CPSC-406 Report

Everett Prussak
Chapman University

April 14, 2023

Abstract

Consisting of CPSC 406 Material at Chapman University with Professor Alexander Kurz. This report will include an Introduction, Weekly Homework, and a Paper on the group project, which is done throughout the semester.

Contents

1 Introduction

This report...

2 Homework

This section contains solutions to homework.

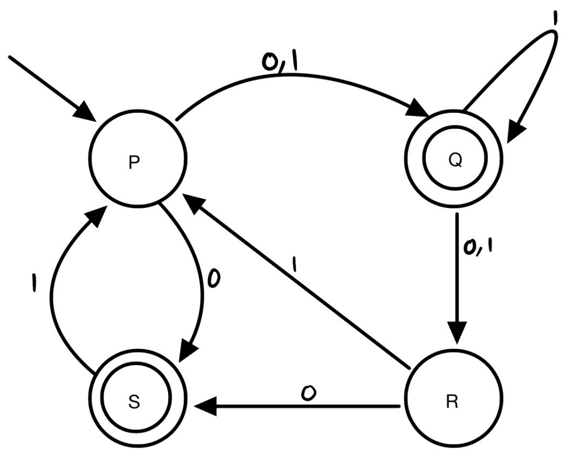
2.1 Week 2 (Homework 1)

This week's homework was to solve the following NFA:

Exercise 2.3.2: Convert to a DFA the following NFA:

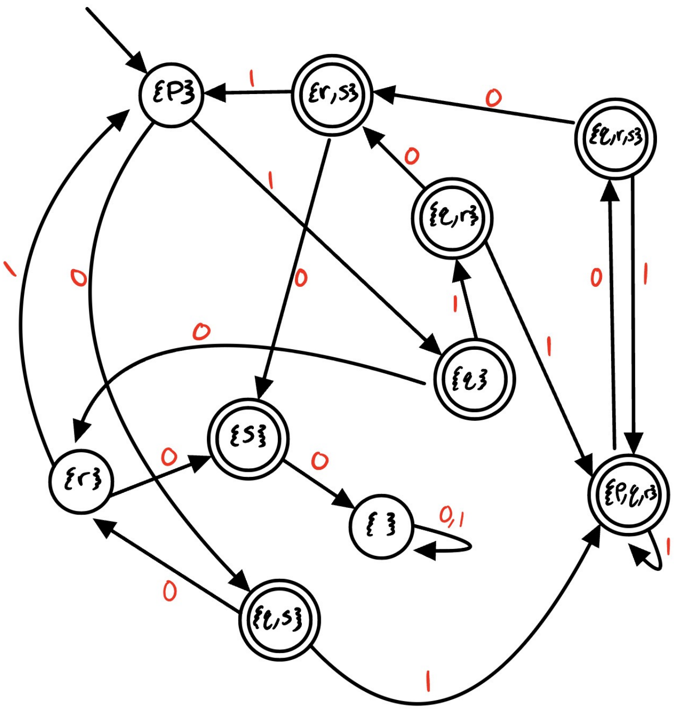
	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\emptyset$	$\{p\}$

This is the following NFA but drawn out:



From this NFA, the following DFA table can be made:

	0	1
→ {p}	{q, s}	{q}
*{p, s}	{r}	{p, q, r}
*{q}	{r}	{q, r}
{r}	{s}	{p}
*{p, q, r}	{q, r, s}	{p, q, r}
*{q, r}	{r, s}	{p, q, r}
*{s}	Ø	{p}
*{q, r, s}	{r, s}	{p, q, r}
*{r, s}	{s}	{p}
Ø	Ø	Ø



This DFA diagram will allow for the correct initial state, final states, and correct path for each input.

2.2 Week 3 (Homework 2)

Week 3 consisted of 2 Questions. They are in their respective sections.

2.2.1 Question 1

For Week 3 Question the object was to write the steps of the unification algorithm for each pair.

