c)$ to estimate an upper bound for the distance between a plan P_c and any plan-completion of P_p :

$$\mathcal{UB}_h(P_p, P_c) = l - |\{i \mid \text{act}_{P_p}(i) = \text{act}_{P_c}(i), i \in \text{dom } \text{act}_{P_p}, 1 \leq i \leq |P_c|\}|.$$

For instance, for the partial plan P_p and P_1 above,

$$\begin{aligned} \mathcal{UB}_h(P_p, P_1) &= 7 - |\{i \mid \text{act}_{P_p}(i) = \text{act}_{P_1}(i), i \in \text{dom } \text{act}_{P_p}, 1 \leq i \leq 6\}| \\ &= 7 - |\{2, 4\}| = 7 - 2 = 5 \end{aligned}$$

and $\mathcal{UB}_h(P_p, P_1) \geq D_h(P_1, P_2)$. Indeed, the following proposition expresses that this upper bound function does not underestimate the distance between P_c and any plan-completion X of P_p .

Proposition 7. *For a partial plan P_p and a plan P_c for the planning problem P , $\mathcal{UB}_h(P_p, P_c)$ is admissible.*

Proof of Proposition 7. Take any plan-completion X of partial plan P_p . Consider two cases.

Case 1: $|X| \leq |P_c|$. Our goal is to prove that

$$\mathcal{UB}_h(P_p, P_c) \geq D_h(X, P_c)$$

where

$$\begin{aligned} \mathcal{UB}_h(P_p, P_c) &= l - |\{i \mid \text{act}_{P_p}(i) = \text{act}_{P_c}(i), i \in \text{dom act}_{P_p}\}|, \\ D_h(X, P_c) &= |\{i \mid \text{act}_X(i) \neq \text{act}_{P_c}(i), 1 \leq i \leq |X|\}| + |P_c| - |X|. \end{aligned}$$

Since $|X| \leq |P_c|$ and X is a plan-completion of P_p , the set

$$\{i \mid \text{act}_{P_p}(i) = \text{act}_{P_c}(i), i \in \text{dom act}_{P_p}\}$$

does not intersect with the set

$$Y = \{i \mid \text{act}_X(i) \neq \text{act}_{P_c}(i), 1 \leq i \leq |X|\} \cup \{i \mid |X| < i \leq |P_c|\}.$$

Then

$$\{1, \dots, l\} \setminus \{i \mid \text{act}_{P_p}(i) = \text{act}_{P_c}(i), i \in \text{dom act}_{P_p}\}$$

is a superset of Y . Therefore,

$$\begin{aligned} \mathcal{UB}_h(P_p, P_c) &= l - |\{i \mid \text{act}_{P_p}(i) = \text{act}_{P_c}(i), i \in \text{dom act}_{P_p}\}| \\ &\geq |\{i \mid \text{act}_X(i) \neq \text{act}_{P_c}(i), 1 \leq i \leq |X|\}| + |P_c| - |X| \\ &= D_h(X, P_c). \end{aligned}$$

Case 2: $|X| > |P_c|$. Our goal is to prove that

$$\mathcal{UB}_h(P_p, P_c) \geq D_h(P_c, X)$$

where

$$\begin{aligned} \mathcal{UB}_h(P_p, P_c) &= l - |\{i \mid \text{act}_{P_p}(i) = \text{act}_{P_c}(i), i \in \text{dom act}_{P_p}, 1 \leq i \leq |P_c|\}|, \\ D_h(P_c, X) &= |\{i \mid \text{act}_X(i) \neq \text{act}_{P_c}(i), 1 \leq i \leq |P_c|\}| + |X| - |P_c|. \end{aligned}$$

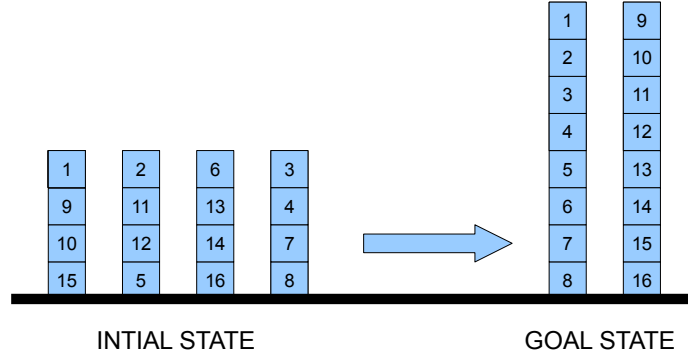


Figure 5.1: Blocks World problem.

Since $|X| > |P_c|$ and X is a plan-completion of P_p , the set

$$\{i \mid act_{P_p}(i) = act_{P_c}(i), i \in \text{dom } act_{P_p}, 1 \leq i \leq |P_c|\}$$

does not intersect with the set

$$Y = \{i \mid act_X(i) \neq act_{P_c}(i), 1 \leq i \leq |P_c|\} \cup \{i \mid |P_c| < i \leq |X|\}.$$

Then

$$\{1, \dots, l\} \setminus \{i \mid act_{P_p}(i) = act_{P_c}(i), i \in \text{dom } act_{P_p}, 1 \leq i \leq |P_c|\}$$

is a superset of Y . Therefore,

$$\begin{aligned} \mathcal{UB}_h(P_p, P_c) &= l - |\{i \mid act_{P_p}(i) = act_{P_c}(i), i \in \text{dom } act_{P_p}, 1 \leq i \leq |P_c|\}| \\ &\geq |\{i \mid act_X(i) \neq act_{P_c}(i), 1 \leq i \leq |P_c|\}| + |X| - |P_c| \\ &= D_h(P_c, X). \end{aligned}$$

□

5.3 Experimental Results

We performed some experiments (based on the methods in the previous section) to find 2 most similar plans, 2 most diverse plans, 3 most similar plans, 3 most diverse plans, 6 most similar plans for the Blocks World instance in Figure 5.1. We solve these optimization problems by iteratively solving the corresponding problems (n k -SIMILAR/DIVERSE PLANS). In the experiments, we consider the plans of length at most 22. Table 5.1 summarizes the results of these experiments. All CPU times are in seconds, for a workstation with a 1.5GHz Xeon processor and 4x512MB RAM, running Ubuntu Server (Version

10.10). For the offline method, online method 1, and online method 2 we used the ASP solver CLASP (Version 2.0.1), for the online method 3, we used CLASP-NK. We used GRINGO (Version 3.0.3) as the grounder for both CLASP and CLASP-NK.

It can be observed that the planning problem in Figure 5.1 has too many solutions (more than 50.000), and it is intractable to compute all of them in advance and then the distances between all pairwise solutions. Therefore, instead of computing all the solutions in advance, we compute a subset of them (around 200) which is small enough to construct a distance graph, and apply our Offline Method in this way (as mentioned in Section 3.2.1). However, these 200 solutions are not diverse enough, and thus, although we can find many very similar solutions, it is hard to find diverse solutions; for instance, we can find 6 1-similar plans but we can find only 3 6-diverse plans.

Online Method 1 performs worst in comparison with the other online methods, as in our experiments with phylogeny reconstruction problems, due to the large ASP program (Figure A.7 in Appendix A) used for computing n distinct plans.

Online Method 2 is comparable with Online Method 3 in terms of computing similar solutions. After computing a solution, computing a 1-close plan has a very small search space and CLASP can find a similar solution in a short time. On the other hand, computing a 21-distant solution has a huge search space. Therefore, performance of computing diverse solutions with Online Method 2 is worse than that of Online Method 3.

Online Method 3 deals with the Hamming distance computation at the search level. In addition, it does not restart the search process to compute a new plan; instead, it learns the conflicts caused by distance difference while computing a new plan and backtracks to approximate levels to compute similar/diverse plans. Especially, for the computation of diverse plans, such a search strategy creates a significant performance gain.

Table 5.1: Computing similar/diverse plans for the blocks world problem. OM denotes “Out of memory.”

Problem	Offline Method	Online Methods		
		Reformulation	Iterative Comp.	Incremental Comp.
2 most similar ($k = 1$)	-	6 min. 45 sec.	6 min. 53 sec.	7 min. 17 sec.
	OM	106 MB	73 MB	111 MB
	-	$k = 1$	$k = 1$	$k = 1$
2 most diverse ($k = 22$)	-	33 min. 28 sec.	11 min.	7 min. 40 sec.
	OM	213 MB	73 MB	112 MB
	-	$k = 22$	$k = 22$	$k = 21$
3 most similar ($k = 1$)	-	7 min. 5 sec.	7 min. 3 sec.	7 min. 21 sec.
	OM	141 MB	73 MB	112 MB
	-	$k = 1$	$k = 1$	$k = 2$
3 most diverse ($k = 22$)	-	78 min 42 sec.	18 min. 49 sec.	12 min. 40 sec.
	OM	333 MB	73 MB	167 MB
	-	$k = 22$	$k = 21$	$k = 21$
6 most similar ($k = 1$)	-	64 min. 42 sec.	7 min. 32 sec.	7 min. 18 sec.
	OM	584 MB	73 MB	112 MB
	-	$k = 1$	$k = 1$	$k = 2$

Chapter 6

Finding Similar/Diverse Genes

Recent advances in health and life sciences have led to generation of a large amount of biomedical data. To facilitate access to its desired parts, such a big mass of data has been represented in structured forms, like biomedical ontologies and databases. On the other hand, representing these biomedical ontologies and databases in different forms, constructing them independently from each other, and storing them at different locations have brought about many challenges for answering queries about the knowledge represented in these ontologies and databases. Consider, for instance, the following complex query which requires integrating several knowledge resources:

Q1 What are the genes that are targeted by the drug Epinephrine and that interact with the gene DLG4?

In order to answer this query, an expert needs to obtain information about drug-gene relations and gene-gene relations which can be found in two different databases or web servers. Therefore, the expert should have the knowledge of finding these sources and retrieve the relevant information to perform the reasoning manually.

In addition, some complex queries have multiple answers which necessitates further analysis. Consider, for instance, the query “What are the genes that are targeted by the drug Epinephrine?”, which has more than thirty answers according to the CTD database. In order to analyze Epinephrine and discover a new drug that targets these genes, it might be desirable to group the genes with respect to their functionality. Therefore, an expert might be interested in finding answers to complex queries related to similar/diverse genes as follows:

Q2 What are the 3 most similar genes that are targeted by the drug Epinephrine?

To answer this query, an expert needs to find the genes that are targeted by the drug Epinephrine using one knowledge resource, and needs to obtain information about the

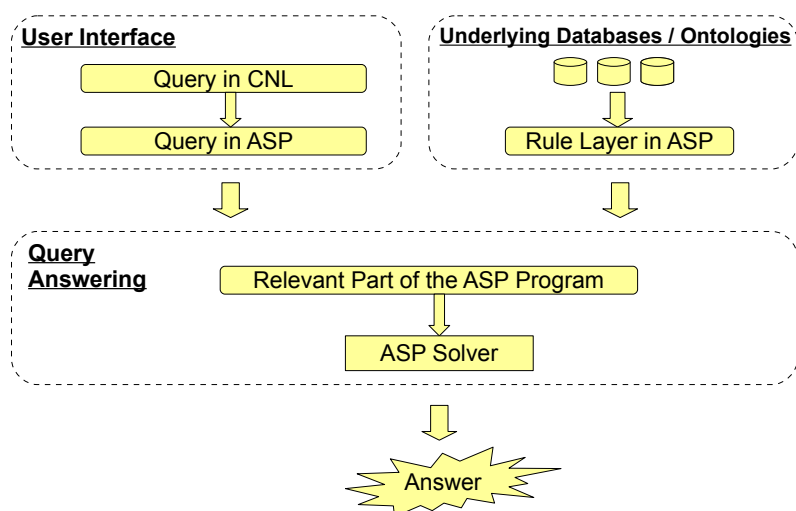


Figure 6.1: System overview of BIOQUERY-ASP.

functional similarity of these genes from another resource. However, because the manual discovery of this knowledge requires significant time and effort, there is a crucial need to build automated tools that can answer such complex queries. With this motivation, we develop an ASP-based system called BIOQUERY-ASP¹ to answer complex biomedical queries. As a part of BIOQUERY-ASP, we apply our methods for computing similar/diverse solutions to answer complex queries related to similar/diverse genes. In the following, we explain BIOQUERY-ASP in detail.

6.1 BIOQUERY-ASP

The system overview of the query answering part of the BIOQUERY-ASP is given in Figure 6.1. As can be seen from the figure, the system consists of three parts.

Underlying Databases/Ontologies BIOQUERY-ASP uses several knowledge resources about relations among biomedical concepts. In our experiments, we used large biomedical knowledge resources about genes, drugs and diseases, such as PHARMGKB² DRUG-BANK³, BIOGRID⁴, CTD⁵, and SIDER⁶. The types of the relations retrieved from these databases are shown in Figure 6.1. The data coming from these sources are converted into and stored as ASP facts. We defined a “rule layer” over these knowledge resources. This ASP program contains rules to integrate the knowledge resources, such as:

¹<http://krr.sabanciuniv.edu/projects/BioQuery-ASP/>

²<http://www.pharmgkb.org/>

³<http://www.drugbank.ca/>

⁴<http://thebiogrid.org/>

⁵<http://ctd.mdibl.org/>

⁶<http://sideeffects.embl.de/>

Table 6.1: The retrieved relations among biomedical concepts.