1. $f(X, f(X,Y)) \stackrel{?}{=} f(f(Y,a), f(U,b))$
2. $f(g(U), f(X,Y)) \stackrel{?}{=} f(X, f(Y,U))$
3. $h(U, f(g(V),W), g(W)) \stackrel{?}{=} h(f(X,b), U, Z)$

For number 1 of question 1 I got the following answer:

```
1. f(X,f(X,Y)) = f(f(Y,a),f(U,b))

  1. X = f(Y,a)
```

```
      o1 = [f(Y,a)/X]

  2. f(X,Y) = f(U,b)
      3. X = U
          o3 = [U/X]
      4. Y = b
          o4 = [b/Y]

  5. X(o3 * o4) = U, f(o3 * o4)(Y,a) = f(b,a)
      U = f(b, a)
      o5 = [f(b,a)/U]


o = o3 * o4 * o5 = [U/X, b/Y, f(b,a)/U]
X = U, Y = b, U = f(b,a)
```

Note: Sigma Symbol was not working in Verbatim, thus a simple lowercase o was subsitituted.

For number 2 of question 1 I got the following answer:

```
2. f(g(U),f(X,Y)) = f(X,f(Y,U))

  1. g(U) = X
       o1 = [g(U)/X]

  2. f(X,Y) = f(Y,U)
       3. X = Y
          o3 = [Y/X]
       4. Y = U
          o4 = [U/Y]

o = o1 * o2 * o3 = [X/U, Y/X, g(Y)/Y]
```

For number 3 of question 1 I got the following answer:

```
3. h(U,f(g(V),W),g(W)) = h(f(X,b),U,Z)

  1. U = f(X,b)
     o1 = [f(X,b)/U]

  2. f(g(V),W) = U
     o2 = [f(g(V),W)/U]

  3. g(W) = Z
     o3 = [g(W)/Z]

  4. U(o1 * o2) --> f(X,b) = f(g(V),W)

     5. X = g(V)
        o5 = [g(V)/X]
```

```
  6. b = W
     o6 = [b/W]

7. Z(o3 * o6) = g(W), W = b
   o7 = [g(b)/Z]

8. U(o1 * o2 * o6) --> f(X,b) = f(g(V), b)
   o8 [f(g(V),b)/U]


o = o5 * o6 * o7 * o8 = [f(g(V),b)/U, g(V)/X, b/W, g(b)/Z]


U = f(g(V),b), W = b, X = g(V), Z = g(b)
```

2.2.2 Question 2

For question 2, the task was to draw a SLD Recursion Tree for the following:

Question 2. Consider the following variant of the network connections problem.

```
% addr(X,Y) = X holds the address of Y
% serv(X) = X is an address server
% conn(X,Y) = X can initiate a connection to Y
% twoway(X,Y) = either end can initiate a connection

addr(a,d).
addr(a,b).
addr(b,c).
addr(c,a).

serv(b).

conn(X,Y):- addr(X,Y).
conn(X,Y):- addr(X,Z), serv(Z), addr(Z,Y).

twoway(X,Y):- conn(X,Y), conn(Y,X).
```

Draw the complete SLD-tree for this program together with the goal

```
?- twoway(W,a).
```

I got the following SLD Tree:

```
              twoway(W,a)
             /           \
            /             \
     conn(W,a)          conn(a,W)
     /        \          /       \
   W=b        W=d   W=c         no
               |              /  |  \
               no            b   d   c
```

twoway(W,a) becomes conn(W,a) and conn(a,W) because of the rule twoway(X,Y):- conn(X,Y), conn(Y,X). This becomes the first part of the tree. Then for the conn(X,Y), it becomes conn(W,a). This side of the tree will split into W=b and W=d. W=d eventually fails because there is no serv(d). However, W=b is successful because we have addr(a,b) and a a serv(b), which allows for the conn(b,a) to be true. On the right side of the tree, we have conn(a,W). This will split into w=c and no. Since c holds the address of a, and b holds the address of c, and serv(b) exists, we can create a conn(a,c) because of these factors. We see in the equation addr(X,Z), serv(Z), addr(Z,Y) can be applied here. In our case it would look similar to addr(a,b) serv(b) addr(b,c), which allows conn(b,c) to be true.