Source	Relation
BIOGRID	<i>gene-gene</i>
DRUGBANK	<i>drug-drug</i> <i>drug-category</i>
SIDER	<i>drug-sideeffect</i>
PHARMGKB	<i>drug-disease</i> <i>drug-gene</i> <i>disease-gene</i>
CTD	<i>drug-disease</i> <i>drug-gene</i> <i>disease-gene</i>

```
drug_gene(D, G) :- drug_gene_pharmgkb(D, G) .
drug_gene(D, G) :- drug_gene_ctd(D, G) .
```

which integrates the knowledge extracted from PHARMGKB and CTD, about “which drug targets which gene.” The rule layer also includes auxiliary definitions, such as chains of gene-gene relations from a starting gene Y whose length is at most L :

```
gene_reachable_from(X, 1) :- gene_gene(X, Y), start_gene(Y) .
gene_reachable_from(X, N+1) :- gene_gene(X, Z),
    gene_reachable_from(Z, N), 0 < N, N < L, max_chain_length(L) .
```

User Interface BIOQUERY-ASP allows users to construct queries like Q1 and Q2 as shown in the Figure 6.2. Using the auto-completion feature, users can construct queries in the grammar of BIOQUERY-ASP. Then, the user interface transforms these queries into ASP programs. For example, query Q1 is represented in ASP as follows:

```
what_be_genes(GN) :- drug_gene("Epinephrine", GN),
    gene_gene("DLG4", GN) .
```

Query Answering The reasoner of BIOQUERY-ASP takes the ASP program coming from the rule layer and the ASP program of the query, and finds an answer to the query using the efficient solvers of ASP. Let Π be the program that corresponds to the rule layer (including the facts coming from knowledge resources) and Q be the program that represents the query. Then an answer set of $\Pi \cup Q$ corresponds to the answer of the query. Most of the queries do not require the entire knowledge coming from the rule layer. Consider, for instance, the query Q1 which requires knowledge about drugs and genes. Then, in order to answer this query, knowledge about diseases and side-effects are

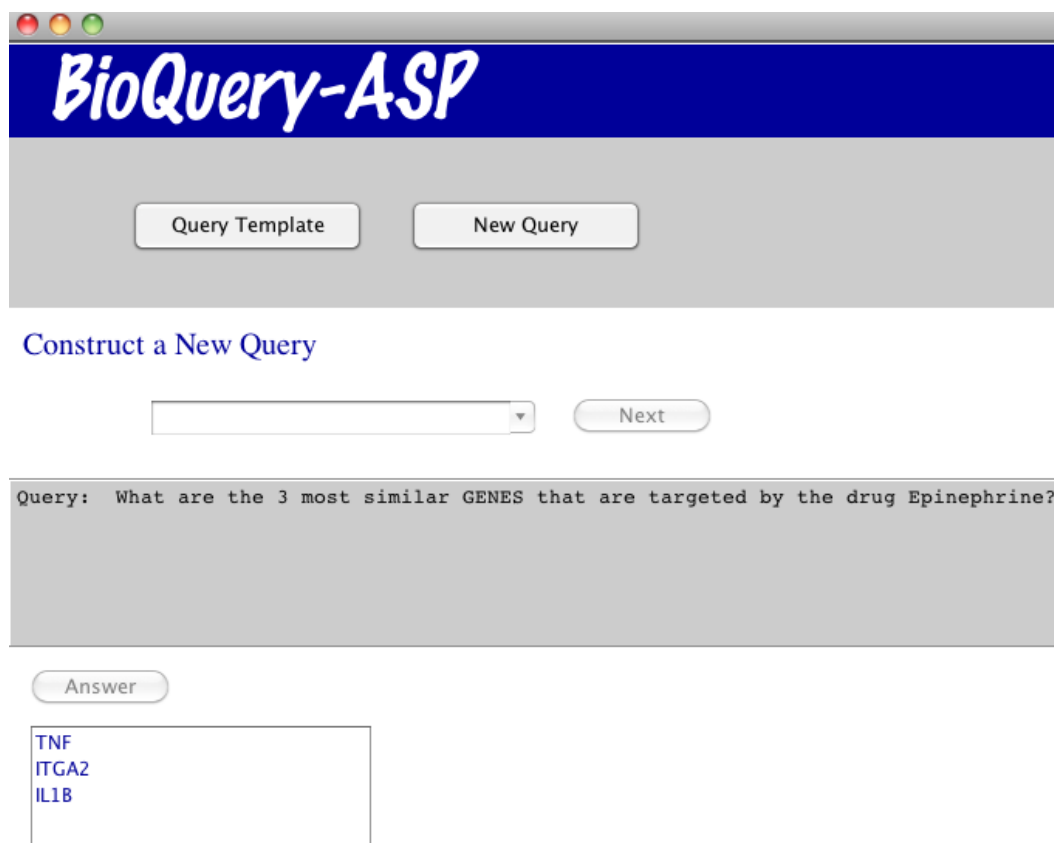


Figure 6.2: A screenshot of BIOQUERY-ASP. Users construct queries with the help of the intelligent user interface.

unnecessary. In that sense, in [38], we introduced an approach for finding the “relevant part” of the rule layer. Intuitively, $R \subseteq \Pi$ is a relevant part of the rule layer Π with respect to Q , if answer sets of $\Pi \cup Q$ and $R \cup Q$ coincide. Therefore, we can compute an answer to a query without considering the entire knowledge; instead we just use the relevant part of the rule layer. This helps BIOQUERY-ASP compute answers of the queries more efficiently in terms of computation time and memory.

6.2 Computing Similar/Diverse Genes

Functional similarity/diversity of genes is useful to perform further analysis while answering complex queries about genes. There are various measures to compute the similarity of two genes ([65, 75, 86, 66, 68]). We consider one of the recent systems GOSEMSIM⁷ to measure the similarity of genes. GOSEMSIM uses the gene ontology to measure the semantic and functional similarity of genes as in [108]. It takes two gene IDs as input and outputs a value between $[0, 1]$ that corresponds to the similarity of the genes. Let

⁷<http://www.bioconductor.org/packages/2.4/bioc/html/GOSemSim.html>

$S_g(G_1, G_2)$ be the similarity of the genes G_1 and G_2 due to GOSEMSIM. Then, the distance $D_g(G_1, G_2)$ between G_1 and G_2 is defined as follows:

$$D_g(G_1, G_2) = 1 - S_g(G_1, G_2)$$

We use this distance measure to compute the answers of the queries asking similar/diverse genes. Analogous to the n k -similar/diverse solutions, we define n k -similar/diverse genes. Since each solution is characterized by a gene, the ASP programs that represent similar/diverse queries should describe a single gene. For example, BIOQUERY-ASP transforms the query Q2 into the following ASP program:

```
1{similargene(GN):condition1(GN)}1.
condition1(GN) :- drug_gene("Epinephrine", GN).
answer_exists :- similargenes(GN).
:- not answer_exists.
```

We apply Online Method 3 to compute similar/diverse solutions for such queries. The distance function D_g is integrated into CLASP-NK. Whenever, we encounter a gene in the answer set, we compute the distance D_g by calling the GOSEMSIM software. Recall that in order to use CLASP-NK, we need to implement an admissible heuristic function to estimate the distance between any completion of a partial solution and the previously computed solutions. However, in queries about genes, each solution corresponds to a single gene; therefore, a partial solution can only be an empty set. Hence, we set the lower bound (resp. upper bound) for a partial solution as 0 (resp. ∞).

6.3 Experimental Results

Recall, in Section 6.1, that BIOQUERY-ASP identifies the relevant part of the rule layer in order to perform efficient reasoning. In order to show the effectiveness of this approach, we applied our methods to find answers to the queries Q1 and Q2.

To answer these queries, we considered the biomedical knowledge resources about genes, drugs and diseases, such as PHARMGKB, DRUGBANK, BIOGRID, CTD and SIDER. In particular, we extracted 347965 triples (as ASP facts) from BIOGRID, 17266 triples from DRUGBANK, 61102 triples from SIDER, 1809 triples from PHARMGKB, 1877799 triples from CTD.

Table 6.2 shows the computation times and the program sizes, with the complete rule layer and with the relevant part of the rule layer. All computation times are for a workstation with two 1.60GHz Intel Xeon E5310 Quad-Core Processor and 16 GB RAM, running Centos 64bit (Version 5.3). We used the ASP solver CLASP (Version 2.0.1) and CLASP-NK along with the grounder GRINGO (Version 3.0.3).

Table 6.2: Experimental results for answering queries Q1 and Q2.

Query	with the complete program	with the relevant part
Q1	36.1 sec. Rules: 3662195	7.3 sec. Rules:797129
Q2	104 sec. Rules: 3662159	65 sec. Rules: 309488

For the query Q1, CLASP takes 36.1 seconds to find an answer with the complete program containing 3662195 rules, whereas it takes 7.3 seconds to find an answer with the relevant part of the program containing 797129 rules. For the query Q2, CLASP-NK takes 104 seconds with the complete program containing 3662159 rules, whereas it takes 65 seconds to find an answer with the relevant part of the program containing 309488 rules. As can be seen from the results for both queries, it is advantageous to apply our method of query answering with respect to the relevant part of the program. In [38], we show the results of a more comprehensive list of queries which also indicates the effectiveness of identifying relevant part of a program for a given query.

Chapter 7

Related Work

Finding similar/diverse solutions has been studied in other areas such as propositional logic [2], constraint programming [58, 57], and planning [99].

Related Work in Propositional Logic In [2], the authors propose two algorithms, $DP_{distance}$ and $DP_{distance+lasso}$, to solve DISTANCE-SAT—determining that a propositional CNF formula has a model that disagrees with a given partial interpretation on at most d variables. Our modification of CLASP’s algorithm is similar to the first algorithm in that both algorithms check whether a partial interpretation computed in the DPLL-like search obeys the given distance constraints. On the other hand, unlike $DP_{distance}$, CLASP also uses conflict-driven learning: when it learns a conflicting set of literals, it will never try to select them in the later stages of the search. $DP_{distance+lasso}$ offers manipulations while selecting a new variable: it creates a set of candidate variables with respect to the distance function, computes weights of these variables relative to the distance function, and selects one with the maximum weight. On the other hand, in SELECT, CLASP creates a set of candidate variables, and selects one of the candidates to continue the search. Using the idea of $DP_{distance+lasso}$, we can modify CLASP further to manipulate the selection of variables with respect to the distance function.

Related Work in Constraint Programming In [58, 57], the authors study various computational problems related to finding similar/diverse solutions, considering Hamming distance as in [2]. They present an offline method (similar to our method) that applies clustering methods, and two online methods: one based on reformulation (similar to Online Method 1), the other based on a greedy algorithm (similar to Online Method 2) that iteratively computes a solution that maximizes similarity to previous solutions. The computation of a k -close solution is due to a Branch & Bound algorithm (similar to the idea behind Online Method 3) that propagates some similarity/diversity constraints specific to the given distance function. Our offline/online methods are inspired by these methods of

[58, 57].

Related Work in Planning In [99], the authors study domain independent approaches to compute diverse plans. They use Hamming distance to measure the distance among plans. They present a method (similar to our Online Method 1), where they add global constraints to the underlying constraint satisfaction solver of the GP-CSP planner [26]. As another method they present a greedy approach (similar to our Online Method 2), where they add global constraints which force the solver to compute k -diverse solutions in each iteration until it computes n solutions. They also present a method (similar to our Online Method 3) which modifies an existing planner's [53] heuristic function and computes n k -similar solutions in the search level.

Discussion Our work can be considered as complementary with this line of research, since we have studied finding similar/diverse solutions in the context of ASP. On the other hand, our methods have three main advantages compared to other approaches:

- they are not restricted to some domain-independent distance function, like (partial) Hamming distance considered in all the methods/tools mentioned above;
- depending on the particular ASP-based method, we can represent domain-independent or domain-specific distance functions in ASP or implement them in C++;
- we can use the definitions of distance functions modularly, without modifying the main problem description or without modifying the search algorithm or the implementation of the solver.

Thus, our ASP-based methods/tools for computing similar/diverse or close/distant solutions are applicable to various problems with different (often domain-specific) distance measures.

It might be possible to extend the SAT, CP, and planning based methods mentioned above to consider domain-specific distance measures. But even in such a case, our ASP-based methods/tools may be preferred when it is easier to represent the main problem in ASP, due to advantages inherited from the expressive representation language of ASP, which allows sophisticated definitions. For instance, reachability in a graph can be defined in ASP with a few rules, whereas with other approaches we need numerous rules to enumerate all possible paths. Some sample applications that exploit such features and point out the advantages of ASP over other approaches include phylogenetic network reconstruction [39] and wire routing [21, 41].

Chapter 8

Conclusion

In this thesis, we have studied computing similar/diverse solutions in the context of ASP. Our contributions are as follows:

- We have defined mainly two kinds of computational problems related to similar/diverse solutions: n k -similar (resp. k -diverse) solutions, and k -close (resp. k -distant) solution.
- We have introduced an Offline Method and three Online Methods to solve these problems.
- We have showed the applicability and effectiveness of these general methods on specific application domains such as phylogeny reconstruction, planning, and biomedical query answering. We have compared offline/online methods from the point of view of computational time and memory. The results of this work are summarized in [30, 31]. In this thesis, we have performed the experiments in [31] with newer versions of GRINGO (Version 3.0.3) and CLASP (Version 2.0.1), and we have improved the ASP programs in [31] with the help of the recent advances in the input language of these systems. In addition, we have developed and used a newer version of CLASP-NK that extends CLASP Version 2.0.1. In these new experiments, we have observed the followings.
 - Offline Method is useful when the solution space is small and it is easy to compute a single solution (e.g., phylogeny experiments).
 - Online Method 1 is preferable when we want to guarantee to find n k -similar (resp. k -diverse) solutions in a large solution space (so that Offline Method is not applicable).
 - Online Method 2 and Online Method 3 are efficient when computing approximate solutions in a large solution space (e.g., planning experiments).

- Online Method 3 is useful when the distance function cannot be represented as an ASP program (e.g., computing similar (resp. diverse) genes using GOSEM-SIM software).
- We have developed novel tools PHYLOCOMPARE-ASP and PHYLORECONSTRUCTN-ASP for comparing a given set of phylogenies, and reconstructing similar (resp. diverse) phylogenies directly. There was no such phylogenetic system that help experts analyze phylogenies by comparing or grouping them.
- No planner could compute similar (resp. diverse) plans with respect to a domain-specific measure; our methods have fulfilled this need in planning.
- We have developed BIOQUERY-ASP that can answer queries about the relations of biomedical concepts (e.g., drugs, diseases, genes, etc.). There was no biomedical query answering system that can integrate several knowledge resources to answer complex queries related to similar (resp. diverse) genes. In that sense, BIOQUERY-ASP is useful for drug discovery research which requires answering such complex queries. The results of this work are summarized in [38].