2.3 Week 6 (Homework 3)

The goal this week was to solve the following:

Use the *Method of Indirect Truth Tables* to show that the following formulas are valid (tautologies).

$$P \vee \neg P$$
$$(P \to Q) \to (\neg Q \to \neg P)$$
$$P \to (Q \to P)$$
$$(P \to Q) \vee (Q \to P)$$
$$((P \to Q) \to P) \to P$$
$$(P \vee Q) \wedge (\neg P \vee R) \to Q \vee R$$

The purpose of these exercises is not only to learn an algorithm but also to learn some laws of logic that are valid for reasoning in general. It is worth spending some time and trying to understand what these formulas mean. Can you find examples of how to use these formulas in an everyday argument?

Use the *Method of Indirect Truth Tables* to show that the following formulas are not valid, that is, find an interpretation of the propositional variables that makes the formula false.

$$(P \vee Q) \to (P \wedge Q)$$
$$(P \to Q) \to (\neg P \to \neg Q)$$

I split this into to two parts (Top and Bottom Questions).

2.3.1 Part 1 (Top)

Here is my answer for number 1:

```
1. P v ¬P

a. Abstract Syntax Tree:
   v
 /   \
P    ¬P


b. Label Root as 0:
   0
 /   \
P    ¬P


c. Use the truth table of the
connective at the root to draw
conclusions about the truth
values of the children:

P | ¬P | P v ¬P
T    F    T
F    T    T

d. Proceed recursively through the
syntax tree:
   T
 /   \
T     F

   T
 /   \
F     T


e. All of the possibilities result in a T. This is valid.
```

Here is my answer for number 2:

```
2. (P → Q) → (¬Q → ¬P)

a.
     →
   /   \
  →     →
/  \  /  \
P   Q  ¬Q  ¬P


b.
     0
   /   \
  →     →
/  \  /  \
P   Q  ¬Q  ¬P

c.

P | Q | ¬Q | ¬P | P → Q | ¬Q → ¬P | (P → Q) → (¬Q → ¬P)
F | F | T  | T  | T     |    T    |      T
F | T | F  | T  | T     |    T    |      T
T | F | T  | F  | F     |    T    |      T
T | T | F  | F  | T     |    T    |      T

d.
    __T__
   /     \
  → (T)   → (T)
/  \     /  \
T   T   ¬F  ¬F

e. All of the possibilities result in a T. This is valid.
```

Here is my answer for number 3:

```
3. P → (Q → P)

a.
   →
  / \
 P   →
    / \
   Q   P

b.
   0
  / \
 P   →
    / \
   Q   P

c.
P | Q | Q → P | P → (Q → P)
F | F |   T   |     T
F | T |   F   |     T
T | F |   T   |     T
T | T |   T   |     T



d.
   0 (T)
  / \
 F   → (F)
    / \
   T   F

e. All of the possibilities result in a T. This is valid.
```

Here is my answer for number 4:

```
4. (P → Q) v (Q → P)

a.
      __v__
     /     \
    →       →
   / \     / \
  P   Q   Q   P

b.
      __0__
     /     \
    →       →
   / \     / \
  P   Q   Q   P


c.
P | Q | P → Q | Q → P | (P → Q) v (Q → P)
F | F |   T   |   T   |       T
F | T |   T   |   F   |       T
T | F |   F   |   T   |       T
T | T |   T   |   T   |       T

d.
      __0__  (T)
     /     \
    → (F)      → (T)
   / \     / \
  T   F   F   T

e. All of the possibilities result in a T. This is valid.
```

Here is my answer for number 5:

```
5. ((P → Q) → P) → P

a.
          →
         / \
        →   P
       / \
      →   P
     / \
    P   Q


 b.
          0
         / \
        →   P
       / \
      →   P
     / \
    P   Q


c.

P | Q | P → Q | (P → Q) → P | ((P → Q) → P) → P
F | F |   T   |    T        |    T
F | T |   T   |    F        |    T
T | F |   F   |    T        |    T
T | T |   T   |    T        |    T
```

```
d.
             0 (T)
         /          \
       → (T)         F
     /      \
   → (T)     F
  /  \
 F    F
```

```
e. All of the possibilities result in a T. This is valid.
```