Future Work One line of future research can be improving the efficiency of the methods to find similar/diverse solutions. In particular:

- After computing all solutions with Offline Method, different clustering methods, such as nearest neighborhood search [93], can be applied to compute a set of similar/diverse solutions or a close/distant solution. Especially, when the distance measure has the metric property, we can model the problem as a proximity problem, where each solution corresponds to a point. Then we can apply the techniques mentioned in [95] to find collections of close/distant points.
- It might be possible to obtain a complete method using Online Method 2 or Online Method 3. Since the incompleteness of these methods is due to the initial solutions, by restarts one can guarantee to find n k -similar/diverse solutions.
- In addition, we may improve the performance of CLASP-NK by allowing it to propagate new literals with respect to the distance function which may lead to a more efficient solver.

Another line of future work could be to apply our methods for computing similar/diverse solutions to other appealing application domains. For instance, we may compute similar/diverse puzzles with the puzzle generation methods of [104], or configure similar/diverse products with respect to the user preferences using the approaches in [96].

It may also be useful to extend BIOQUERY-ASP to allow more specific and useful queries about biomedical concepts. For instance, we can define distance measures for drugs and answer complex queries related to similar/diverse drugs, which could help perform better decision making in drug discovery research.

Computing similar/diverse solutions could be useful to the fields outside the ASP. For instance, computing similar/diverse solutions may be helpful for better clustering; therefore, such an approach could be useful for learning techniques. We may also consider investigating the studies in local search since it is one of most widely used approaches in combinatorial optimization. It might be interesting to study finding similar/diverse approximate solutions with local search techniques and compare them with our ASP-based methods.

Appendix A

ASP Formulations

```
% generate n rooted trees
solution(1..x).
vertex(0..2*k).
root(2*k).

internal(X) :- vertex(X), not leaf(X).
2 {edge(S,X,Y) : vertex(Y) : X > Y} 2 :- internal(X), solution(S).

reachable(S,X,Y) :- edge(S,X,Y), X > Y.
reachable(S,X,Y) :- edge(S,X,Z), reachable(S,Z,Y), X > Z.
:- vertex(Y), not reachable(S,X,Y), root(X), Y != X, solution(S).
:- reachable(S,X,X).

maxY(S,X,Y) :- edge(S,X,Y), edge(S,X,Y1), Y > Y1.
:- maxY(S,X,Y), maxY(S,X1,Y1), Y > Y1, X < X1.
```

Figure A.1: A reformulation of the phylogeny reconstruction program of Brooks et. al., to find n distinct phylogenies: Part 1

```

% ensure that no tree has more than n incompatible characters
g0(N,X,I,S) :- f(X,I,S), informative_character(I),
  essential_state(I,S), solution(N). g0(N,Y,I,S) :- g0(N,X,I,S),
  g0(N,X1,I,S), edge(N,Y,X), edge(N,Y,X1), X>X1.
marked(N,X,I) :- g0(N,X,I,S).

g(N,X,I,S) :- g0(N,X,I,S).
{g(N,X,I,S): essential_state(I,S)} 1 :- internal(X),
  not marked(N,X,I), informative_character(I), solution(N).

{root_is(N,X,I,S)} :- g(N,X,I,S).
:- root_is(N,X,I,S), root_is(N,Y,I,S), X < Y.

% we need to consider every antecedent of x below
:- root_is(N,X,I,S), g(N,Y,I,S), reachable(N,Y,X), Y > X.

reachable_is(N,X,I,S) :- root_is(N,X,I,S).
reachable_is(N,X,I,S) :- g(N,X,I,S), reachable_is(N,Z,I,S),
  edge(N,Z,X), Z > X.

incompatible(N,I) :- g(N,X,I,S), not reachable_is(N,X,I,S).
:- n+1 {incompatible(N,I) : informative_character(I)}, solution(N).

% make sure that these n trees are distinct
different(S1,S2) :- edge(S1,X1,Y), edge(S2,X2,Y),
  S1 != S2, X1 != X2.
:- not different(S1,S2), solution(S1;S2), S1 != S2.

```

Figure A.2: A reformulation of the phylogeny reconstruction program of Brooks et. al., to find n distinct phylogenies: Part 2

```

maxdist(0..m).
vertexvertexdistancedomain(1..k+1).

% length of a path between vertices X and Y
tempnodaldistance(S,X,Y,T1+T2) :- distance_v(S,CA,X,T1),
    distance_v(S,CA,Y,T2), X < Y, leaf(X;Y).
notminnodal(S,X,Y,D1) :- tempnodaldistance(S,X,Y,D1),
    tempnodaldistance(S,X,Y,D2), D2 < D1.

% compute the nodal distances using distance_v.

% nodaldistance(S,X,Y,T): the nodal distance between X and Y
% in the S'th tree is T.
nodaldistance(S,X,Y,D) :- tempnodaldistance(S,X,Y,D),
    not notminnodal(S,X,Y,D).

% distance_v(S,X,Y,T): the distance between the vertex X and
% its descendant Y is T in the S'th tree.
distance_v(S,X,Y,1) :- edge(S,X,Y).
distance_v(S,X,Z,D+1) :- distance_v(S,X,Y,D), edge(S,Y,Z),
    vertexvertexdistancedomain(D).

% compute the differences of nodal distances of each pairs of
% leaves in each pairs of trees.
diffnodal(S1,S2,X,Y,#abs(D1-D2)) :- nodaldistance(S1,X,Y,D1),
    nodaldistance(S2,X,Y,D2), S2 > S1.

% compute the distance between each pairs of trees.
distance_t(S1,S2,K) :- solution(S1;S2), maxdist(K), S1 < S2,
    K#sum[diffnodal(S1,S2,X,Y,D) : leaf(X;Y) = D]K.

```

Figure A.3: A formulation of the nodal distance function D_n in ASP.

```

% Consider the vertices whose depth is 1,2,3
depthRange(0..r).
dist(0..m).
w(0,0).w(1,4).w(2,3).w(3,2).w(4,1).

% at each solution N, define reachability of leaf Y from X
reachableN(N,X,Y) :- edge(N,X,Y), vertex(X), leaf(Y), X > Y,
    solution(N).
reachableN(N,X,Y) :- edge(N,X,Z), reachableN(N,Z,Y), X > Z.

% at each solution S, assign depths to vertices Y
depth(S,2*k,0) :- solution(S).
depth(S,Y,T+1) :- depth(S,X,T), edge(S,X,Y), T<r.

diff(N1,V1,N2,V2) :- solution(N2), vertex(V2), N1 < N2,
    reachableN(N1,V1,X), not reachableN(N2,V2,X).
diff(N1,V1,N2,V2) :- solution(N1), vertex(V1), N1 < N2,
    not reachableN(N1,V1,X), reachableN(N2,V2,X).

fN(N1,V1,N2,V2,1) :- diff(N1,V1,N2,V2), N1 < N2.
fN(N1,V1,N2,V2,0) :- not diff(N1,V1,N2,V2), solution(N1;N2),
    vertex(V1;V2), N1 < N2.

depthFN(D,N1,N2,V1,V2,VAL) :- fN(N1,V1,N2,V2,VAL),
    depth(N1,V1,D), depth(N2,V2,D).

gN(0,N1,N2,0) :- solution(N1;N2), N1 < N2.
gN(D,N1,N2,D3) :- gN(D-1,N1,N2,D2),
    D1#sum[depthFN(D,N1,N2,V1,V2,VAL) : vertex(V1;V2) = VAL]D1,
    w(D,W), minmaxdepth(N1,N2,Y), D < Y+1, D3 = D2 + W*D1,
    dist(D1;D3), D > 0.

maxdepth(N1,N2,X) :- depth(N1,Y1,X), depth(N2,Y2,X), N1 < N2.
minmaxdepth(N1,N2,X) :- maxdepth(N1,N2,X),
    not maxdepth(N1,N2,X+1), N1 < N2.

% distance_t finds the distance between 2 phylogenies
distance_t(N1,N2,X) :- gN(D,N1,N2,X), N1 < N2, minmaxdepth(N1,N2,D).