Here is my answer for number 6:

```
    6. (P ∨ Q) ∧ (¬P ∨ R) → Q ∨ R

   a.
                    →
                  /   \
                 /     \
                /       \
               ∧          ∨
            /    \       / \
           ∨      ∨     Q   R
          / \    /  \
         P   Q  ¬P   R

   b.
                    0
                  /   \
                 /     \
                /       \
               ∧          ∨
            /    \       / \
           ∨      ∨     Q   R
          / \    /  \
         P   Q  ¬P   R


   c.
```

P	Q	R	(P ∨ Q)	(¬P ∨ R)	(P ∨ Q) ∧ (¬P ∨ R)	Q ∨ R	(P ∨ Q) ∧ (¬P ∨ R) → Q ∨ R
F	F	F	F	T	F	F	T
F	F	T	F	T	F	T	T
F	T	F	T	T	T	T	T
F	T	T	T	T	T	T	T
T	F	F	T	F	F	F	T
T	F	T	T	T	T	T	T
T	T	F	T	F	F	T	T
T	T	T	T	T	T	T	T

```
d.
                 0 (T)
               /    \
              /      \
             /        \
           ∧ (F)          ∨ (F)
         /     \         / \
       ∨ (F)  ∨ (T)     F   F
      / \     /  \
     F   F  T    F

e. All of the possibilities result in a T. This is valid.
```

2.3.2 Part 1 (Bottom)

Here is my answer for number 1:

```
1. (P∨Q)→(P∧Q)

a.
         →
       /    \
      ∨      ∧
   /  \    /  \
  P   Q   P    Q

b.
         0
       /    \
      ∨      ∧
   /  \    /  \
  P   Q   P    Q


c.
P       | Q |   P ∨ Q   | P ∧ Q | (P ∨ Q) → (P ∧ Q)
F       | F |     F       |   F         | T
F       | T |     T       |   F         | F
T       | F |     T       |   F         | F
T       | T |     T       |   T         | T


d.
         0 ()
       /    \
      ∨ (T)  ∧ (F)
   /  \    /  \
  F    T   F   T


e. Not each one is true. We have some contrast here.
Only when P and Q are the same value will the overall be a T.
```

Here is my answer for number 2:

```
2. (P→Q)→(¬P→¬Q)

a.
        →
      /   \
     →       →
   /  \    /   \
  P   Q   ¬P   ¬ Q

b.
        0
      /   \
     →       →
   /  \    /   \
  P   Q   ¬P   ¬ Q


c.
P       | Q |   P → Q   | (¬P→¬Q)       | (P → Q) → (¬P→¬Q)
F       | F |     T       |   T         |    T
F       | T |     T       |   F         |    F
T       | F |     F       |   T         |    T
T       | T |     T       |   T         |    T


d.        0 (F)
      /   \
     → (T)      → (F)
   /  \    /   \
  F   T   T     F


e. Not each one is true. We have some contrast here.
Only when P and Q are the same value will the overall be a T.
```

Note: I had trouble converting some of the symbols I used in my .txt file in my Verbatim. I used many screenshots instead. I will add my .txt and .jpg's in a folder called homeworkMedia on github!

2.4 Week 9

This weeks tasks were to experiment with Spin, and solve 4 exercises.

Formulas:

```
[]  (success && bobAlice -> aliceBob)
[]  (success && aliceBob -> bobAlice)
```

Exercise 1: Go through the program, find these variables, and describe in plain language the meaning of these propositions above.

The 3 variables that will be discussed are success, bobAlice, and aliceBob. The variable success is a boolean variable. If the protocol runs to completion, then it will be set as 'True'. If it breaks somewhere down the line, then the protocol is not a 'success' and would be set to 'False'. The bobAlice variable is another boolean variable. If Bob receives a message from Alice then it would be set as 'True'. If Bob does not receive a message from Alice then it would be set as 'False'. Lastly, aliceBob is another boolean variable. If Alice receives a message from Bob then it would be set to 'True', if not then it would be 'False'.

With all of these variables explained, the first formula would read if there is a success in the completion of the protocol and Bob is receives a message from Alice, then Bob is communicating with Alice. The next formula would read if there is a success in the completion of the protocol and Alice is receives a message from Bob, then Alice is communicating with Bob.

Exercise 2: Which of the two formulas (from above) is verified as correct and which one is violated? What do we learn from this about the correctness of the protocol?