```

Figure A.4: An ASP formulation of the descendant distance function D_l for two phylogenies.

```

% distance of a set of phylogenies
notmaxdistance_t(P1,P2,T1) :- distance_t(P1,P2,T1),
    distance_t(P3,P4,T2), T1 < T2.
delta(T) :- distance_t(P1,P2,T), not notmaxdistance_t(P1,P2,T).

% constraints on the distance function, for similarity
:- delta(T), T > k.

```

Figure A.5: An ASP formulation of the distance function Δ_D for a set of phylogenies, and the constraints for k -similarity.

```

goal :- time(T), goal(T).
:- not goal.

% effect of moving a block
on(B,L,T1) :- moveop(B,L,T), next(T,T1).

% a block can be moved only when it's clear
:- moveop(B,L,T), on(B1,B,T).

% any two blocks cannot be on the same block at the same time
:- 2{on(B1,B,T):block(B1)}, time(T), block(B).

% wherever a block is, it's not anywhere else
non(B,L1,T) :- location(L1), on(B,L,T), L != L1.

% every block is supported by the table
supported(B,T) :- on(B,table,T).
supported(B,T) :- on(B,B1,T), supported(B1,T), B != B1.
:- block(B), time(T), not supported(B,T).

% no concurrency
:- 2{moveop(B,L,T):block(B):location(L)}, time(T).

% inertia
on(B,L,T1) :- on(B,L,T), not non(B,L,T1), next(T,T1).

% initial values and actions are exogenous
1{non(B,L,0), on(B,L,0)}1 :- block(B), location(L).

{moveop(B,L,T)} :- block(B), location(L), time(T), T < lasttime.
:- non(B,L,T), on(B,L,T).

% auxiliary predicates
time(0..lasttime).
next(T,T+1) :- time(T), T < lasttime.
location(L) :- block(L).
location(table).

```

Figure A.6: Blocks World Formulation.

```

solution(1..x).

goal(S) :- goal(S,T).
:- not goal(S), solution(S).

% effect of moving a block
on(S,B,L,T1) :- moveop(S,B,L,T), next(T,T1).

% a block can be moved only when it's clear
:- moveop(S,B,L,T), on(S,B1,B,T).

% any two blocks cannot be on the same block at the same time
:- 2{on(S,B1,B,T):block(B1)}, time(T), block(B), solution(S).

% wherever a block is, it's not anywhere else
non(S,B,L1,T) :- on(S,B,L,T), location(L1), L != L1.

% every block is supported by the table
supported(S,B,T) :- on(S,B,table,T).
supported(S,B,T) :- on(S,B,B1,T), supported(S,B1,T), B != B1.
:- block(B), time(T), not supported(S,B,T), solution(S).

% no concurrency
:- 2{moveop(S,B,L,T):block(B):location(L)},time(T), solution(S).

% inertia
on(S,B,L,T1) :- on(S,B,L,T), not non(S,B,L,T1), next(T,T1).

% initial values and actions are exogenous
1{non(S,B,L,0),on(S,B,L,0)}1 :- block(B), location(L), solution(S).
{moveop(S,B,L,T)} :- block(B), location(L), time(T), T <
    lasttime, solution(S).
:- non(S,B,L,T), on(S,B,L,T).

% auxiliary predicates
time(0..lasttime).
next(T,T+1) :- time(T), T < lasttime.
location(L) :- block(L).
location(table).

% find distinct solutions
different(S1,S2) :- moveop(S1,X,Y,T), not moveop(S2,X,Y,T),
    solution(S2), S1 < S2.
different(S1,S2) :- not moveop(S1,X,Y,T), moveop(S2,X,Y,T),
    solution(S1), S1 < S2.
:- not different(S1,S2), solution(S1;S2), S1 < S2.

```

Figure A.7: A reformulation of the Blocks World program shown in Fig. A.6, to compute n distinct plans.

```

% for every time step T, check that the T'th actions
% of Plans P1 and P2 are different:
maxDist(0..lasttime).
different(S1,S2,T) :- moveop(S1,X,Y,T), not moveop(S2,X,Y,T),
    S1 < S2, solution(S2).
different(S1,S2,T) :- not moveop(S1,X,Y,T), moveop(S2,X,Y,T),
    S1 < S2, solution(S1).

% and define the hamming distance between two plans P1 and P2
% in terms of these differences:
hammingdistance(S1,S2,H) :- H{different(S1,S2,T): time(T)}H,
    maxDist(H), solution(S1;S2), S1 < S2.

```

Figure A.8: An ASP formulation of the Hamming distance D_h for two plans.

```

somedistance(H) :- hammingdistance(P1,P2,H).
notmaxdistance(H1) :- somedistance(H1), somedistance(H2), H2 > H1.
totaldistance(H) :- not notmaxdistance(H), somedistance(H).
:- totaldistance(H), H > k.

```

Figure A.9: An ASP formulation of the distance Δ_h for a set of plans and the constraint for k -similarity.

Bibliography

- [1] M. Alviano, W. Faber, and N. Leone. Disjunctive asp with functions: Decidable queries and effective computation. *TPLP*, 10(4-6):497–512, 2010.
- [2] O. Bailleux and P. Marquis. DISTANCE-SAT: complexity and algorithms. In *Proc. of AAAI*, pages 642–647, 1999.
- [3] M. Balduccini and M. Gelfond. Diagnostic reasoning with a-prolog. *CoRR*, cs.AI/0312040, 2003.
- [4] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
- [5] C. Baral, M. Gelfond, and J. N. Rushton. Probabilistic reasoning with answer sets. *TPLP*, 9(1):57–144, 2009.
- [6] C. Baral and C. Uyan. Declarative specification and solution of combinatorial auctions using logic programming. In *Proc. of LPNMR*, pages 186–199, 2001.
- [7] A. Biere, M. Heule, H. Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. 2009.
- [8] J. Bluis and D. G. Shin. Nodal distance algorithm: Calculating a phylogenetic tree comparison metric. In *Proc. of BIBE*, page 87, Washington, DC, USA, 2003. IEEE Computer Society.
- [9] G. Boenn, M. Brain, M. D. Vos, and J. Fitch. Anton: Composing logic and logic composing. In *Proc. of LPNMR*, pages 542–547, 2009.
- [10] G. Boenn, M. Brain, M. D. Vos, and J. Fitch. Automatic music composition using answer set programming. *CoRR*, abs/1006.4948, 2010.
- [11] G. Brewka. Preferences, contexts and answer sets. In *Proc. of ICLP*, page 22, 2007.
- [12] D. R. Brooks, E. Erdem, S. T. Erdogan, J. W. Minett, and D. Ringe. Inferring phylogenetic trees using answer set programming. *J. Autom. Reasoning*, 39(4):471–511, 2007.