The second formula was verified as correct

```
[]  (success && aliceBob -> bobAlice)
```

While the first formula was violated

```
[]  (success && bobAlice -> aliceBob)
```

This tells us that the variables **success** and **bobAlice** were True boolean variables. The variable **aliceBob** was False, and this formula would result in violated truth table, with 1 -¿ 0 being an outcome of 0 as well, meaning violated.

Relevant Output (Violated Formula):

```
spin -a ns.pml; cc -o pan pan.c; ./pan -a

pan:1: assertion violated
!( !(( !((((statusA==1)&&(statusB==1))&&(partnerB==10)))||(partnerA==9)))) (at depth 83)
pan: wrote ns.pml.trail
```

Exercise 3: The property that is violated produces an execution sequence. How long is that execution sequence?

The number of execution steps I read from my terminal using this command:

```
spin -p -t ns.pml
```

was **84 execution sequences**.

Relevant Output:

```
vi ns.pml.trail

81:2:41
82:0:116
83:1:20
84:0:113
```

Exercise 4: Use Spin to produce an MSC that represents a successful attack on the protocol. Explain in detail why the MSC constitutes a successful attack.

To be considered a **successful attack**, the LTL-correctness properties are to be verified.

...

3 Paper

3.1 Introduction

Prolog is a logic programming language that has many useful applications. Used in a wide range of fields, such as artificial intelligence, computational linguistics, and more, Prolog has proven to be a versatile language that can handle reasoning. This paper will explore the history, features, and applications of Prolog. Features that will be included in this paper will consist of logical paradigm, unification, backtracking, and pattern matching, which all play vital roles for the success of Prolog. Applications such as natural language process, expert systems, and automated planning will be examined through an algorithm analysis. Comparisons of common languages such as Java, C++, and Python will also be made versus Prolog to get a better understanding of the importance for the relatively unknown language. Highlighting the strengths and weaknesses of Prolog in an algorithm analysis and comparison will allow Prolog to be used more efficiently and correctly. With this information, the possibilities of Prolog for future development will be more clear for the reader. This paper will examine important aspects of Prolog to showcase the value it brings to software developments and its ability to solve complex problems with relative ease.

3.2 Background

Prolog was created in the early 1970's by French computer scientist Alan Colmerauer and Phillip Roussel in Marseille, France (1) , who both attended the University of Aix-Marseille. The two computer scientists had started working on a programming language that could handle reasoning and symbolic computation. While building the programming language, Colmerauer and Roussel collaborated with American-British computer scientist Robert Kowalski (2). Kowalski had proposed the idea of a language based on Horn Clauses, which ended up being the basis of logical expressions in Prolog. In 1972, Roussel developed the first Prolog interpreter and artificial intelligence specialist David Warren created the first compiler (3), also known as Prolog I. This marked a significant milestone in the evolution of Prolog. Prolog, which stands for logic programming, also sometimes written as programming logic depending on who is asked, was aimed to be a programming language that was capable of handling reasoning and symbolic computation. Logic programming, uses logic to represent knowledge and the use of deduction to solve problems by deriving logical consequences (3),. The language was made to be declarative3, a high-level programming language that specifies what is to be done, rather than how to do it (4). In 1973, Prolog II was created, which has many of the common and important features we see in Prolog today (5). This version of the language featured the use of backtracking (5) and unification, more on these features later. Over the years, Prolog continued to evolve and gain popularity, with new features and implementations being continually developed. Today there are many versions of Prolog, which include GNU Prolog and Sicstus Prolog. The most notable version is SWI-Prolog, which is open source (6) and a community based version that highlights industry and degree use. The impact on the entire field of computer science cannot be overstated and allowed for advancements of artificial intelligence, natural language processing, and expert systems. Programming languages can seem to have some similarities or inspiration for Prolog, such as Haskell.

3.3 Features

3.3.1 Logical Paradigm

Prolog has many important features that can be applied to different things. One feature that makes Prolog distinct from other programming languages is its logical paradigm. Logical paradigm takes a declarative approach to solving problems, with its knowledge being represented in logical rules (1). As the program receives logical assertions based on various situations, prolog establishes them as facts. This idea is called Horn-Clauses. Logical rules created use Horn-Clauses to represent the rules and facts in the query written. Horn-Clauses have a head (left side) and a body (right side), but there are often times when there are headless Horn-Clauses which would disregard the head and only have the body (2). Here is few examples of using logical rules being expressed as Horn-Clauses:

```
Headless Horn-Clauses:
cat(Bob)
cat(Bob, Leon)
```

The first script would tell us Bob is a Cat.
The next script would explain more, telling us that Bob is a cat owned by Leon.

```
Headed Horn-Clauses:
animal(Bob):- cat(Bob, Leon)
```

This tells us that Bob is an animal, if Bob is a cat owned by Leon.

As you can see, the more rules and facts that get added, the more complicated the entire program will get for us to interpret. This is a huge reason why Prolog can be so useful because it can quickly identify relationships and solve the problem being asked. Many Prolog programs can be handled via pen and paper, but this would prove to be a much slower process than simply using the impressive programming language.

summary, the logical paradigm is a vital feature for Prolog. Providing Horn-Clauses and the basis for a declarative language, it solves user problems with relative ease. This feature is one of backings for the success of Prolog, and will continue to be.

To continue with the logical paradigm in Prolog, being a declarative programming language is another important aspect. In the Background, there was mention of Prolog being made to solve the task, rather than how to do the task. In conventional programming languages, an algorithm is formulated as a sequence of instructions that the computer must follow precisely in order to solve a problem (3). As previously mentioned, Horn-Clauses are used to create rules and facts in Prolog. Prolog uses these Horn-Clauses to create relationships between objects that the user created. The programmer will specify what it is they want to solve for, and Prolog will use the relationships, rules, and facts given and determine an answer. In some instances, depending on what is being asked, Prolog will answer the question with True, False, or many different data-types (4).

3.3.2 Unification

Unification is another important feature that sets Prolog apart from typical programming languages. Unification involves comparing two terms to see if they are identical or can be made to be identical. If they are already identical, then unification can be made. If they are not, then a process called binding variables must be done in order to make the terms unify with each other (5). Terms can be numbers, variables, atoms, or other structures.

When the process of unification between two terms takes place, Prolog will compare each term structure from left to right. When two constants are unified, Prolog will check if they are the same value. If they are not the same, then unification fails and the program backtracks, more on this later. If they are the same value, then unification is successful (6). Here is an example:

```
?- a = a.
Yes

?- a = b.
No
```

As you can see here, we know that a would be the same as a, and likewise that a is not the same as b. Prolog does this computation using unification, as mentioned above.

If they are not constants, but instead variables, Prolog binds them to the same value (6). Here is an example:

```
?- X = a.
X = a
Yes

?- X = Y.
X = _UNIQUE
Y = _UNIQUE
Yes
```

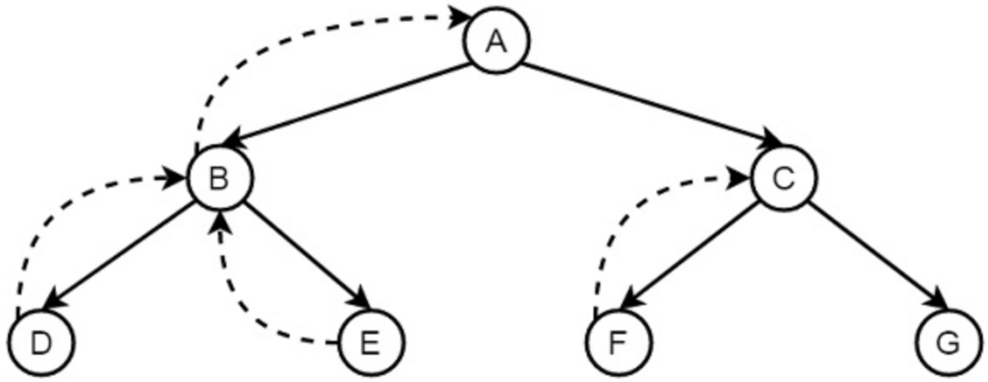
Unification will bind these two variables to a single, unique variable name. Likely the variable would not be called **underscore UNIQUE** but instead possibly an underscore followed by a sequence of random-like numbers. In Prolog, assignment of values to variables is different from typical programming languages. Instead of directly assigning the value to the variable, the variable is unified directly with the term. This means when the value of the term changes, the value of the variable changes as well (7). Unification is also used to match a query with a rule or fact in the knowledge base. To determine if the query is true or false, the query will be unified with the rule or fact. If the result is true, then the unification process generates a set of values for the variables in the query.