- [13] D. R. Brooks and D. A. McLennan. *Phylogeny, Ecology, and Behavior: A Research Program in Comparative Biology*. University of Chicago Press, Chicago, IL, 1991.
- [14] D. Cakmak. Reconstructing weighted phylogenetic trees and phylogenetic networks using answer set programming. Master’s thesis, Sabanci University, 2010.
- [15] D. Cakmak, E. Erdem, and H. Erdogan. Computing weighted solutions in asp: Representation-based method vs. search-based method. 2011. To Appear in AMAI.
- [16] O. Caldiran, K. Haspalamutgil, A. Ok, C. Palaz, E. Erdem, and V. Patoglu. Bridging the gap between high-level reasoning and low-level control. In *Proc. of LP-NMR*, pages 342–354, 2009.
- [17] D. Calvanese, T. Eiter, and M. Ortiz. Regular path queries in expressive description logics with nominals. In *Proc. of IJCAI*, pages 714–720, 2009.
- [18] J. H. Camin and R. R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19:311–326, 1965.
- [19] G. Chafle, K. Dasgupta, A. Kumar, S. Mittal, and B. Srivastava. Adaptation in web service composition and execution. *Proc. of ICWS*, pages 549–557, 2006.
- [20] K. L. Clark. Negation as failure. *Logic and Databases*, pages 293–322, 1977.
- [21] E. Coban, E. Erdem, and F. Ture. Comparing ASP, CP, ILP on two challenging applications: Wire routing and haplotype inference. In *Proc. of the 2nd International Workshop on Logic and Search (LaSh 2008)*, 2008.
- [22] B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp, and L. Zhang. On distances between phylogenetic trees. In *Proc. of SODA*, pages 427–436, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.
- [23] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
- [24] J. P. Delgrande, T. Grote, and A. Hunter. A general approach to the verification of cryptographic protocols using answer set programming. In *Proc. of LPNMR*, pages 355–367, 2009.
- [25] J. Dix, T. Eiter, M. Fink, A. Polleres, and Y. Zhang. Monitoring agents using declarative planning. *Fundam. Inform.*, 57(2-4):345–370, 2003.
- [26] M. B. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001.

- [27] D. East and M. Truszczynski. More on wire routing with asp. In *Proc. of ASP*, 2001.
- [28] A. Edwards and L. L. Cavalli-Sforza. Reconstruction of evolutionary trees. *Phenetic and Phylogenetic Classification*, pages 67–76, 1964.
- [29] T. Eiter, G. Brewka, M. Dao-Tran, M. Fink, G. Ianni, and T. Krennwallner. Combining nonmonotonic knowledge bases with external sources. In *Proc. of FroCos*, pages 18–42, 2009.
- [30] T. Eiter, E. Erdem, H. Erdogan, and M. Fink. Finding similar or diverse solutions in answer set programming. In *Proc. of ICLP*, pages 342–356, 2009.
- [31] T. Eiter, E. Erdem, H. Erdogan, and M. Fink. Finding similar or diverse solutions in answer set programming. 2011. To Appear in TPLP.
- [32] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlvsystem. *AI Communications*, 12(1-2):99–111, 1999.
- [33] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer set planning under action costs. *Artificial Intelligence Research*, 19:25–71, 2003.
- [34] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Well-founded semantics for description logic programs in the semantic web. In *Proc. of RuleML*, pages 81–97, 2004.
- [35] E. Erdem. *Theory and applications of answer set programming*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, 2002.
- [36] E. Erdem. Phylo-asp: Phylogenetic systematics with answer set programming. In *Proc. of LPNMR*, pages 567–572, 2009.
- [37] E. Erdem, O. Erdem, and F. Türe. Haplo-asp: Haplotype inference using answer set programming. In *Proc. of LPNMR*, pages 573–578, 2009.
- [38] E. Erdem, Y. Erdem, H. Erdogan, and U. Oztok. Finding answers and generating explanations for complex biomedical queries. In *Proc. of AAAI*, 2011.
- [39] E. Erdem, V. Lifschitz, and D. Ringe. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming*, 6(5):539–558, 2006.
- [40] E. Erdem, V. Lifschitz, and M. F. Wong. Wire routing and satisfiability planning. In *In Proceedings CL-2000*, pages 822–836. Springer-Verlag. LNCS, 2000.
- [41] E. Erdem and M. D. F. Wong. Rectilinear Steiner Tree construction using answer set programming. In *Proc. of ICLP*, pages 386–399, 2004.

- [42] J. Felsenstein. Phylip. <http://evolution.genetics.washington.edu/phylip.html>, 2009.
- [43] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.
- [44] R. Finkel, V. W. Marek, and M. Truszczyński. Constraint lingo: A program for solving logic puzzles and other tabular constraint problems, 2002.
- [45] M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In *Proc. of KR*, 2010.
- [46] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In *Proc. of LPNMR*, pages 345–351, 2011.
- [47] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Proc. of LPNMR*, pages 260–265, 2007.
- [48] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In *Proc. of LPNMR*, pages 136–148, 2007.
- [49] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proc. of IJCAI*, pages 386–392, 2007.
- [50] M. Gebser, B. Kaufmann, and T. Schaub. Solution enumeration for projected Boolean search problems. In *Proc. of CPAIOR*, pages 71–86, 2009.
- [51] M. Gebser, T. Schaub, S. Thiele, and P. Veber. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming*, 11(2):1–38, 2011.
- [52] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of ICLP*, pages 1070–1080. MIT Press, 1988.
- [53] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in lpg. *J. Artif. Int. Res.*, 20(1):239–290, 2003.
- [54] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Automated Reasoning*, 36:345–377, 2006.
- [55] G. Gutin. *Handbook of Graph Theory*, chapter 5.3 Independent sets and cliques, pages 389–402. CRC Press, 2003.
- [56] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

- [57] E. Hebrard, B. Hnich, B. O’Sullivan, and T. Walsh. Finding diverse and similar solutions in constraint programming. In *Proc. of AAAI*, pages 372–377, 2005.
- [58] E. Hebrard, B. O’Sullivan, and T. Walsh. Distance constraints in constraint satisfaction. In *Proc. of IJCAI*, pages 106–111, 2007.
- [59] K. Heljanko and I. Niemela. Bounded ltl model checking with stable models. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 200–212. Springer-Verlag, 2003.
- [60] W. Hennig. Grundzuege einer theorie der phylogenetischen systematik. *Deutscher Zentralverlag*, 1950.
- [61] W. Hennig. Phylogenetic systematics. *Annu. Rev. Entomol.*, 10:97–116, 1965.
- [62] W. Hennig. Phylogenetic systematics. *University of Illinois Press*, 1966.
- [63] W. K. Hon, M. Y. Kao, and T. W. Lam. *Algorithms and Computation*, chapter Improved Phylogeny Comparisons: Non-shared Edges, Nearest Neighbor Interchanges, and Subtree Transfers, pages 369–382. Springer Berlin / Heidelberg, 2000.
- [64] K. Inoue and C. Sakama. Abductive framework for nonmonotonic theory change. In *IJCAI*, pages 204–210, 1995.
- [65] J. J. Jiang and David W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. In *Proc. of ROCLING*, 1997.
- [66] E. V. Kriventseva, W. Fleischmann, E. M. Zdobnov, and R. Apweiler. Clustr: a database of clusters of swiss-prot+trembl proteins. *Nucleic Acids Res*, 29:33–36, 2001.
- [67] M. Kunher and J. Felsenstein. A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. *Mol Biol Evol*, 3:459–468, 1994.
- [68] S. G. Lee, J. U. Hur, and Y. S. Kim. A graph-theoretic modeling on go space for biological interpretation of gene clusters. *Bioinformatics*, 20:381–388, 2004.
- [69] N. Leone, G. Greco, G. Ianni, V. Lio, G. Terracina, T. Eiter, W. Faber, M. Fink, G. Gottlob, R. Rosati, D. Lembo, M. Lenzerini, M. Ruzzi, E. Kalka, B. Nowicki, and W. Staniszkis. The infomix system for advanced integration of incomplete and inconsistent data. In *Proc. of SIGMOD*, pages 915–917, 2005.
- [70] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlvs system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7:499–562, 2006.

- [71] V. Lifschitz. Answer set planning. In *Proc. of ICLP*, pages 23–37, 1999.
- [72] V. Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.
- [73] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.
- [74] V. Lifschitz. What is answer set programming? In *Proc. of AAI*, 2008.
- [75] D. Lin. An information-theoretic definition of similarity. In *In Proceedings of the 15th International Conference on Machine Learning*, pages 296–304, 1998.
- [76] F. Lin and Y. Zhao. Assat: Computing answer sets of a logic program by sat solvers. pages 115–137, 2004.
- [77] L. Liu and M. Truszczyński. Pmodels - software to compute stable models by pseudoboolean solvers. In *Proc. of LPNMR*, pages 410–415, 2005.
- [78] D. V. Marina and D Vermeir. Logic programming agents and game theory, 2001.
- [79] S. Mcilraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proc. of KR*, pages 482–496, 2002.
- [80] A. Mileo, D. Merico, and R. Bisiani. Wireless sensor networks supporting context-aware reasoning in assisted living. In *Proc. of PETRA*, pages 1–2, 2008.
- [81] A. Mileo, D. Merico, and R. Bisiani. Non-monotonic reasoning supporting wireless sensor networks for intelligent monitoring: The sindi system. In *Proc. of LPNMR*, pages 585–590, 2009.
- [82] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proc. of DAC*, pages 530–535, 2001.
- [83] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *Proc. of LPNMR*, pages 421–430, 1997.
- [84] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An a-prolog decision support system for the space shuttle. In *Proc. of PADL*, pages 169–183. Springer, 2001.
- [85] T. M. Nye, P. Lio, and W. P. Gilks. A novel algorithm and web-based tool for comparing two alternative phylogenetic trees. *Bioinformatics*, 22(1):117–119, January 2006.

- [86] P. Resnik. Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. *Journal of Artificial Intelligence Research*, 11:95–130, 1999.
- [87] D. Ringe, T. Warnow, and A. Taylor. Indo-European and computational cladistics. *Transactions of the Philological Society*, 100(1):59–129, 2002.
- [88] D. F. Robinson and L. R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131–147, February 1981.
- [89] L. Ryan. Efficient algorithms for clause-learning sat solvers, 2004. MS Thesis, Simon Fraser University.
- [90] C. Sakama. Learning by answer sets. In *Proc. of AAAI Spring Symposium: Answer Set Programming*, 2001.
- [91] T. Schaub and S. Thiele. Metabolic network expansion with answer set programming. In *Proc. of ICLP*, pages 312–326, 2009.
- [92] T. Schaub and K. Wang. A comparative study of logic programs with preference. In *Proc. of IJCAI*, pages 597–602, 2001.
- [93] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing)*. The MIT Press, 2006.
- [94] P. Simons, I. Niemelä, and T. Soinen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.
- [95] M. Smid. *Handbook of Computational Geometry*, chapter Closest-point problems in computational geometry, pages 877–935. Elsevier Science B. V., 2000.
- [96] T. Soinen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proc. of PADL*, pages 305–319, 1998.
- [97] T. C. Son, E. Pontelli, and C. Sakama. Logic programming for multiagent planning with negotiation. In *Proc. of ICLP*, pages 99–114, 2009.
- [98] T. C. Son and C. Sakama. Reasoning and planning with cooperative actions for multiagents using answer set programming. In *Proc. of DALI*, pages 208–227, 2009.
- [99] B. Srivastava, T. A. Nguyen, A. Gerevini, S. Kambhampati, M. B. Do, and I. Serina. Domain independent approaches for finding diverse plans. In *Proc. of IJCAI*, pages 2016–2022, 2007.

- [100] D. L. Swofford. Paup*: Phylogenetic analysis under parsimony (and other methods), 2003. version 4.0. Sinauer Associates, Sunderland, Mass.
- [101] S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.C. Rousset, and R. A. Schmidt, editors. *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*. Springer, 2009.
- [102] J. Tiihonen, T. Soininen, and R. Sulonen. A practical tool for mass-customising configurable products. In *Proc. of the International Conference on Engineering Design*, pages 1290–1299, 2003.
- [103] N. Tran and C. Baral. Reasoning about triggered actions in ansprolog and its application to molecular interactions in cells. In *Proc. of KR*, pages 554–564, 2004.
- [104] M. Truszczynski, V. W. Marek, and R. A. Finkel. Generating cellular puzzles with logic programs. In *Proc. of ICAI*, pages 403–407, 2006.
- [105] F. Türe and E. Erdem. Efficient haplotype inference with answer set programming. In *Proc. of AAAI*, pages 1834–1835, 2008.
- [106] M. D. Vos, T. Crick, J. Padget, M. Brain, O. Cliffe, and J. Needham. A multi-agent platform using ordered choice logic programming. In *In Declarative Agent Languages and Technologies (DALT'05)*, pages 72–88, 2005.
- [107] M. D. Vos and D. Vermeir. Extending answer sets for logic programming agents. *Ann. Math. Artif. Intell.*, 42(1-3):103–139, 2004.
- [108] J. Z. Wang, Z. Du, R. Payattakool, S. P. Yu, and C. F. Chen. A new method to measure the semantic similarity of go terms. *Bioinformatics*, 23:1274–1281, 2007.
- [109] J.P White and J.F. O’Connell. *A Prehistory of Australia, New Guinea, and Sahul*. Academic, San Diego, CA, 1982.