The use of Unificication proves to be a vital feature of Prolog. By unifying constants, atoms, variables, and more, to make terms identical, it allows Prolog to represent and reason about knowledge and find an answer for the user.

3.3.3 Backtracking

The next vital feature of Prolog that will be discussed is backtracking. Backtracking makes Prolog a much more powerful programming language because this concept allows Prolog to search multiple solutions and select the best. Backtracking allows Prolog to search for the truth value of different predicates by trying out various solutions and checking if they are correct (8). While backtracking is not unique to only Prolog, fields such as Artificial Intelligence use the backtracking algorithm as well, it can be a super useful tool and has found success in the Prolog language.

When the backtracking algorithm is computing, it will explore multiple paths of the search tree until a solution, or the best solution is found. When it runs into a leaf node with no solution found, the algorithm stores the current state of the program and will try another path. This will continue to occur until a solution is found, or every path has been taken (9)



There are numerous advantages of the backtracking algorithm for use in Prolog. By exploring multiple solutions to a single problem, the best choice can be decided. In typical single-pass algorithms this would not be possible. Another advantage is that backtracking makes Prolog much more flexible. This means that code can be adjusted and changed, but the answer can still find the same exact solution with enough information.

One main disadvantage is the possible computation time. Since there could be a lot of rules and facts given to Prolog, and many different possibilities, a solution may take a long time to find. This also means that the search space of the algorithm is very large as well. Computationally efficient algorithms are very important to have, but in most cases Prolog does compute in a reasonable amount of time.

3.3.4 Pattern Matching

The last Prolog feature that will be mentioned in this paper is pattern matching. Pattern matching is core feature of Prolog that allows manipulation of complex data with relative ease. Pattern matching is closely related and shares many of the same details and commands as Unification. The main difference is that Unification is the process of making two terms the same, while pattern matching is testing terms on a specific pattern (10).

The rules created by the user defines a pattern that can be followed. The pattern matching algorithm will determine if a set of terms matches the specific pattern. This will allow for the Prolog language as a whole to use new facts with other data. Simply, the program will be able match the new term to patterns that it has already defined from previous rules. Pattern matching is extremely important for Prolog, as Prolog follows the declarative programming paradigm that utilizes logical rules for defining relationships among entities. Pattern matching serves as an ideal algorithm to check if a specific input data works with a particular rule.

3.4 Applications

3.4.1 Natural Language Processing

Natural Language Processing, also known as NLP, is a field of computer science that focuses on giving machines the ability to comprehend text and write or speak similarly to humans (1). Immediately, many people can see the connection to Artificial Intelligence from NLP. By combing computational linguistics, statistical analysis, and deep learning models, NLP allows computers to process human langauges by writing in text or even speaking.

The features of Prolog allow Natural Language Processing to be accelerated in some instances. By allowing developers to define rules and relationships, Prolog can correctly interpret user input phrases easily (2). Again, since Prolog is a Declarative Programming Language, the answer is what the User would be provided with, rather than many programming steps. This is important because efficiency of receiving an answer is very important in today's world, where the longer it takes, the more impatient a user will become.

Another advantage of using Prolog for Natural Language Processing is the ability to directly translate one human language to another. More on this... ...

4 Conclusions

(approx 400 words) A critical reflection on the content of the course. Step back from the technical details. How does the course fit into the wider world of software engineering? What did you find most interesting or useful? What improvements would you suggest?

References

[ALG] Algorithm Analysis, Chapman University, 2023.

Paper Sources:
(1) a
(2) b
(3) c
(4) d
(5) e
(6) f

Features Sesction:
(1) 1
(2) 2
(3) 3
(4) 4
(5) 5
(6) 6
(7) 7
(8) 8
(9) 9
(10) 10

Applications Section:
(1) 1
(2) 2